HIGH TECH

ГЛЕН СМИТ
ПИТЕР ЛЕДБРУК

Grails

Гибкость Groovy и надежность Java

СИМВОЛ®

По договору между издательством «Символ-Плюс» и Интернет-магазином «Books.Ru – Книги России» единственный легальный способ получения данного файла с книгой ISBN 978-5-93286-185-1, название «Grails. Гибкость Groovy и надежность Java» – покупка в Интернет-магазине «Books.Ru – Книги России». Если Вы получили данный файл каким-либо другим образом, Вы нарушили международное законодательство и законодательство Российской Федерации об охране авторского права. Вам необходимо удалить данный файл, атакже сообщить издательству «Символ-Плюс» (piracy@symbol.ru), где именно Вы получили данный файл.

Grails in Action

Glen Smith, Peter Ledbrook

HIGH TECH

Grails

Гибкость Groovy и надежность Java

Глен Смит, Питер Ледбрук

Символ®

Санкт-Петербург — Москва
2010

Серия «High tech»

Глен Смит, Питер Ледбрук

Grails
Гибкость Groovy и надежность Java

Перевод А. Киселева

Главный редактор	*А. Галунов*
Зав. редакцией	*Н. Макарова*
Выпускающий редактор	*П. Щеголев*
Редактор	*О. Меркулова*
Корректор	*С. Минин*
Верстка	*К. Чубаров*

Смит Г., Ледбрук П.

Grails. Гибкость Groovy и надежность Java. – Пер. с англ. – СПб.: Символ-Плюс, 2010. – 656 с., ил.

ISBN 978-5-93286-185-1

«Grails. Гибкость Groovy и надежность Java» – книга для практиков от практиков. Grails – сверхпродуктивная веб-платформа с открытыми исходными текстами, позволяющая создавать полезные приложения, прикладывая минимум усилий. Благодаря использованию языка Groovy она предоставляет в распоряжение разработчика мощный Java-подобный язык, а также полный доступ ко всем Java-библиотекам. Кроме того, появляется возможность изменять поведение приложения прямо во время его выполнения без перезагрузки сервера.

Книга является всеобъемлющим руководством по платформе Grails. Сначала рассматриваются основы: модели предметной области, контроллеры, представления и службы. Дальше начинается практика – читателям предлагается приступить к созданию приложения в стиле Twitter, которое будет включать функции удаленного взаимодействия AJAX/JSON, анимационные эффекты, возможность поиска и даже интеграцию с системой обмена мгновенными сообщениями Jabber. Авторы знакомят читателя с множеством полезных расширений, которые могут придать приложению дополнительную функциональность, а также описывают особенности интеграции с существующими системами Java с помощью Spring и Hibernate.

ISBN 978-5-93286-185-1
ISBN 978-1-933988-93-1 (англ)

© Издательство Символ-Плюс, 2010
Authorized translation of the English edition © 2009 Manning Publications Co. This translation is published and sold by permission of Manning Publications Co., the owner of all rights to publish and sell the same.

Все права на данное издание защищены Законодательством РФ, включая право на полное или частичное воспроизведение в любой форме. Все товарные знаки или зарегистрированные товарные знаки, упоминаемые в настоящем издании, являются собственностью соответствующих фирм.

Издательство «Символ-Плюс». 199034, Санкт-Петербург, 16 линия, 7,
тел. (812) 324-5353, www.symbol.ru. Лицензия ЛП N 000054 от 25.12.98.

Подписано в печать 26.02.2010. Формат 70×100 $^{1}/_{16}$. Печать офсетная.
Объем 41 печ. л. Тираж 1200 экз. Заказ №
Отпечатано с готовых диапозитивов в ГУП «Типография «Наука»
199034, Санкт-Петербург, 9 линия, 12.

Моей Кайли,
которая изо дня в день позволяет мне
воплощать мои сумасшедшие мечты.
Я люблю тебя, мой друг!

Глен Смит

Моим родителям за то, что всегда рядом.

Питер Ледбрук

Оглавление

Вступительное слово 11
Предисловие 13

I. Введение в Grails 23

Глава 1. Коротко о Grails 25
1.1. Как появилась платформа Grails? 26
1.2. Установка 33
1.3. Пример программы Web 2.0 QOTD 34
1.4. Создание модели предметной области 44
1.5. Добавление действий в пользовательский интерфейс 47
1.6. Улучшение архитектуры 52
1.7. В заключение 59
2. Основы Grails 61
2.1. Введение 61
2.2. Исследование системы типов данных 70
2.3. Особенности, позволяющие экономить время 78
2.4. Специальные возможности Groovy 86
2.5. В заключение 100

II. Основы Grails 103

3. Моделирование предметной области 105
3.1. Hubbub: приступаем к созданию приложения 105
3.2. Первый объект класса предметной области 110
3.3. Проверка: барьер на пути мусора 116
3.4. Определение модели данных – 1:1, 1:m, m:n 123
3.5. В заключение 137
4. Включение модели в работу 139
4.1. Создание пользовательского интерфейса с помощью механизма скаффолдинга 139
4.2. Извлечение данных динамическими методами и механизм Query by Example 155
4.3. Более сложные способы выполнения запросов 165
4.4. Начальная загрузка исходных данных 172
4.5. В заключение 174

5. Управление потоком выполнения приложения 176
5.1. Основы контроллеров 176
5.2. Службы: повышаем надежность и удобство сопровождения приложения 191
5.3. Привязка данных 196
5.4. Управляющие объекты 202
5.5. Работа с изображениями 206
5.6. Обработка запросов фильтрами 211
5.7. Создание собственных отображений URL 215
5.8. В заключение 219

6. Разработка привлекательных представлений, форм и макетов 221
6.1. Основные теги форм 222
6.2. Расширение представлений собственными тегами 234
6.3. Добавление привлекательных макетов 240
6.4. Применение тегов Ajax 251
6.5. В заключение 260

7. Создание надежных приложений 262
7.1. Зачем необходимо тестировать программное обеспечение? 262
7.2. Модульное тестирование 265
7.3. Интеграционное тестирование 285
7.4. Функциональное тестирование 292
7.5. В заключение 299

III. Grails на каждый день 301

8. Расширения: добавляем поддержку Web 2.0 за 60 минут 303
8.1. Использование чужого труда в своих интересах 304
8.2. Добавление графиков и диаграмм 313
8.3. Добавление поддержки электронной почты 320
8.4. Полнотекстовый поиск: реализуем свою процедуру поиска . 325
8.5. Улучшение интерфейса с помощью GrailsUI 338
8.6. В заключение 346

9. Реализация мастеров и последовательностей с помощью веб-потоков 348
9.1. Что такое веб-поток? 349
9.2. Работа с веб-потоками 356
9.3. Дополнительные особенности веб-потоков 363
9.4. Тестирование веб-потоков 371
9.5. В заключение 378

10. Не впускайте незнакомцев, или к вопросу о безопасности 381
10.1. Почему безопасность имеет такое значение 381
10.2. Защита от злоумышленников 382
10.3. Управление доступом 395
10.4. Дальнейшее исследование Spring Security 410
10.5. В заключение 418

11. Удаленный доступ 421
11.1. Решение на базе REST 422
11.2. Согласование формата представления 434
11.3. Практика реализации интерфейса REST 443
11.4. Командно-ориентированные удаленные взаимодействия 451
11.5. В заключение 458

12. Обмен сообщениями и планирование заданий 461
12.1. Автостопом по работе с сообщениями 462
12.2. Использование расширения JMS для платформы Grails 468
12.3. Механизм планирования заданий платформы Grails 476
12.4. Дополнительные возможности планирования 480
12.5. В заключение 488

IV. Расширенные возможности Grails 491

13. Тонкое искусство владения механизмом GORM 493
13.1. Искусство моделирования предметной области 494
13.2. Искусство кэширования: переходим от 2 пользователей к 2^{10} 499
13.3. Интеграция с имеющимся ПО: работа с несколькими источниками данных 511
13.4. Сложности взаимодействий с существующими базами данных 513
13.5. В заключение 528

14. Платформа Spring и транзакции 531
14.1. Spring и Grails 532
14.2. Использование транзакций в механизме GORM 545
14.3. В заключение 555

15. После компиляции, тестирования и запуска 557
15.1. Система сборки платформы Grails 559
15.2. Автоматизированная сборка – не для любителей 577
15.3. Обслуживание изменений в модели данных 585
15.4. В заключение 590

16. Разработка расширений 592
16.1. Создание нашего первого расширения 593
16.2. Публикация расширений 603
16.3. Интеграция с платформой Grails 608
16.4. В заключение 626
Алфавитный указатель 628

Вступительное слово

Эта книга для практиков от практиков. Серия статей «20-часовой марафон» Глена, в которой он описал, как в течение 20 часов ему удалось реализовать сайт *www.groovyblogs.org* на платформе Grails, стала легендой. Этот сайт до сих пор остается для меня единственным источником самой свежей информации о языке программирования Groovy и платформе Grails. Мало того что Глен является одним из лидеров проекта Grails, он также всегда готов поделиться своими достижениями и знаниями с другими: предоставляет свой программный код, публикует статьи в блоге и обсуждает проблемы с коллегами. Я давно хотел, чтобы он собрал весь свой опыт, оформив его в виде книги, и просто не верится, что сейчас я держу эту книгу в своих руках.

«Фронтовой» опыт Глена идеально дополняется глубокими знаниями Питера, которые он демонстрировал тысячи раз, отвечая на вопросы пользователей Grails в рассылке. Если кто-то на этой планете и знает все ответы на вопросы о Grails, то это он.

Создание хорошей книги о Grails – тяжелый труд. Как автор вы испытываете желание представить как можно больше особенностей, но при этом вам также хочется объяснить, насколько упрощает платформа Grails разработку веб-приложений на языке Java.

Глен и Питер сумели найти золотую середину, обеспечив простое введение в мир Grails и охватив проблемы, с которыми пользователю, вероятно, придется столкнуться при использовании платформы Grails в проектах с высокими требованиями.

Данная книга о Grails – это практическое руководство. Она проведет читателя через весь процесс создания приложения на платформе Grails и по пути расскажет о некоторых тонкостях, к числу которых относятся различные составляющие технологии: модели, тесты, представления, контроллеры, службы и прочие, а также, что, пожалуй, еще более важно, последовательно объяснит, что и за чем следует.

Я могу только посоветовать идти путем, предложенным авторами, при создании своего первого приложения. Когда позднее у вас появится желание отклониться от предлагаемого пути, вы найдете все необходимые рекомендации в последних главах.

Чтение этой книги немного напоминает беседу с Гленом. За строками почти явно слышится его австралийский акцент, а из-за искрометного и полного юмора стиля изложения он напоминает мне «восторженного коалу».

Связь между манерой изложения, принятой в этой книге, и платформой Grails не случайна. Такая манера упрощает объяснение сложных решений.

Удачи!

Дирк Кёниг (*Dierk König*)
автор книги «Groovy in Action»[1]

[1] Издательство Символ-Плюс планирует выпустить перевод этой книги осенью 2010 года.

Предисловие

Мы оба пришли к платформе Grails несколько лет тому назад, пытаясь найти более простой и более удобный способ создания веб-приложений на языке Java. Хотя существует большое количество весьма интересных платформ, написанных на языке Java, те из них, которые мы попробовали применять, показались нам неудобными и требовали кропотливой настройки. В то время в самом разгаре была рекламная шумиха вокруг платформы Ruby on Rails. В ней привлекала скорость и простота разработки веб-приложений, но ее использование означало бы уход от языка Java.

И в это время появилась платформа Grails. Для работы с ней достаточно было использовать простой текстовый редактор, а производительность труда при этом оказывалась чрезвычайно высокой. Она стала глотком свежего воздуха для давних разработчиков на языке Java, таких как мы. Даже с помощью версии 0.1 мы могли создавать полезные приложения, прикладывая минимум усилий. С тех пор платформа Grails стала одним из ведущих игроков и может использоваться для воплощения широкого круга проектов, как больших, так и маленьких. Я занялся переносом моих незавершенных проектов SpringMVC на платформу Grails, и мне это удалось! Мое первое приложение SMS Gateway вышло на платформе Grails 0.2 beta. В то же время Питер тайно, по-партизански внедрял платформу Grails в своей компании.

Как многое хорошее в нашей жизни, эта книга началась с чашки горячего кофе. Будучи на конференции Groovy/Grails Experience (2GX) в Вашингтоне, я болтал с Дирком Кёнигом (Dierk König) (легендой Groovy) за завтраком. Поговорив о том, как движется его работа над книгой «Groovy in Action» для издательства Manning (которая остается нашей самой любимой книгой о программировании), мы перешли к обсуждению возможности создания книги о платформе Grails, которая содержала бы практические сведения, действительно необходимые разработчикам, а не описание всех возможных ситуаций. Дирк тогда сказал мне: «Глен, если ты напишешь эту книгу, я куплю первый же экземпляр». Такая поддержка обеспечила почву для проекта. К тому времени Питер потратил уже много часов на поддержку Grails и собрал множество фактических данных о насущных потребностях разработчиков.

Примерно через месяц или два издательство Manning обратилось с вопросом – нет ли желающих написать такую книгу. Момент был подходящий, и мы ухватились за эту возможность. Несмотря на бум по-

пулярности платформы Grails, долгое время существовала всего одна книга, посвященная этой платформе, поэтому все говорило о необходимости книги «Grails in Action». Сообщество Grails чрезвычайно отзывчиво; когда я показал Грэму Рочеру (Graeme Rocher) первый набросок оглавления, он предложил обратиться к Питеру Ледбруку (Peter Ledbrook) и Джеффу Брауну (Jeff Brown) как к потенциальным соавторам. Однако Джефф тогда как раз заканчивал работу над книгой Грэма, а вот Питер взошел на борт корабля «Grails». Такое сотрудничество наглядно демонстрирует дух дружбы и взаимовыручки в экосистеме Grails.

Несмотря на то что мы «географические антиподы» (Питер живет в Лондоне, а я в столице Австралии), мы нашли способ еженедельно связываться друг с другом и обсуждать проблемы создания книги, задействовав службу звонков Skype (в редкие свободные часы). Питер является одним из основных разработчиков Grails и вложил в поддержку этой платформы больше, чем кто-либо другой на планете. Именно благодаря ему приобрела конкретные очертания идея подготовить такой материал, который насущно необходим каждому разработчику приложений на Grails. Кроме того, им написано множество основных компонентов платформы (включая подсистему тестирования, расширение JSecurity и большую часть подсистемы расширений). Его знание «внутренней сущности» Grails внесло свои коррективы в темы, обсуждаемые в книге, и превратило ее в полное собрание материалов, необходимых для изучения ключевых компонентов платформы. Чувство юмора и дружелюбная натура Питера сделали его идеальным соавтором. О таком сотрудничестве приятно вспомнить!

На страницах этой книги мы попытались не просто научить пользоваться платформой Grails, но и поделиться нашим богатым практическим опытом работы с ней. Надеемся, что она станет бесценным помощником для профессиональных разработчиков. Честно признаться, мы могли бы добавить в книгу еще больше информации, но думаем, что нам удалось найти оптимальное соотношение между объемом содержимого и его полезностью. Хотелось бы верить, что книга понравится нашим читателям и поможет им быстро и легко освоить весь потенциал Grails.

Глен Смит

Благодарности

Как и с любым новым делом, никогда не знаешь, во что впутываешься, когда его начинаешь. Это мы ясно осознали, приступив к работе над своей первой книгой. К счастью, нам помогала отличная команда сотрудников издательства Manning, которая направляла наш труд и всячески помогала создать нечто такое, чем мы могли бы впоследствии гордиться. Мы хотели бы сказать спасибо всем, кто участвовал в процессе выпуска этого издания. Особое спасибо нашему редактору-консультанту

Синтии Кейн (Cynthia Kane), которая сумела вынести все – даже наши ночные телефонные конференции – и помогла нам сохранить высокую мотивацию. Наш редактор Анди Кэрролл (Andy Carroll) отлично справился с «выжиманием воды» из нашей прозы и не раз указывал на различные неудобочитаемые или противоречивые места.

Мы надеемся, что нам удалось соблюсти 100-процентную точность сведений в этой книге, чего было бы сложно добиться без острого глаза Барта Беквиза (Burt Beckwith). Он помог нам обеспечить высочайшую точность формулировок, выявив технические ошибки и двусмысленности. Мы хотели бы также выразить свою благодарность нашим рецензентам за то, что они согласились потратить свое время на составление подробных отзывов о рукописи на разных этапах ее создания: Джошу Риду (Josh Reed), Полу Стасиаку (Paul Stusiak), Биллу Флаю (Bill Fly), Кеннету Делонгу (Kenneth DeLong), Лестеру Лобо (Lester Lobo), Питеру Джонсону (Peter Johnson), Роберту О'Коннору (Robert O'Connor), Джо Макти (Joe McTee), Джонасу Банди (Jonas Bandi), Тиму Муру (Tim Moore), Кэрол Макдональд (Carol McDonald), Дейву Клейну (Dave Klein), Дугу Уоррену (Doug Warren), Гордону Диккенсу (Gordon Dickens), Джону Гутри (John Guthrie), Джону Дж. Ледо (John G. Ledo), Кенрику Чину (Kenrick Chien), Ли Баттсу (Lee Butts), Марку Иглу (Mark Eagle), Полу Кингу (Paul King), Роберту Флетчеру (Robert Fletcher) и Зану Трашу (Zan Thrash).

Первые подписчики тоже оказали нам немалую помощь, обратив внимание на опечатки, грамматические ошибки, запутывающие формулировки и другие проблемы. Возможно, еще более важными для нас были полученные от них положительные отзывы, которые помогли нам сохранить вдохновение.

Спасибо также Дирку Кёнигу (Dierk König) за его поддержку и за согласие написать предисловие к нашей книге. И напоследок, но не в последнюю очередь, мы хотели бы поблагодарить Грэма Рочера (Graeme Rocher) за создание такой великолепной платформы, которая снова превратила разработку программного обеспечения в увлекательное занятие.

Глен Смит

Когда я был ребенком, один из моих учителей, Пол Ле Льевре (Paul Le Lievre), говорил мне: «Глен, нет такой вещи, как бесплатный обед. Он бесплатный лишь потому, что кто-то уже заплатил за него». Хорошее замечание. Всегда кто-то платит. Так произошло и с этой книгой – она оплачена удивительным долготерпением моей супруги Кайли. Более года она жила жизнью «книжной вдовы», практически в одиночку воспитывая наших замечательных деток и терпя мой раздражительный и взрывной характер все время, пока готовилось издание. Друг мой, ты – лучшая! Считай, что этой книге мне пришлось принести в жертву многие часы нашего с тобой семейного досуга и несчетного числа беззаботных выходных. Мои милые детки, Исаак и Зои, также долгое время

не имели полноценной возможности общаться с отцом. Я очень люблю вас, ребята! Папа дома, и впереди вас ждут продолжительные прогулки на велосипедах и бесконечные крепкие объятия!

Мои родители, Элби и Ян Смит, и родители жены, Стив и Джой Салтер, оказали большое содействие этому проекту и здорово помогли в воспитании детей. Спасибо вам за поддержку!

Джо Гиддингс (Jo Giddings), мой непосредственный начальник, дал зеленый свет этой книге, великодушно согласившись предоставить мне сокращенный рабочий день, чтобы я мог заниматься ею. Дин Макаулай (Dean Macaulay) нередко брал часть моих обязанностей на себя и постоянно подбадривал меня в работе. Вы удивительные люди!

Свен Хайгс (Sven Haiges), мой соведущий по подкасту Grails (*grailspodcast.com*), с большим пониманием относился к этому проекту и мужественно выносил гремучую смесь моего характера и вспышек раздражительности от сумасшедшего метания по часовым поясам на заключительной стадии работы. До встречи на следующих 100 передачах, приятель!

Большая часть этих глав была написана в часы обеда в местном кафе «The Tea Caddy» (где, кстати, готовят лучшие в Канберре сэндвичи с беконом, томатами и салатом), что долгое время позволяло мне существенно удлинять мой рабочий день. Спасибо вам!

Наконец, мой соавтор, Питер Ледбрук, был для меня спокойным и мудрым голосом поддержки, когда я приходил в неистовство, утопая в ворохе незавершенных дел. Он очень скромный и сдержанный парень, который всегда стремится помочь товарищу. Он знает о Grails больше, чем кто-либо, и в этой книге его перу принадлежит весь технически сложный материал. В ходе работы над проектом он стал моим настоящим другом, и я счастлив и горд этим обстоятельством.

Питер Ледбрук

Прежде всего я хотел бы поблагодарить моего соавтора, Глена Смита, чьи энергия и энтузиазм не знают границ. Без его оптимизма и поддержки написать эту книгу было бы гораздо сложнее.

Мой друг Линда Джордан заслуживает отдельного упоминания за ее бесценные советы литературного плана, за то, что поддерживала меня, когда я в этом нуждался, и за то, что помогла мне не пасть духом и закончить начатое. Спасибо также моим родителям за то, что кормили меня и следили, чтобы я делал перерывы, когда это было необходимо.

Я хотел бы также сказать спасибо ребятам из G2One/SpringSource за то, что дали мне время поработать над книгой, и за то, что обеспечивали меня работой, благодаря которой я мог заниматься платформой Grails полный рабочий день. И да продолжит катить дальше мой тяжелый грузовик Groovy и Grails!

Об этой книге

Данная книга – это всеобъемлющее введение в платформу Grails, детально охватывающее все ее базовые компоненты: контроллеры, представления, службы, веб-потоки, библиотеки тегов, поддержку Ajax и расширения. Однако она представляет собой нечто большее, чем просто введение, – книга наполнена практическими знаниями и методами, способными помочь решить проблемы, с которыми часто сталкиваются разработчики веб-приложений.

Краткое содержание книги

Эта книга представляет собой углубленное, но в то же время доступное введение в мир гибкой разработки веб-приложений.

Она делится на следующие четыре части:

- Часть 1 «Введение в Grails»
- Часть 2 «Основы Grails»
- Часть 3 «Grails на каждый день»
- Часть 4 «Расширенные возможности Grails»

Первая часть знакомит читателя с платформой Grails и поможет создать первое простое приложение на этой платформе –«Цитата дня». Здесь дается представление обо всех компонентах экосистемы Grails и информация о том, как объединяются эти компоненты в законченные приложения. Но для создания более сложных приложений нам потребуется знание Groovy – динамического языка программирования, на котором основывается разработка приложений для платформы Grails. Поэтому мы потратим некоторое время на изучение его основ.

Во второй части мы приступим к более глубокому изучению основных компонентов Grails. Здесь рассказывается, как взаимодействуют между собой модели, представления и контроллеры, а также дается глубокий анализ всех важнейших особенностей, составляющих основу приложений на платформе Grails. Мы представим пример простого приложения Hubbub, реализующего социальную сеть и использующего все особенности, которые часто можно обнаружить в приложениях на платформе Grails: моделирование предметных областей, выполнение запросов, оформление внешнего вида, обработку форм и многое другое. Закончив изучение второй части, можно уже вполне уверенно браться за разработку своих собственных несложных приложений.

Но так как создание настоящих приложений заключается не только в работе с формами и базами данных, то в третьей части мы займемся развитием навыков, которые потребуются для вывода своих приложений в мир. Мы исследуем различные стратегии тестирования, позволяющие гарантировать корректную реализацию программного кода, и покажем, как задействовать в приложениях особенности Web 2.0 за счет

использования сторонних расширений. Ни одно приложение не может считаться законченным, если в нем не используется какая-либо модель обеспечения безопасности, поэтому мы также исследуем проблемы безопасности приложений на платформе Grails в Интернете. В заключение мы познакомимся со стратегиями удаленных взаимодействий с использованием прикладного интерфейса REST и завершим эту часть обзором асинхронных технологий, которые приобретают все большую популярность при разработке масштабируемых приложений.

В четвертой части мы завершим наш тур по платформе Grails изучением наиболее продвинутых особенностей платформы. Мы посмотрим, как можно организовать доступ к данным для достижения максимальной производительности и обеспечить взаимодействие с базами данных, существовавшими ранее, а также гораздо глубже заглянем внутрь платформы Grails и познакомимся с технологиями Spring и Hibernate, лежащими в ее основе. Кроме того, мы покажем, как интегрировать Grails с существующими процедурами сборки. В заключение мы детально обсудим вопросы создания и распространения ваших собственных расширений для платформы Grails.

Кому адресована книга

Если вы закаленный программист на языке Java, готовый окунуться в воды динамических веб-платформ, или опытный веб-разработчик, желающий осуществить переход к новейшей парадигме программирования преимущества соглашений перед настройкой (Convention over Configuration), тогда эта книга для вас. Она предоставит все необходимое для обеспечения высокой производительности труда и практические знания, которые помогут выйти из возможной тупиковой ситуации. Предполагается, что читатель обладает некоторым опытом веб-разработки (в частности, CSS, HTML и JavaScript) и знаком с основами программирования. Наличие предыдущего опыта веб-программирования на языке Java желательно, тем не менее мы будем объяснять некоторые базовые понятия веб-программирования на языке Java во врезках, где это будет необходимо. Если у вас имеется опыт программирования на других языках (таких как Ruby, Perl или PHP), переход на платформу Grails покажется вам вполне естественным.

Соглашения по оформлению программного кода

В книге приводится множество примеров, демонстрирующих принципы использования каждого из рассматриваемых аспектов. Программный код в листингах или в тексте оформляется моноширинным шрифтом, чтобы отделить его от обычного текста. Имена классов и методов, свойства объектов и другие элементы программного кода, а также значения переменных будут представлены тем же моноширинным шрифтом.

Фрагменты программного кода и примеры данных, вводимых в командной строке, могут иметь значительный объем. Во многих случаях исходный программный код (доступный для загрузки) был дополнительно отформатирован – мы добавили разрывы строк и отступы, чтобы уместить его по ширине книжной страницы. В редких случаях, когда даже этого оказывалось недостаточно, мы добавили значки, чтобы указать места разрывов длинных строк.

Многие листинги сопровождаются пояснениями, где описываются наиболее важные понятия, поэтому в некоторых случаях в листинге присутствуют цифры в кружочках, которые указывают на дополнительные пояснения, следующие за листингом.

Доступность примеров программного кода

Все примеры программного кода, использованного в книге, можно найти на сайте издательства по адресу: *http://www.manning.com/gsmith/*. Все исходные тексты кода проекта находятся на сайте GitHub (*github.com*) – коммерческой компании, позволяющей размещать репозитории Git. Исходный программный код разбит по главам, так, например, вы можете загрузить */source-code/ch06* и получить полную копию исходных текстов вплоть до этого пункта в книге.

Обновление

Мир Grails изменяется очень быстро. Существенные изменения в платформе Grails, произошедшие за последний год, подтолкнули нас к доработке книги «Grails in Action». В результате перехода от версии Grails 1.0 к версии 1.1 появилось множество новых разделов, а существующие разделы были переписаны – в них были включены описания новых особенностей!

Мы писали эту книгу в прицеле на версию Grails 1.1, однако к тому моменту, когда вы будете читать эти строки, может появиться следующая версия Grails. Новые версии Grails несут в себе новые функциональные возможности, но в них редко появляются серьезные изменения, нарушающие обратную совместимость, поэтому все знания, которые вы приобретете здесь, пригодятся вам при работе со следующими версиями платформы.

Информацию о возможных модификациях примеров программного кода, необходимых для обеспечения их работоспособности в следующих версиях платформы, можно найти на форуме «Author Online» (Диалог с автором) книги «Grails in Action» (*www.manning.com/GrailsinAction*).

Диалог с авторами

Приобретая данную книгу, вы получаете свободный доступ к частному веб-форуму на сайте издательства Manning Publications, где можно оставлять свои отзывы о книге, задавать технические вопросы и обращаться за помощью к авторам и другим пользователям. По адресу *www.manning.com/GrailsinAction* находится информация о том, как пользоваться форумом после регистрации, какого рода помощь там можно получить и правила поведения на нем. Здесь также размещены ссылки на исходный программный код примеров, список ошибок и другие материалы, доступные для загрузки.

Издательство Manning обязуется предоставить своим читателям возможность ведения содержательного диалога с авторами. Это не гарантирует обязательное присутствие авторов на форуме; любое их участие на форуме «Author Online» (Диалог с автором) является исключительно добровольным (и не оплачивается). Мы предлагаем задавать авторам сложные вопросы, чтобы вызвать у них интерес!

Форум «Author Online» (Диалог с автором) и архивы предыдущих дискуссий будут доступны на веб-сайте издательства, пока книга продолжает печататься.

Об авторах

Глен Смит (Glen Smith) начал заниматься программированием в тот день, когда в его школу поступила первая партия компьютеров Hitachi Peach (в начале 80-х годов), и продолжает заниматься им до сих пор. Он программировал на языке C для операционной системы UNIX, был знатоком языка Perl и писал программы даже на Visual Basic (правда, он говорит, что это был лишь временный этап). Познакомившись с языком Java, он потерял интерес ко всему остальному. В настоящее время Глен занимается консультированием по вопросам использования технологий Java EE в правительстве Австралии.

В сообщество Grails Смит был вовлечен во времена версии Grails 0.1 и свое первое общедоступное приложение (SMS-шлюз) написал на платформе Grails версии 0.2. Он регулярно выступает на канале, посвященном Groovy и Grails, является соведущим подкаста Grails (*http://grails-podcast.com*) и администрирует сайт *groovyblogs.org*.

Глен живет в Канберре (Австралия). Его блог: *http://blogs.bytecode.com.au/glen* и микроблог: *http://twitter.com/glen_a_smith*.

Питер Ледбрук (Peter Ledbrook) начинал карьеру программиста, еще будучи подростком, изучая программирование в родительском доме. Пережив тяжелое испытание языками C и C++, он переключился на Java при выполнении своего первого задания и с тех пор не разлучается с ним.

Как истинный поклонник открытого программного обеспечения Питер всегда интересовался новыми и полезными решениями, рождающими-

ся в этом сообществе. Он открыл для себя платформу Grails, когда знакомился с платформой Ruby on Rails, и был удивлен легкостью создания веб-приложений на базе этой платформы. С тех самых пор и начался его бурный роман с Grails, плоды которого может наблюдать каждый.

Он автор нескольких популярных расширений (Remoting, JSecurity и GWT). В конце 2007 года, поступив на работу в компанию G2One в качестве консультанта, Питер стал одним из основных разработчиков Grails. Он обладает богатым опытом разработки приложений, предоставляет консультации и помогает командам, широко использующим платформу Grails.

I

Введение в Grails

В области разработки платформ для создания веб-приложений на языке Java были достигнуты значительные успехи, однако создание новых приложений на их основе все еще является весьма трудоемким делом. Главное преимущество платформы Grails заключается в возможности быстро создавать веб-приложения, поэтому мы сразу перейдем к написанию нашего первого приложения.

В главе 1 мы представим вам основные компоненты платформы Grails, создав с нуля простое приложение «Цитата дня». Вы узнаете, как сохранять и извлекать информацию из базы данных, как разрабатывать бизнес-логику приложения, как писать тесты и даже добавлять некоторую поддержку технологии AJAX. По окончании этой главы вы будете иметь ясное представление обо всех основных компонентах платформы Grails.

Для разработки серьезных приложений на платформе Grails вам необходимо надежно освоить Groovy – динамический язык программирования, на котором реализованы все механизмы Grails. В главе 2 мы в быстром темпе исследуем основные концепции языка Groovy и познакомим вас с основными синтаксическими конструкциями.

К концу первой части вы ощутите всю мощь Groovy и Grails и будете готовы свернуть горы. Не бойтесь пробовать – платформа Grails дружелюбна к экспериментаторам. И возможно, вам захочется перейти ко второй части книги, где мы расскажем об основных компонентах Grails более подробно.

В этой главе:

- Что такое Grails
- Основы философии Grails
- Установка Grails
- Ключевые компоненты приложения на платформе Grails
- Разработка и развертывание первого приложения на платформе Grails

1

Коротко о Grails

«Помогите! Я потерял Mojo!» Вероятно, это самое краткое восклицание, которое можно услышать от программистов, работающих с одной из многочисленных веб-платформ на языке Java. Так много времени приходится тратить на редактирование файлов с настройками и файлов web.xml, на написание встраиваемых определений, на подгонку сценариев сборки, на модификацию структуры страниц, и после каждого изменения необходимо перезапускать приложения, о-о-о-о! «Куда делось веселье? Почему это так утомительно? Мне лишь требовалось на скорую руку набросать небольшое приложение, чтобы отслеживать регистрацию клиентов! Должен же быть более простой путь...» Мы слышим вас.

Grails – платформа «нового поколения» разработки веб-приложений на языке Java, которая позаимствовала оптимальные инструментальные средства веб-разработки, приемы и методики из существующих платформ Java и объединила их с мощным и передовым динамическим языком программирования. В результате получилась платформа, предоставляющая стабильные технологии, знакомые и любимые многими, и защищающая от утомительной настройки, сложностей проектирования и необходимости писать шаблонный программный код, который делает разработку веб-приложений на языке Java такой утомительной. Платформа Grails позволяет сосредоточиться на реализации функциональных возможностей, а не на редактировании файлов XML.

Но Grails далеко не первая платформа, делающая подобные заявления. Вероятно, вы подумали: «Не предлагайте мне еще одну YAJWF!» (Yet Another Java Web Framework – еще одна веб-платформа на языке Java). Как известно, в мире разработки на языке Java существует масса платформ для веб-приложений: Struts, WebWork, JSF, Spring MVC, Seam, Wicket, Tapestry, Stripes, GWT. Этот список можно продолжать и продолжать, причем каждая из них имеет собственные файлы с настройками, идиомы, языки шаблонов и прочие фишки. А теперь нам предлагается еще одна?

Спешим вас обрадовать: это не веб-платформа, построенная на древних принципах; мы переместим вас на совершенно иной уровень, где вы сможете не только делать свою работу, но и делать ее без головной боли. Платформа Grails замечательна тем, что, на наш взгляд, она возвращает нас к временам, когда разработка веб-приложений на языке Java была увлекательнейшим делом! Это было время, когда можно было за полдня набросать что-нибудь, способное сделать безумно счастливым вашего босса, клиента или весь Интернет. Настолько хороша платформа Grails.

В этой главе мы вместе с вами разработаем ваше первое приложение на платформе Grails. Не безделушку, а приложение, которое вы сможете развернуть и продемонстрировать своим друзьям. Приложение, которое управляется данными, поддерживает функциональные возможности Ajax, обладает полной реализацией комплекса операций CRUD (create, read, update, delete – создание, чтение, изменение, удаление), имеет шаблон, управляющий структурой страницы, и даже модульные тесты. Мы напишем за полчаса меньше 100 строк программного кода. Мы говорим это вполне серьезно.

Но прежде чем мы приступим к написанию программного кода, возможно, нам необходимо привести еще несколько убедительных доводов, почему платформа Grails должна заинтересовать вас. До того как запустить среду разработки, давайте ненадолго обратимся к истории и посмотрим, как платформа Grails изменила правила игры.

1.1. Как появилась платформа Grails?

Grails – платформа следующего поколения разработки веб-приложений на языке Java, которая увеличивает производительность разработчика благодаря совместному использованию динамического языка программирования, философии преимущества соглашений перед настройками (Convention over Configuration), мощных практичных инструментов и гибких методик, заимствованных из лучших парадигм веб-разработки.

1.1.1. Сначала была Rails...

Некоторые считают, что платформа Grails является реализацией Ruby on Rails на языке Java, однако это ошибочное мнение легко опровергается следующими сведениями о Grails:

- Платформа Grails привнесла удивительные инновации в сектор разработки корпоративных приложений под своим секретным соусом.
- На Grails оказал влияние широкий диапазон платформ, таких как Ruby, Python, PHP и Java.
- Многие особенности, имеющиеся в арсенале Grails, отсутствуют в Rails – особенности использования корпоративного окружения,

созданные в течение длинной истории развития виртуальных машин Java (JVM).

Тем не менее платформа Grails действительно вобрала в себя многие из нововведений, впервые появившихся в платформе Rails. Ruby on Rails тяготела к корпоративному сектору еще со времени своего первого появления на рынке разработки веб-приложений в 2004 году. Когда же в 2006 году эта платформа получила широкое признание, начало появляться множество новых идей, касающихся разработки веб-приложений.

Ни одна из идей сама по себе не отличалась новизной, но их воплощение было поистине ошеломляющим. Такие вещи, как преимущество соглашений перед настройками, скаффолдинг (scaffolding), шаблоны программного кода и простота интеграции с базой данных, обеспечили возможность молниеносной самонастройки приложений. Казалось невероятным, что Дэвид Хейнмейер Ханссон (David Heinemeier Hansson) (основатель платформы Rails) создал на демонстрации приложение блога, управляемое базой данных, всего за 15 минут. У всех наблюдавших это просто челюсти отвисли.

Истинная мощь этих идей была выведена на передний план в платформе Rails за счет использования динамического языка программирования (Ruby), на котором было реализовано все волшебство работы с метаклассами. Для тех из нас, кто использовал Java в корпоративном секторе, эквивалента не существовало. Все упиралось в язык программирования со статической типизацией, который не обладал гибкостью, необходимой для работы с метаклассами, обеспечивающими элегантность реализации.

1.1.2. Как платформа Grails изменила правила игры

Тогда же, в 2006 году, появилась платформа Grails. Получив все преимущества динамического языка программирования Groovy, платформа Grails сделала возможными создание объекта `Book` и обращение к нему с помощью динамических методов, таких как `Book.findByTitle("Grails in Action")` или `Book.findAllByDatePublishedGreaterThanAndTitleLike(myDate, "Grails")`, несмотря на то, что в действительности объект `Book` не обладал ни одним из этих методов.

Более того, платформа позволила обращаться к любому программному коду на языке Java и к уже существующим библиотекам, а синтаксис языка Groovy был настолько близок к языку Java, что изучение его не составляло труда. Но самое главное, что к концу дня вы получали файл WAR, который был готов к развертыванию на существующем сервере Java-приложений и не требовал ни специальной инфраструктуры, ни специфических знаний.

Еще одна положительная особенность заключалась в том, что платформа Grails была построена на основе платформы Spring, Hibernate и дру-

гих библиотек Java, которые ранее уже использовались разработчиками для создания своих приложений. По сути, имело место что-то вроде максимального улучшения существующих приемов разработки без отказа от надежных и проверенных технологий.

Произошел взрыв популярности Grails. Наконец-то разработчики веб-приложений на Java получили способ взять на вооружение потрясающие идеи, привнесенные платформой Rails, и применять их для разработки надежных и мощных веб-приложений, не отказываясь от приобретенного опыта, библиотек и созданной инфраструктуры.

Вероятно, этого вполне достаточно для объяснения, почему веб-платформа Grails приобрела такую популярность. Но если вам (или вашему руководителю) требуются дополнительные убедительные аргументы в пользу того, что платформа Grails является отличным кандидатом для реализации вашего очередного крупного веб-проекта, прочитайте следующие подразделы, где рассматриваются семь основных доводов (схематически изображенных на рис. 1.1), обеспечивающих платформе Grails доминирующее положение на рынке веб-платформ на языке Java.

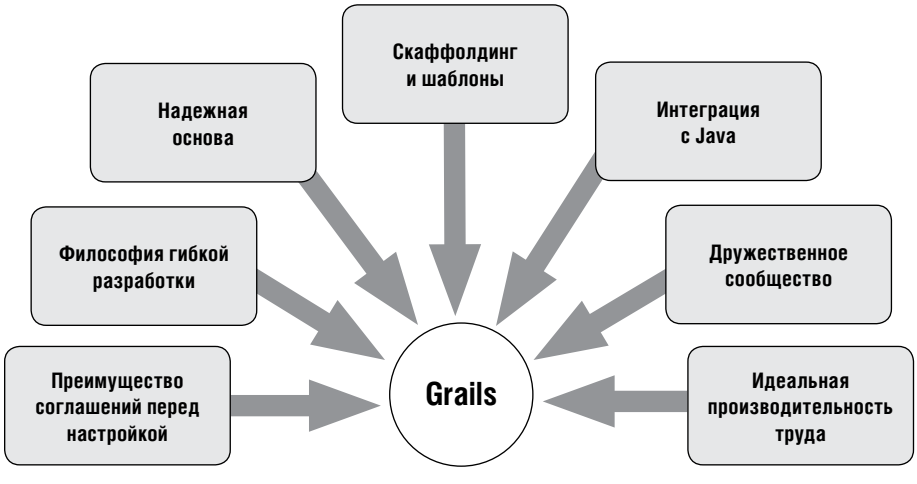


Рис. 1.1. *Экосистема Grails – это мощное слияние людей, идей и технологий*

1.1.3. Основной довод №1: преимущество соглашений перед настройками

При разработке приложений на платформе Grails вы, несомненно, обратите внимание, как мало в ней файлов с настройками. Большинство решений принимается платформой Grails на основе достаточно разумных значений по умолчанию в программном коде:

- Добавьте класс контроллера `Shop` с действием `order`, и платформа Grails обеспечит к нему доступ с адресом URL */yourapp/shop/order*.

- Поместите файлы представления в каталог с именем /views/shop/order, и платформа Grails создаст все необходимые связи без единой строчки в файлах с настройками.
- Создайте новый класс предметной области с именем `Customer`, и платформа Grails автоматически создаст таблицу с именем `customer` в базе данных.
- Добавьте некоторые поля в объект `Customer`, и платформа Grails автоматически создаст необходимые поля в таблице `customer` (используя правильные типы данных, основываясь на ограничениях, заложенных вами). При этом вам не потребуется писать код SQL.

Однако, как справедливо отметил Джейсон Рудольф (Jason Rudolph), в основе Grails лежит *преимущество* соглашений *перед* настройками, а не использование соглашений *вместо настроек*. Если вам потребуется отладить поведение по умолчанию, для этого имеются средства. Платформа Grails обеспечивает простую возможность переопределения значений по умолчанию, не требующую редактирования файлов XML. Если же вы пожелаете использовать существующие файлы XML с настройками из библиотеки Hibernate со всеми их сложностями, Grails не будет стоять у вас на пути.

1.1.4. Основной довод №2: философия гибкой разработки

Платформа Grails обеспечивает серьезные возможности, дающие ей право называться гибкой веб-платформой, и к концу этой главы вы поймете почему. Благодаря использованию динамического языка программирования (Groovy) платформа Grails позволяет легко и просто создавать вещи, которые при работе на языке Java вызывали настоящую головную боль. Будь то обработка форм, реализация библиотек тегов или написание тестов, в вашем распоряжении имеются лаконичность и выразительность платформы, которые сделают эти операции более простыми в осуществлении и в сопровождении одновременно.

Инфраструктура Grails предохраняет вас от выполнения повторяющихся операций, устраняя помехи, стоящие у вас на пути. Представьте себе, что вы можете запускать локальную копию приложения и в процессе его работы добавлять новые контроллеры, представления и библиотеку тегов, и при этом вам не требуется перезапускать его! Более того, вы можете тестировать эти новые особенности, выполнять настройки и щелкать на кнопке обновления в броузере для просмотра обновленного представления. Разве это не счастье!

Платформа Grails выводит гибкость разработки веб-приложений на языке Java на совершенно новый уровень, и как только вы создадите свое первое приложение, чем мы будем заниматься в течение следующих 30 минут, вы по достоинству оцените уникальные особенности Grails.

1.1.5. Основной довод №3: надежная основа

Хотя сама платформа Grails наполнена новыми, ультрасовременными идеями, основу ее составляют испытанные технологии: Spring и Hibernate. Данные технологии по сей день используются во многих проектах на Java, и на то есть веские причины: они надежны и проверены временем.

Базирование на Spring и Hibernate означает также, что за кулисами скрыто не так много необычного. Если вам потребуется что-либо подкорректировать в настройках (с помощью конфигурационного класса библиотеки Hibernate) или во время выполнения (получая ссылку на `ApplicationContext` в платформе Spring), то ничего нового изучать не придется. Время, потраченное на освоение Spring и Hibernate, не пропадет даром.

Если вы плохо знакомы с платформой Grails и не имеете опыта работы с платформой Spring и библиотекой Hibernate, это не имеет значения. В действительности существует совсем мало ситуаций, когда необходимо опуститься до их уровня, но вы узнаете о таких случаях, когда это потребуется.

Та же философия опоры на лучшее была перенесена и в другие области экосистемы Grails – в частности, на сторонние расширения. Так, модуль планировщика выполнения заданий построен на базе механизма Quartz, модуль поиска – на базе библиотек Lucene и Compass, а механизм отображения – на базе SiteMesh. Повсюду в экосистеме вам будут встречаться популярные библиотеки языка Java, обернутые в удобные для использования модули расширения. Это обеспечит вам душевное спокойствие и удивительную производительность труда!

Еще одно немаловажное для разработчиков обстоятельство – гарантированное предоставление квалифицированных услуг, обучения и организационной поддержки. Когда в ноябре 2008 года компания SpringSource приобрела фирму G2One, Groovy и Grails получили поддержку крупной компании, имеющей большой опыт работы с полным комплексом программного обеспечения Groovy и Grails. Кроме того, было организовано круглосуточное предоставление широкого диапазона услуг для организаций, нуждающихся в поддержке по вопросам, касающимся языка Groovy и платформы Grails.

1.1.6. Основной довод №4: скаффолдинг и шаблоны

Если вы попробуете вручную развернуть приложение на платформе Spring MVC, вы узнаете, насколько это неудобно. Вам потребуется создать каталог для файлов JAR, множество файлов с определениями специальных классов бинов (bean), набор файлов web.xml с настройками, связку аннотированных объектов POJO, несколько файлов с настройками для библиотеки Hibernate, сценарий создания базы данных, по-

сле чего система сборки сможет превратить все это в работающее приложение. Работа выйдет сложная, и вы, вероятно, потратите целый день на ее выполнение.

Напротив, достаточно всего одной строки: `grails createapp myapp`, чтобы собрать работающее приложение на платформе Grails; затем вы можете запустить его командой `grails run-app` и наблюдать его работу в броузере. За кулисами выполняются те же операции, но они опираются на соглашения и разумные значения по умолчанию, а не на настройки, определяемые вручную.

Если вам потребуется добавить новый класс контроллера, команда `grails create-controller` сгенерирует его заготовку (вместе с тестом). То же относится к представлениям, службам, классам доменов и ко всем остальным компонентам приложения. Такой шаблонный подход обеспечивает фантастическую производительность труда, благодаря которой вы сможете сконцентрироваться на решении насущных проблем и не отвлекаться на написание шаблонного программного кода.

Кроме того, платформа Grails обладает удивительной особенностью, получившей название «скаффолдинг» (scaffolding). На основе классов, описывающих модель (структуру) базы данных, платформа Grails может динамически генерировать набор представлений и контроллеров, позволяющих выполнять простейшие CRUD-операции – creating (создание), reading (чтение), updating (изменение) и deleting (удаление) – без единой строчки программного кода.

1.1.7. Основной довод №5: интеграция с Java

Одна из уникальных особенностей сообщества Groovy и Grails состоит в том, что в отличие от некоторых других языков, поддерживаемых JVM, мы любим язык Java! Мы признаем, что есть задачи и решения, которые лучше реализуются на языке со статической типизацией, поэтому мы не испытываем проблем при создании классов обработки вебформ на языке Groovy и реализации высокопроизводительных вычислительных алгоритмов на языке Java. Для нас это лишь вопрос выбора правильного инструмента.

Мы с любовью относимся и к экосистеме Java. Другими словами, мы не выбрасываем потрясающий набор библиотек на языке Java, которые мы знаем и любим. Будь то наши собственные файлы JAR с классами DTO для выполнения расчетов или великолепная новая библиотека Java для взаимодействия с Facebook, при переходе на использование платформы Grails вам ничего не придется выбрасывать, за исключением длинных файлов XML с настройками. Однако, как мы уже говорили, у вас сохраняется возможность при желании повторно использовать имеющиеся файлы отображений Hibernate и ресурсов Spring!

1.1.8. Основной довод №6: дружественное сообщество

Одна из самых привлекательных особенностей экосистемы Grails заключается в наличии фантастически доброжелательного сообщества пользователей. Рассылка Groovy и Grails представляет собой поле бурной деятельности, где в равной степени приветствуются и опытные ветераны, и начинающие пользователи. На сайте *Grails.org* имеется своя wiki, содержащая информацию и документацию о платформе Grails.

Вокруг Grails возникло также множество сторонних веб-сайтов сообществ. Например, на сайте *groovyblogs.org* собирается информация о событиях, происходящих в сфере Groovy и Grails, и имеется масса интересных статей. А на таких сайтах, как *grailscrowd.com*, Facebook и LinkedIn, присутствуют Grails-элементы социальных сетей. Существует даже свой подкаст Grails (*grailspodcast.com*), который обновляется каждые две недели, чтобы держать вас в курсе новых событий, а также знакомить с интервью и дискуссиями, ведущимися вокруг Grails.

Но одна из самых замечательных черт сообщества – это постоянно растущий список сторонних расширений для Grails. Требуется ли вам расширение для реализации полнотекстового поиска, виджеты Ajax, инструменты составления отчетов, средства обмена мгновенными сообщениями, компоненты для работы с лентами RSS или управления файлами журналов, инструменты профилирования производительности или средства интеграции со службами Twitter, здесь есть все и на все случаи жизни. Существуют буквально сотни расширений, доступных для использования (в главе 8 мы познакомим вас с наиболее популярными из них).

1.1.9. Основной довод №7: идеальная производительность труда

Платформа Grails не просто позволяет создавать веб-приложения. Она быстро исполняет задуманное вами, благодаря чему вы можете заняться другими более важными «житейскими делами». Для нас производительность – это настоящая находка, и использование платформы Grails, каждой ее особенности вернет вам вашу жизнь. Когда вы заметите, что можете за один день выполнить работу, которую раньше делали за две недели, вы поймете, как это хорошо, когда есть возможность уходить домой пораньше. Работа с такой продуктивной платформой поможет даже свой досуг проводить более увлекательно. И вы сможете воплотить в жизнь все свои идеи по созданию веб-сайта Web 2.0, которые вы видели во сне, но реализация которых завершилась написанием наполовину законченных приложений на платформе Struts или Spring MVC.

Возможность быстро разрабатывать надежные приложения позволит вам освободить время для других, более важных занятий: общения

со своей семьей, прогулок с собакой, обучения игре на гитаре или выращивания гигантских кабачков на своем участке. Веб-приложения приходят и уходят, а кабачки остаются. Использование продуктивной платформы Grails позволит понять всю мудрость такой точки зрения. В этой главе мы дадим вам представление о том, какой производительности можно добиться при переходе на платформу Grails.

Большинство наших знакомых программистов по характеру являются людьми нетерпеливыми, поэтому в этой главе мы потратим всего 30 минут на разработку легко разворачиваемого веб-сайта Web 2.0, управляемого данными, использующего механизмы Ajax и снабженного модульными тестами. Попутно мы познакомимся с основными составляющими приложений на платформе Grails: моделями, представлениями, контроллерами, библиотеками тегов и службами. Итак, мы начинаем.

1.2. Установка

Чтобы получить работающую платформу Grails, необходимо выполнить процедуру установки, представленную на рис. 1.2.

Рис. 1.2. *Процедура установки платформы Grails*

Прежде всего необходимо установить JDK (версии 1.5 или выше; чтобы узнать, какая версия установлена у вас в системе, запустите команду `javac-version`). В наше время большинство компьютеров поставляет-

ся с предустановленной системой Java, поэтому вы, вероятно, можете пропустить этот шаг.

Если вам посчастливилось и JDK уже установлен у вас, загрузите дистрибутив с последней версией Grails с сайта *www.grails.org* и распакуйте его в каталог установки.

После этого вам необходимо определить переменную окружения `GRAILS_HOME`, записать в нее путь к каталогу установки Grails и добавить путь `GRAILS_HOME/bin` в путь поиска файлов. В Mac OS X и Linux это можно сделать, отредактировав сценарий ~/.profile и добавив в него следующие строки:

```
export GRAILS_HOME=/opt/grails
export PATH=$PATH:$GRAILS_HOME/bin
```

В Windows вам надо открыть диалог Свойства системы (System Properties), определить переменную `GRAILS_HOME` и изменить переменную `PATH`.

Корректность установки платформы Grails можно проверить, выполнив команду `grails help` в командной строке. Эта команда выведет список всех команд Grails, а также подтвердит, что все работает, как ожидается, и что переменная окружения `GRAILS_HOME` содержит правильный путь к каталогу установки:

```
grails help

Welcome to Grails 1.1 - http://grails.org/
Licensed under Apache Standard License 2.0
Grails home is set to: /opt/grails
```

Похоже, что все в порядке.

При разработке более сложных приложений на платформе Grails вам, вероятно, потребуется поддержка некоторых изумительных сред разработки для платформы Grails IDE. В настоящее время существуют расширения, обеспечивающие поддержку платформы Grails в таких интегрированных средах разработки, как IntelliJ, NetBeans и Eclipse. Какую бы среду разработки вы ни выбрали, для нее найдется расширение, которое позволит вам приступить к работе. В этой главе мы не будем писать большие объемы программного кода, поэтому достаточно будет воспользоваться простым текстовым редактором. Запустите ваш любимый редактор, и приступим к разговору о нашем примере приложения.

1.3. Пример программы Web 2.0 QOTD

Раз уж мы собрались заняться написанием небольшого приложения, добавим немного веселья. Пусть нашим примером будет веб-приложение «Цитата дня» (Quote Of The Day, QOTD), которое будет получать и отображать известные выражения, произнесенные звездами программирования в разное время. Мы дадим пользователю возможность добавлять, изменять и просматривать цитаты и даже добавим в приложе-

ние некоторую поддержку Ajax, чтобы придать ему ощущение соответствия идеям Web 2.0. Для нашего приложения желательно будет определить короткий адрес URL, поэтому назовем его «qotd».

Примечание

Со страницы книги на сайте *manning.com* вы можете загрузить все примеры приложений, включая таблицы стилей CSS и файлы с изображениями.

Приступим к созданию приложения «Цитата дня»; создание всех проектов на платформе Grails будет начинаться аналогичным образом. Прежде всего выберите каталог, в котором вы будете работать. Затем создайте приложение:

```
grails create-app qotd
cd qotd
```

Вот и все. Вы создали свое первое приложение на платформе Grails. Вы можете заметить, что платформа Grails создала подкаталог qotd для хранения файлов приложения. Теперь перейдите в этот подкаталог – в нем мы останемся до конца главы.

После выполнения всей тяжелой работы по созданию приложения было бы глупо не насладиться плодами своего труда. Давайте запустим его:

```
grails run-app
```

В состав платформы Grails входит копия Jetty (встраиваемого вебсервера Java – ходят слухи, что в будущие версии будет включен вебсервер Tomcat), который используется платформой Grails для развертывания приложений на этапе разработки и тестирования. Когда вы запустите команду `grails run-app`, платформа Grails скомпилирует и запустит веб-приложение. Когда все будет готово, вы увидите в консоли примерно такое сообщение:

```
Server running. Browse to http://localhost:8080/qotd
```

Это означает, что можно запустить свой любимый броузер и вызвать приложение: *http://localhost:8080/qotd/*. На рис. 1.3 показано, как выглядит наше приложение QOTD в окне броузера.

Проверив работу приложения, его можно остановить, нажав комбинацию клавиш Ctrl-C, или оставить его работающим и запускать команды Grails в другом окне терминала.

1.3.1. Наш первый контроллер

Мы создали и развернули наше первое приложение, но пока мы никаким образом не обеспечили взаимодействие с пользователем. Прежде чем двинуться дальше, давайте познакомимся с тем, как с помощью *контроллеров* платформа Grails организует взаимодействие с пользователем.

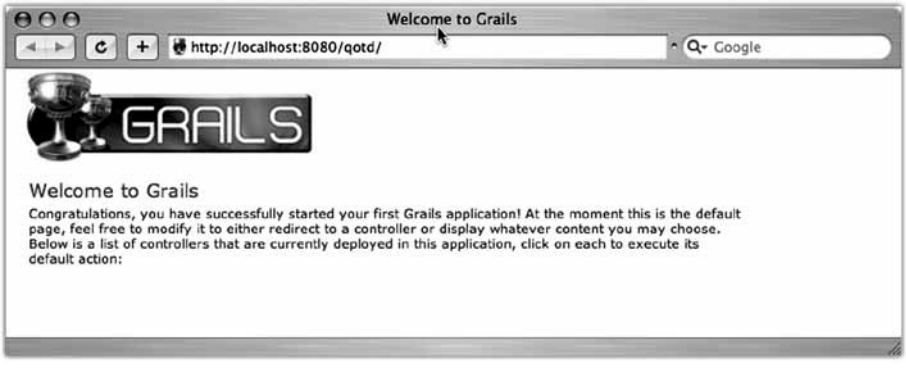


Рис. 1.3. *Наше первое приложение*

Использование нестандартного порта (отличного от 8080)

Если порт 8080 вам недоступен (например, потому что он занят другим выполняющимся процессом, таким как Tomcat), вы можете изменить порт, на котором встроенный сервер приложений Grails будет принимать соединения, использовав аргумент командной строки `-Dserver.port`. Допустим, если вы хотите, чтобы платформа Grails использовала порт 9090, вы можете запустить приложение следующей командой:

```
grails -Dserver.port=9090 run-app
```

Если вы решили постоянно запускать некоторое приложение на определенном порту, то вы можете создать файл /grails-app/conf/BuildConfig.groovy и задать в нем параметр настройки `grails.server.port.http=9090`, указывая тем самым, что выбранный вами номер порта является портом по умолчанию. Или вы можете определить значение порта для всей системы, отредактировав файл с глобальными настройками $HOME/.grails/settings.groovy. Подробнее об этих файлах рассказывается в главе 15.

Контроллеры – основа любого приложения на платформе Grails. Они принимают ввод от веб-броузера пользователя, взаимодействуют с бизнес-логикой и моделью данных и обеспечивают отправку пользователю корректных страниц. Без контроллеров веб-приложение было бы лишь связкой статических страниц.

Как и в случае многих других компонентов приложения, вы можете позволить платформе Grails сгенерировать скелет контроллера с помощью команд Grails. Создадим простой контроллер для обработки цитат:

```
grails create-controller quote
```

Платформа Grails создаст скелет контроллера в файле /grails-app/controllers/QuoteController.groovy. Обратите внимание, что Grails автоматически выбирает регистр символов в именах. В листинге 1.1 показано, как выглядит простой скелет контроллера.

Листинг 1.1. Наш первый контроллер

```
class QuoteController {
    def index = { }
}
```

Не впечатляет, правда? Определение `index` в листинге 1.1 – это *действие* платформы Grails, к которому мы вернемся через минуту. А пока добавим действие `home`, которое будет возвращать броузеру некоторый текст, как показано в листинге 1.2.

Листинг 1.2. Добавление операции вывода

```
class QuoteController {
    def index = { }

    def home = {
        render "<h1>Real Programmers do not eat Quiche</h1>"
    }
}
```

Для непосредственной отправки некоторого содержимого обратно броузеру платформа Grails предоставляет метод `render()`. Он приобретет для нас особое значение, когда мы погрузимся в воды Ajax, а пока будем использовать его для отправки заголовка «Real Programmers» (Настоящие программисты).

Как теперь вызвать наше действие в броузере? Если бы мы имели дело с веб-приложением на Java, мы объявили бы адрес URL для этого действия в файле с настройками, но в случае платформы Grails мы поступим иначе. В этом месте мы должны представить вам парадигму преимущества соглашений перед настройками.

В платформе Ruby on Rails была использована идея, согласно которой можно избежать необходимости создавать тонны настроек в формате XML (а также настроек в любом другом формате), если позволить платформе самой определять необходимые соответствия. Платформа Grails заимствовала эту философию. Так как наш контроллер называется `QuoteController`, платформа Grails обеспечит доступ к его действиям посредством адреса URL /qotd/quote/youraction. На рис. 1.4 наглядно показано, как платформа Grails преобразует адреса URL в объекты.

В случае нашего действия `hello` нам необходимо перейти по адресу URL:

```
http://localhost:8080/qotd/quote/home
```

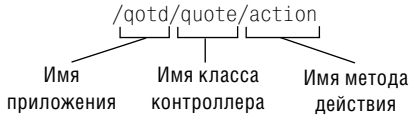


Рис. 1.4. *Порядок преобразования адреса URL в объект Grails*

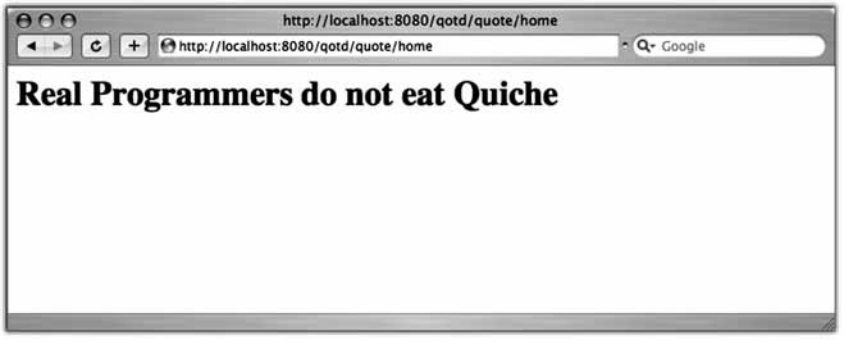


Рис. 1.5. *Вид страницы после добавления первого функционального элемента*

На рис. 1.5 показан внешний вид нашего нового приложения, созданного и запущенного без единой строчки XML.

Если вас интересует, что делает метод `index()` в листинге 1.1, заметим, что этот метод вызывается, когда пользователь опускает имя действия. Если мы решаем, что все обращения по адресу /qotd/quote/ должны приводить к адресу /qotd/quote/home, мы должны сообщить об этом платформе Grails с помощью действия `index`, как показано в листинге 1.3.

Листинг 1.3. Перенаправление

```
class QuoteController {
    def index = {
        redirect(action: home)
    }

    def home = {
        render "<h1>Real Programmers do not each quiche!</h1>"
    }
}
```

Пока все выглядит неплохо, но необходимость внедрения тегов HTML в исходный программный код кажется малопривлекательной. Теперь, когда мы немного узнали о контроллерах, самое время познакомиться с представлениями.

1.3.2. Вывод содержимого: представление

Внедрение разметки HTML в программный код – всегда плохая идея. Не только потому, что такой код сложно читать и сопровождать, но и поскольку в этом случае вашему дизайнеру потребуется иметь доступ к исходным текстам, чтобы реализовать оформление страниц. Обойти данную проблему можно, переместив логику управления отображением в отдельный файл, называемый *представлением*, причем на платформе Grails сделать это достаточно просто.

Если вы уже работали с веб-приложениями на языке Java, вы должны быть знакомы с JavaServer Pages (JSP). Страницы JSP отображают разметку HTML вашего веб-приложения перед пользователем. Приложения на платформе Grails, в свою очередь, используют Groovy Server Pages (GSP). Эти две концепции очень близки.

Мы уже обсуждали парадигму преимущества соглашений перед настройками; представления используют ее применительно к внешнему оформлению. Если поместить файлы представлений в нужном каталоге, они будут задействованы без дополнительной настройки.

Сначала мы реализуем действие `random`, как показано в листинге 1.4. А затем побеспокоимся о представлении.

Листинг 1.4. Действие выбора случайной цитаты

```
def random = {
    def staticAuthor = "Anonymous"
    def staticContent = "Real Programmers don't eat much quiche"
    [ author: staticAuthor, content: staticContent]
}
```

Что это за фрагмент в квадратных скобках? Он описывает, какую информацию должен передавать контроллер в представление. Если у вас имеется опыт разработки сервлетов, можете представить себе этот фрагмент как данные, возвращаемые по запросу. Оператор `[:]` в языке Groovy создает отображение (тип данных Map); таким образом, мы реализовали передачу нашему представлению последовательности пар ключ/значение.

Как представление вписывается во все это и где должен находиться файл GSP, чтобы платформа Grails смогла обнаружить его? Мы будем использовать то же соглашение об именовании, которое использовали при создании контроллера и действия, и поместим файл GSP сюда: /grails-app/views/quote/random.gsp. Если следовать этому соглашению, никакой дополнительной настройки не потребуется.

Создадим файл GSP, как показано в листинге 1.5, и посмотрим, как можно сослаться на данные в отображении.

Листинг 1.5. Реализация нашего первого представления

```
<html>
<head>
    <title>Random Quote</title>
</head>

<body>

    <q>${content}</q>
    <p>${author}</p>

</body>

</html>
```

Формат `${content}` и `${author}` известен как GSP Expression Language (язык выражений GSP); если вам когда-либо приходилось иметь дело со страницами JSP, данная информация, вероятно, не будет для вас новой. Если же ранее вы не работали со страницами JSP, можете представить себе эти теги `${}` как способ отображения содержимого переменных. Запустим броузер и сделаем попытку обратиться к приложению. На рис. 1.6 показано наше первое представление в действии.

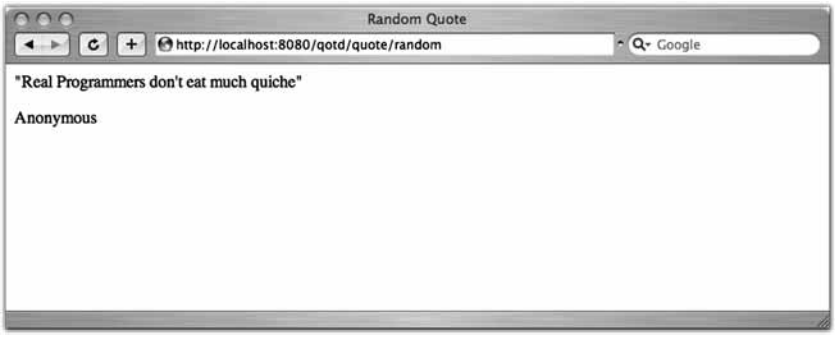


***Рис. 1.6.** Наше первое представление в действии*

1.3.3. Добавление стилей оформления с помощью макетов Grails

Теперь у нас имеется реализация первой части логики приложения. Но результат смотрится не слишком привлекательно – на странице нет плавных переходов цветов, нет крупных заголовков, нет закругленных углов. Страница словно из середины 90-х.

Вы, вероятно, подумали, что настало время ввести в действие таблицы стилей CSS, но давайте сначала немного порассуждаем. Если мы реа-

лизуем оформление страницы random.gsp с помощью CSS, то окажемся перед необходимостью добавлять ссылки на таблицы стилей в заголовок каждой страницы в приложении. Существует более простой способ – макеты Grails.

Макеты обеспечивают возможность определить шаблоны размещения определенных частей приложения. Например, мы можем все страницы с цитатами (выбираемыми случайно, по имени автора или по дате) снабдить общим заголовком и ссылками для навигации и изменять только текст цитат. Для этого добавим в нашу страницу некоторый идентификатор, который затем можно будет использовать в таблицах стилей CSS. Исправленное представление показано в листинге 1.6.

Листинг 1.6. Исправленное представление

```
<html>
<head>
    <title>Random Quote</title>
</head>

<body>

    <div id="quote">
        <q>${content}</q>
        <p>${author}</p>
    </div>

</body>

</html>
```

Как теперь задействовать макет (заголовок и элементы навигации), обсуждавшийся выше? Как и все остальное в платформе Grails, макеты следуют принципу преимущества соглашений перед настройками. Чтобы все действия контроллера `QuoteController` использовали один и тот же макет, мы создадим файл с именем /grails-app/views/layouts/quote.gsp. Платформа Grails не предусматривает команд создания макетов, поэтому нам придется сделать это вручную. В листинге 1.7 представлена наша первая попытка написать макет.

Листинг 1.7. Добавление макета

```
<html>
  <head>
    <title>QOTD &raquo; <g:layoutTitle/></title> ← ❶ Добавляет текст заголовка из целевой страницы
    <link rel="stylesheet" href="
          <g:createLinkTo dir='css' file='snazzy.css' /> ← ❷ Относительная ссылка на файл CSS
    " />
    <g:layoutHead /> ← ❸ Добавляет элементы заголовка из целевой страницы
  </head>
  <body>
```

```
    <div id="header">
      <img src="
        <g:createLinkTo dir='images' file='logo.png'/>
      " alt="logo"/>
    </div>
    <g:layoutBody /> ← ❹ Добавляет элементы тела из целевой страницы
  </body>
</html>
```

Образовалась масса угловых скобок, будем разбираться по порядку. Важно не забывать, что это шаблон страницы, значит, перед тем, как страница будет отправлена броузеру, содержимое целевой страницы (random.gsp) будет включено в данный шаблон. Внутри платформа Grails использует SiteMesh, популярный механизм шаблонов на языке Java, который автоматически выполняет объединение страницы и шаблона. На рис. 1.7 схематически показано, как механизм SiteMesh выполняет объединение.

Рис. 1.7. *Механизм SiteMesh декорирует простой файл GSP стандартным набором заголовков и колонок*

Для работы нашего макета, приведенного в листинге 1.7, необходимо предоставить ему возможность доступа к элементам целевой страницы (это требуется, например, для объединения заголовка макета и целевой страницы). Такой доступ обеспечивается с помощью библиотек тегов шаблонов платформы Grails, поэтому сейчас мы дадим вам общее понятие о библиотеках тегов.

Если раньше вам никогда не приходилось сталкиваться с библиотеками тегов, вы можете представить их как группы нестандартных тегов HTML, которые могут вызывать выполнение программного кода. В листинге 1.7 мы задействовали теги `g:createLinkTo`, `g:layoutHead` и `g:layoutBody`. Когда броузер запросит страницу, платформа Grails заместит все эти теги разметкой HTML, и содержимое разметки HTML будет зависеть от того, что генерируют отдельные теги. Например, первый тег `createLinkTo` ❷ сгенерирует фрагмент ссылки, такой как /qotd/css/snazzy.css.

В блоке заголовка страницы мы включили название приложения QOTD с двумя последующими угловыми скобками (>>), которые представляет метасимвол HTML `»`, и затем добавили заголовок самой целевой страницы ❶.

В конце блока заголовка мы использовали тег `layoutHead`, чтобы добавить содержимое раздела `HEAD` целевой страницы ❸. Это может быть важно для методики оптимизации поисковых систем (Search Engine Optimization, SEO), согласно которой отдельные целевые страницы могут содержать свои собственные теги `META` для повышения значимости в поисковой системе Google.

Наконец мы добрались до тела страницы. Здесь мы выводим тег `div` с заголовком, чтобы реализовать плавный переход цвета в стиле Web 2.0 и вывести привлекательные ярлыки, и затем используем тег `<g:layoutBody>`, чтобы отобразить раздел `BODY` целевой страницы ❹.

Обновите страницу в броузере, чтобы увидеть, что получилось в результате. На рис. 1.8 изображена наша страница с новым оформлением.

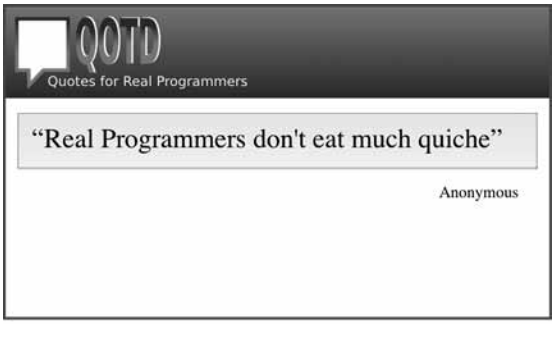


Рис. 1.8. *Внешний вид страницы приложения QOTD с привлекательным оформлением стилями CSS*

Наше приложение выглядит совсем неплохо. Что самое интересное, для этого нам не пришлось вносить изменения в файл random.gsp. Отсутствие косметической разметки в страницах представлений существенно снижает сложность их сопровождения. И если вам потребуется изменить заголовок, добавить сценарии JavaScript или внедрить дополнительные файлы CSS, все это вы будете делать в одном месте – в макете.

Фантастика. Мы создали и запустили контроллер, представление и макет. Но сами данные, представленные на странице, все еще остаются статическими. Вероятно, мы немного запоздали с изучением механизмов, используемых платформой Grails для работы с информацией в базе данных. Однако как только мы познакомимся с ними, мы сможем вернуться назад и реализовать действие случайного выбора цитат.

1.4. Создание модели предметной области

Мы получили работающее приложение и можем развертывать его для тестирования в веб-контейнере. Но не будем преувеличивать наши достижения – пока что компания Google не собирается нас покупать. Наше приложение испытывает недостаток определенных возможностей. Нам необходимо включить в него немного интерактивности, чтобы позволить пользователям добавлять новые цитаты в базу данных. Чтобы получить возможность сохранять эти цитаты, нам необходимо узнать, как платформа Grails работает с моделями данных.

Для описания объектов данных, которые могут сохраняться в базе данных, в платформе Grails используется термин «класс предметной области». Для нужд приложения QOTD нам потребуется несколько классов предметной области, но начнем мы с самого необходимого: с класса предметной области, обеспечивающего возможность сохранения цитат.

Создадим класс предметной области `Quote`:

```
grails create-domain-class quote
```

В приложениях на платформе Grails классы доменов всегда сохраняются в каталоге /grails-app/domain. Взгляните на скелет класса, созданный платформой Grails в файле /grails-app/domain/Quote.groovy:

```
class Quote {

  static constraints = {
  }

}
```

В нем нет ничего необычного. Нам нужно добавить в модель данных несколько полей для хранения различных элементов цитат. Дополним наш класс так, чтобы он позволял сохранять содержимое цитаты, имя автора и дату добавления записи, как показано в листинге 1.8.

Листинг 1.8. Наш первый класс предметной области с дополнениями

```
class Quote {

  String content
  String author
  Date created = new Date()

  static constraints = {
  }

}
```

Теперь, когда у нас появилась модель данных, нам нужно создать свою схему базы данных, правильно? Нет, неправильно. Платформа Grails

автоматически выполнит всю эту трудную работу за нас. Опираясь на определения типов в листинге 1.8 и применяя некоторые простые соглашения, платформа Grails создаст таблицу `quote` с полями типа `varchar` для хранения строк и с полем типа `Date` для хранения даты. Когда в следующий раз мы запустим команду `grails run-app`, платформа автоматически создаст модель данных.

Но как она узнает, в какой базе данных нужно создавать таблицы? Пришло время настроить источник данных.

1.4.1. Настройка источника данных

Платформа Grails «из коробки» поддерживает работу с базой данных, находящейся в памяти, поэтому, если не предпринять дополнительных действий, данные будут сохраняться в оперативной памяти компьютера. Эта идея немного нервирует большинство программистов, так что давайте посмотрим, как можно создать более постоянную базу данных.

В каталоге /grails-app/conf/ вы найдете файл DataSource.groovy. В этом файле можно определить источник данных (базу данных), который будет использоваться приложением – вы можете определить несколько баз данных: для нужд разработки, для тестирования и для работы в нормальном режиме. Когда вы выполняете команду `grails run-app` для запуска локального веб-сервера, задействуется источник данных для разработки. В листинге 1.9 приводится фрагмент стандартного файла DataSource, где находится определение источника данных по умолчанию.

Листинг 1.9. Определение источника данных, расположенного в памяти

```
development {
  dataSource {
    dbCreate = "create-drop" ← Создавать базу данных при каждом запуске
    url = "jdbc:hsqldb:mem:devDB" ← База данных располагается в памяти
  }
}
```

Здесь имеются две проблемы. Во-первых, стратегия `dbCreate` предписывает платформе Grails уничтожать и пересоздавать базу данных при каждом запуске. Это не совсем то, что нам требуется, поэтому изменим ее так, чтобы платформа Grails оставляла базу данных в неприкосновенности между запусками приложения (но дадим ей право добавлять столбцы, если в этом возникнет необходимость).

Вторая проблема имеет отношение к адресу URL – здесь используется база данных HSQLDB, хранящаяся в памяти. Это прекрасно подходит для тестирования, но малопригодно для работы в нормальном режиме. Заменим ее файловой версией HSQLDB, чтобы обеспечить постоянство хранения данных.

Измененный файл приводится в листинге 1.10.

Листинг 1.10. Определение постоянного источника данных

```
development {
  dataSource {
    dbCreate = "update" ← Сохранять базу данных между запусками
    url = "jdbc:hsqldb:file:devDB;shutdown=true" ← База данных располагается
  }                                                в файле
}
```

Теперь у нас имеется база данных, обеспечивающая долговременное хранение данных, поэтому далее посмотрим, как можно заполнить ее примерами данных.

1.4.2. Исследование операций с базой данных

Мы еще ничего не сделали с пользовательским интерфейсом, но было бы неплохо сначала сохранить некоторые данные в таблице `quote` и попробовать получить их. Для этого мы воспользуемся консолью платформы Grails – небольшим приложением с графическим интерфейсом, которое запускается независимо от веб-сервера и предоставляет консоль, в которой можно выполнять команды Groovy.

Чтобы опробовать модель данных до того, как приложение будет готово к запуску, можно выполнить команду `grails console`. По этой команде будет произведена настройка приложения QOTD, и на экране появится графический интерфейс консоли, ожидающей ввода программного кода. На рис. 1.9 демонстрируется операция сохранения новой цитаты с помощью консоли.

Рис. 1.9. *Консоль Grails предоставляет графический интерфейс для запуска команд*

В нашем первом исследовании модели данных было бы хорошо создать и сохранить несколько объектов Quote. Введите в окне консоли следующий код и затем щелкните на кнопке Run (Выполнить) (крайняя правая на панели инструментов):

```
new Quote(author: 'Larry Wall',
    content: 'There is more than one method to our madness.').save()
```

В нижней части окна консоли появятся результаты выполнения операции:

```
Result: Quote : 1
```

Интересно, откуда взялась процедура save()? Платформа Grails автоматически снабжает классы доменов определенными методами. Создадим еще несколько записей, чтобы получить более полное представление о запросах:

```
new Quote(author: 'Chuck Norris Facts', content: 'Chuck Norris always uses his
     own design patterns, and his favorite is the Roundhouse Kick').save()
new Quote(author: 'Eric Raymond', content: 'Being a social outcast helps you
     stay concentrated on the really important things, like thinking and
     hacking.').save()
```

Теперь задействуем другой динамический метод (count()), чтобы убедиться, что данные правильно сохраняются в базе данных:

```
println Quote.count()
3
```

Пока все идет неплохо. Пришло время закатать рукава и выполнить несколько запросов к базе данных цитат. Чтобы упростить поиск в базе данных, платформа Grails добавляет к классу предметной области специальные методы запросов, которые называются *динамическими методами поиска*. Имена этих специальных методов включают в себя имена полей в модели предметной области, что обеспечивает максимальную простоту выполнения запросов:

```
def quote = Quote.findByAuthor("Larry Wall")
println quote.content
There is more than one method to our madness.
```

Сейчас, когда мы знаем, как сохранять и извлекать данные, можно вернуться и доработать наше веб-приложение. Закройте консоль Grails, и мы посмотрим, как можно вывести цитаты в веб-странице.

1.5. Добавление действий в пользовательский интерфейс

Попробуем вывести цитату в веб-странице. Сначала нам необходимо добавить в контроллер QuoteController действие, возвращающее случай-

ную цитату из базы данных. Случайным выбором данных мы займемся чуть ниже, а пока немного схитрим и «состряпаем» пример данных, как показано в листинге 1.11.

Листинг 1.11. Выборка не совсем случайных данных

```
def random = {
  def staticQuote = new Quote(author: "Anonymous",
        content: "Real Programmers Don't eat quiche")
  [ quote : staticQuote]
}
```

Нам также необходимо изменить файл /grails-app/views/quote/random.gsp и задействовать в нем наш новый объект `Quote`:

```
<q>${quote.content}</q>
<p>${quote.author}</p>
```

Здесь нет ничего нового, просто улучшена модель данных. Теперь можно обновить страницу в броузере и убедиться, что статическая цитата передается представлению. Позвольте этой попытке уверить вас, что все работает правильно.

Теперь, когда вы узнали, как объекты модели передаются в представление и какую опасность могут представлять собой запросы, переделаем метод действия, как показано в листинге 1.12, реализовав выборку действительно случайной цитаты из базы данных.

Листинг 1.12. Выборка случайных данных из базы данных

```
def random = {

  def allQuotes = Quote.list()  <-- ❶ Получить список цитат
  def randomQuote
  if (allQuotes.size() > 0) {
    def randomIdx = new Random().nextInt(allQuotes.size())  <-- ❷ Выбрать случайную цитату
    randomQuote = allQuotes[randomIdx]
  } else {
    randomQuote = new Quote(author: "Anonymous",  <-- ❸ Сгенерировать цитату по умолчанию
        content: "Real Programmers Don't eat Quiche")
  }
  [ quote : randomQuote]  <-- ❹ Передать цитату в представление

}
```

Благодаря переработанному действию `random` мы начинаем использовать настоящие данные из базы данных. С помощью метода `list()` ❶ мы получаем полное множество объектов `Quote` из таблицы `quote` в базе данных и заполняем ими коллекцию `allQuotes`. Если в коллекции присутствует хотя бы одна запись, мы выбираем случайную ❷, опираясь на

индекс в коллекции. В противном случае мы используем статическую цитату ❸. Выполнив всю тяжелую работу, мы возвращаем объект `randomQuote` в представление в виде переменной с именем `quote` ❹, к которой мы сможем обратиться в файле GSP.

Теперь, когда мы реализовали выборку случайной цитаты, вернемся на страницу *http://localhost:8080/qotd/quote/random*, чтобы увидеть, как это работает. На рис. 1.10 показана выборка случайной цитаты в действии.

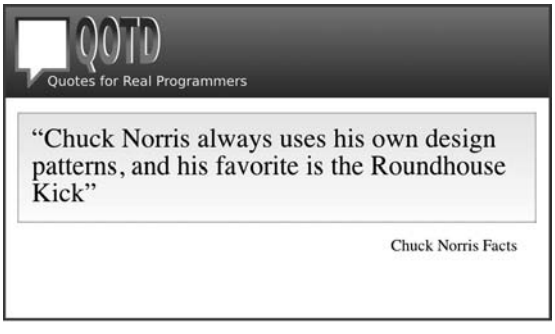


***Рис. 1.10.** Выборка случайной цитаты в действии*

1.5.1. Скаффолдинг: просто добавим ракетного топлива

Мы выполнили всю тяжелую работу по созданию модели данных. Теперь нам требуется дополнить контроллер методами выполнения CRUD-операций, чтобы дать пользователям возможность добавлять свои цитаты в базу данных.

Но это необходимо, если мы хотим выполнить искусную реализацию. А если нам нужно быстро получить результат, можно воспользоваться потрясающей особенностью платформы Grails, называемой *скаффолдингом*. Эта особенность динамически создает простейшие методы действий контроллера и представления, которые дополняют модель данных типичными операциями CRUD.

Как задействовать скаффолдинг для создания страниц, позволяющих добавлять и изменять информацию о цитатах? Делается это одной строкой программного кода в объявлении класса `QuoteController`, как показано в листинге 1.13.

Листинг 1.13. Включение скаффолдинга

```
class QuoteController {
    def scaffold = true
    // далее следует остальной программный код...
}
```

Вот и все. Когда платформа Grails обнаружит контроллер, помеченный как `scaffold = true`, она автоматически создаст несколько простейших действий контроллера и представлений GSP. Если хотите увидеть эту особенность в действии, откройте страницу *http://localhost:8080/qotd/quote/list*, и вы получите доступную для редактирования страницу, показанную на рис. 1.11.

Рис. 1.11. *Метод list() в действии*

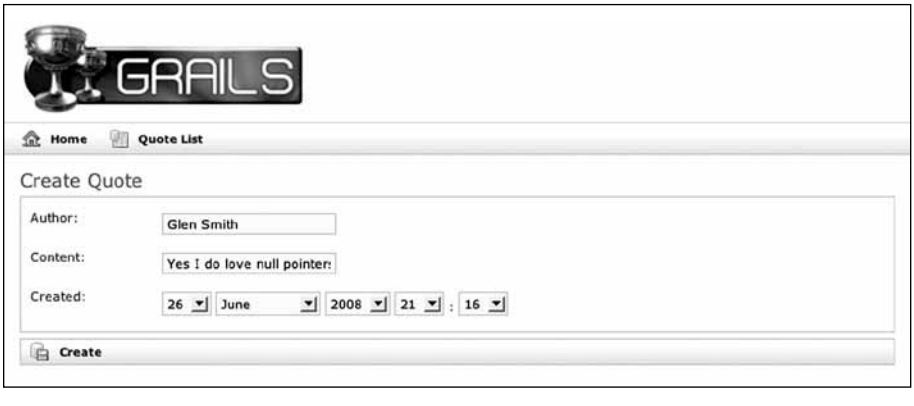


Рис. 1.12. *Трудно представить себе более простой способ добавления цитат*

Щелкните на кнопке New Quote (Новая цитата), и перед вами откроется страница, где вы сможете добавить новую цитату, как показано на рис. 1.12.

Это слишком мощная возможность, чтобы получить ее бесплатно. Сгенерированные методы и представления, вероятно, недостаточно привлекательны для действующего веб-сайта, но они являются совершенно фантастическим приобретением для административного раздела

и прекрасно подходят для возни с базой данных в процессе разработки (когда нет желания загружать себя созданием пакета страниц для выполнения операций с базой данных).

1.5.2. Выживание в наихудших случаях

Наша модель выглядит весьма неплохо, а скаффолдинг – просто великолепно, но мы упустили из виду проблему обеспечения надежности приложения. Было бы нежелательно, чтобы пользователи помещали в базу данных опасные данные, поэтому сейчас мы займемся проверкой данных.

Проверка выполняется объектом `Quote`, поэтому мы просто заполним замыкание `constraints` необходимыми нам правилами. Сначала убедимся, что пользователи всегда заполняют поля `author` и `content`, как показано в листинге 1.14.

Листинг 1.14. Добавление простейших проверок

```
class Quote {

    String content
    String author
    Date created = new Date()

    static constraints = {
        author(blank:false)
        content(maxSize:1000, blank:false)
    }

}
```

Правила проверки данных

Эти ограничения сообщают платформе Grails, что ни поле `author`, ни поле `content` не могут оставаться пустыми (то есть эти поля не могут иметь значение `null` или значение нулевой длины). Если не указывать размер для полей типа `String`, они будут определены в базе данных как `VARCHAR(255)`. Возможно, этого вполне достаточно для поля `author`, но содержимое поля `content` может оказаться немного больше. По этой причине мы добавили ограничение `maxSize`.

Правила в замыкании `constraints` также оказывают влияние на работу скаффолдинга. Например, порядок следования правил в замыкании `constraints` определяет порядок следования полей в создаваемых страницах. Поля, которые согласно ограничению должны содержать больше 255 символов, отображаются в разметке HTML не как поля `TEXT`, а как элементы `TEXTAREA`. На рис. 1.13 показано, как выглядит сообщение об ошибке, выводимое при нарушении ограничений.

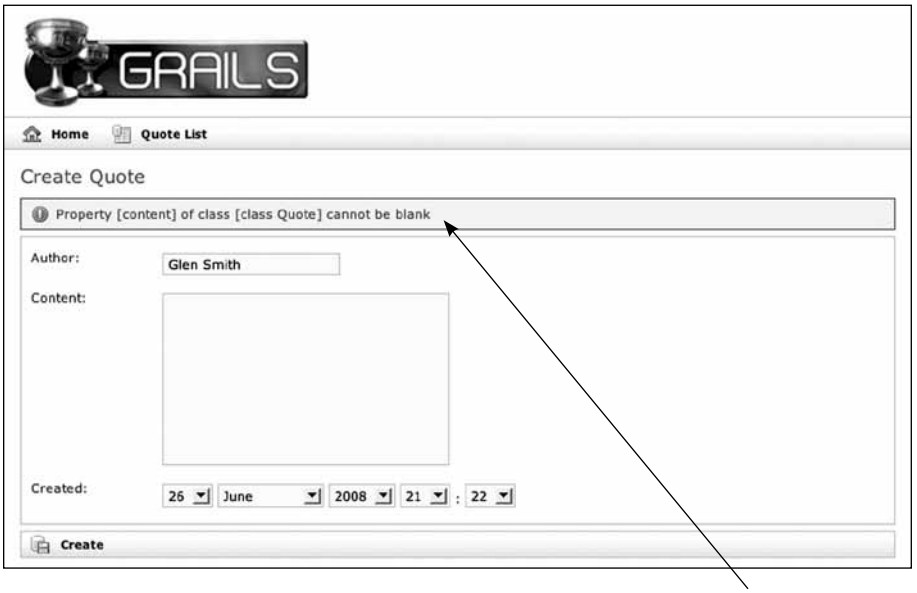


Рис. 1.13. *Когда обнаруживается нарушение ограничения, красным цветом выводится сообщение*

1.6. Улучшение архитектуры

Распределение логики работы приложения по методам действий выглядит достаточно хорошо. Мы довольно легко можем определить, что и где выполняется в нашем маленьком приложении; сопровождение его также не представляет большого труда. Но по мере роста приложения все эти вещи будут усложняться. Для нас желательно было бы определить логику программы так, чтобы ее можно было многократно использовать в разных методах контроллера и даже в разных контроллерах. Пришло время навести порядок в логике приложения, и самый лучший способ сделать это в платформе Grails – использовать службу.

Создадим нашу службу и приступим к знакомству с ней в процессе реализации:

```
grails create-service quote
```

Эта команда создаст скелет службы цитат в файле /grails-app/services/QuoteService.groovy:

```
class QuoteService {

  boolean transactional = true

  def serviceMethod() {

  }
}
```

Обратите внимание, что служба может быть объявлена с поддержкой транзакций, но подробнее об этом мы поговорим ниже. А пока переместим логику выборки случайной цитаты в отдельный метод службы, как показано в листинге 1.15.

Листинг 1.15. Расширение службы

```
class QuoteService {

  boolean transactional = false

  def getStaticQuote() {
    return new Quote(author: "Anonymous",
      content: "Real Programmers Don't eat quiche")
  }

  def getRandomQuote() {

    def allQuotes = Quote.list()
    def randomQuote = null
    if (allQuotes.size() > 0) {
      def randomIdx = new Random().nextInt(allQuotes.size())
      randomQuote = allQuotes[randomIdx]
    } else {
      randomQuote = getStaticQuote()
    }
    return randomQuote
  }
}
```

Наша служба реализована. А как задействовать ее в контроллере? Здесь снова вступают в игру соглашения по именованию. Мы просто добавим в контроллер новое поле с именем `quoteService`, и платформа Grails внедрит службу в контроллер. В листинге 1.16 приводится измененная реализация контроллера.

Листинг 1.16. Вызов службы

```
class QuoteController {

  def scaffold = true

  def quoteService

  def random = {
    def randomQuote = quoteService.getRandomQuote()
    [ quote : randomQuote ]
  }

}
```

Программный код теперь выглядит более опрятно, не так ли? Наша служба QuoteService реализует всю бизнес-логику, связанную с цитатами, а контроллер QuoteController снабжается всеми необходимыми методами. Если у вас есть опыт использования принципа инверсии управления (Inversion of Control, IoC) в таких платформах, как Spring или Google Guice, вы легко узнаете шаблон проектирования приложений внедрения зависимости (Dependency Injection, DI). Платформа Grails выводит принцип DI на совершенно новый уровень путем использования соглашения об именах переменных для определения того, какая зависимость внедряется. Но нам еще требуется написать тесты для проверки бизнес-логики, поэтому сейчас мы займемся исследованием поддержки возможности тестирования в платформе Grails.

1.6.1. Ваш первый испытательный тест на платформе Grails

Тестирование – неотъемлемая составляющая методики гибкой разработки; поддержка тестирования в платформе Grails встроена в саму платформу. Платформа Grails настолько требовательна к тестированию, что когда мы создали нашу службу QuoteService, она автоматически создала испытательный модульный тест в файле /test/unit/QuoteServiceTests.groovy с целью направить нас на верный путь. Но модульные тесты (которые мы будем исследовать в главе 7) требуют, чтобы была создана имитация обращений к базе данных. Сейчас нам необходим интеграционный тест (который создаст в памяти «настоящую» базу данных для нужд тестирования). Создадим его следующей командой:

```
grails create-integration-test QuoteServiceIntegration
```

В результате этой команды Grails создаст файл /test/integration/QuoteServiceIntegrationTests.groovy. Мы использовали суффикс «IntegrationTests», чтобы избежать конфликтов имен классов с существующим модульным тестом в файле /test/unit/QuoteServiceTests.groovy. В листинге 1.17 показано, как выглядит интеграционный тест сразу после создания.

Листинг 1.17. Наш первый испытательный тест

```
import grails.test.*

class QuoteServiceIntegrationTests extends GrailsUnitTestCase {

  protected void setUp() {
    super.setUp()
  }

  protected void tearDown() {
    super.tearDown()
  }
```

```
    void testSomething() {
    }
}
```

Не так много, но для начала вполне достаточно. К тестам применяется то же правило преимущества соглашений перед настройками, поэтому дополним наш испытательный тест `QuoteServiceIntegrationTests` так, чтобы внедрить в него службу, под которой будет выполняться тестирование, как показано в листинге 1.18.

Листинг 1.18. Добавление действующих тестов

```
class QuoteServiceIntegrationTests extends GrailsUnitTestCase {

  def quoteService

  void testStaticQuote() {
    def staticQuote = quoteService.getStaticQuote()
    assertEquals("Anonymous", staticQuote.author)
    assertEquals("Real Programmers Don't eat quiche", staticQuote.content)
  }
}
```

Метод `getStaticQuote()` не такой большой, чтобы в нем могла возникнуть ошибка, тем не менее создадим тест и для него, чтобы обеспечить полноту реализации. Чтобы запустить тесты, выполните команду `grails test-app`. Вы должны увидеть результаты, подобные показанным в листинге 1.19.

Листинг 1.19. Результаты тестирования

```
-------------------------------------------------------
Running 1 Integration Tests...
Running test QuoteServiceIntegrationTests...
              testStaticQuote...SUCCESS
Integration Tests Completed in 284ms
-------------------------------------------------------
```

Листинг 1.19 показывает, что наши тесты прошли успешно. Платформа Grails генерирует также HTML-версию результатов тестирования, которую можно открыть в веб-броузере, по адресу /test/reports/html/index.html. Здесь можно посмотреть результаты испытаний всего проекта, а также отдельных его компонентов, чтобы выяснить, какие тесты потерпели неудачу и почему, как показано на рис. 1.14.

В главе 7 мы узнаем, как повысить эффективность тестов, а пока у нас имеется работоспособный тест, и мы знаем, как просматривать результаты тестирования.

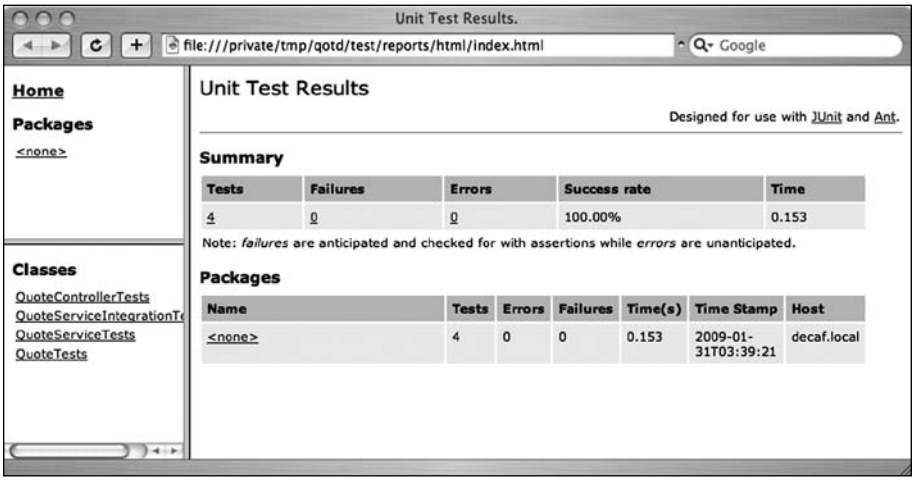


Рис. 1.14. *Отчет в формате HTML о прохождении интеграционного теста*

1.6.2. Web 2.0: представление с применением технологии Ajax

Наш пример приложения не был бы полным без небольшого количества секретной приправы в виде технологии Ajax (Asynchronous JavaScript and XML – асинхронный JavaScript и XML). Если вы ничего не слышали об Ajax, заметим, что применение этой технологии позволяет обновлять отдельные фрагменты веб-страниц с помощью JavaScript. Использование технологии Ajax может сделать наше приложение намного более отзывчивым благодаря обновлению текста цитаты без повторной загрузки баннеров в заголовке и прочего содержимого страницы. Кроме того, оно дает нам возможность заглянуть в библиотеки тегов платформы Grails.

Добавим поддержку технологии Ajax в наше представление random.gsp. Сначала нам следует подключить библиотеку Ajax в элементе `<head>` (мы будем использовать библиотеку Prototype, однако платформа Grails позволяет использовать также YUI, Dojo и любые другие библиотеки). Измененный фрагмент файла random.gsp приводится в листинге 1.20.

Листинг 1.20. Подключение библиотеки поддержки технологии Ajax

```
<head>
    <title>Random Quote</title>
    <g:javascript library="prototype" />
</head>
```

Затем мы добавим раздел меню в тело страницы random.gsp, которое позволит пользователям отображать новые цитаты или переходить

в административный раздел. Обе ссылки, как с поддержкой Ajax, для обновления цитаты, так и обычная, для перехода в административный раздел, будут созданы с помощью библиотеки тегов платформы Grails. Разметка HTML нашего нового меню приводится в листинге 1.21. Мы добавим этот фрагмент перед тегом , который окружает тело страницы.

Листинг 1.21. Использование возможностей технологии Ajax

```
<ul id="menu">
  <li>
    <g:remoteLink action="ajaxRandom" update="quote">
      Next Quote
    </g:remoteLink>
  </li>
  <li>
    <g:link action="list">
      Admin
    </g:link>
  </li>
</ul>
```

Вы уже сталкивались с подобными библиотечными тегами выше в этой же главе (в разделе 1.3.3), когда использовали их для создания стандартизированного макета страниц нашего приложения. В данном примере мы используем тег g:remoteLink, представляющий в платформе Grails гиперссылку с поддержкой Ajax, и тег g:link, создающий стандартную гиперссылку.

По щелчку на этой ссылке платформа Grails будет вызывать метод действия ajaxRandom контроллера, отправившего эту страницу, в данном случае контроллера QuoteController, и поместит полученную разметку HTML внутрь элемента div со значением quote в атрибуте id. Но мы еще не написали метод ajaxRandom, поэтому сделаем это прямо сейчас. В листинге 1.22 приводится измененный фрагмент файла QuoteController.groovy с реализацией нового метода.

Листинг 1.22. Ajax-метод на стороне сервера

```
def ajaxRandom = {
  def randomQuote = quoteService.getRandomQuote()
  render "<q>${randomQuote.content}</q>" +
                  "<p>${randomQuote.author}</p>"
}
```

Мы уже выполнили всю тяжелую работу в службе QuoteService, поэтому здесь мы можем просто воспользоваться ею. В данном случае нам необходимо предотвратить декорирование вывода макетом Grails, поэтому мы отправляем ответ напрямую броузеру (более элегантные способы бу-

дут представлены в следующих главах). Обновите страницу приложения в броузере, теперь она должна выглядеть, как показано на рис. 1.15.

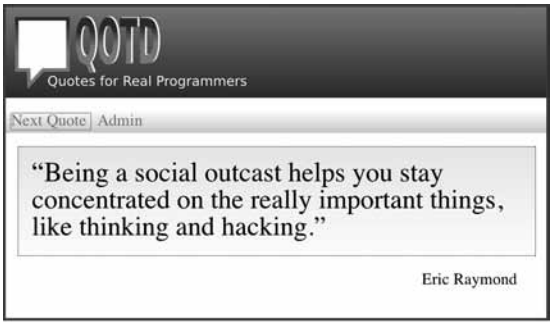


Рис. 1.15. *Представление с поддержкой Ajax в действии*

Чтобы убедиться, что приложение действительно использует прелести технологии Ajax, щелкните на пункте `Next Quote` (Следующая цитата) меню несколько раз. Обратили внимание, что теперь пропал раздражающий эффект перерисовывания страницы? Вы попали в заветный мир Web 2.0!

1.6.3. Сборка конечного продукта: создание файла WAR

Посмотрите, как много нам удалось сделать всего за полчаса! Но что проку запускать приложение на своем ноутбуке? Вам нужно подготовить его и развернуть на настоящем сервере, где-нибудь в Интернете. Для этого следует создать файл WAR, что легко сделать с помощью одной команды платформы Grails:

```
grails war
```

Взгляните на вывод команды, и вы увидите, что платформа Grails упаковала все необходимые файлы JAR, а также файлы приложения, и создала в корневом каталоге проекта файл WAR:

```
Done creating WAR /Users/glen/qotd/qotd-0.1.war
```

Теперь все готово, чтобы его развернуть.

1.6.4. И всего 55 строк кода

Мы узнали достаточно много. И написали немало программного кода. Но не будем верить мне на слово – позвольте платформе Grails подсчитать все количественные характеристики, выполнив команду `grails stats`. Результат работы команды `grails stats` приводится в листинге 1.23.

Листинг 1.23. Определение количественных характеристик: команда grails stats в действии

```
grails stats
```

```
+----------------------+-------+-------+
| Name                 | Files | LOC   |
+----------------------+-------+-------+
| Controllers          |     1 |    13 |
| Domain Classes       |     1 |     9 |
| Services             |     1 |    17 |
| Integration Tests    |     3 |    16 |
+----------------------+-------+-------+
| Totals               |     6 |    55 |
+----------------------+-------+-------+
```

Всего 55 строк кода (столбец LOC)! Может быть, мы и не реализовали столько, сколько нам хотелось. Однако следует заметить, что 55 строк – это не так плохо для приложения, обладающего поддержкой Ajax, способного выводить случайные цитаты и дающего пользователям возможность добавлять свои и изменять существующие цитаты.

Это было введение в платформу Grails. Мы познакомились с моделями, представлениями, контроллерами, службами, библиотеками тегов, макетами и модульными тестами. И многое еще предстоит изучить. Однако прежде чем двинуться дальше, вероятно, будет нелишним исследовать язык программирования Groovy.

1.7. В заключение

Поздравляем, вы написали и развернули свое первое приложение на платформе Grails и теперь вы представляете, как с нуля создать законченный проект. Удивительная продуктивность платформы может вызывать только изумление.

Ниже приводятся несколько важных положений, которые вы должны вынести из этой главы:

- *Ключевое значение имеет быстрота итераций*. Самое важное, что следует вынести из этой главы, – платформа Grails способствует быстрым итерациям операций в процессе разработки, что позволяет создавать и запускать приложение в рекордно короткий срок, и при этом вы получаете массу удовольствия.
- *Устранение помех упрощает сопровождение и ускоряет процесс разработки*. Благодаря использованию принципа преимущества соглашений перед настройками платформа Grails позволяет ликвидировать тонны файлов XML с настройками, которые обычно снижают привлекательность веб-платформ на языке Java.
- *Самонастройка приложения экономит время*. В некоторых случаях, когда возникает необходимость в применении скаффолдинга (на-

пример, при проектировании пользовательского интерфейса), платформа Grails генерирует весь шаблонный программный код, готовый к использованию. Это еще одна особенность платформы Grails, позволяющая экономить время.

- *Тестирование предусмотрено по умолчанию.* Платформа Grails существенно упрощает процедуру написания испытательных тестов. Более того, она даже создает заготовки испытательных тестов. Найдите время, чтобы изучить философию тестирования, заложенную в платформу Grails (которая подробно будет рассматриваться в главе 7), и практикуйте ее применение в повседневной работе.

Вам еще многому предстоит научиться. В оставшейся части книги мы во всех подробностях расскажем вам о разработке функциональных, надежных и простых в сопровождении приложений на платформе Grails и попутно будем давать вам советы, рекомендации и указывать на типичные ловушки.

В этой главе:

- Введение в язык программирования Groovy
- Различия между языками Java и Groovy
- Динамические особенности языка Groovy

2

Основы Grails

Grails – замечательная платформа, существенно упрощающая разработку веб-приложений. Вы уже видели, насколько легко с ее помощью создаются с нуля удобные приложения. Продуктивность этой платформы в значительной степени обусловлена тем, что в качестве базового языка программирования в ней используется язык Groovy.

Цель данной главы заключается в том, чтобы познакомить разработчиков на языке Java с основами Groovy, которые пригодятся в остальной части книги. Информация будет подаваться в быстром темпе, но вам не нужно стараться понять все сразу. Мы предлагаем вам периодически возвращаться к этой главе, когда по мере накопления опыта вам потребуется чуть углубиться в Groovy. Если вы уже знакомы с этим языком программирования, мы все равно рекомендуем вам прочитать эту главу, так как мы будем говорить о некоторых тонкостях и ловушках, о которых вы можете быть не осведомлены.

Сначала мы рассмотрим небольшие различия, существующие между языками Groovy и Java, большую часть которых составляют различия в синтаксисе. Затем мы подчеркнем особенности языка Java, отсутствующие в языке Groovy (их совсем немного), после чего перейдем к знакомству с замыканиями и динамическими особенностями языка. Даже после быстрого прочтения этой главы вы будете знать достаточно, чтобы следовать за примерами в книге и писать свой собственный программный код. Более того, в этой главе достаточно информации, чтобы вы могли стать компетентным разработчиком приложений на языке Groovy (и на платформе Grails!).

2.1. Введение

Мы могли бы сразу же перейти к описанию различий между языками Groovy и Java, но нет ничего дурного в том, чтобы сначала получить некоторое представление о самом языке Groovy. Для этого мы приведем

пару команд платформы Grails, которые позволят вам просто и безопасно поэкспериментировать с ним. Затем мы расскажем вам о нескольких небольших, но важных различиях между двумя языками, которые дадут вам возможность следовать за остальными примерами этой главы.

2.1.1. Поиграем с Groovy!

У вас уже установлена платформа Grails, и теперь вы без лишней суеты можете опробовать язык программирования с таким эпатажным названием, как Groovy. Какие же возможности имеются в вашем распоряжении? В первой главе вы уже видели, как можно использовать команду `create-app` для создания проекта на платформе Grails. Существуют еще две команды, дающие доступ к интерактивной оболочке для запуска программного кода на языке Groovy. Они могут использоваться для экспериментов с приложением на платформе Grails в интерактивном режиме.

Версии Grails и Groovy

В данной книге рассматривается платформа Grails версии 1.1, которая распространяется с собственной копией языка Groovy 1.6, поэтому именно к этой версии Groovy относится все, о чем рассказывается в главе 2. Большая часть сведений применима к версии Groovy 1.5, однако некоторые примеры могут не работать с ней.

Первая команда – `shell`. Она запускает интерактивную оболочку или командную строку, где вы можете вводить выражения на языке Groovy и просматривать результаты. На рис. 2.1 показан сеанс работы с такой оболочкой. Как видите, чтобы завершить работу с оболочкой, достаточно просто ввести команду `quit`.

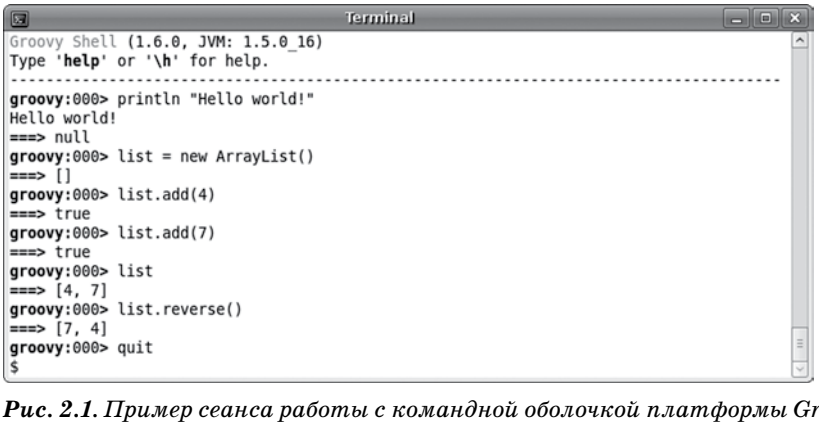


Рис. 2.1. *Пример сеанса работы с командной оболочкой платформы Grails, которая запускается командой shell*

Оболочка прекрасно подходит для вычисления выражений и проверки результатов вызовов методов; с ее помощью можно даже определять новые классы. Однако она не позволяет править уже существующий программный код, поэтому имеет несколько ограниченный круг применений.

Консоль платформы Grails, запускаемая командой `console`, – это приложение на базе библиотеки Swing, которое позволяет писать и запускать простые сценарии. Более того, она даже умеет подсвечивать синтаксис! Консоль состоит из двух панелей: верхняя предназначена для ввода программного кода на языке Groovy, а нижняя отображает результаты работы сценария. Главное преимущество консоли перед командной оболочкой состоит в том, что она позволяет внести изменения в сценарий и тут же опробовать их. Консоль позволяет также сохранять сценарии и загружать их позже. На рис. 2.2 изображено приложение консоли в действии.

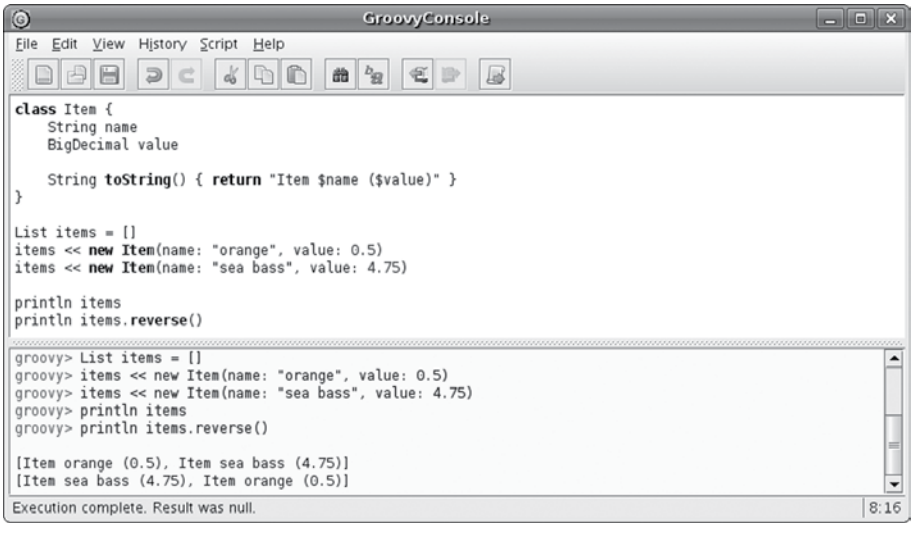


***Рис. 2.2.** Консоль платформы Grails в действии*

Примечание

В пределах проекта на платформе Grails вы можете запускать только команды `shell` и `console`. Следует заметить, что эти две команды используют стандартные командную оболочку и консоль языка Groovy соответственно, что объясняет наличие ссылок «Groovy Shell» и «GroovyConsole» в интерфейсах. Однако платформа Grails привносит нечто свое, позволяющее вам взаимодействовать с проектом Grails способами, невозможными из стандартной командной оболочки или консоли языка Groovy.

Сценарий на рис. 2.2 позволяет заметить несколько отличий между языками Groovy и Java, о которых мы собираемся рассказать, но сначала мы должны пояснить, что этот сценарий делает.

Что делает этот сценарий?

Чтобы выполнить программный код на языке Java, необходимо создать класс и определить в нем статический метод `main()`. Данная необходимость не кажется слишком обременительной, но она удивительным образом эффективно препятствует созданию простых приложений. Чтобы устранить это препятствие, язык Groovy обеспечивает поддержку сценариев: коллекцию инструкций в виде текстовых файлов, которые выполняются в порядке их следования. Сценарий можно представить себе как тело метода `main()` без определения класса и метода. Как видно на рис. 2.2, поддерживается даже возможность включать в сценарии определения классов.

Попробуйте выполнить следующий простой сценарий в консоли Grails:

```
println "Hello world!"
```

Для запуска сценария можно воспользоваться либо пунктом меню Script (Сценарий) → Run (Запустить), либо комбинацией клавиш Ctrl-R. Итак! Вы только что написали и выполнили простой сценарий на языке Groovy.

Теперь, когда вы знаете, как пользоваться консолью, можно приступать к исследованию примеров. Рассмотрим некоторые из основных особенностей языка, которые вы должны знать.

2.1.2. Некоторые основные отличия от языка Java

С некоторыми исключениями программный код, допустимый на языке Java, будет считаться допустимым и на языке Groovy. Но применение особенностей языка Groovy подразумевает использование его некоторых дополнительных преимуществ. Сначала рассмотрим несколько понятий языка Java, которые отличаются в языке Groovy.

Меньше инструкций импортирования

Как вы знаете, все классы в пакете `java.lang` в языке Java импортируются автоматически. Язык Groovy расширил эту особенность и автоматически импортирует следующие пакеты:

- `java.io`
- `java.math`
- `java.net`
- `java.util`
- `groovy.lang`
- `groovy.util`

Это удобно при разработке сценариев, так как классы из данных базовых пакетов используются чрезвычайно часто, и было бы слишком утомительно запускать сценарий только для того, чтобы обнаружить ошибку, связанную с отсутствием инструкции импорта одного из указанных пакетов.

Проверки

Во многих примерах в этой главе используется ключевое слово `assert`:

```
assert 1 == 1
assert 'c' != 'z'
```

Все подобные проверки возбуждают исключение `java.lang.AssertionError`, если выражение в результате дает значение `false` (в соответствии с понятием истины в языке Groovy, о чем рассказывается ниже). Инструкция `assert` должна быть вам знакома по языку Java, но, несмотря на похожую семантику в языке Java, она может включаться и выключаться с помощью аргумента `-ea` виртуальной машины JVM.

Истина

Вы могли бы ожидать, что «Механизм определения истины в Groovy» даст ответ на вопрос о смысле жизни, Вселенной и всего сущего. К сожалению, ответ на этот вопрос вам придется поискать в другом месте. В действительности понятие истины в Groovy имеет более прозаический смысл и описывает, как язык Groovy определяет значения `true` и `false`.

В языке Java условные операторы работают только с логическими (Boolean) значениями, что часто приводит к необходимости писать избыточный код:

```
if (mySet == null || mySet.isEmpty()) {
    ...
}
```

Язык Groovy помогает уменьшить эту избыточность и повысить удобочитаемость программного кода за счет приведения других объектов к типу Boolean. Например, и ссылка `null`, и пустая коллекция будут оцениваться как значение `false`, благодаря чему эквивалентный программный код на языке Groovy мог бы выглядеть, как показано ниже:

```
if (!mySet) {
    ...
}
```

В табл. 2.1 перечислены варианты приведения типов, используемые механизмом истины Groovy. Обратите внимание, что ссылка со значением `null` рассматривается как значение `false` независимо от типа объекта, на который она ссылается, поэтому, например, и пустая ссылка на отображение, и пустое отображение будут интерпретироваться как значение `false`.

Таблица 2.1. Действие механизма определения истины в Groovy

Тип	Интерпретируется как -
Boolean	false, Boolean.FALSE
Ссылка на объект	Null
Number	0
String, GString	Строка нулевой длины
Collection	Пустая коллекция
Map	Пустое отображение
Iterator	hasNext(), возвращающий false
Enumerator	hasMoreElements(), возвращающий false
java.util.regex.Matcher	find(), возвращающий false

Будьте внимательны при использовании механизма поиска истины в языке Groovy – действительно ли вы проверяете именно то, что вам требуется? Спросите себя, хотите ли вы проверить на равенство значению null или на равенство с каким-либо другим значением. Например, если известно, что выполняется проверка коллекции и ссылку null необходимо интерпретировать так же, как пустую коллекцию, то использование механизма определения истины в Groovy вполне уместно. Однако если необходимо, чтобы пустая коллекция и значение null интерпретировались по-разному, проверку необходимо выполнять явно, например: obj == null. Это ясно обозначит ваши намерения для любого, кто будет читать ваш код.

По умолчанию используется область видимости Public

Как известно, в языке Java существуют четыре области видимости: private, protected, public и по умолчанию (в пределах пакета). Создатели языка Groovy решили, что последняя из них избыточна, поэтому в Groovy поддерживаются только первые три области видимости. Это, в свою очередь, привело к тому, что было принято решение сделать область видимости public областью видимости по умолчанию, поскольку она используется наиболее часто. Если в объявлении класса, поля или метода отсутствует явное определение области видимости, он по умолчанию получает область видимости public, а не область видимости пакета по умолчанию в языке Java.

Контролируемые исключения перехватывать необязательно

Обработка ошибок в языке Groovy выполняется так же, как в языке Java. Синтаксически обработка исключений в этих двух языках производится идентично, с помощью конструкций try...catch...finally. Различие заключается в том, что в языке Groovy вы не обязаны обрабатывать контролируемые исключения (checked exceptions). Это означает,

что контролируемые исключения, независимо от вида и назначения, становятся полностью идентичны исключениям времени выполнения.

В общем и целом это приводит к уменьшению шаблонного кода, потому что от вас требуется перехватывать только те исключения, которые вы знаете, как обрабатывать. Но иногда бывают ситуации, когда важно помнить, что вы имеете дело с контролируемыми исключениями. Например, транзакции платформы Spring по умолчанию работают только с исключениями времени выполнения (как будет показано в главе 14). Если в данный момент для вас это ничего не значит, не волнуйтесь – просто помните, что контролируемые исключения остаются контролируемыми исключениями, даже если вы их не перехватываете!

Пока все совсем неплохо, не правда ли? Теперь у нас есть фундамент, отталкиваясь от которого, можно перейти к другим, более тонким различиям, каждое из которых призвано облегчить вам жизнь.

2.1.3. Некоторые новые операторы

Безусловно, вы знакомы с такими операторами, как: `+ - .` и так далее. Язык Groovy поддерживает те же операторы, что и язык Java, но в нем появилось несколько новых операторов, которые способны облегчить вам жизнь и сделать ваш код более удобочитаемым.

?. (Безопасный способ работы с пустыми ссылками)

Приходилось ли вам когда-нибудь испытывать раздражение от необходимости проверять ссылку на равенство значению `null`, прежде чем вызывать метод относительно нее? Разработчики языка Groovy испытывали, поэтому они добавили разновидность оператора точки (`.`), обеспечивающую безопасный способ работы с пустыми ссылками.

Взгляните на следующий, распространенный в языке Java, прием, который проверяет, не равен ли объект или одно из его свойств значению `null`:

```
if (obj != null && obj.value != null) {
    ...
}
```

В языке Groovy эту же проверку можно реализовать так:

```
if (obj?.value != null) {
    ...
}
```

В этом примере обращение к полю `value` происходит, только если ссылка `obj` не равна `null`. В противном случае результатом всего выражения `obj?.value` будет значение `null` и, как результат, – никаких больше исключений `NullPointerException`! Вы можете и даже должны составлять цепочки с помощью этого оператора. В противном случае остается вероятность появления исключений. Поэтому делайте так:

```
user.company?.address?.state?.name
```

а не так:

```
user.company?.address.state.name
```

Единожды задействовав в цепочке оператор ?., вы должны использовать его и в остальной части цепочки.

?: (Элвис!)

И в Java, и в Groovy имеется трехместный оператор, представляющий собой сокращенную форму записи условного оператора if...then...else:

```
<условие> ? <выражение1> : <выражение2>
```

То есть, если условие истинно (true), вычисляется выражение1; в противном случае вычисляется выражение2.

Ниже приводится типичный пример использования этого оператора:

```
String name = (person.getName() != null) ? person.getName() : "<unknown>"
```

Проще говоря, если имя человека задано, мы используем его. В противном случае используется значение по умолчанию.

Этот оператор применяется настолько часто (и настолько в избыточной форме записи), что в язык Groovy была введена сокращенная форма записи этого оператора:

```
String name = person.getName() ?: "<unknown>"
```

Это оператор Элвис; он возвращает значение левого выражения, если оно имеет значение true (согласно механизму определения истины в Groovy), и значение правого выражения в противном случае.

Почему «Элвис»? Поверните оператор на 90° по часовой стрелке, и все станет понятно!

*. (Оператор точка с подстановкой)

Ни один оператор не маскируется под своим названием лучше, чем оператор «точка с подстановкой». Что он делает? Как это ни удивительно, кое-что полезное: он вызывает метод каждого элемента коллекции и создает список, содержащий результаты вызова этих методов.

Например, представьте, что у вас имеется список объектов, каждый из которых имеет метод getName(). Тогда следующий программный код создаст новый список, содержащий названия оригинальных элементов:

```
List names = people*.getName()
```

Элементы в получившемся списке будут располагаться в том же порядке, что и элементы в оригинальном списке, поэтому в предыдущем примере имена будут располагаться в том же порядке, что и информация о людях в оригинальном списке. Оператор может использоваться с кол-

лекциями и других типов, но в результате всегда будет возвращаться список, элементы которого будут располагаться в порядке выполнения итераций в исходной коллекции.

Мы могли бы использовать также синтаксис свойств, который будет описан в одном из следующих разделов:

```
List names = people*.name
```

Это, пожалуй, наиболее часто используемый оператор, который встретится вам как в примерах, так и в действующем программном коде.

<=> (Сравнивание с корабликом)

Язык Java обладает множеством операторов сравнения двух значений, зачем же понадобился еще один? Если когда-либо вам доводилось писать методы `Comparable.compareTo()` или `Comparator.compare()`, вы поймете зачем.

Ниже приводится типичная реализация метода сравнения целых чисел:

```
public int compare(int i1, int i2) {
    if (i1 == i2) return 0;
    else if (i1 < i2) return -1;
    else return 1;
}
```

Для чисел эту реализацию можно сократить до одной строки, но в случае других типов данных это возможно не всегда. Как бы мог выглядеть метод, если бы у нас имелся оператор с поведением, идентичным поведению метода `compare()`? Давайте посмотрим:

```
public int compare(int i1, int i2) {
    return i1 <=> i2;
}
```

Этот оператор в языке Groovy называется «кораблик», и он может использоваться для сравнения любых объектов, реализующих интерфейс `Comparable`. Он особенно удобен при реализации собственных методов сравнения, а в комбинации с некоторыми другими особенностями языка Groovy, с которыми мы познакомимся ниже, – для сортировки списков. Он возвращает отрицательное число, если значение слева меньше, чем значение справа, ноль – если операнды равны, и положительное число – в противном случае. Он не гарантирует абсолютное значение ненулевого результата – только его знак.

== (equals())

При использовании оператора `==` в языке Groovy генерируется программный код, вызывающий метод `equals()`. Он также обеспечивает безопасную работу с пустыми ссылками, то есть он будет работать, даже если оба операнда являются пустыми ссылками (`null`).

Следующий пример демонстрирует поведение оператора при работе со строками и использовании инструкции `assert`:

```
String str = null
assert str == null
assert str != "test"

str = "test"
assert str != null
assert str == "test"

str += " string"
assert str == "test string"
```

Этот пример недостаточно наглядный и имеет некоторые исключения из правил: если объект слева реализует интерфейс `java.lang.Comparable`, то оператор `==` вместо метода `equals()` использует `compareTo()`. В данном случае результат будет равен значению `true`, если метод `compareTo()` вернет `0`, и `false` – в противном случае.

Если вас интересует вопрос, почему Groovy поступает именно так, вспомните, что значения 0.3 и 0.30 с точки зрения метода `BigDecimal.equals()` не эквивалентны, тогда как метод `compareTo()` считает их равными. Опять же в соответствии с принципом «наибольшей предсказуемости» (или «наименьшего удивления» – «least surprise») выражение `0.3 == 0.30` должно возвращать значение `true`, что и происходит в Groovy.

Наконец, как в языке Java получить поведение, аналогичное оператору `==`? Используйте метод `is()`:

```
BigDecimal x = 0.234
BigDecimal y = x

assert y.is(x)
assert !y.is(0.234)
assert y == 0.234
```

Как видите, значение `true` возвращается, только если два объекта являются одним и тем же экземпляром.

Пока описанные операторы кажутся непривычными, но это продлится недолго, до тех пор, когда вы не начнете использовать их в своем программном коде. После такого плавного введения можно приступить к знакомству с системой типов данных в языке Groovy.

2.2. Исследование системы типов данных

Если вы предположите, что система типов данных языка Groovy, как и в случае большинства других его элементов, эквивалентна системе языка Java, то вы окажетесь недалеки от истины. Тем не менее при ра-

боте с типами данных между этими двумя языками начинают проявляться некоторые существенные различия, о которых вам необходимо знать, чтобы раскрыть весь потенциал языка.

2.2.1. Обзор простых типов

Все простые и объектные типы данных, которые можно встретить в языке Java, присутствуют и в языке Groovy: `boolean`, `int`, `double`, `String` и так далее. Это обеспечивает относительную простоту перехода на Groovy и интеграции с языком Java. Но в целом системы типов этих двух языков не являются идентичными. Возьмите простые типы данных: даже если переменная объявляется с простым типом, ее все равно можно интерпретировать как объект, вызывая его методы. Наиболее ярко эта особенность проявляется при работе с числами.

Числа

Не только переменные простых типов могут интерпретироваться как объекты, такое поведение распространяется и на числовые литералы. Например, литерал `1` – это экземпляр класса `Integer`, а литерал `546L` – это экземпляр класса `Long`. Ниже демонстрируется, как можно вызвать метод относительно литерала целого числа:

```
assert (-12345).abs() == 12345
```

Малозаметная, но важная особенность языка Groovy состоит в том, что он реализует вещественную арифметику на базе типа `BigDecimal`, а не `double` или `float`. Сначала это может показаться странным, но в языке Groovy преследуется принцип «наибольшей предсказуемости» (или «наименьшего удивления» – «least surprise»), тогда как в Java одной из самых удивительных особенностей является то, что выражение `0.1D * 3` дает результат `0.300...04`, а не `0.3`. Так как в языке Groovy не ставится цель добиться высокой производительности в вычислениях с плавающей точкой, предпочтение в нем было отдано типу `BigDecimal`, благодаря чему выражение `0.1 * 3` дает в результате `0.3`.

Возможно, вас беспокоит необходимость использовать незнакомый интерфейс класса `BigDecimal`, но не волнуйтесь! Арифметические операторы в языке Groovy обеспечивают прозрачную работу с этим классом, например:

```
assert 0.25 instanceof BigDecimal
assert 0.1 * 3 == 0.3
assert 1.1 + 0.1 == 1.2
assert 1 / 0.25 == 4
```

Любой, кто пытался произвести деление значений типа `BigDecimal` в языке Java, по достоинству оценит простоту выполнения этой операции в Groovy.

Строки в Groovy

Вы уже видели, что строковые литералы Java могут использоваться и в языке Groovy, однако они немного отличаются в этих двух языках.

Прежде всего заметим, что, в отличие от Java, в Groovy отсутствуют символьные литералы. Апострофы (‘’) окружают литералы строк, а не символов. Тогда в чем отличие между апострофами и кавычками? Последние позволяют внедрять в строку выражения Groovy. Значит, вместо того, чтобы использовать оператор конкатенации строк, как это принято в Java, можно записать такой код:

```
String name = “Ben”
String greeting = “Good morning, ${name}”
assert greeting == ‘Good morning, Ben’
String output = “The result of 2 + 2 is: ${2 + 2}”
assert output == “The result of 2 + 2 is: 4”
```

Все, что находится внутри конструкции ${}, будет вычислено, преобразовано в строку с помощью метода `toString()` и вставлено в данную позицию. Если строка содержит символ доллара, следует либо заключить такую строку в апострофы, либо экранировать символы доллара обратным слэшем (“\$”). Этот тип строк известен как `GString`.

Обе формы строковых литералов имеют *многострочные* версии. Это позволяет легко создавать строковые литералы, включающие разрывы строк. Если раньше вам приходилось собирать такие строки в языке Java, вы поймете, насколько удобна указанная возможность. Возьмем для примера простой метод, возвращающий тело электронного письма для человека с заданным именем. В языке Groovy реализация такого метода могла бы выглядеть, как показано ниже:

```
String getEmailBody(String name) {
    return “””Dear ${name},

Thank you for your recent inquiry. One of our team members
will process it shortly and get back to you. Some time in
the next decade. Probably.

Warmest and best regards,

Customer Services
“””
}
```

Обратили внимание, как мы внедрили в строку переменную `name`? Просто и элегантно. Однако следует заметить, что любые пробельные символы, присутствующие между тройными кавычками, будут включены в строку. Поэтому, если вы *не хотите*, чтобы в строке текст имел отступы, начинайте отдельные строки с первой позиции.

Для создания строковых литералов можно также использовать символ слэша (/), но такая форма записи не имеет многострочной версии. Данная возможность может показаться вам немного странной, однако позднее вы увидите, что она широко применяется при работе с регулярными выражениями.

Все, что было сказано выше, относится к строковым литералам, но возникает вопрос – а как объявить символьный литерал? Сделать это можно двумя способами: либо присвоить односимвольную строку переменной типа `char`, либо использовать ключевое слово `as`. Ниже приводятся примеры использования обоих способов:

```
char ch = 'D'
assert ch instanceof Character

String str = "Good morning Ben"
str = str.replace(' ' as char, '+' as char)
assert str == "Good+morning+Ben"
```

Несмотря на то что в предыдущем примере для определения символьных литералов мы использовали апострофы, вы точно так же можете использовать и кавычки. Выбор в пользу апострофов или кавычек во многом зависит от личных предпочтений, но у нашего метода есть свои преимущества – предыдущий пример совместим с языком Java. Уберите два элемента `as char`, и вы получите чистый Java-код! Другое преимущество использования апострофов для определения символьных литералов и кавычек для определения строковых литералов состоит в том, что разработчики на языке Java будут увереннее чувствовать себя при изучении такого программного кода – таким достоинством не стоит пренебрегать.

Раз уж разговор зашел о разработчиках на языке Java, отметим, что ключевое слово `as` будет новым для них. Для чего оно используется?

Ваш услужливый друг: as

В языке Java можно выполнить приведение объекта к типу при условии, что этот объект фактически является экземпляром этого типа. Язык Groovy поддерживает приведение типов языка Java, но позднее вы поймете, что в языке с динамической типизацией данная операция избыточна. Однако иногда бывает полезно иметь возможность простого преобразования типов.

Допустим, что у нас имеется строка `"12345"`. Она выглядит как число, но, и в Java, и в Groovy она будет интерпретироваться как строка. Чтобы преобразовать ее в число в языке Java, необходимо вызвать метод `Integer.parseInt()` или похожий на него. Все это хорошо, но как быть, если стоит задача преобразовать массив в список? Для этого можно воспользоваться методом `Arrays.asList()`. Аналогичным образом в языке Java можно выполнять и другие преобразования, но для них вам потребуется искать соответствующие методы, что не всегда просто.

В таких ситуациях на помощь приходит оператор as языка Groovy. Он позволяет преобразовывать строки в числа или массивы в списки с помощью непротиворечивого синтаксиса. Помимо этого он делает программный код более удобочитаемым, потому что ключевое слово as реализовано как оператор. Ниже приводятся несколько примеров его использования:

```
assert 543667 as String == "543667"
assert 1234.compareTo("34749397" as int) < 0
```

Язык Groovy поддерживает несколько преобразований типов по умолчанию, часть которых перечислена в табл. 2.2. Вникать во все преобразования пока не обязательно, далее в этой книге мы встретимся со многими из них.

Таблица 2.2. Преобразования, поддерживаемые оператором as

Исходный тип	Конечный тип	Описание
String	Число (int, double, Long, BigDecimal и т. д.)	Интерпретирует строку, как если бы это было число
List	Array	Преобразует список в массив. Например, myList as String[] (в языке Groovy отсутствует поддержка массивов литералов, присущая языку Java)
List или Array	Set	Преобразует список или массив во множество
Любой тип	boolean	С помощью механизма языка Groovy определения истины преобразует значение в true или false
Collection	List	Создает список, который является копией оригинальной коллекции. Элементы списка располагаются в порядке выполнения итераций в коллекции
String	List	Интерпретирует строку как последовательность символов и преобразует ее в список

Имеется к тому же возможность добавлять собственные операции приведения типов за счет реализации метода asType(Class). Именно так платформа Grails добавляет собственные приведения типов, такие как конвертеры XML и JSON, о которых будет рассказываться в главе 11 при описании архитектуры REST.

В значительной степени все вышесказанное относится к базовым типам данных. Надеемся, что вы, так же как и мы, найдете тип GString и операции приведения типов удобными для себя. Они определенно представляют собой более элегантные альтернативы операциям конкате-

нации строк и приведения типов с помощью методов. И тем не менее все эти небольшие улучшения меркнут перед поддержкой коллекций и отображений в языке Groovy.

2.2.2. Синтаксический подсластитель для работы со списками, отображениями и диапазонами

Списки, отображения и диапазоны получили настолько широкое распространение, что в некоторых языках программирования они были встроены в систему типов. Однако язык Java имеет лишь ограниченную поддержку этих типов, как будто она была реализована второпях. Язык Groovy продвинулся в этом направлении намного дальше и обеспечивает превосходную поддержку этих типов на уровне языка.

Литералы списков и отображений

Если вам когда-нибудь доводилось создавать список или отображение ссылок на данные в языке Java, вы знаете, насколько это сложно: сначала необходимо создать экземпляр коллекции, а затем добавить в нее каждый элемент по отдельности. В этом отношении язык Groovy способен существенно облегчить нам жизнь, предоставляя возможность определять литералы списков и отображений. Литерал списка – это последовательность элементов, разделенных запятыми, заключенная в квадратные скобки. Литерал отображения представляет собой то же самое, только каждый «элемент» отображения – это разделенная двоеточием пара «ключ/значение». В листинге 2.1 приводится сравнение синтаксических конструкций создания списков и отображений в языках Groovy и Java и одновременно демонстрируется использование литералов списков и отображений.

Листинг 2.1. Инициализация списков и отображений в языке Groovy

```
List myList = new ArrayList()
myList.add("apple")
myList.add("orange")
myList.add("lemon")

Map myMap = new HashMap()
myMap.put(3, "three")
myMap.put(6, "six")
myMap.put(2, "two")
```

← Создание списков и отображений в стиле языка Java

```
List myList = [ "apple", "orange", "lemon" ]
Map myMap = [ 3: "three", 6: "six", 2: "two" ]
```

← Создание списков и отображений в Groovy

```
List l = []
Map m = [:]
```

← Создание пустого списка и отображения

Эти литералы являются экземплярами классов `java.util.List` и `java.util.Map`, поэтому они обладают всеми методами и свойствами этих классов. Например, в инструкции

```
assert 3 == [ 5, 6, 7 ].size()
```

для определения количества элементов в литерале списка используется стандартный метод `size()` класса `List`.

Форма записи списков и отображений в виде массивов

Язык Java имеет довольно неуклюжий синтаксис доступа к элементам списков и отображений, потому что он опирается на применение методов. В языке Groovy эта ситуация улучшена весьма существенно за счет расширения синтаксиса обращения к спискам и отображениям как к массивам. Лучше всего понять это можно на примере. В листинге 2.2 демонстрируются чтение и запись значений обоих типов.

Листинг 2.2. Доступ к элементам списков и отображений

```
List numbers = [ 5, 10, 15, 20, 25 ]
assert numbers[0] == 5          <-- Чтение элементов списка
assert numbers[3] == 20

assert numbers[-1] == 25        <-- ❶ Индексация относительно
assert numbers[-3] == 15            конца списка

numbers[2] = 3                  <-- Замена значения элемента списка
assert numbers[2] == 3

numbers << 30                   <-- ❷ Добавление элемента в список
assert numbers[5] == 30

Map items = [ "one": "apple",
              "two": "orange",
              "three": "pear",
              "four": "cherry" ]
assert items["two"] == "orange"   <-- Чтение элемента отображения
assert items["four"] == "cherry"

items["one"] = "banana"           <-- Замена значения элемента отображения
assert items["one"] == "banana"

items["five"] = "grape"           <-- Добавление элемента в отображение
assert items["five"] == "grape"
```

Как видите, работа со списками и отображениями в языке Groovy реализована просто и понятно. Особо следует отметить возможность индексации списков (и массивов!) относительно конца, что показано в строке ❶ (индекс –1 соответствует последнему элементу). Кроме того, в листинге 2.2 демонстрируется действие оператора сдвига влево ❷. В язы-

ке Groovy это стандартный прием добавления элементов в конец списка (или любого другого типа, реализующего интерфейс `Collection`).

Новый тип данных: диапазон

Тип *диапазон* в языке Groovy имеет некоторое отношение к спискам. В действительности все объекты диапазонов реализуют интерфейс `java.util.List`. Полное перечисление всех возможностей этого типа данных выходит далеко за рамки данной книги, однако вам необходимо поближе познакомиться с целочисленными литералами диапазонов и разобраться с принципами их использования. Более подробную информацию можно найти на веб-сайте проекта Groovy: *http://groovy.codehaus.org/Collections*.

В сущности, диапазон – это комбинация нижней и верхней границ, которые в случае целочисленных диапазонов представляют все числа, заключенные между этими границами. Нижняя граница всегда включается в диапазон, а верхняя граница может как включаться, так и не включаться в диапазон.

В листинге 2.3 демонстрируется программный код, который выполняет итерации через список строк и выводит на консоль сначала элементы списка, а затем подмножество элементов.

Листинг 2.3. Диапазоны

```
List fruit = [
    “apple”,
    “pear”,
    “lemon”,
    “orange”,
    “cherry” ]

for (int i in 0..<fruit.size()) {                    ← ❶ Обход диапазона, исключающего
    println “Fruit number $i is ‘${fruit[i]}’”          верхнюю границу
}

List subList = fruit[1..3] ← ❷ Извлечение фрагмента списка

for (int i in 0..<subList.size()) {
    println “Sub list item $i is ‘${subList[i]}’”
}
```

Листинг демонстрирует два наиболее часто применяемых способа использования диапазонов: для выполнения итераций ❶ и как аргумент инструкции обращения к массиву ❷.

Возможно, синтаксис реализации цикла ❶ покажется вам незнакомым, но не стоит беспокоиться. Этот фрагмент программного кода просто выполняет итерации по целым числам от 0 до 4 включительно (список содержит 5 элементов) и присваивает текущее целочисленное значение переменной `i`. В данном случае используется форма записи, исклю-

чающая верхнюю границу: `lower..<upper`. Обратите также внимание, что в качестве границ диапазона могут использоваться переменные и выражения.

В строке ❷ демонстрируется способ получения фрагмента списка, или, в терминах языка Java, подсписка. Здесь для извлечения со 2 по 4 элементы списка `fruit` использована форма записи, включающая верхнюю границу. Обратите внимание, что изменения оригинального списка отражаются и на фрагменте. Если значение «lemon» третьего элемента списка `fruit` заменить значением «lime», второй элемент фрагмента также получит значение «lime».

Если все это не вызывает у вас изумления, значит, вы уже имеете опыт работы с языком программирования, обладающим богатой поддержкой коллекций и отображений! Те, кто пришел из языка Java, с удивлением будут вспоминать, как они обходились без этих возможностей раньше.

На этом мы заканчиваем знакомство с системой типов данных. Далее мы рассмотрим некоторые особенности языка, которые несколько сложнее в освоении.

2.3. Особенности, позволяющие экономить время

Прежде чем переходить к более существенным особенностям языка Groovy, рассмотрим некоторые вспомогательные возможности. Начнем с безобидных точек с запятой в конце инструкций.

2.3.1. Зачем нужны точки с запятой?

Одно из самых очевидных различий между языками Java и Groovy состоит в том, что в последнем можно не добавлять точки с запятой в конце строк или, если говорить более точно, в конце инструкций. Если у вас появится желание поместить несколько инструкций языка Groovy в одной строке, вы все же должны будете отделять их друг от друга точкой с запятой, как в языке Java.

Эта гибкость имеет свою цену. Вам необходимо проявлять осторожность при переносе инструкции на другую строку, потому что некоторые способы форматирования программного кода могут вызывать ошибку компилятора. В листинге 2.4 демонстрируются примеры допустимых и недопустимых способов переноса на другую строку.

Листинг 2.4. Допустимые и недопустимые способы переноса инструкции на другую строку

```
private void invalidMethod ← Недопустимый перенос объявления метода
        (String name) {
    String fruit = “orange, apple, pear, “   ← Недопустимый перенос
        + “banana, cherry, nectarine”          операции конкатенации
```

```
}

private int validMethod( ← Допустимый перенос объявления метода
        String name) {
    String furniture = "table, chair, sofa, bed, " +
        "cupboard, wardrobe, desk"

    String fruit = "orange, apple, pear, " \ ← ❶ Допустимое объединение строк
        + "banana, cherry, nectarine"

    int i = 0; i++; return i; ← Допустимое использование
                                нескольких инструкций
}
```

← Допустимый перенос операции конкатенации

Эти ограничения можно преодолеть с помощью оператора объединения строк ❶, который фактически говорит: «эта строка еще не окончена». Вы могли бы предположить, что расстановка точек с запятой позволит добиться того же результата, но это не так. Наткнувшись на многострочную инструкцию, интерпретатор Groovy встретит конец строки прежде, чем точку с запятой, завершающую инструкцию, поэтому он все же выдаст сообщение об ошибке.

Вероятно, на данном этапе вас начинает волновать вопрос о правилах, определяющих верное оформление инструкций. Если вся строка представляет собой допустимую инструкцию, интерпретатор Groovy воспримет ее как инструкцию, а все, что встретится на следующей строке, будет рассмотрено как новая инструкция. В противном случае Groovy попробует объединить первую строку с последующей и интерпретировать комбинацию строк как одну инструкцию. Вообще говоря, если из-за этой проблемы у вас возникают ошибки компиляции, используйте оператор объединения строк.

И зачем нужны круглые скобки?

Очень скоро вы обнаружите, что при вызове методов необязательно указывать круглые скобки. Классическим примером является вызов метода `println()`, предоставляемого Groovy JDK (будет рассматриваться в разделе 2.4.3):

```
println "Hello, world!"
```

Методы, принимающие замыкание в виде единственного аргумента (см. раздел 2.4.1), обычно тоже вызываются без круглых скобок с целью повышения удобочитаемости. Однако если вызов метода производится без аргументов или первым аргументом является литерал списка или отображения, круглые скобки обязательны:

```
println()
println([1, 2, 3, 4])
```

Вероятно, существуют и другие ситуации, когда применение круглых скобок обязательно, поэтому мы рекомендуем использовать их в большинстве случаев, опуская их лишь в наиболее типичных случаях (например, при вызове метода `println()` и при использовании замыканий). Соблюдение этой рекомендации облегчит понимание вашего программного кода для разработчиков на языке Java.

Обязательно ли использовать инструкцию return?

Может показаться странным, но в языке Groovy инструкция `return` является необязательной. В случае отсутствия инструкции `return` метод, возвращающий результат, будет возвращать результат последнего вычисленного выражения.

Например, следующий метод возвращает результат выражения `value + 1`:

```
int addOne(int value) { value + 1 }
```

У вас может возникнуть вопрос о причине введения этой возможности. Ответ прост: лаконичность. Данная возможность особенно удобна в однострочных замыканиях, с которыми вы познакомитесь в разделе 2.4.1. Она не препятствует использованию инструкции `return` при наличии такого желания или для повышения удобочитаемости для глаз, настроенных на чтение Java-кода.

Необязательные элементы, подобные перечисленным, заслуживают дополнительного обсуждения, однако у нас есть кое-что, куда действительно стоит вонзить свои зубы.

2.3.2. Встроенная поддержка регулярных выражений

Поддержка регулярных выражений появилась в ядре Java в версии 1.4. Groovy поддерживает регулярные выражения на уровне самого языка. В язык были введены новые литералы, новые операторы сопоставления с регулярными выражениями и синтаксическая конструкция извлечения содержимого групп, что обеспечивает более продуктивную работу с регулярными выражениями.

Литералы

Начнем с новой формы записи строковых литералов, в которой используются символы слэша (/). Для чего необходим и даже желателен еще один тип строковых литералов?

В регулярных выражениях для экранирования служебных символов используется символ обратного слэша (\), например последовательность `\b` представляет метасимвол границы слова. К сожалению, в строковых литералах языка Java символ обратного слэша также используется как экранирующий символ, поэтому регулярные выражения в языке Java быстро превращаются в лес из символов обратного слэша. В качестве примера можно привести регулярное выражение, которое

совпадает с символом обратного слэша, за которым следует произвольный символ слова: "\\\\\\w".

Строковые литералы, окруженные символами слэша, появившиеся в языке Groovy, решают эту проблему. В них символ обратного слэша не имеет специального назначения (за исключением случаев, когда необходимо употребить символ слэша как литерал), что упрощает запись регулярных выражений. При использовании символа / в качестве разделителя предыдущее выражение приобретает вид: /\\\w/. Мы смогли вдвое сократить количество обратных слэшей!

Вам также следует знать, что эти строковые литералы имеют тип GString, что позволяет встраивать в них выражения на языке Groovy. Например, /${name}: \w+/.

Если вы предпочитаете явно иметь дело с экземплярами класса java.util.regex.Pattern, вы можете создавать их, предваряя строковые литералы символом ~:

```
assert ~"London" instanceof java.util.regex.Pattern
assert ~/\w+/ instanceof java.util.regex.Pattern
```

Этот прием удобно применять при необходимости многократного использования шаблона, но такая необходимость встречается нечасто.

Сопоставление

Литералы шаблонов не единственная особенность регулярных выражений, поддерживаемая на уровне языка Groovy. В нем появились также два новых оператора сопоставления, которые должны быть знакомы разработчикам на языках Perl и Ruby.

Первый из них – оператор =~, который отыскивает совпадение шаблона справа в строке слева. Например, проверка

```
assert "Speaking plain English" =~ /plain/
```

будет признана успешной, потому что испытуемая строка содержит подстроку «plain». Противоположный результат дает применение оператора ==~:

```
assert !("Speaking plain English" ==~ /plain/)
```

Это обусловлено тем, что испытуемая строка совпадает с шаблоном не полностью – шаблону соответствует только часть испытуемой строки. Добавление в шаблон пары метасимволов с квантификаторами обеспечивает точное совпадение:

```
assert "Speaking plain English" ==~ /.*plain.*/
```

Однако эти два оператора имеют более фундаментальные отличия. Оператор =~ возвращает экземпляр класса java.util.regex.Matcher, тогда как оператор ==~ возвращает логическое значение.

Но почему тогда допускается прямое использование оператора =~ в инструкции assert? Потому что в случае необходимости интерпретатор Groovy автоматически преобразует объект Matcher в значение логического типа. За кулисами интерпретатор вызывает метод find() объекта Matcher и возвращает его результат.

Группы

Некоторые регулярные выражения содержат сохраняющие группы, содержимое которых можно получить с использованием синтаксиса обращения к массивам.

Пример в листинге 2.5 извлекает все слова, содержащие последовательность символов «ain», и выводит предшествующие ей и следующие за ней символы, присутствующие в каждом совпадении. Если вы запустите его, в консоли будут выведены такие результаты:

```
Found: 'rain' - prefix: 'r', suffix: ''
Found: 'Spain' - prefix: 'Sp', suffix: ''
Found: 'mainly' - prefix: 'm', suffix: 'ly'
Found: 'plain' - prefix: 'pl', suffix: ''
```

Листинг 2.5. Сохраняющие группы в регулярных выражениях

```
import java.util.regex.Matcher

String str = "The rain in Spain falls mainly on the plain"
Matcher m = str =~ /\b(\w*)ain(\w*)\b/ ← Регулярное выражение с двумя
                                         сохраняющими группами
if (m) {
    for (int i in 0..<m.count) {
        println "Found: '${m[i][0]}' - " +
                "prefix: '${m[i][1]}'" +
                ", suffix: '${m[i][2]}'"
    }
}
```

❶ Доступ ко всему совпадению и к подгруппам

Как видите, строковые литералы в слэшах, оператор поиска совпадений и группы обеспечивают намного более краткую форму записи операций с регулярными выражениями, чем в языке Java. Единственная сложная для понимания часть – это использование синтаксиса обращения к массивам ❶.

В листинге 2.5 испытуемая строка содержит четыре слова, имеющих совпадения с шаблоном: «rain», «Spain», «mainly» и «plain». Первый индекс позволяет выбрать требуемое совпадение, так, индекс 1 указывает на совпадение «Spain». Второй индекс позволяет получить содержимое определенной группы в этом совпадении. Индекс со значением 0 соответствует всему совпадению (например, «Spain»), а индексы со значением 1 и выше соответствуют группам. Так, в нашем примере индекс со значением 1 используется для извлечения символов префикса, а индекс со значением 2 – суффикса ❶.

Это все, что касается поддержки регулярных выражений. Мы рекомендуем вам максимально полно использовать описанные преимущества в своем программном коде, но не забывайте, что у вас сохраняется возможность вернуться к синтаксису языка Java и использовать его классы и методы, если особенности языка Groovy покажутся вам неудобными.

Нам осталось обсудить одну небольшую особенность, но она очень важна, поскольку используется практически в каждом программном коде на языке Groovy. Ладно, возможно, это преувеличение, но не очень сильное.

2.3.3. Форма записи свойств

Спецификация JavaBeans ввела понятие *свойств объекта*, но оставила нам громоздкий синтаксис: методы доступа к свойствам. В языке Groovy эта проблема была исправлена за счет обеспечения возможности обращаться к свойствам JavaBeans как к полям.

Возьмем в качестве примера класс `java.util.Date`. У этого класса имеются методы `long getTime()` и `void setTime(long milliseconds)`, которые соответствуют спецификации JavaBeans и представляют свойство типа `long` с именем `time`. В языке Groovy к этому свойству можно обратиться как к полю:

```
Date now = new Date()
println "Current time in milliseconds: ${ now.time }"

now.time = 103467843L
assert now.time == 103467843L
```

Начав использовать этот синтаксис, вы не сможете отказаться от него. Он выглядит намного естественнее вызова методов, требует меньше ввода с клавиатуры и более удобочитаемый. Что еще можно добавить к сказанному?

Свойства – это хорошо, но вам по-прежнему необходимо определять методы доступа к ним, не так ли?

Определение свойств

Создавать методы доступа приходится настолько часто, что интегрированные среды разработки позволяют генерировать их реализацию автоматически. Но даже при этих условиях класс получается наполненным шаблонными методами, которые не содержат никакой значимой логики. К счастью, Groovy обеспечивает более краткую форму записи.

Если при объявлении поля не указать область видимости, Groovy автоматически создаст частное поле и сгенерирует внутренние методы доступа к нему. При желании вы можете даже переопределять эти методы. В следующем примере все поля, за исключением последнего, являются свойствами:

```
class MyProperties {
    static String classVar
```

```
    final String constant = "constant"
    String name

    public String publicField
    private String privateField
}
```

Вот и все – что может быть проще? Примечательно, что для логических свойств Groovy сгенерирует даже методы `is...()`. Для тех, кто не знаком с языком Java: спецификатор `final` означает, что переменная не может изменять свое значение после инициализации (которая может быть выполнена в конструкторе или при объявлении, как в данном примере).

Итак, осталось рассмотреть лишь порядок создания и инициализации компонентов JavaBean.

Создание компонентов JavaBean

При определении компонента JavaBean необходимо предусмотреть конструктор по умолчанию, и язык Groovy удовлетворяет это требование, предоставляя сокращенную форму инициализации таких классов. Все, что вам нужно сделать, – это определить список имен аргументов, соответствующих свойствам, которые требуется инициализировать.

В следующем примере данный подход используется для создания нового экземпляра класса `java.text.SimpleDateFormat`:

```
DateFormat format = new SimpleDateFormat(
    lenient: false,
    numberFormat: NumberFormat.getIntegerInstance(),
    timeZone: TimeZone.getTimeZone("EST"))
```

Список именованных аргументов напоминает отображение, только без квадратных скобок, где роль ключей играют имена свойств класса. В действительности за кулисами Groovy использует экземпляр класса `LinkedHashMap`. Вы можете передавать конструктору даже собственные экземпляры отображения. Этот прием часто используется при создании экземпляров классов доменов из параметров веб-запроса, как вы увидите в главе 5.

Заговорив об отображениях, следует заметить, что Groovy позволяет получать и изменять значения элементов отображений путем использования нотации свойств. Например:

```
Map values = [ fred: 1, peter: 5, glen: 42 ]
assert values.fred == 1
values.peter = 10
assert values.peter == 10
```

Данный прием применяется очень широко, и его особенно удобно использовать в комбинации с приемом грубого определения типа, с которым вы познакомитесь в следующем разделе. Таким образом, вы можете работать с отображениями как с объектами!

Примечание

В языке Groovy ключи отображения интерпретируются как строки, если они не являются числами или не заключены в квадратные скобки. В предыдущем примере `fred` – это строка `"fred"`, а не переменная. Кавычки можно не использовать, если ключ не содержит пробелов, дефисов или других символов, недопустимых в идентификаторах Groovy. Если бы `fred` было именем переменной и потребовалось бы использовать ее значение в качестве ключа, вы могли бы заключить ее в квадратные скобки: `[ (fred): 1 ]`.

Итак, вы узнали о многих особенностях, реализованных в языке Groovy сверх того, что имеется в языке Java, а существуют ли особенности, присутствующие в Java, но отсутствующие в Groovy?

2.3.4. Есть ли что-нибудь в Java, чего нет в Groovy?

Как уже отмечалось выше, в большинстве случаев код Java является допустимым кодом и в языке Groovy. Однако некоторые конструкции в языке Groovy отсутствуют, поэтому весьма важно упомянуть о них.

Символьные литералы

В предыдущем разделе мы показали, что в языке Groovy отсутствуют символьные литералы, а апострофы используются для определения строковых литералов. Но у вас есть возможность преобразовать односимвольную строку в значение типа `char` с помощью оператора `as`.

Цикл «for» языка Java

В Groovy имеется лишь ограниченная поддержка стандартного для Java цикла `for`, поэтому вы не можете использовать оператор запятой (`,`). Следующий пример будет работать:

```
for (int i = 0; i < 10; i++) { ... }
```

А этот – нет:

```
for (int i = 0, j = 0; i < 10; i++, j++) { ... }
```

do...while

Язык Groovy не поддерживает цикл `do...while`, но мы сомневаемся, что кто-то будет оплакивать его отсутствие. Нам редко доводилось встречать его в природе, а для получения того же эффекта вы всегда можете использовать циклы `while` и `for`.

Вложенные и анонимные классы

Вы не можете объявлять вложенные или анонимные классы. В будущем ожидается появление их поддержки, но в общем случае возможность объявления замыканий и нескольких классов в одном файле на

языке Groovy означает лишь наличие действительной проблемы в библиотеках и платформах, основанных на нем.

2.4. Специальные возможности Groovy

Теперь вы должны уверенно понимать и писать программный код на языке Groovy. До сих пор большая часть различий между Groovy и Java относилась к категории «синтаксического подсластителя». В настоящий момент мы переходим к обсуждению двух важных концепций, отличающих Groovy от Java на более фундаментальном уровне: замыканий и динамической природы языка.

2.4.1. Замыкания

Вероятно, вы уже знакомы с методом `sort()` интерфейса `java.util.Collections`, выполняющим сортировку списка с использованием заданной функции сравнения. Следующий фрагмент программного кода на языке Java отсортирует список строк без учета регистра символов и выведет его на консоль:

```
List<String> fruit = Arrays.asList(
    “apple”, “Orange”, “Avocado”,
    “pear”, “cherry”);

Collections.sort(fruit, new Comparator<String>() {
    public int compare(String str1, String str2) {
        return str1.compareToIgnoreCase(str2);
    }
});

System.out.println(“Sorted fruit: “ + fruit);
```

Результаты его работы:

```
Sorted fruit: apple, Avocado, cherry, Orange, pear
```

Вы заметили, что реализация метода не содержит ничего, кроме функции сравнения? Все, что мы хотим сделать, – это передать методу `sort()` функцию, принимающую два аргумента, но в языке Java это невозможно, поэтому нам необходим интерфейс, описывающий функцию, и объект, реализующий его. К счастью для нас, JDK предоставляет интерфейс `java.util.Comparator` и анонимные классы. Но как быть, если вы разработали алгоритм обхода дерева, который должен вызывать функцию для каждой его вершины? Вам придется написать интерфейс, описывающий функцию, и передавать в алгоритм объект, реализующий этот интерфейс.

Решение проблемы оказывается сложнее, чем хотелось бы, и даже при использовании анонимных классов остается желание иметь более эле-

гантную альтернативу. Здесь, как обычно, Groovy предлагает свое решение: замыкания. Вы можете представить их как анонимные функции, которые можно присваивать переменным, передавать методам и даже возвращать из методов. Их можно представить также как анонимные классы, содержащие один метод.

Сначала синтаксис замыканий может показаться сложным, поэтому мы продемонстрируем его на примере. Следующий фрагмент делает то же, что и фрагмент на языке Java, который приводился выше, но в нем используется метод `sort()`, сравнивающий элементы с помощью замыкания:

```
List fruit = [ "apple", "Orange", "Avocado", "pear", "cherry" ]

fruit.sort { String a, String b -> a.compareToIgnoreCase(b) }

println "Sorted fruit: ${fruit}"
```

У вас может появиться вопрос, откуда взялся этот странный метод `sort()`, – не волнуйтесь, в языке Groovy он добавляется в реализацию класса `List` посредством библиотеки Groovy JDK, которая будет кратко представлена в разделе 2.4.3.

В этом примере мы не только объявили замыкание, но и передали его в качестве аргумента методу `sort()`. Видите, как похоже замыкание на тело метода без имени? Основное отличие заключается в том, что аргументы объявляются *внутри* фигурных скобок.

Прежде чем двинуться дальше, необходимо сделать два замечания к предыдущему примеру. Во-первых, метод `sort()` вызывается без круглых скобок. Такой способ вызова типичен для методов, принимающих единственный аргумент, который является при этом замыканием. Во-вторых, замыкание использует возможность неявного возврата результата, что помогает сохранить компактность программного кода.

Стандартный синтаксис замыканий имеет вид:

```
{ <аргументы> -> <тело> }
```

где `<аргументы>` – это список типизированных или нетипизированных аргументов замыкания, а `<тело>` – это эквивалент тела обычного метода. Замыкание, не имеющее аргументов, может определяться без объявления списка аргументов перед стрелкой: `{-> ... }`. Кроме того, имеется особый случай, о котором вам следует знать; пример ниже демонстрирует его:

```
[ "apple", "pear", "cherry" ].each { println it }
```

Определение, в котором отсутствует оператор `->`, создает замыкание с одним неявным аргументом, к которому можно обратиться с помощью специальной переменной `it`. Так, в предыдущем примере метод `each()` вызывает замыкание для каждого элемента списка и передает

ему текущий элемент списка в виде аргумента. Благодаря этому в переменной `it` оказывается текущий элемент списка (название фрукта), и производится вывод каждого элемента списка на консоль.

Что в имени тебе моем?

Термин *замыкание* имеет в информатике особый смысл, и в прошлом не раз велись споры, должны ли замыкания в языке Groovy иметь имена. Главное, о чем следует помнить, – они не простые блоки программного кода; они сохраняют ссылки на переменные и методы, находящиеся за пределами своих определений. Попробуйте запустить в консоли следующий простой пример:

```
def n = 14
def msg = "'n' is ${->n}"
println msg
n = 10
println msg
```

Второй вызов `println` выведет сообщение со значением переменной `n`, равным `10`, потому что в строку встроено замыкание, хранящее ссылку на переменную, а не ее значение. Нечто похожее можно наблюдать при использовании анонимных классов, но они обладают доступом только к переменным, объявленным со спецификатором `final`.

Вы видели, как замыкания объявляются в точке их использования, в качестве аргумента метода, но мы говорили, что они могут также присваиваться переменным и передаваться из одного фрагмента программы в другой как своеобразные указатели на функции. В листинге 2.6 представлена модифицированная версия предыдущего примера реализации сортировки на языке Groovy, где демонстрируется, как можно использовать ссылки на замыкания.

Листинг 2.6. Присваивание замыкания переменной

```
Closure comparator = { String a, String b -> ← Присваивание замыкания переменной
    a.compareToIgnoreCase(b)
}

List fruit = [ "apple", "Orange", "Avocado", "pear", "cherry" ]

fruit.sort(comparator) ← ❶ Ссылка на замыкание используется как аргумент

println "Sorted fruit: ${fruit}"

assert comparator("banana", "Lemon") < 0 ← ❷ Прямой вызов замыкания
```

Передача замыкания методу по ссылке выглядит, как обычный вызов метода ❶. Мы включили в пример также прямой вызов замыкания, как если бы оно было методом ❷. Такая форма записи не кажется неожиданной, если рассматривать замыкания как анонимные функции. Когда выполняется присваивание замыкания переменной, ему присваивается имя, то есть оно фактически превращается в именованную функцию.

Мы рассмотрели достаточно, чтобы вы могли следовать за большинством примеров в этой книге, где используются замыкания. Но чтобы чувствовать себя по-настоящему уверенно при работе с ними в контексте платформы Grails, необходимо знать чуть больше.

Во-первых, замыкания с одним аргументом являются уникальными: их можно вызывать с или без аргумента; например, `myClosure(1)` или `myClosure()`. В последнем случае замыканию передается значение `null`.

Во-вторых, на специальный синтаксис, использовавшийся в предыдущих примерах в этом разделе, имеется простое ограничение. Рассмотрим следующий пример, в котором вычисляется сумма элементов списка. Не стоит беспокоиться, от вас пока не требуется понимать, как он работает; в настоящий момент нас интересует только синтаксис.

```
List list = [ 1, 3, 5, 6 ]
assert 15 == list.inject(0) { runningTotal, value -> runningTotal + value }
```

Замыкание объявлено *после* закрывающей скобки в вызове метода, но при этом оно все равно передается методу `inject()` в виде второго аргумента. Это один из синтаксических трюков, которые в языке Groovy используются для повышения удобочитаемости программного кода. Вызов метода можно было бы записать как

```
list.inject(0, { runningTotal, value -> runningTotal + value })
```

но такая форма записи не работает с длинными, многострочными замыканиями. Что же это за ограничение, о котором упоминалось выше? Первую форму можно использовать, только когда замыкание передается методу в последнем аргументе, тогда как последняя форма записи может применяться независимо от положения замыкания в списке аргументов.

И последнее, самое важное замечание: замыкания – это не методы! Способ, которым они используются, часто напоминает вызовы методов, но фактически они являются объектами, тогда как полноправные методы в языке Groovy компилируются в обычные методы Java. Во многих случаях данное отличие является несущественным, но вы должны понимать, что стандартные правила переопределения методов неприменимы к замыканиям. Например, при расширении такого класса

```
Class SuperClass {
    Closure action = {
        ...
    }
}
```

мы не рекомендуем объявлять свойство `action` в своем подклассе. Если вы собираетесь использовать принцип полиморфизма, используйте методы. Наследование – не единственная область, где использование замыканий может порождать проблемы. Вы можете быть незнакомы с аспектно-ориентированным программированием (АОП), но его популярность продолжает расти в последние несколько лет. Многие библиотеки и инструменты АОП работают с методами, поэтому, если вы начнете повсеместно применять замыкания, вы потеряете возможность интеграции с инструментами АОП.

Мы надеемся, вы будете расценивать эти слова только как предостережение, а не как повод вообще отказаться от использования замыканий. Они мощны и позволяют получать более простые решения некоторых проблем. Но, как и в случае любой новой игрушки, есть опасность использования их не по назначению.

2.4.2. Динамическое программирование

Название «динамический язык программирования» звучит довольно интригующе, не правда ли? В нем присутствует слово «динамический». Но что означает этот термин?

На протяжении всей главы мы исходили из предположения, что вы знакомы с языком Java. В языке Java все переменные, свойства, аргументы методов и возвращаемые значения должны объявляться с обязательным указанием типа данных. Это позволяет компилятору выполнять проверку типов и устранять потенциальные ошибки еще до запуска приложения.

Теперь представьте, что имеется класс `Person` со свойствами `givenName`, `familyName`, `age` и `country`. В языке Java мы легко можем отсортировать список объектов класса `Person` по любому из этих свойств, например:

```
public void sortPeopleByGivenName(List<Person> personList) {
    Collections.sort(personList, new Comparator<Person>() {
        public int compare(Person p1, Person p2) {
            return p1.getFamilyName().compareTo(p2.getFamilyName());
        }
    });
}
```

Но как быть, если заранее не известно, по какому из свойств потребуется отсортировать список на этапе выполнения программы? Это может случиться, например когда пользователю предоставляется возможность выбрать свойство, по которому требуется отсортировать. В языке Java подобное обстоятельство неожиданно начинает осложнять нашу жизнь. Вам придется либо писать для каждого свойства отдельный метод сортировки, аналогичный приведенному в примере, и каким-то образом связывать получившиеся методы с пользовательским интерфейсом, либо использовать механизм рефлексии в функции сравнения.

Первый вариант означает, что придется дублировать программный код, выполняющий сравнение, что не только утомительно, но и чревато ошибками. Для реализации второго варианта придется писать настолько противный программный код, что мы предпочитаем избавить вас от неприятности видеть его.

Лучше давайте посмотрим, как можно реализовать решение данной задачи на языке Groovy, воспользовавшись нетипизированными переменными и возможностью *динамического вызова*. Как и требуется, этот метод выполняет сортировку списка людей по заданному свойству:

```
def sortPeople(people, property) {
    people.sort { p1, p2 -> p1."${property}" <=> p2."${property}" }
}
```

Мы бросили вас в пучину довольно сложного примера, но верьте нам – вы научитесь плавать в нем в мгновение ока! Разобьем этот пример на части, начав с сигнатуры метода. Она выглядит так же, как сигнатура любого другого метода, не считая наличия в ней ключевого слова `def` и отсутствия имен типов аргументов. Ключевое слово `def` позволяет определять как нетипизированные переменные, так и нетипизированные методы; то есть переменные, объявленные с помощью ключевого слова `def`, могут принимать значения любых типов. Вы можете в одной инструкции присвоить такой переменной строку, а в другой – целое число. В случае методов это означает, что метод может возвращать значения любых типов или вообще ничего не возвращать. Фактически ключевое слово `def` подобно использованию `java.lang.Object`.

Чтобы объявить нетипизированный метод и аргументы замыкания, не требуется даже ключевое слово `def`. Отбросьте объявление типа, и они будут интерпретироваться как нетипизированные. Несмотря на почти (если не совершенно) уникальную особенность языка Groovy, отличающую его от других динамических языков программирования и заключающуюся в поддержке возможности объявления типов, большинство разработчиков быстро привыкают к использованию нетипизированных переменных, аргументов и методов, потому что программный код становится более компактным и наглядным.

В замыкании, передаваемом методу `sort()`, можно заметить странную форму записи, в которой присутствуют объекты типа `GString`. Первое, что важно уяснить, – в языке Groovy допускается использование строки в качестве имен свойств и методов. Чтобы понять, что мы имеем в виду, взгляните на следующий пример, где обе строки совершенно идентичны по своему назначению:

```
peopleList.sort()
peopleList."sort"()
```

Форма записи в виде строк особенно удобна при работе с классами-компоновщиками Groovy, такими как класс создания разметки XML, который будет представлен в разделе 2.4.4, но в данном случае она по-

зволяет определить имя свойства или метода во время выполнения, что очень важно. Так, если аргумент `property` будет содержать строку `age`, тело замыкания в примере метода `sortPeople()` приобретет вид

```
p1."age" <=> p2."age"
```

то есть метод выполнит сортировку списка по свойству `age`. Тот же программный код может использоваться для сортировки по любому свойству класса `Person`. В действительности можно заметить, что метод вообще никак не зависит от класса `Person`, то есть этот метод может использоваться для сортировки списков любых объектов, при условии что они обладают заданным свойством!

Вы только что видели, какой силой обладают нетипизированные переменные в комбинации с возможностью динамического вызова. Этот шаблон программирования имеет собственное название.

Утиная типизация

Никакое обсуждение динамического языка программирования не может считаться полным без упоминания механизма *утиной типизации*. Своим происхождением этот термин обязан фразе: «Если это ходит, как утка, и крякает, как утка, скорее всего, это утка». Применительно к динамическим языкам программирования данная фраза означает, что если объект имеет определенное свойство или сигнатуру метода, тип объекта становится неважным – вы все равно имеете возможность вызвать этот метод или обратиться к этому свойству.

Такое возможно только в динамических языках программирования, потому что разрешение имен свойств и методов выполняется *на этапе выполнения*. Обсуждаемая особенность позволяет также добавлять методы и свойства в класс без изменения самого класса. Именно таким способом сам язык Groovy расширяет некоторые классы JDK – некоторые примеры дополнительных методов и свойств, реализованных библиотекой Groovy JDK, можно найти в разделе 2.4.3.

Таким образом, всякий раз, когда вам попадается ссылка на метод или свойство, которое не имеет объявления, есть вероятность, что они были внедрены в класс платформой Grails. Не волнуйтесь, мы будем упоминать о таких элементах при встрече с ними в наших примерах, и теперь вы знаете, откуда они возьмутся!

Внимание – опасность!

Помните, что большая власть
влечет большую ответственность.

Дядюшка Бен (Uncle Ben)
«Человек-паук», 2002

При использовании языка Groovy первое время возникает ощущение, что с вас сняли смирительную рубашку. Он дает большую свободу про-

граммирования и обеспечивает более простую возможность выражения своих идей в программном коде. Но в этой силе кроется опасность, о которой вы должны знать.

Во-первых, компилятор не в состоянии выявить несоответствия в типах данных или определить отсутствие свойств и методов, потому что их наличие определяется на этапе выполнения. Вероятно, на ранних стадиях изучения языка Groovy вы очень близко познакомитесь с исключениями `MissingPropertyException` и `MissingMethodException`. Позднее вы будете удивляться, насколько редко они возникают, однако вам следует с самого начала взять в привычку создание тестов для вашего кода, о которых мы подробно поговорим в главе 7.

Во-вторых, приложения на языке Groovy более сложны в отладке, особенно при наличии динамически создаваемых свойств и методов. Использование функции отладчика «войти в» («step into»), вероятно, будет слишком утомительным, поэтому старайтесь вместо нее использовать точки останова и функцию «выполнить до позиции курсора» («run to cursor»).

Наконец, чрезмерное увлечение динамическими особенностями и даже нетипизированными переменными может помешать пониманию программного кода. В любом случае очень важно использовать описательные имена, а при динамическом добавлении свойств и методов обязательно документируйте, какие свойства и методы добавляются и что они делают. Кроме того, по возможности объявляйте типы аргументов методов и замыканий: это не только упростит использование ваших методов другими людьми, но и позволит средам разработки подсказывать о потенциальных ошибках в вызывающем программном коде.

При разумном использовании динамических свойств и методов вы сможете увеличить свою производительность за счет расширения классов в мирном направлении. Пожалуй, лучшим примером такого подхода может служить библиотека Groovy JDK, которая представляет собой набор расширений базовых классов JDK.

2.4.3. Groovy JDK

Приходилось ли вам когда-либо расстраиваться из-за отсутствия какого-либо метода в JDK или задаваться вопросом, почему необходимо использовать вспомогательные классы, такие как `java.util.Collections`? Язык Groovy стремится восполнить любые недостатки, имеющиеся в JDK, и сделать программный код более однородным и объектно-ориентированным, чем Java, посредством Groovy JDK, набора расширений для некоторых из наиболее часто используемых классов JDK. Мы не в состоянии описать все дополнительные свойства и методы, потому что их слишком много, но мы рассмотрим некоторые наиболее полезные из них. Полный их перечень вы найдете в электронном справоч-

ном руководстве по Groovy JDK по адресу: *http://groovy.codehaus.org/groovy-jdk/*.

Пожалуй, самые полезные дополнения имеют отношение к коллекциям. Большинство из них касаются массивов и строк, последние из которых обычно интерпретируются как списки символов. Кроме того, существует несколько замечательных расширений для `java.io.File`. В заключение мы расскажем о некоторых усовершенствованиях.

Метод size() для коллекций/массивов/строк

Для обеспечения непротиворечивости Groovy реализует метод `size()` для массивов и строк. По своему действию он соответствует свойству массивов `length` и методу `length()` класса `String`. Вам не нужно больше запоминать, какое свойство или метод следует использовать с тем или иным типом данных!

Метод each(closure) для коллекций/массивов/строк

В некоторых примерах этой главы вы уже видели метод `each()` в действии. Он выполняет обход всех элементов коллекции, применяя заданное замыкание к каждому из них. Не забывайте, что строки интерпретируются как последовательности символов, поэтому следующий код

```
"Test".each { println it }
```

выведет на консоль:

```
T
e
s
t
```

Метод find(closure) для коллекций/массивов/строк

Начав однажды пользоваться методом `find()` (и родственным ему `findAll()`), вы станете удивляться, как вам удавалось обходиться без него. Он делает в точности то, что говорит его имя: отыскивает элемент коллекции. Точнее, он возвращает первый элемент, для которого заданное замыкание возвращает значение `true`. Если искомый элемент отсутствует в коллекции, метод вернет `null`. Замыкание в данном случае действует, как критерий поиска.

В следующем примере из списка извлекается первый объект, свойство `firstName` которого имеет значение «Glen»:

```
def glen = personList.find { it.firstName == "Glen" }
```

Метод `find()` возвращает первый элемент, соответствующий критерию поиска, а метод `findAll()` возвращает список всех элементов, соответствующих критерию. Если не будет найдено ни одного соответствующего элемента, метод вернет пустой список, а не значение `null`.

Метод collect(closure) для коллекций/массивов/строк

Метод `collect()` является одним из тех методов, которые требуют некоторой привычки, но потом быстро становятся незаменимыми. В некоторых других языках программирования его задачи выполняет функция «map». Он выполняет обход коллекции, применяя заданное замыкание к каждому элементу и добавляя возвращаемое значение в новый список.

Например, допустим, что у нас имеется список строк, и нам требуется получить список длин этих строк. Сделать это можно, как показано ниже:

```
def names = [ "Glen", "Peter", "Alice", "Graham", "Fiona" ]
assert [ 4, 5, 5, 6, 5 ] == names.collect { it.size() }
```

Обратите внимание, что новый список имеет тот же размер и порядок следования элементов, что и оригинал.

Метод sort(closure) для коллекций/массивов

Метод `sort()` выполняет сортировку коллекций, используя заданное замыкание в качестве функции сравнения. Замыкание может принимать один аргумент, и тогда оно должно возвращать значение, допускающее возможность сравнивания, например строку; или два аргумента (подобно методу `compareTo()`), и тогда оно должно возвращать целое число, определяющее порядок следования двух значений.

Чтобы продемонстрировать все вышесказанное на практике, мы отсортируем список имен по их длине обоими способами. Сначала используем замыкание с одним аргументом:

```
def names = [ "Glen", "Peter", "Ann", "Graham", "Veronica" ]
def sortedNames = names.sort { it.size() }
assert [ "Ann", "Glen", "Peter", "Graham", "Veronica" ] == sortedNames
```

А затем – с двумя аргументами:

```
def names = [ "Glen", "Peter", "Ann", "Graham", "Veronica" ]
def sortedNames = names.sort { name1, name2 ->
    name1.size() <=> name2.size()
}
assert [ "Ann", "Glen", "Peter", "Graham", "Veronica" ] == sortedNames
```

В обоих случаях мы получаем один и тот же результат, поэтому первый вариант выглядит предпочтительнее. Более сложный способ сравнивания может потребовать применения второго.

Метод join(string) для коллекций/массивов

Метод `join()` создает строку за счет конкатенации всех элементов коллекции, используя заданную строку как разделитель. Если сами элементы не являются строками, перед операцией конкатенации вызывается метод `toString()`.

Одним из наиболее частых случаев использования этого метода является преобразование списка строк в «список» строк, разделенных запятыми:

```
def names = [ "Glen", "Peter", "Alice", "Fiona" ]
assert "Glen, Peter, Alice, Fiona" == names.join(", ")
```

Обратное действие можно выполнить с помощью стандартного метода `String.split(String)`, входящего в состав JDK.

File.text

При обращении к свойству `text` выполняются чтение заданного файла и возвращение его содержимого в виде строки.

File.size()

Метод `File.size()` возвращает размер файла в байтах. Он соответствует методу `File.length()`.

File.withWriter(closure)

Для потоков и файлов существует несколько различных методов `with...()`, но здесь мы рассмотрим только один из них. Описание остальных вы найдете в электронном справочном руководстве по Groovy JDK.

Метод `File.withWriter()` создает экземпляр класса `java.io.Writer` из файла и передает его замыканию. Как только замыкание завершит свою работу, использовавшийся поток вывода будет закрыт автоматически и безопасно (не будет возбуждаться никаких исключений). Теперь, чтобы выполнить простую запись данных в файл, вам больше не потребуется использовать конструкцию `try...catch`!

Matcher.count

Свойство `count` возвращает количество совпадений, обнаруженных данным объектом класса `Matcher`. Оно никак не связано с количеством групп в соответствующем шаблоне.

Number.abs()

Наконец-то, вы сможете вызывать метод `abs()` относительно самих чисел!

Number.times(closure)

Метод `times()` вызывает заданное замыкание *n* число раз, где *n* – это число, относительно которого вызывается метод. Обратите внимание, что применять этот метод имеет смысл только относительно целых чисел. Замыканию передается текущий порядковый номер итерации, начиная с 0.

Мы показали лишь небольшую часть свойств и методов, имеющихся в Groovy JDK, но они являются наиболее типичными и широко исполь-

зуются в примерах этой книги. По мере обретения опыта работы с языком Groovy вы обязательно должны взять в привычку проверку наличия в Groovy JDK полезных свойств и методов у классов, с которыми имеете дело. Никогда не знаешь, какие драгоценности могут быть там скрыты!

Есть еще две особенности, с которыми мы хотели бы познакомить вас, прежде чем закончить главу. Они используются повсюду в платформе Grails, поэтому важно понимать, что они делают и как они действуют. Кроме того, они превращают работу с форматом XML в увлекательное занятие!

2.4.4. Простой способ создания и парсинга документов XML

Вселенная Java наполнена средствами для работы с форматом XML: библиотеки для создания данных в формате XML, библиотеки для парсинга формата XML и масса файлов с настройками в формате XML! Поскольку платформа Grails находится в этой вселенной, для вас будет отличной новостью тот факт, что Groovy сильно упрощает создание и парсинг документов XML. Поддержка XML присутствует в форме «классов конструирования разметки XML», используемых для создания документов XML, и «классов чтения разметки XML», используемых для их парсинга.

В листинге 2.7 демонстрируется использование обеих особенностей в одном примере. Пример сначала выполняет запись следующего документа XML в файл с помощью конструктора разметки:

```
<root>
  <item qty="10">
    <name>Orange</name>
    <type>Fruit</type>
  </item>
  <item qty="6">
    <name>Apple</name>
    <type>Fruit</type>
  </item>
  <item qty="2">
    <name>Chair</name>
    <type>Furniture</type>
  </item>
</root>
```

А затем он выполняет парсинг этого файла с помощью класса чтения. В конце примера демонстрируются извлечение различных элементов информации из документа XML и вывод их на консоль.

Листинг 2.7. Действия с документом XML средствами языка Groovy

```
import groovy.xml.MarkupBuilder
import groovy.util.XmlSlurper
```

```
def file = new File("test.xml")
def objs = [
    [ quantity: 10, name: "Orange", type: "Fruit" ],
    [ quantity: 6, name: "Apple", type: "Fruit" ],
    [ quantity: 2, name: "Chair", type: "Furniture" ] ]
def b = new MarkupBuilder(new FileWriter(file))
b.root {                                    ← Создание документа XML
    objs.each { o ->
        item(qty: o.quantity) {             ← ❶ Создание
            name(o.name)                        элементов XML
            type(o.type)
        }
    }
}

def xml = new XmlSlurper().parse(file)
assert xml.item.size() == 3                 ← ❷ Навигация по документу XML
assert xml.item[0].name == "Orange"             в памяти
assert xml.item[0].@qty == "10"
println "Fruits: ${xml.item.findAll {it.type == 'Fruit'}*.name }"
println "Total: ${xml.item.@qty.list().sum {it.toInteger()} }"
```

Приходилось ли вам когда-либо видеть столь компактный код, создающий документ XML и извлекающий информацию из него? Начнем с программного кода, создающего документ XML. Все, что мы делаем, – создаем новый экземпляр класса `MarkupBuilder` и вызываем его метод `root()`. Сам класс не имеет метода с именем «root», поэтому, прежде чем возбудить исключение, он пытается интерпретировать вызов метода как начало нового элемента XML. Как можно было бы предположить, в качестве названия элемента XML принимается имя метода.

История о двух конструкторах разметки

Самое интересное, что класс `MarkupBuilder` – не единственный класс, позволяющий конструировать документы XML. Кроме него в языке Groovy присутствует также класс `StreamingMarkupBuilder`. Внешне он похож на использованный нами класс конструирования разметки, но его реализация отличается в корне.

С точки зрения пользователя, класс `StreamingMarkupBuilder` обладает более широкими возможностями, но он не предусматривает возможности оформления отступов (весь документ XML располагается в одной строке). За дополнительной информацией обращайтесь к электронной документации языка Groovy.

Вложенные элементы реализуются с помощью замыканий ❶, где замыканием является последний аргумент соответствующего метода. В листинге 2.7 `item()`, `name()` и `type()` – неизвестные методы, поэтому

они превращаются в элементы XML. Атрибуты объявляются как именованные аргументы методов, тогда как текстовое содержимое передается как неименованный строковый аргумент. Все это чисто технические аспекты, но вы должны понять основную идею, изучив программный код примера.

Следует отметить, что вы можете смешивать обычный программный код с разметкой, как сделано в цикле `each`. Это обеспечивает компактность программного кода, создающего объемные документы XML. Как только вы привыкнете использовать такую форму разметки, вы поймете, насколько неудобно и сложно было работать с документами XML раньше. Мы вас предупредили!

На следующем этапе выполняется парсинг созданного документа XML с помощью класса `XmlSlurper`. При таком подходе вы получаете представление документа XML в памяти, из которого можно извлекать информацию, применяя форму записи обращений к свойствам ❷. После того как класс `XmlSlurper` выполнит парсинг документа XML, он вернет представление корневого элемента документа в виде объекта. В действительности такой объект является экземпляром класса `GPathResult`, но это не имеет большого значения. Нас больше интересует его поведение.

Как видно из листинга 2.7, к элементам документа XML можно обращаться как к свойствам с теми же именами. При необходимости доступ к вложенным элементам может осуществляться как к вложенным свойствам использованием оператора точки (`.`). Так, чтобы получить элемент `name`, необходимо сначала обратиться к свойству `item`:

```
def names = xml.item.name
```

При взгляде на этот небольшой пример возникает вопрос: что возвращают элементы `item` и `name`?

Опираясь на листинг 2.7, можно было бы подумать, что свойства возвращают списки. В определенном смысле такое предположение верно, потому что они обладают некоторыми чертами списков, однако они не реализуют интерфейс `List`. Это важно понять. Вы можете вызывать методы `size()` и `findAll()` объекта результата, но действительный список можно получить только при вызове специализированного метода `list()`.

И последнее замечание: чтобы получить значения атрибутов, необходимо лишь добавить перед именем свойства символ `@`.

Вот и все – мы закончили краткий обзор возможностей языка Groovy работы с документами XML. Мы включили этот обзор главным образом потому, что документы XML широко используются в платформе Grails, однако надеемся, что мы вас заинтересовали и вы будете применять эти возможности не только в проектах на платформе Grails!

Хотелось бы надеяться, что данное обсуждение Groovy и XML продемонстрировало, как замыкания и динамические методы могут взаимодействовать друг с другом для обеспечения новых, удобных решений типич-

ных проблем. Даже если некоторые концепции показались вам не вполне ясными, не волнуйтесь. На данном этапе от вас не требуется их глубокое понимание, а в следующих главах вы увидите еще массу примеров, которые помогут вам осмыслить описанные особенности.

2.5. В заключение

Мы рассмотрели особенности языка Groovy в достаточно быстром темпе, поэтому давайте приостановимся и вспомним все, о чем рассказывалось. После первых разделов и примеров этой главы вы должны были почувствовать себя достаточно уверенно и обрести интерес к программированию на языке Groovy. Даже если вы пока что-то не поняли, не волнуйтесь. В данной главе приводится больше информации, чем необходимо, чтобы приступить к работе с платформой Grails. По мере продвижения по остальным главам книги возвращайтесь сюда всякий раз, когда в примерах вам будут встречаться неясные конструкции.

Для начала вам достаточно уметь распознавать синтаксические конструкции, уникальные для языка Groovy. Позднее понимание динамической природы Groovy поможет вам избежать многих ловушек. Мы полагаем, что вскоре выразительность и мощь языка Groovy начнут приносить вам удовольствие. Только не осуждайте нас, если у вас пропадет желание программировать на простом языке Java!

Ниже приводятся несколько важных положений, которые вы должны вынести из этой главы:

- *Используйте особенности языка Groovy*. Синтаксические особенности языка Groovy способны сделать вашу жизнь легче, а программный код – выразительнее, поэтому используйте их. Вы обнаружите также, насколько проще следовать за примерами программного кода Groovy, когда привыкнете использовать различные идиомы Groovy, отличающие его от языка Java. После этой главы во всех примерах программного кода будут использоваться идиомы Groovy, а не языка Java.
- *Экспериментируйте*. Командная оболочка и консоль платформы Grails позволяют вам экспериментировать с программным кодом Groovy. Это отличный способ изучения языка и выяснения границ своих возможностей.
- *Используйте методы везде, где это уместно*. Несмотря на то что замыкания являются мощным инструментом, не забывайте о методах. Пусть методы нельзя передавать в виде аргументов или интерпретировать их как объекты, но они обеспечивают более тесную интеграцию с языком Java и полностью соответствуют идеологии наследования.
- *Используйте явные определения типов в сигнатурах методов*. Весьма заманчиво повсеместно использовать нетипизированные пере-

менные и аргументы, но не стоит пренебрегать возможностью явного определения типов аргументов и возвращаемых значений методов. Они не только позволяют документировать, что ожидает получить метод и что он возвращает, но могут также использоваться интегрированной средой разработки для выявления случаев неправильного их применения. И в случае вызова метода с аргументами неправильного типа сообщение об ошибке во время выполнения позволит быстро выявить причины, ее породившие.

С другой стороны, когда имеет место явная выгода от поддержки методом принципа грубого определения типа или когда метод способен работать с определенным аргументом, который может содержать данные разных типов, есть смысл использовать один или более нетипизированных аргументов. Только не забывайте при этом указывать в комментариях, что ожидает получить метод в нетипизированных аргументах.

Прежде чем перейти к изучению основ Grails, вспомните, что Groovy – это не просто язык программирования, созданный для платформы Grails. Он может использоваться для решения широкого круга задач: для создания сценариев, для поддержки сценариев в приложениях на языке Java, для быстрого моделирования и многого другого! Более полный обзор языка вы найдете в книге «Groovy in Action»[1], написанной Дирком Кёнигом (Dierk Köenig) в соавторстве с Эндрю Гловером (Andrew Glover), Полом Кингом (Paul King), Гильомом Лафоржем (Guillaume Laforge) и Ионом Скитом (Jon Skeet).

Вы увидели, как быстро можно создать приложение с нуля. Теперь вы должны уметь без страха писать программный код на языке Groovy. А сейчас самое время перейти к изучению фундаментальных концепций платформы Grails, и начнем мы с классов доменов.

[1] Издательство Символ-Плюс планирует выпустить перевод этой книги осенью 2010 года.

II

Основы Grails

В первой части книги мы кратко познакомили вас с основными компонентами платформы Grails и с особенностями языка Groovy, лежащего в ее основе. Во второй части мы приступаем к более детальному исследованию трех базисных компонентов экосистемы Grails: моделей, контроллеров и представлений.

В главе 3 мы рассмотрим моделирование предметной области – сердце любого приложения на платформе Grails. Вы узнаете, как сохранять и изменять классы предметной области, исследуете множество способов проверки данных и познакомитесь со всеми основными видами отношений между классами предметной области (1:1, 1:m, m:n).

В главе 4 вы обретете навыки моделирования, исследовав многочисленные механизмы выполнения запросов, которые предлагаются платформой Grails для поиска вашей модели предметной области. Мы также поближе познакомимся с фантастическими возможностями механизма скаффолдинга в платформе Grails, которые позволяют создавать функциональные пользовательские интерфейсы в рекордные сроки.

К тому времени, когда мы перейдем к главе 5, вы будете готовы приступить к исследованию веб-ориентированных особенностей платформы Grails. В частности вы узнаете, как с помощью контроллеров платформы Grails управлять перемещением пользователя по страницам приложения. Кроме того, мы рассмотрим вопросы привязки данных из веб-форм, напишем фильтр запроса и даже создадим собственный механизм отображения адресов URL на более дружественные постоянные ссылки в вашем приложении.

В главе 6 мы обратим свое внимание на компоненты пользовательского интерфейса приложения на платформе Grails, исследуем теги Grails, используемые для конструирования пользовательского интерфейса. Мы покажем, как быстро придать приложению целостный и привлека-

тельный внешний вид и даже как создавать сменные темы оформления для приложения. В заключение мы расскажем о поддержке технологии AJAX в платформе Grails и покажем, как можно добавить некоторые анимационные эффекты в приложение.

В главе 7 мы научим вас создавать надежные тесты для разрабатываемого программного кода, с помощью которых вы сможете, прежде чем распространять свой программный код, убедиться в том, что все действует безупречно. Сначала мы познакомимся с модульными тестами для всех основных составляющих платформы Grails, разработкой которых вы занимались до сих пор. После этого, опираясь на навыки избирательного тестирования, мы перейдем к применению более широких понятий интегрального и функционального тестирования. По окончании главы 7 тестирование прочно займет свое место в арсенале ваших привычек.

Изучив эту часть книги, вы будете иметь четкое представление обо всех основах платформы Grails и прочно встанете на путь становления продуктивного разработчика приложений на платформе Grails. В третьей части мы познакомим вас с более специфическими особенностями платформы Grails, которые сделают ваши приложения по-настоящему готовыми к работе.

В этой главе:

- Что такое GORM и как он действует
- Определение классов модели предметной области
- Как сохранять и обновлять классы предметной области
- Приемы проверки и обеспечения целостности полей
- Отношения между классами предметной области (1:1, 1:m, m:n)

3

Моделирование предметной области

В этой главе мы рассмотрим поддержку моделей данных в приложениях на платформе Grails, и если вы волнуетесь, что мы будем погружаться слишком глубоко, то вы будете приятно удивлены. Мы не напишем ни строчки на языке SQL и не собираемся заниматься настройкой отображений Hibernate XML. Мы воспользуемся преимуществами парадигмы преимущества соглашений перед настройками, а, значит, меньше времени будем уделять настройкам и больше достижению главной цели.

Большую часть времени мы будем исследовать основы организации сохранения классов предметной области в платформе Grails с использованием GORM – реализации механизма объектно-реляционного отображения в платформе Grails (Grails Object Relational Mapping). Вы также познакомитесь с различными видами отношений между моделями GORM («один ко многим», «многие ко многим» и так далее).

Но мы – практики, а не теоретики, поэтому все эти темы мы будем обсуждать в процессе создания основы приложения Hubbub, которое будем использовать в качестве примера на протяжении всей книги. В этой главе мы не будем уделять очень много времени созданию пользовательского интерфейса (UI), однако те концепции, что будут здесь рассматриваться, являются фундаментальными для построения надежных моделей данных, служащих опорой ваших приложений.

Посмотрим на наше приложение без лишней суеты.

3.1. Hubbub: приступаем к созданию приложения

Наша цель в этой книге – довести вас до уровня продуктивного разработчика приложений на платформе Grails. Мы передадим вам все навыки, необходимые для создания приложений мирового уровня в ре-

кордно короткие сроки. И сделаем мы это, задействовав вас в разработке настоящего приложения.

На протяжении всей оставшейся части книги в качестве примера мы будем использовать приложение Hubbub – простое приложение для микроблогинга в стиле Twitter. Вы можете представлять его как систему, позволяющую писать однострочные сообщения о том, чем вы занимаетесь прямо сейчас. Друзья могут следить за вашими сообщениями, в частности, за тем, как вы набираетесь опыта, и вдохновляться на проверку некоторых своих идей. Вы тоже можете следить за сообщениями своих друзей. На рис. 3.1 показана окончательная версия приложения Hubbub в действии.

Совет

Если ранее вам не приходилось пользоваться системой Twitter, сейчас самое время посетить сайт *twitter.com* и посмотреть, о чем речь. Если вам нужны друзья, за сообщениями которых вы могли бы следить, загляните к пользователям @glen_a_smith и @pledbrook. Мы часто говорим на профессиональном жаргоне, но обычно мы безобидны.

Рис. 3.1. *Приложение Hubbub, к которому мы направляемся*

Базовая модель предметной области приложения Hubbub достаточно проста. На рис. 3.2 модель «сущность-связь» (Entity Relationship, ER) приложения Hubbub показана во всей своей красе.

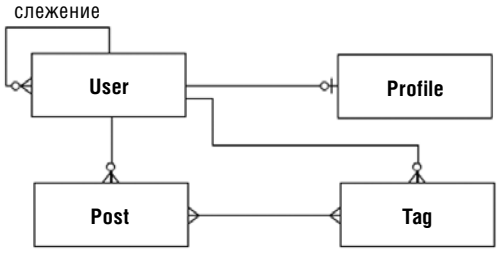


Рис. 3.2. *Базовая модель данных приложения Hubbub демонстрирует основные типы отношений*

Класс `User` хранит информацию, идентифицирующую пользователя (идентификатор пользователя и пароль). Все хорошие приложения социальных сетей позволяют ассоциировать массу информации с каждым пользователем (например, адрес электронной почты, адрес блога, часовой пояс, имя любимой рок-звезды), поэтому мы включили в модель класс `Profile` (который связан отношением 1:1 – каждому экземпляру класса `User` в соответствие ставится один экземпляр класса `Profile`, а каждому экземпляру класса `Profile` соответствует один и только один экземпляр класса `User`).

Одна из задач приложения Hubbub заключается в том, чтобы дать пользователям возможность создавать сообщения – однострочные записи в блоге, описывающие, над чем прямо сейчас они работают. Пользователь может написать множество сообщений, но каждый экземпляр класса `Post` будет иметь единственного автора, то есть здесь наблюдается классическое отношение 1:m.

Но разве может приложение, написанное в стиле Web 2.0, не иметь поддержки тегов? Каждый раз, когда пользователь создает сообщение (экземпляр класса `Post`), он может отметить его несколькими тегами – каждому экземпляру класса `Post` может соответствовать множество тегов. Обратное утверждение также является истинным. Каждый тег (экземпляр класса `Tag`) может соответствовать нескольким сообщениям. Налицо отношение m:n («многие ко многим»). Кроме того, мы собираемся связать отношениями экземпляры классов `Tag` и `User` (потому что было бы весьма удобно иметь возможность увидеть все теги, имеющиеся у пользователя, без необходимости отыскивать их во всех сообщениях).

Напоследок мы припасли самую хитрую часть модели: объект `User` имеет несколько ссылок на самого себя. Пользователь может следить за множеством других пользователей (мы назвали такое отношение «слежением»). Похоже, что мы исчерпали требования к реализации, причем все оказалось достаточно просто.

Не надо пока волноваться о том, чтобы уложить все сказанное в своей голове. Мы уделим еще массу времени этим классам на протяжении

следующих нескольких глав, и со временем все встанет на свои места. Вам нужно только получить общее представление о назначении основных объектов, и можно двигаться дальше.

3.1.1. Архитектура, управляемая предметной областью

Если вы столь же нетерпеливы, как и мы, то, наверно, уже задались вопросом, почему мы начали с проектирования предметной области. Почему мы не начали с чего-то более увлекательного, например с реализации функции автодополнения на основе технологии Ajax? Не волнуйтесь, мы еще этим займемся.

Grails – интерактивная платформа для гибкой разработки. Это означает, что начать можно с чего угодно: с рефакторинга, с внесения изменений – и все равно в результате получится фантастическое приложение. Можно начать с создания UI (user interface) – пользовательского интерфейса, с моделирования предметной области, с классов служб и даже с испытательных тестов, если вам этого захочется.

Но в ходе работы над созданием приложения на платформе Grails первое, что мы обычно делаем, – создаем эскиз дизайна страницы на обычном листе бумаги в клеточку, как показано на рис. 3.3. Это дает нам возможность понять, как пользователь будет взаимодействовать с системой, и получить некоторое представление о том, как будет выглядеть конечный продукт. Это позволяет нам создать образ будущего приложения у себя в голове и способствует рождению идей, касающихся действий пользователя.

Рис. 3.3. *Предварительный эскиз страницы UI приложения Hubbub*

Но UI – только часть истории. Закончив разработку эскизов UI, мы переходим к определению модели предметной области, описывающей взаимосвязи между данными, хранимыми в системе. Это позволяет получить полное представление о том, как будут взаимодействовать основ-

ные компоненты системы, и помогает подробно изложить наши идеи. Платформа Grails настолько упрощает процесс моделирования, что мы обычно сразу же воплощаем модель данных в программном коде, минуя этап моделирования на бумаге. По этой причине данную главу мы начали именно с моделирования: создание модели предметной области – отличная отправная точка при разработке приложений на платформе Grails.

После того как мы определили модель данных, мы создаем черновой рабочий вариант UI, применяя механизм скаффолдинга в платформе Grails (бесценную возможность автоматического создания UI мы представили в главе 1, а в этой главе мы познакомимся с ней еще ближе). Получив автоматически сгенерированный UI, можно почувствовать, что дело сдвинулось – у нас имеется простейшее приложение, способное сохранять информацию в базе данных. Таким образом, у нас появляется стимул перейти к следующему этапу развития функциональности приложения и приступить к воплощению эскизов на миллиметровке в действующий UI.

Возможно, вы более дисциплинированы, и вам не нужен дополнительный стимул для продолжения исследования, но вы столкнулись с нами в этой главе, поэтому давайте вместе доведем приложение Hubbub до того момента, когда мы сможем увидеть его в окне броузера.

3.1.2. Запускаем Hubbub: от 0 до первой загрузки

У нас имеется набросок внешнего вида страницы на бумаге, и у нас есть примерная модель данных, поэтому теперь можно попробовать запустить наше приложение. Давайте создадим его:

```
grails create-app hubbub
```

Мы считаем хорошим признаком выполнение команды `cd hubbub`, а затем `grails run-app` для запуска вновь созданного приложения. Введите в адресной строке броузера адрес `http://localhost:8080/hubbub/`, чтобы увидеть работающее приложение, как показано на рис. 3.4.

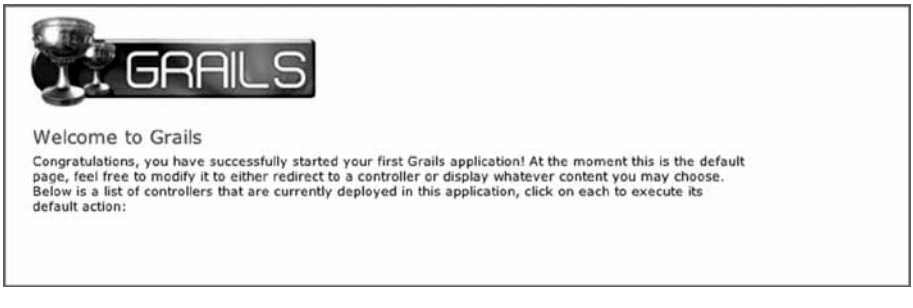


Рис. 3.4. *Только что созданное приложение Hubbub*

Итак, мы создали обертку приложения Hubbub, и теперь настало время нарастить мясо на костях. Давайте сгенерируем наш первый класс предметной области.

3.1.3. Введение в GORM (механизм объектно-реляционного отображения в платформе Grails)

Прежде чем сгенерировать наш первый класс предметной области, давайте сначала коротко познакомимся с механизмом GORM, реализацией механизма объектно-реляционного отображения в платформе Grails.

Объектно-реляционное отображение (Object-Relational Mapping, ORM) представляет собой процесс сохранения и извлечения объектов из реляционной базы данных. Это означает, что вы можете (почти) не задумываться о запросах SQL, которые выполняются за кулисами, и продолжать программирование. Например, когда выполняется инструкция `user.firstName = "Glen"`, механизм ORM выполняет инструкцию `UPDATE` языка SQL, чтобы обеспечить изменение объекта, хранимого в базе данных. В приложениях на языке Java эта роль обычно возлагается на такие ORM, как Hibernate или JPA; в Grails эту работу выполняет механизм GORM, который использует преимущества динамической типизации языка Groovy для максимального упрощения доступа к данным.

Если вам уже приходилось использовать Hibernate, TopLink или другую библиотеку реализации ORM в языке Java, вы знаете, что на их настройку требуется тратить немало времени. Часто приходится создавать множество файлов XML с описанием правил отображения. Механизм GORM, как и большинство компонентов платформы Grails, использует парадигму преимущества соглашений перед настройками, чтобы обеспечить работоспособность без единой строчки XML.

Теперь, когда вы немного знакомы с механизмом GORM, можно определить наш первый объект модели предметной области и понаблюдать за ним в действии.

3.2. Первый объект класса предметной области

Предварительную модель мы выработали в начале этого раздела, и у нас имеется оболочка приложения, поэтому теперь можно определить классы модели данных. В первую очередь нам нужно определить объект `User`, чтобы наши пользователи могли зарегистрироваться и приступить к работе с системой.

Сначала мы попросим платформу Grails создать заготовку класса предметной области:

```
grails create-domain-class com.grailsinaction.User
```

Эта команда создаст новый файл с определением класса, /grails-app/domain/com/grailsinaction/User.groovy, (и соответствующий модульный

тест в файле /test/unit/com/grailsinaction/UserTests.groovy). Предпочтительнее сохранять классы в пакетах, а не в области видимости по умолчанию, поэтому мы будем сохранять все файлы с исходными текстами в пакете с именем com.grailsinaction.

Теперь настало время задуматься о полях, которые было бы желательно определить в новых учетных записях класса User. Нам не хотелось бы, чтобы процесс регистрации новых пользователей был слишком утомительным, но нам необходимо будет получить от пользователей некоторые основные сведения о них. В листинге 3.1 приводится определение нашего первого базового объекта User.

Листинг 3.1. Наш первый базовый объект User

```
package com.grailsinaction

class User {

    String userId
    String password
    String homepage
    Date dateCreated

}
```

Типы данных, которые можно использовать в классах предметной области

В нашем объекте User мы пока использовали поля типа String и Date, но вообще можно использовать более широкий диапазон типов: Integer, Long, Float, Boolean (и соответствующие им элементарные типы), Date, Calendar, URL и byte[] – все эти типы входят в перечень поддерживаемых. Кроме того, платформа Grails обеспечивает разумный выбор соответствующих им типов полей в базе данных для отображения структуры сохраняемых данных. Полный перечень поддерживаемых типов вы найдете в описании библиотеки Hibernate.

Платформа Grails обеспечивает специальную поддержку полей типа Date, имеющих имена dateCreated и lastUpdated. Если у вас имеются поля с такими именами, платформа Grails автоматически будет записывать в поле dateCreated текущие дату и время при первом сохранении объекта, а в поле lastUpdated – текущие дату и время при каждом сохранении. Мы использовали поле dateCreated, чтобы сохранить время регистрации пользователя.

Теперь мы можем сохранять информацию о пользователе. У нас пока нет UI для ввода каких-либо данных, зато у нас имеется заготовка ис-

пытательного теста, которая была автоматически создана платформой Grails. Это может с самого начала вызвать чувство вины, поэтому давайте облегчим нашу совесть.

3.2.1. Сохранение и извлечение информации о пользователях с помощью тестов

Впервые идея автоматизированного тестирования была представлена в главе 1, когда мы создавали тесты для службы `QuoteService`. Однако тесты полезно использовать для проверки любых компонентов приложения. Настолько полезно, что мы выделили отдельную главу (главу 7) для обсуждения стратегий тестирования на всех этапах цикла разработки.

Однако сейчас тесты дадут нам возможность показать вам, как механизм GORM сохраняет объекты в базе данных и как извлечь их обратно. Тестирование является отличным способом подправить что-нибудь с самого начала. Давайте напишем интеграционный испытательный тест.

Как уже отмечалось, платформа Grails создала заготовку модульного теста в файле /test/unit/com/grailsinaction/UserTest.groovy. Но нам необходим интеграционный тест, поскольку мы хотим проверить взаимодействие с базой данных. Вспомните, в главе 1 говорилось, что интеграционные тесты создаются командой:

```
grails create-integration-test com.grailsinaction.UserIntegration
```

Модульные или интеграционные тесты?

При создании любых компонентов из командной строки платформа Grails автоматически генерирует соответствующий модульный тест в файле /grails-app/test/unit/YourArtifactTest.groovy. Модульные тесты работают изолированно и опираются на имитацию операций (о которой мы поговорим в главе 7), то есть от вас потребуется сымитировать работу базы данных или любой другой необходимой инфраструктуры.

В этой главе мы будем работать с интеграционными тестами. Для интеграционного теста платформа Grails подготавливает базу данных и собирает все компоненты, воссоздавая окружение действующего приложения. Это означает, что мы можем моделировать создание, сохранение и удаление объектов предметной области, и при этом нам не придется задумываться о хитростях имитации. Интеграционные тесты выполняются немного медленнее, но они фантастически хороши для обучения и проведения экспериментов, то есть для того, чем мы будем заниматься в этой главе.

Она создаст файл /test/integration/com/grailsinaction/UserIntegrationTests.groovy.

Сначала мы создадим объект `User` (для пользователя joe) и сохраним его в базе данных. А затем посмотрим, возможно ли обратиться к базе данных и отыскать пользователя по идентификатору. В листинге 3.2 представлен первый тест, выполняющий сохранение.

Листинг 3.2. Сохранение объекта предметной области в базе данных и извлечение его оттуда

```
package com.grailsinaction

import grails.test.*

class UserIntegrationTests extends GrailsUnitTestCase {

  void testFirstSaveEver() {

    def user = new User(userId: 'joe', password: 'secret',
      homepage: 'http://www.grailsinaction.com')
    assertNotNull user.save() <-- ❶ Проверить успешность вызова save()
    assertNotNull user.id <-- ❷ Проверить, установил ли метод save() значение id

    def foundUser = User.get(user.id) <-- ❸ Извлечь объект User по значению ID
    assertEquals 'joe', foundUser.userId

  }

}
```

Процесс создания нового экземпляра объекта предметной области обычно включает в себя конструирование объекта и последующий вызов метода `save()` ❶. Когда происходит вызов метода `save()`, механизм GORM генерирует запрос SQL, вставляющий наш объект `User` в базу данных. Механизм GORM возвращает сохраненный объект `User` (или значение `null`, если метод `save()` потерпел неудачу, о чем мы поговорим позднее). После сохранения объекта `User` в базе данных его свойству `id` присваивается значение одноименного поля в базе данных ❷. Мы можем затем использовать это значение `id` в вызове метода `get()` ❸ для извлечения объекта (можно использовать также метод `read()`, если желательно получить копию объекта, доступную только для чтения).

Помимо методов `get()` и `read()` существуют более привлекательные способы извлечения объекта из базы данных, и мы расскажем о них, когда перейдем к знакомству с *динамическими методами поиска* в следующей главе, а пока нам вполне подойдет метод `get()`.

Теперь самое время проверить работоспособность нашего испытательного теста. Давайте попросим платформу Grails выполнить наш тест:

```
grails test-app -integration
```

При желании запустить оба теста, модульный и интеграционный, можно использовать команду `grails test-app`, но сейчас нас интересует только интеграционный тест. В процессе выполнения теста в консоли появится масса информации, но где-то в середине вы обнаружите хорошие известия, которые мы ждали:

```
-------------------------------------------------------
Running 1 Integration Test...
Running test com.grailsinaction.UserIntegrationTests...
                  testFirstSaveEver...SUCCESS
Integration Tests Completed in 1772ms
-------------------------------------------------------
```

И мы все в зелени (так говорят, когда тест прошел благополучно, потому что в большинстве интегрированных сред разработки пройденные тесты сопровождаются отображением строк зеленого цвета). Слово «SUCCESS» (успешно) сообщает, что все инструкции `assert` выполнены благополучно, как мы и ожидали.

Что делает метод save() за кулисами?

За кулисами метод `save()` использует сеанс Hibernate, который в текущем потоке выполнения открывает платформа Spring, и добавляет наш объект `User` в этот сеанс. На языке Hibernate это означает, что временный объект `User` становится постоянным.

Фактическая запись в базу данных (выполнение запроса SQL, осуществляющего вставку) из сеанса Hibernate происходит при завершении потока выполнения, однако имеется возможность принудительно выполнить фактическую запись, для этого нужно явно вызвать с аргументом `user.save(flush: true)`. Мы еще вернемся к этому.

Подробнее о сохранении объектов мы поговорим в главе 13.

3.2.2. Изменение свойств пользователя

Мы сохранили наш первый объект, теперь попробуем реализовать процедуру изменения. Изменение объекта – это особый случай сохранения, давайте попробуем изменить пароль пользователя joe программным способом.

Примечание

Мы будем должны создавать учетную запись пользователя «joe» при каждом запуске цикла тестирования, потому что интеграционные тесты всегда возвращают базу данных к состоянию, предшествовавшему тестированию. Наши из-

менения выполняются в рамках транзакции, которая откатывается обратно в конце каждого теста, чтобы обеспечить восстановление прежнего состояния базы данных.

Как и в предыдущем тесте, мы начнем со знакомых методов `save()` и `get()`, затем модифицируем некоторые поля объекта и повторно вызовем методы `save()` и `get()`, чтобы убедиться, что процедура изменения работает. В листинге 3.3 приводится реализация теста сохранить-и-изменить.

Листинг 3.3. Изменение информации о пользователе за счет изменения значений полей с последующим вызовом save()

```
void testSaveAndUpdate() {

  def user = new User(userId: 'joe', password: 'secret',
    homepage: 'http://www.grailsinaction.com')
  assertNotNull user.save()

  def foundUser = User.get(user.id)
  foundUser.password = 'sesame'  <-- ❶ Модифицировать извлеченный объект User
  foundUser.save()               <-- ❷ Сохранить изменения в базе данных

  def editedUser = User.get(user.id)
  assertEquals 'sesame', editedUser.password <-- ❸ Проверить, сохранился ли пароль

}
```

Здесь использована та же последовательность вызовов методов `save()` и `get()`, что и в предыдущем тесте. Но обратите внимание: достаточно лишь изменить значения свойств ❶ и вызвать метод `save()` ❷, чтобы сохранить эти изменения в базе данных. После внесения изменений мы повторно извлекаем объект из базы данных, чтобы убедиться, что пароль действительно изменился ❸.

Давайте подтвердим работоспособность процедуры изменения, запустив еще раз команду `grails test-app -integration`:

```
-------------------------------------------------------
Running 2 Integration Tests...
Running test com.grailsinaction.UserIntegrationTests...
                  testFirstSaveEver...SUCCESS
                  testSaveAndUpdate...SUCCESS
Integration Tests Completed in 1939ms
-------------------------------------------------------
```

3.2.3. Удаление пользователей

Теперь мы неплохо представляем, как выполняются загрузка и сохранение. Но назойливые боты очень быстро наполнят нашу базу данных

регистрационной информацией ложных пользователей, поэтому нам также требуется иметь возможность удалять объекты User.

Настало время создать третий, и последний, испытательный тест. В листинге 3.4 демонстрируется, как выполняется удаление объектов из базы данных с помощью метода delete().

Листинг 3.4. Удаление объектов из базы данных выполняется одной строкой

```
void testSaveThenDelete() {

  def user = new User(userId: 'joe', password: 'secret',
    homepage: 'http://www.grailsinaction.com')
  assertNotNull user.save()

  def foundUser = User.get(user.id)
  foundUser.delete() ← ❶ Удалить объект из базы данных

  assertFalse User.exists(foundUser.id) ← ❷ Проверить наличие объекта
                                          с заданным значением ID в базе данных
}
```

Реализация процедуры удаления дает нам возможность познакомиться с двумя новыми методами классов предметной области: delete() и exists(). Вы можете вызвать метод delete() для любого класса предметной области, извлеченного из базы данных ❶. После удаления объекта из базы данных его экземпляр в памяти не исчезает, благодаря чему мы можем обратиться к свойству foundUser.id в последующем вызове метода exists() даже несмотря на то, что объект foundUser был удален из базы данных.

Проверить существование любого объекта предметной области можно с помощью метода exists() ❷. Метод exists(), как и следовало ожидать, возвращает значение true, если объект с указанным значением идентификатора присутствует в базе данных.

Теперь мы знаем, как выполнять сохранение, изменение и удаление объектов User. И хотя мы проверили работоспособность метода save(), мы еще не сталкивались с ситуациями, когда вызов метода save() терпел бы неудачу. Основная причина подобной неудачи заключается в нарушении ограничений целостности одним из полей класса предметной области. Итак, пора познакомить вас с возможностями выполнения проверок, предоставляемыми платформой Grails.

3.3. Проверка: барьер на пути мусора

Мы создали новый объект User и благополучно протестировали операцию его сохранения в базе данных, но, возможно, вы уже немного задумались и о защите: «Как помешать клиентам записывать всякий мусор (включая пустые значения) в свойства объектов и сохранять их?» От-

вет – пока никак. Для нас это повод поговорить о проверке правильности данных.

Платформа Grails предоставляет свой способ упростить типичные проверки, а когда существующие возможности проверки не совсем точно соответствуют вашим потребностям, их несложно настроить. Допустим, нам необходимо гарантировать, что все пароли будут состоять не менее чем из шести символов и не более чем из восьми. Такого рода ограничение можно наложить с помощью специального замыкания `constraints`, в котором определения ограничений записываются на предметно-ориентированном языке (Domain Specific Language, DSL). Существуют валидаторы, ограничивающие размеры полей, не допускающие наличие пустых полей, выполняющие проверку (с помощью шаблонов) на наличие в поле адреса URL, адреса электронной почты или номера кредитной карты, и так далее.

Давайте добавим в наш объект `User` несколько простых ограничений. Нам необходимо установить пределы для размеров полей `userId` и `password` и убедиться, что поле `homepage` содержит допустимый адрес URL. В листинге 3.5 приводится доработанный класс предметной области с новыми ограничениями.

Листинг 3.5. Платформа Grails упрощает наложение ограничений

```
package com.grailsinaction

class User {

    String userId
    String password
    String homepage
    Date dateCreated

    static constraints = {
        userId(size:3..20, unique: true)   ❶ Задает мин. и макс. длину поля
        password(size: 6..8)
        homepage(url: true, nullable: true)   ❷ Сравнивает URL с шаблоном
    }                                            и допускает, что поле может
                                                 быть пустым
}
```

Ограничение `size` ❶ гарантирует, что поле `userId` будет иметь размер от 3 до 20 символов. В случае применения к полю типа `String` ограничение `size` проверяет длину строки. А при применении к числовому полю проверяется попадание числа в указанный диапазон. Например, на поле типа `Integer` с именем `quantity` можно наложить ограничение, гарантирующее, что пользователь не сможет заказать более 10 экземпляров, записав такое определение: `quantity (size: 0..10)`. Мы наложили также ограничение `unique`, гарантировав тем самым невозможность существования двух пользователей с одинаковыми значениями `userId`.

Нет необходимости перечислять все поля в блоке `constraints` – достаточно указать только те, к которым необходимо применить какие-либо проверки. Однако следует отметить, что все поля по умолчанию не допускают возможность хранения пустых значений, поэтому если поле может оставаться пустым, необходимо явно определить ограничение `nullable`. Мы допускаем, что поле с адресом URL домашней страницы пользователя ❷ может быть пустым (`nullable`), но если оно заполнено, мы проверяем соответствие шаблону URL. Такая возможность комбинирования ограничений позволяет описывать проверки кратко и выразительно.

Что произойдет, если пользователь попытается сохранить объект, который не удовлетворяет установленным ограничениям? Давайте напишем испытательный тест и посмотрим. Сейчас самое время познакомиться с методом `validate()`, имеющимся в распоряжении всех классов предметной области. Когда производится вызов метода `validate()`, платформа Grails проверяет соответствие объекта установленным ограничениям и предоставляет объект `errors`, с помощью которого можно выяснить, какие поля нарушают ограничения.

Листинг 3.6 добавляет в наш файл UserIntegrationTests.groovy еще один тест, который пытается сохранить экземпляр, не удовлетворяющий установленным ограничениям.

Листинг 3.6. Получение результатов неудачной проверки

```
void testEvilSave() {
    def user = new User(userId: 'chuck_norris',
        password: 'tiny', homepage: 'not-a-url')

    assertFalse user.validate() <-- ❶ Вернет false
    assertTrue user.hasErrors()

    def errors = user.errors <-- ❷ errors хранит список ошибочных значений

    assertEquals "size.toosmall",
        errors.getFieldError("password").code <-- ❸ Код, описывающий ошибку
    assertEquals "tiny",
        errors.getFieldError("password").rejectedValue <-- ❹ Ошибочное значение

    assertEquals "url.invalid",
        errors.getFieldError("homepage").code
    assertEquals "not-a-url",
        errors.getFieldError("homepage").rejectedValue

    assertNull errors.getFieldError("userId") <-- ❺ Проверка отсутствия поля
                                                  в списке полей с ошибками
}
```

Как уже говорилось выше, метод `validate()` ❶ проверяет, соблюдаются ли все ограничения, установленные для класса предметной области,

и возвращает `true` или `false`. В результате очень часто в котроллерах платформы Grails можно встретить такой прием:

```
if (user.validate()) {
    user.save()
} else {
    // дать пользователю еще одну попытку... или
    user.discard()
}
```

После выполнения проверки можно обратиться к свойству `errors` объекта предметной области, чтобы определить, где была допущена ошибка ❷. Возвращаемый объект `errors` содержит коллекцию объектов `fieldError`, каждый из которых представляет отдельное поле объекта предметной области. Каждый объект `fieldError` хранит в свойстве `code` ❸ код, описывающий тип проверки, которая потерпела неудачу, а в свойстве `rejectedValue` ❹ данные, введенные пользователем. Если поле удовлетворяет ограничениям, для него будет возвращаться пустая ссылка на объект `fieldError`, как это произошло в случае с полем `userId` ❺.

Если вам интересно узнать коды ошибок, мы перечислим всех их в следующем разделе (в табл. 3.1). А пока достаточно знать, что мы можем точно определить причины нарушения ограничений. В главе 7 мы покажем, как выполнять все эти проверки в модульных тестах, которые делают их еще более краткими.

Теперь, когда мы знаем, как вызвать ошибку (нарушив ограничения), напишем испытательный тест, который будет устранять ошибки после неудачной попытки сохранения. Это будет не совсем то, что обычно приходится делать при обработке веб-запросов, однако позволит нам продемонстрировать работу валидаторов. В листинге 3.7 приводится испытательный тест, который сначала пытается вызвать метод `save()` для сохранения недопустимых данных, а затем устраняет ошибки и повторно сохраняет данные вызовом метода `save()`.

Листинг 3.7. Исправление ошибок после неудачной проверки

```
void testEvilSaveCorrected() {
  def user = new User(userId: 'chuck_norris',
    password: 'tiny', homepage: 'not-a-url') ← ❶ Недопустимые URL и пароль
  assertFalse(user.validate())
  assertTrue(user.hasErrors()) ← ❷ Вернет true
  assertNull user.save()

  user.password = "fistfist"
  user.homepage = "http://www.chucknorrisfacts.com"
  assertTrue(user.validate())
  assertFalse(user.hasErrors()) ← ❸ Ошибки были устранены
  assertNotNull user.save()

}
```

Наш оригинальный объект `User` содержит недопустимые адрес URL и пароль ❶, что вызывает неудачу при проверке объекта ❷. После исправления ошибок метод `validate()` благополучно проверяет соответствие ограничениям и сбрасывает объект `errors` ❸.

Итак, мы реализовали ограничения и получили некоторую уверенность в том, что поля базы данных будут хранить допустимые данные. К настоящему моменту мы рассмотрели только ограничения, проверяющие размер и допустимость адресов URL, однако в платформе Grails существует множество других валидаторов, которые будут описаны далее.

3.3.1. Стандартные валидаторы

Теперь, когда вы знаете, как действует механизм ограничений, можно ознакомиться со списком валидаторов, реализованных в платформе Grails по умолчанию. Всего существует достаточно много валидаторов, наиболее часто используемые из них перечислены в табл. 3.1.

Таблица 3.1. Платформа Grails по умолчанию реализует множество валидаторов

Имя	Описание	Пример	Свойство объекта errors
`blank`	Проверяет, не пустая ли строка (или не содержит ли поле значение `null`)	`password(blank:false)`	`blank`
`email`	Проверяет допустимость адреса электронной почты	`userEmail(email:true)`	`email.invalid`
`inList`	Проверяет присутствие значения в заданной коллекции или ее диапазоне	`country(inList:['Australia', 'England'])`	`not.inList`
`matches`	Проверяет совпадение поля с заданным регулярным выражением	`userId(matches: '[0-9]{7} [A-Za-z]')`	`matches.invalid`
`maxSize`	Ограничивает максимальный размер поля в базе данных	`orderQuantity(maxSize:100)`	`maxSize.exceeded`
`minSize`	Ограничивает минимальный размер поля в базе данных	`orderQuantity(minSize:10)`	`minSize.notmet`
`nullable`	Определяет допустимость значения `null` в поле	`password(nullable: false)`	`nullable`

size	Определяет минимальную и максимальную длину строки, границы целочисленного значения или размер коллекции	userId(size:3..20)	size.toosmall или size.toobig
unique	Определяет, должно ли значение быть уникальным	userId(unique:true)	unique
url	Проверяет допустимость адреса URL	homepage(url:true)	url.invalid
validator	Позволяет определить собственную функцию проверки в виде замыкания	Описывается в разделе 3.3.3	validator.error

Полный перечень валидаторов, доступных в платформе Grails, вы найдете в справочной документации, по адресу: *http://grails.org/doc/1.1/*.

Пустое значение не равно null?

Возможно, вы заметили, что в табл. 3.1 присутствуют отдельные валидаторы `nullable` и `blank`. Это важно, поскольку когда вы посылаете форму HTML с пустыми полями, они представляются платформой Grails как «пустые» (blank) поля, удовлетворяющие ограничению `nullable:true`. Запомните, если вам требуется, чтобы пользователь всегда вводил какое-либо значение в поле, используйте ограничение `blank:false`. Если поле не является обязательным и может не заполняться, используйте ограничение `nullable:true`.

3.3.2. Реализация собственных проверок с помощью регулярных выражений

Безусловно, ваши правила проверки могут быть иными. И тогда вам потребуется реализовывать собственные проверки.

Если проверка может быть выполнена с помощью регулярного выражения, тогда ее можно реализовать с применением ограничения `matches`. Допустим, вы пишете систему учета студентов для своего университета, причем все идентификаторы, присваиваемые студентам, состоят из семи цифр, за которыми следует алфавитный символ. Тогда ограничение можно реализовать с помощью простого регулярного выражения:

```
static constraints = {
    userId(matches: '[0-9]{7}[A-Za-z]')
}
```

Регулярные выражения – очень мощный инструмент. Но иногда одних регулярных выражений бывает недостаточно.

3.3.3. Специальные приемы проверки полей

Регулярные выражения способны помочь во многих случаях, но временами может возникать необходимость наложить ограничение, в котором участвуют сразу несколько полей. Например, рассмотрим типичное правило, когда значение поля `password` не должно совпадать со значением `userId`. В такого рода ситуациях приходится пользоваться ограничением `validator`. Это не самое простое ограничение, но оно таит в себе огромные возможности!

Когда определяется ограничение `validator`, ему передается замыкание, принимающее один или два аргумента. В первом аргументе передается значение, которое пользователь пытается записать в поле, а во втором, если вы определили его, передается ссылка на сам экземпляр класса предметной области. Если проверяемые данные являются допустимыми, замыкание должно вернуть значение `true`.

В нашем случае нам потребуется определить замыкание с двумя аргументами, потому что нам необходимо сравнить значение поля `password` со значением поля `userId`:

```
static constraints = {
  password(size: 6..8,
    validator: { passwd, user ->
      return passwd != user.userId
    })
  homepage(url: true, nullable: true)
}
```

Теперь осуществляется более сложная проверка. Когда выполняется сохранение класса предметной области, валидаторы поля `password` проверяют, содержит ли пароль от 6 до 8 символов включительно и не совпадает ли пароль с идентификатором пользователя `userId`. С помощью собственных валидаторов вы можете организовывать такие проверки, какие вам только заблагорассудится, потому что они позволят вам программно проверить все, что угодно.

Совет

Некоторые из ограничений (такие, как `size`, `maxSize` и `nullable`) оказывают прямое влияние на то, как платформа Grails будет создавать поля в базе данных. Например, если указано ограничение `maxSize` со значением 8, платформа Grails создаст в базе данных поле размером в 8 символов. Дополнитель-

ную информацию о том, как некоторые ограничения влияют на создание полей базы данных, вы найдете в справочном руководстве.

3.4. Определение модели данных – 1:1, 1:m, m:n

Теперь мы знаем, как выполняются операции CRUD, как применять проверки к полям наших классов доменов и даже как быстро генерировать заготовку UI. Но для работы приложения Hubbub требуется намного больше, чем простой класс `User`, поэтому теперь мы поговорим об отношениях в модели данных.

Использование механизма ORM еще не означает, что вам придется искать компромиссные решения при создании своих классов доменов. Платформа Grails обладает гибкостью, позволяющей вам определять необходимые отношения: «один к одному» (1:1), «один ко многим» (1:m) или «многие ко многим» (m:n). Более того, механизм GORM заботится о создании таблиц с соответствующей структурой, используя соглашения об именах.

3.4.1. Отношения «один к одному»

Сначала смоделируем отношение «один к одному». Это, пожалуй, самый простой для понимания вид отношений.

Пришло время связать в нашем приложении Hubbub поля, идентифицирующие пользователя (идентификатор, пароль, время последней регистрации), и информацию, хранящуюся в учетной записи (адрес домашней страницы, адрес электронной почты, фотографию и все остальное). Мы движемся в направлении реализации нашей оригинальной модели данных для приложения Hubbub (изображенной на рис. 3.2), которая включает в себя объект `Profile`. Соответствующий раздел модели данных изображен на рис. 3.5.

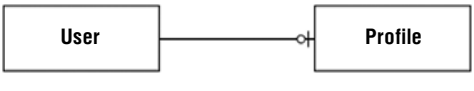


Рис. 3.5. *Каждому экземпляру класса User может соответствовать необязательный экземпляр класса Profile*

Начнем с создания класса предметной области `Profile`:

```
grails create-domain-class com.grailsinaction.Profile
```

Затем дополним вновь созданный класс свойствами, определяющими связь объекта `Profile` с соответствующим ему объектом `User`. Мы переместим сюда поле `homepage` и добавим новые поля для хранения адре-

са электронной почты и даже фотографии. Модифицированная версия объекта `Profile` показана в листинге 3.8.

Листинг 3.8. Модифицированный объект Profile, связанный отношением 1:1 с объектом User

```
package com.grailsinaction

class Profile {

  static belongsTo = User ← ❶ Объект Profile принадлежит объекту User

  byte[] photo ← ❷ Двоичные данные типа byte[]

  String fullName
  String bio
  String homepage
  String email
  String timezone
  String country
  String jabberAddress

  static constraints = {
    fullName(nullable: true)
    bio(nullable: true, maxSize: 1000)
    homepage(url: true, nullable: true)
    email(email: true, nullable: true)
    photo(nullable: true)
    country(nullable: true)
    timezone(nullable: true)
    jabberAddress(email: true, nullable: true)
  }

}
```

Самое очевидное новое свойство в данном классе предметной области – поле `belongsTo` ❶. Данное поле сообщает механизму GORM, что этот объект связан отношением с объектом `User`. Он также сообщает механизму GORM, как следует выполнять каскадные операции изменения и удаления (обсуждаются во врезке).

В листинге 3.8 полю `belongsTo` присваивается класс владельца, что означает однонаправленное отношение. Вы можете получить доступ к объекту `Profile` через объект `User`, но обратной ссылки из объекта `Profile` на объект `User` здесь нет. Существует форма определения поля `belongsTo`, которая позволяет создавать двунаправленную связь, но об этом мы поговорим в разделе 3.4.2, когда будем рассматривать отношения 1:m.

Мы добавили в объект `Profile` несколько новых полей и ограничений. Среди них: поле для ввода полного имени пользователя, краткой биографии, страны проживания и часового пояса. Мы объявили необяза-

тельные поля для ввода адреса домашней страницы и адреса электронной почты и использовали стандартные валидаторы, чтобы обеспечить соответствие предъявляемым требованиям. Так как эти поля не обязательны для заполнения, мы пометили их как `nullable`. Jabber-адреса имеют такое же представление, как и адреса электронной почты, поэтому мы можем применить валидатор `email` и к этому полю.

Нам необходимо также обеспечить возможность сохранения фотографии пользователя в виде поля BLOB (Binary Large Object – большой двоичный объект). В данном случае поле объявлено с типом `byte[]` (массив байтов), что предписывает механизму GORM сохранить его как BLOB ❷.

Поле BelongsTo и каскадные операции

Механизм GORM выполняет каскадные операции только для объектов, обладающих свойством `belongsTo`. В листинге 3.8 класс `Profile` имеет поле `belongsTo`, ссылающееся на класс `User`, поэтому в случае удаления какого-либо объекта `User` будет также удален соответствующий ему объект `Profile`. В случае отсутствия поля `belongsTo` соответствующий объект `Profile` не будет удален. Это обстоятельство имеет особую важность для отношений 1:m, когда желательно обеспечить полное удаление связанных данных.

Кроме того, поле `belongsTo` приобретает особый смысл в отношениях типа m:n, где методы `addTo*()` могут добавляться только на стороне владельца. Но подробнее об этом мы поговорим ниже.

Теперь, когда мы создали класс `Profile`, можно добавить ссылку на него в наш объект `User`. В листинге 3.9 мы создаем в классе `User` поле типа `Profile` и определяем несколько ограничений, регламентирующих работу отношения.

Листинг 3.9. Добавление отношения 1:1 в класс User

```
package com.grailsinaction

class User {

  String userId
  String password

  Date dateCreated
  Profile profile ← ❶ Объявление профиля пользователя

  static constraints = {
    userId(size:3..20, unique: true)
    password(size: 6..8, validator: { passwd, user ->
      passwd != user.userId
```

```
        })
        dateCreated()
        profile(nullable: true) ← ❷ Отмечает профиль, как необязательный элемент
    }

    static mapping = {
        profile lazy:false ← ❸ Предписывает платформе Grails загружать объект
    }                          Profile вместе с объектом User

}
```

Мы добавили в класс `User` несколько новых свойств. В первую очередь мы добавили в класс `User` поле типа `Profile`, чтобы платформа Grails знала, что это отношение 1:1 ❶. (Для описания отношения 1:m потребовалось бы это поле объявить как множество (или список) элементов типа `Profile`.)

Мы также добавили ограничение `nullable`, чтобы обеспечить необязательность профиля ❷. Если опустить это ограничение, платформа Grails будет принудительно создавать экземпляр класса `Profile` всякий раз, когда будет создаваться объект `User`, что слишком накладно и чего мы будем пока избегать.

В конце мы добавили в наш класс `User` новое замыкание `mapping`. Это замыкание позволяет настраивать различные аспекты работы с базой данных, включая кэширование, именование таблиц и полей и стратегии загрузки. Здесь мы определили, что объект `Profile` должен загружаться незамедлительно ❸, то есть всякий раз, когда платформа Grails будет загружать объект `User`, она будет формировать запрос так, что объект `Profile` всегда будет возвращаться в составе того же объекта `ResultSet`.

Не всегда бывает необходимо описывать профиль как `nullable` и требовать незамедлительной его загрузки. Во многих случаях этого не требуется, и для установления отношения 1:1 достаточно в родительском объекте объявить дочерний объект.

Стратегии немедленной и отложенной загрузки

По умолчанию для извлечения присоединенных коллекций механизм GORM использует стратегию отложенной загрузки, согласно которой извлечение из базы данных происходит только в момент непосредственного обращения. В большинстве ситуаций эта стратегия соответствует нашим требованиям. Но при использовании отношения 1:1, когда стратегия доступа должна обеспечить немедленный доступ к связанному объекту (как и в случае нашего объекта `Profile`), есть смысл заставить Hibernate извлекать объект `Profile` одновременно с соответствующим объектом `User`. Это стратегия *немедленного извлечения,* и в данной ситуации она повышает производительность.

При использовании отношения 1:1 в комбинации со стратегией немедленного извлечения можно прибегнуть к возможности композиции в платформе Grails. Она позволит внедрить объекты `Profile` в ту же таблицу, где хранятся объекты `User` (но при этом использует различные ссылки на объекты для работы с каждым из них). Подробнее об этой возможности мы поговорим в главе 13, когда будем рассматривать более передовые приемы использования механизма GORM.

Теперь, когда мы получили некоторый опыт определения отношений 1:1, можно перейти к моделированию более распространенных отношений «один ко многим» (1:m).

3.4.2. Отношения «один ко многим»

В нашем приложении Hubbub каждый пользователь может создавать множество *сообщений*, или *записей*, при этом каждое сообщение может принадлежать одному (и только одному) пользователю, как показано на рис. 3.6. Это классическое отношение «один ко многим» (1:m).

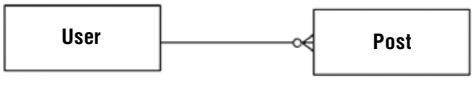


Рис. 3.6. *Каждому объекту User может соответствовать ноль или более объектов Post*

Мы сначала определим отношение, а затем посмотрим, какие операции можно применять к стороне отношения «ко многим».

Определение отношения «один ко многим»

Нам необходим новый класс предметной области `Post`, поэтому давайте создадим его:

```
grails create-domain-class com.grailsinaction.Post
```

Платформа Grails предоставляет два типа свойств классов предметной области для моделирования отношений: `hasMany` (на стороне «один») и `belongsTo` (на стороне «ко многим»). Сначала создадим объект `Post`, в котором достаточно определить поля `content` и `date`. Определение класса приводится в листинге 3.10.

Листинг 3.10. Объект Post, моделирующий все сообщения данного пользователя

```
package com.grailsinaction

class Post {
```

```
    String content
    Date dateCreated

    static constraints = {
        content(blank: false)
    }

    static belongsTo = [ user : User ] ← ❶ Ссылается на объект-владелец
}
```

Мы уже видели поле `belongsTo` ❶ в отношении 1:1 (листинг 3.8) и знаем, как оно влияет на каскадирование операций. В частности, при удалении объекта `User` будут удалены все соответствующие ему объекты `Post`.

Две формы поля belongsTo

Мы уже видели поле `belongsTo` в нашем классе `Profile` (листинг 3.8), где оно ссылалось на класс-владелец как `belongsTo = User`. В том случае создавалась однонаправленная связь – можно получить доступ к объекту класса `Profile` через объект класса `User`, но не наоборот.

В листинге 3.10 поле `belongsTo` определяется как отображение, тем самым создается двунаправленная связь между классами `User` и `Post`. Такое определение создает в классе `Post` новое поле с именем `user`, которое позволяет сослаться на объект-владелец `User`. Это дает нам возможность получить значение, например `post.user.userId`. Данную возможность мы будем использовать ниже, когда будем извлекать сообщения и отображать связанный с ними идентификатор пользователя.

Мы сообщили платформе Grails, что объект `Post` принадлежит объекту `User`, а теперь нам необходимо сообщить, что с объектом `User` может быть связано множество объектов `Post`. Делается это с помощью замыкания `hasMany`:

```
class User {

  // здесь находится уже существующий код

  static hasMany = [ posts : Post ]
}
```

Добавив определения `hasMany` и `belongsTo`, мы завершили создание простого отношения «один ко многим». Но как сообщить платформе Grails, что новые объекты `Post` должны ассоциироваться с заданным объектом `User`? Воспользовавшись волшебством механизма GORM.

После того как между классами `User` и `Post` будет установлено отношение «один ко многим», платформа Grails автоматически добавит к классу `User` два новых метода: `User.addToPosts()` и `User.removeFromPosts()`. Мы

создадим новый интеграционный тест для класса Post, с помощью которого сможем проверить новые возможности. Начнем с привычной уже команды:

```
grails create-integration-test com.grailsinaction.PostIntegration
```

Получив заготовку испытательного теста, добавим в нее программный код, создающий пользователя и добавляющий несколько новых сообщений для данной учетной записи. В листинге 3.11 мы используем новый метод addToPosts(), чтобы сделать нашего пользователя чуть более говорливым.

Листинг 3.11. Метод User.affToPosts() упрощает использование отношения 1:m

```
package com.grailsinaction
import grails.test.*

class PostIntegrationTests extends GrailsUnitTestCase {

    void testFirstPost() {

        def user = new User(userId: 'joe',
                            password: 'secret').save()
        def post1 = new Post(content: "First post... WOOt!")
        user.addToPosts(post1)
        def post2 = new Post(content: "Second post...")
        user.addToPosts(post2)
        def post3 = new Post(content: "Third post...")
        user.addToPosts(post3)
        assertEquals 3, User.get(user.id).posts.size()
    }
}
```

❶ Создать объект User, владеющий сообщениями

❷ Сохранить объект Post, добавив его с помощью объекта User

Обратите внимание, что мы должны вызвать метод save() объекта User, чтобы сохранить его в базе данных ❶. Как только объект User будет сохранен в базе данных, любые дополнения, которые мы будем делать относительно этого объекта (например, добавление новых объектов Post с помощью метода addToPosts() ❷), будут автоматически сохраняться в базе данных. По этой причине от нас не требуется вызывать метод save() для создаваемых нами объектов Post. Если вы скептически относитесь к нашим словам, перезапустите испытательные тесты, чтобы убедиться, что все работает, как и ожидается:

```
grails test-app -integration
```

Воспользовавшись некоторыми волшебными динамическими особенностями механизма GORM, мы добавили нового пользователя и несколько сообщений от его имени. Но как извлечь сообщения, когда это потребуется? Типичный подход заключается в том, чтобы получить требуемый объект User и выполнить итерации по его сообщениям. В ли-

стинге 3.12 представлен испытательный тест, выполняющий обращения ко всем сообщениям, принадлежащим заданному пользователю.

Листинг 3.12. Доступ к сообщениям пользователя посредством обхода графа объекта

```
void testAccessingPosts() {

    def user = new User(userId: 'joe', password: 'secret').save()
    user.addToPosts(new Post(content: "First"))
    user.addToPosts(new Post(content: "Second"))
    user.addToPosts(new Post(content: "Third"))

    def foundUser = User.get(user.id)
    def postNames =
        foundUser.posts.collect { it.content }

    assertEquals(['First', 'Second', 'Third'],
        postNames.sort())
}
```

Добавление сообщений пользователя

❶ Извлечь пользователя по идентификатору

❷ Итерации через все сообщения пользователя

❸ Сортировка сообщений в алфавитном порядке

В этом примере мы получаем информацию о пользователе из базы данных, пользуясь его идентификатором ❶, и затем с помощью метода `collect()` ❷ языка Groovy выполняем обход всех сообщений и извлекаем их содержимое. Метод `collect()` возвращает список значений свойств `content` объектов `Post`, которые мы затем сравниваем с известными значениями, чтобы убедиться, что были извлечены именно требуемые сообщения ❸. По умолчанию неизвестно, в каком порядке будут следовать элементы коллекций отношения 1:m (поскольку они интерпретируются как множества), поэтому в нашем испытательном тесте мы сортируем их в алфавитном порядке, чтобы сравнение имело смысл ❸.

Чтобы обеспечить вывод сообщений пользователя в хронологическом порядке, обычно приходится сортировать их по дате создания в порядке убывания, но сортировка вручную может очень быстро наскучить. В следующем разделе мы покажем способ, позволяющий получать уже отсортированный список сообщений.

Сортировка данных на стороне отношения «ко многим»

При использовании отношений «один ко многим» часто бывает необходимо сохранить определенный порядок следования данных на стороне «ко многим», таких как элементы счета. В подобных ситуациях есть смысл использовать механизм платформы Grails упорядочения по умолчанию. Когда требуется применить иной способ упорядочения, можно воспользоваться более мощными средствами поиска, имеющимися в платформе Grails (такими как прикладной интерфейс Criteria, о котором мы поговорим в главе 4), чтобы выполнить сортировку в нужный момент времени.

Но иногда бывает необходимо извлечь данные на стороне отношения «ко многим» в предопределенном порядке. Например, в приложении

блога бывает желательно обеспечить хронологический порядок следования записей (такой, чтобы на первой странице отображались последние записи). В этих ситуациях платформа Grails позволяет определить собственный механизм упорядочения с помощью замыкания `mapping` (которое мы уже использовали ранее в примере реализации класса `Profile` в листинге 3.9).

Чтобы реализовать такой способ сортировки, необходимо сообщить платформе Grails, что наши объекты `Post` должны возвращаться в упорядоченном виде, отсортированными по дате их создания. Это достигается путем добавления нового блока `mapping` в класс `Post`, как показано в листинге 3.13.

Листинг 3.13. Сортировка сообщений по дате создания

```
package com.grailsinaction

class Post {

    String content
    Date dateCreated

    static constraints = {
        content(blank: false)
    }

    static belongsTo = [ user : User ]

    static mapping = {
        sort dateCreated:"desc" ← Определяет порядок сортировки коллекции объектов Post
    }

}
```

Существует возможность определить порядок сортировки как по возрастанию, так и по убыванию. В этом примере все запросы объектов `Post` будут возвращать их отсортированными в порядке убывания.

Но что, если сообщения необходимо сортировать только при обращении к ним через объект `User` (например, в случае обхода их с помощью метода `user.posts.each`)? В таких ситуациях платформа Grails позволяет указывать порядок сортировки непосредственно в самом определении отношения, а не в объекте `Post`. Мы могли бы изменить класс `User` (а не класс `Post`), добавив в него следующий блок `mapping`:

```
static mapping = {
    profile lazy:false
    posts sort:'dateCreated'
}
```

Эта форма отображения предписывает платформе Grails производить сортировку по полю `dateCreated`, только когда обращение к коллекции

posts выполняется через объект User. Такая возможность была проблематична в версии Grails 1.1 для определенных случаев (смотрите описание проблемы GRAILS-4089 в Grails JIRA), но, скорее всего, она будет отлажена к тому времени, когда вы будете читать эти строки.

Теперь, когда мы познакомились с возможностью сортировки, можно перейти к изучению более сложного типа отношений: «многие ко многим».

3.4.3. Отношения «многие ко многим»

Возможно ли приложение социальных сетей в стиле Web 2.0 без тегов? Теги дают пользователям возможность группировать свои сообщения, просматривать сообщения, отмеченные определенными тегами, и вообще классифицировать свои сообщения. Давайте добавим в нашу модель предметной области возможность помечать сообщения тегами.

Сейчас самое время узнать, как можно было бы использовать теги. Вообразим, что у нас имеются следующие требования:

- Предоставить пользователям возможность создавать облака тегов на их домашних страницах
- Обеспечить создание лент RSS для передачи всех сообщений, помеченных заданным тегом
- Предоставить возможность просматривать все теги для заданного сообщения

Чтобы включить эти требования в нашу модель предметной области, нам необходимо смоделировать два отношения:

- Пользователь может создавать множество тегов, поэтому каждый объект Tag связан с одним объектом User (отношение «один ко многим» (1:m))
- Каждый объект Post может быть связан с множеством объектов Tag, и каждый объект Tag может быть связан с множеством объектов Post (отношение «многие ко многим» (m:n))

Представить, как это выглядит, не так-то просто, однако модель, изображенная на рис. 3.7, немного проясняет ситуацию.

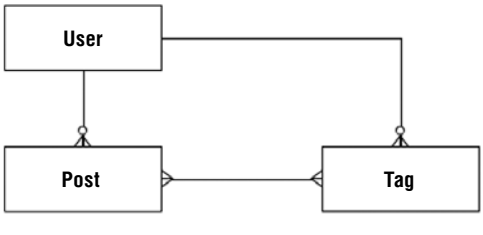


Рис. 3.7. *Модель отношения «многие ко многим» между объектами User, Post, и Tag*

У нас есть для вас хорошее известие, касающееся отношения «многие ко многим»: определение таких отношений содержит совсем мало новых для вас синтаксических конструкций. Если два объекта связаны отношением «многие ко многим», они оба должны содержать определение `hasMany`, указывающее друг на друга. В листинге 3.14 представлена измененная версия класса `Post`, в который добавлено новое отношение `hasMany` с классом `Tag`.

Листинг 3.14. Модель класса Post, который может иметь отношения со многими объектами Tag

```
class Post {

    String content
    Date dateCreated

    static constraints = {
        content(blank: false)
    }

    static belongsTo = [ user : User ]

    static mapping = {
        sort dateCreated:"desc"
    }

    static hasMany = [ tags : Tag ] ← Моделирует отношение класса Post
}                                     со многими объектами Tag
```

Мы уже встречались с определением `hasMany` при создании отношения «один ко многим», и здесь мы использовали то же определение. Отображение `[ tags : Tag ]` говорит о том, что объект `Post` может быть связан отношениями со многими объектами `Tag` и что отношение сохраняется в атрибуте с именем `tags`.

Теперь познакомимся с предметной моделью `Tag`, которая будет связана обратным отношением с объектом `Post`. В листинге 3.15 мы указали, что объект `Tag` может быть связан отношением `hasMany` с множеством объектов `Post`.

Листинг 3.15. Модель объекта Tag, связанного отношениями с объектами Post и User

```
class Tag {

    String name
    User user

    static constraints = {
        name(blank: false)
    }
```

```
    static hasMany = [ posts : Post ]

    static belongsTo = [ User, Post ] ← Указывает, каким сторонним объектам
                                        может принадлежать

}
```

В листинге 3.15 присутствует определение отношения `hasMany`, которое ссылается обратно на объект `Post`. Еще одно важное отличие этого класса состоит в том, что тег (`Tag`) принадлежит (`belongsTo`) обоим объектам – `User` и `Post`. Такое определение `belongsTo` имеет большое значение в контексте отношения «многие ко многим»: оно определяет перечень доступных методов `addTo*()` (дополнительная информация приводится во врезке).

Как определение belongsTo влияет на отношения «многие ко многим»

Поле `belongsTo` определяет, какие динамические методы `addTo*()` могут быть использованы на стороне владельца. В соответствии с определением в листинге 3.15 мы получим возможность вызывать метод `User.addToTags()`, потому что тег (`Tag`) принадлежит (`belongsTo`) объекту `User`. Мы также сможем вызывать метод `Post.addToTags()`, потому что тег (`Tag`) принадлежит (`belongsTo`) объекту `Post`. Но объект `Post` не принадлежит (не содержит определение `belongTo`) объекту `Tag`, поэтому мы не сможем вызывать метод `Tag.addToPosts()`.

Последнее изменение, которое нам необходимо внести, имеет отношение к объекту `User`, где нам требуется описать связь с классами `Post` и `Tag`. В листинге 3.16 приводится измененное определение `hasMany`.

Листинг 3.16. Пользователь может иметь множество сообщений и тегов

```
package com.grailsinaction

class User {

    // .. существующий код

    static hasMany = [ posts : Post, tags : Tag ] ← Определяет, что пользователь
                                                    может иметь множество
                                                    сообщений и тегов
}
```

Теперь мы описали в классе `User` связь с обоими классами `Post` и `Tag` с помощью определения `hasMany`. Собрав воедино все элементы отношения «многие ко многим», давайте создадим испытательный тест, чтобы под-

твердить правильность наших предположений. В листинге 3.17 приводится простой испытательный тест, позволяющий убедиться, что сообщение может быть помечено одним или более тегами.

Листинг 3.17. Комплексная проверка отношения «многие ко многим» между сообщениями и тегами

```
void testPostWithTags() {

    def user = new User(userId: 'joe', password: 'secret').save()

    def tagGroovy = new Tag(name: 'groovy')
    def tagGrails = new Tag(name: 'grails')
    user.addToTags(tagGroovy)
    user.addToTags(tagGrails)

    def tagNames = user.tags*.name
    assertEquals([ 'grails', 'groovy'] , tagNames.sort())

    def groovyPost = new Post(content: "A groovy post")
    user.addToPosts(groovyPost)
    groovyPost.addToTags(tagGroovy)

    assertEquals 1, groovyPost.tags.size()

    def bothPost = new Post(content: "A groovy and grails post")
    user.addToPosts(bothPost)
    bothPost.addToTags(tagGroovy)
    bothPost.addToTags(tagGrails)
    assertEquals 2, bothPost.tags.size()

}
```

← Создание тегов и связывание их с пользователем

← ❶ Добавляются сообщение пользователя и связь с одним тегом

← ❷ Добавляется связь с несколькими тегами

Поскольку наш класс `Tag` связан отношением 1:m с классом `User` и отношением m:n с классом `Post`, мы можем связывать теги с пользователями и с сообщениями. За кулисами обоими полями `User` и `Post` вновь созданного объекта `Tag` управляет сама платформа Grails, гарантируя сохранение двунаправленности всех отношений.

В листинге 3.17 мы создали сообщение `groovyPost` ❶ с одним тегом («groovy») и сообщение `bothPost` ❷ с двумя тегами («groovy» и «grails»). Выполняя несколько вызовов метода `post.addToTags()`, можно связать каждое сообщение с любым количеством тегов.

Как видите, отношения «многие ко многим» являются самыми сложными из стандартных отношений, поэтому очень важно, чтобы вы ясно представляли себе, как действуют методы `addTo*()`. Листинг 3.17 может служить отправной точкой, но мы рекомендуем вам поэкспериментировать со своими вариантами их использования.

Каскадирование: правила выполнения операций удаления и изменения

Механизм GORM выполняет за кулисами большой объем операций, чтобы обеспечить работу всех этих отношений 1:m и m:n. Мы уже исследовали методы `addTo*()`, но пока мы не знаем, как механизм GORM обеспечивает каскадное выполнение операций.

Правила выполнения операций, касающихся отношений 1:m, достаточно просты. Если в нашем приложении Hubbub выполняется удаление объекта `User`, механизм GORM автоматически удалит все связанные с ним объекты `Post`.

Но давайте рассмотрим более сложную ситуацию, когда операции выполняются с объектами `Tag`. Объект `Post` может быть связан с множеством объектов `Tag`, и каждый объект `Tag` может быть связан более чем с одним объектом `Post`. В подобных случаях механизм GORM принимает решения на основе определений `belongsTo`. Если в объекте отсутствует определение `belongsTo`, каскадирование операций не производится ни в одном из направлений, и мы должны выполнять все необходимые операции самостоятельно.

3.4.4. Связь объекта с самим собой

Последнее, что необходимо реализовать в модели данных приложения Hubbub, – это отношение «слежение», благодаря которому один пользователь сможет следить за сообщениями другого пользователя. Модель данных включает ссылку объекта на самого себя, как показано на рис. 3.8.

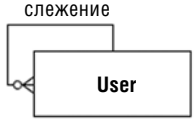


Рис. 3.8. *Моделирование отношения «слежение»*

В ссылке объекта на самого себя нет ничего специального. Это всего лишь особый случай отношения «один ко многим», которое вы уже видели. Чтобы смоделировать необходимое отношение, мы изменим определение `hasMany` в классе `User`, как показано в листинге 3.18.

Листинг 3.18. Моделирование возможности слежения пользователя за другими пользователями

```
class User {

    //... остальной код опущен
```

```
    static hasMany = [ posts : Post, tags : Tag, following : User ]
}
```

Как обычно, напишем испытательный тест, который позволит нам понять, как действует этот вид отношений. В листинге 3.19 производится добавление пользователей, за которыми может следить заданный пользователь.

Листинг 3.19. Простой испытательный тест, выполняющий добавление пользователей

```
void testFollowing() {
    def glen = new User(userId: 'glen', password:'password').save()
    def peter = new User(userId: 'peter', password:'password').save()
    def sven = new User(userId: 'sven', password:'password').save()

    glen.addToFollowing(peter)
    glen.addToFollowing(sven)
    assertEquals 2, glen.following.size()

    sven.addToFollowing(peter)
    assertEquals 1, sven.following.size()
}
```

← Методы addTo*() работают и со ссылками объекта на самого себя

Как видите, метод `addToFollowing()` действует со ссылками объекта на самого себя точно так же, как и в предыдущих вариантах отношений «один ко многим».

Итак, мы исследовали все основные типы отношений, имеющиеся в платформе Grails, и написали полный набор модульных тестов для их проверки. Платформа Grails при этом создала таблицы и поля (включая внешние ключи отношений). Если заглянуть в базу данных приложения Hubbub, можно увидеть, что она содержит пять таблиц, хранящих данные и отношения нашего приложения. На рис. 3.9 изображена полная структурная схема базы данных, которая становится понятна при сопоставлении ее с определениями полей модели предметной области, созданной нами к настоящему моменту.

3.5. В заключение

Мы рассмотрели в этой главе большой объем материала. Многие понятия, введенные здесь, являются фундаментальными и будут весьма часто использоваться в нескольких следующих главах, где мы будем изучать контроллеры и представления.

Мы создали базовые классы предметной области, включая все типичные отношения между данными. Мы также узнали, как выполняется проверка и как определяются собственные ограничения.

Ниже приводятся несколько важных положений, которые вы должны вынести из этой главы:

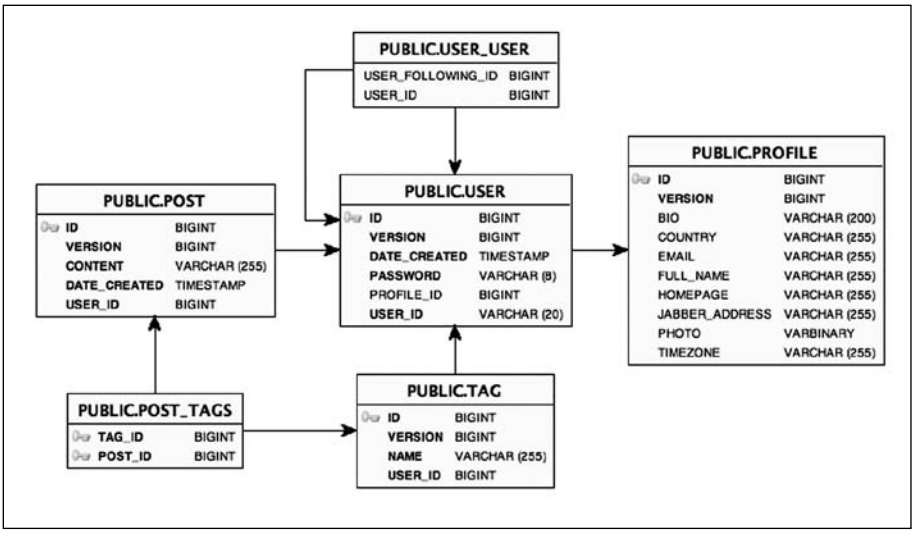


Рис. 3.9. *Окончательная модель данных в приложении Hubbub после всех изменений*

- *Используйте архитектуру, управляемую данными.* Создавайте базовые классы, моделирующие предметную область, на ранних этапах разработки приложений и используйте скаффолдинг, чтобы обеспечить доступ к ним. Это даст вам дополнительный стимул и позволит лучше понять свою предметную область.
- *Подробно изучите основные возможности моделирования.* Вам придется тратить немало времени на создание моделей данных при разработке своих будущих приложений на платформе Grails. Найдите время, чтобы изучить все основные типы отношений, представленные в этой главе. Испытательные тесты дадут вам бесценные экспериментальные сведения.
- *Используйте тесты для экспериментов.* Испытательные тесты для модели предметной области обеспечивают отличный способ экспериментирования со сложными случаями использования метода `save()` и проверки накладываемых ограничений.
- *Не доверяйте пользователям – всегда выполняйте проверки.* Используйте валидаторы при сохранении объектов предметной области. Собственные валидаторы не слишком сложны в реализации, поэтому не бойтесь их использовать, если этого требует ситуация. Лучше включить логику валидаторов в классы предметной области, чем потом мучиться со сложной логикой работы контроллеров.

Вооруженные полученными знаниями, мы должны теперь побольше узнать о создании UI, и после этого мы сможем создать первую полнофункциональную версию приложения Hubbub, от которой нас отделяют всего несколько коротких глав.

В этой главе:

- Создание пользовательского интерфейса с помощью механизма скаффолдинга
- Настройка скаффолдинга для своего проекта
- Использование динамических методов поиска для выполнения запросов
- Дополнительные приемы выполнения запросов

4

Включение модели в работу

В предыдущей главе мы потратили немало времени на создание базовой модели предметной области. Мы не просто определили все необходимые отношения, но и познакомились в процессе работы с фундаментальными основами, которые потребуются вам в каждом новом проекте на платформе Grails. В этой главе мы применим новые знания предметной области и создадим простой пользовательский интерфейс для нашего приложения Hubbub.

Вы узнаете, как можно с помощью механизма скаффолдинга быстро создать пользовательский интерфейс и как можно настроить этот механизм для придания приложению неповторимого внешнего вида. Как только вы получите некоторый опыт использования скаффолдинга для построения простого интерфейса, обеспечивающего возможность взаимодействия с классами предметной области, мы передадим вам некоторые навыки выполнения запросов, что поможет включить в работу свою модель предметной области. Попутно мы реализуем простую форму поиска для приложения Hubbub. На основе этой формы мы расскажем вам о некоторых мощных приемах выполнения запросов и закончим главу созданием облака тегов.

Однако сначала о самом простом. Давайте посмотрим, какой огромный прирост производительности труда обеспечивает механизм скаффолдинга платформы Grails.

4.1. Создание пользовательского интерфейса с помощью механизма скаффолдинга

В предыдущей главе мы создали базовый класс `User` и исследовали порядок выполнения операций CRUD и запросов вручную. Но мы провели вас живописным маршрутом. Мы могли бы реализовать наш набор особенностей одной строкой кода, вообще без какой-либо разметки

HTML, используя преимущества механизма скаффолдинга, реализованного в платформе Grails.

Мы хотели научить вас выполнению простейших операций CRUD вручную, и теперь, когда мы это сделали, пришло время показать вам, как можно реализовать то же самое меньшим количеством кода.

4.1.1. Скаффолдинг классов предметной области приложения Hubbub

В главе 1 мы познакомили вас с механизмом скаффолдинга, реализованным в платформе Grails. Он позволяет всего одной строкой кода автоматически генерировать формы для выполнения операций CRUD и соответствующие им методы действий контроллера для классов предметной области. В самом начале работы над приложением этот механизм обеспечивает фантастическую возможность представить, как будет действовать ваш сайт; скаффолдинг оказывается полезен и позднее для создания страниц административного раздела, которые никогда не будут доступны конечному пользователю.

Давайте применим механизм скаффолдинга к нашему объекту `User`. Первым делом необходимо создать контроллеры, которые реализуют логику выполнения CRUD-операций:

```
grails create-controller com.grailsinaction.User
grails create-controller com.grailsinaction.Profile
grails create-controller com.grailsinaction.Post
grails create-controller com.grailsinaction.Tag
```

После создания контроллеров нужно внести изменения в каждый из созданных классов, чтобы включить механизм скаффолдинга. Например, ниже показано, как следует исправить содержимое файла /grails-app/controllers/com /grailsinaction/UserController.groovy file:

```
package com.grailsinaction

class UserController {
    def scaffold = true
}
```

Проведите такие же действия с файлами ProfileController, PostController и TagController, а затем запустите приложение, чтобы увидеть скаффолдинг в действии. Выполните команду `grails run-app` и откройте в броузере страницу *http://localhost:8080/hubbub*. Щелкните на ссылке UserController, присутствующей на главной странице, и вы увидите пустой список пользователей. Щелкните на кнопке New User, чтобы создать нового пользователя. Платформа Grails отобразит простую форму, которую можно заполнить сведениями о пользователе, как показано на рис. 4.1.

Сгенерированная форма ввода информации о пользователе включает параметр для выбора объекта `Profile`, связанного с объектом `User` отно-

шением «один к одному», который был создан нами в предыдущей главе. Любые проверки, которые мы определили для полей, будут действовать и в сгенерированной форме. Например, текстовое поле, сгенерированное для ввода пароля, позволит ввести не более 8 символов, исходя из ограничения `size: 6..8`, которое мы наложили на это поле.

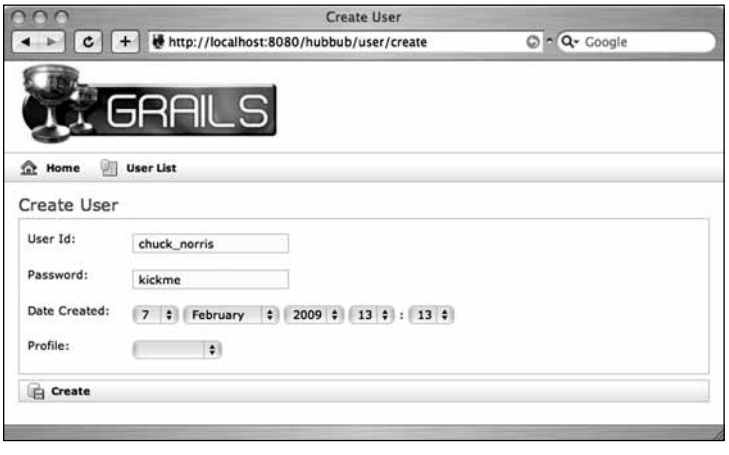


Рис. 4.1. *Механизм скаффолдинга учитывает также установленные отношения и ограничения*

Порядок проверки определяет порядок следования полей в формах, созданных механизмом скаффолдинга

Порядок следования полей в формах, сгенерированных механизмом скаффолдинга, определяется порядком следования ограничений в классах предметной области. Если поля класса вообще не имеют ограничений, порядок следования полей формы будет случайным (что практически всегда нежелательно для форм). Вспомните, что в предыдущей главе мы определили следующие ограничения для класса `User`:

```
static constraints = {
    userId(size:3..20)
    password(size: 6..8, validator: { passwd, user ->
        passwd != user.userId
    })
    dateCreated()
    profile(nullable: true)
}
```

Сравните порядок следования этих ограничений с порядком следования полей формы на рис. 4.1. Заметили совпадение порядка следования?

> Мы включили в замыкание `constraints` поле `dateCreated()`, но не задействовали какую-либо проверку для него. Такой прием удобно использовать, когда необходимо оказать влияние на порядок следования полей в формах, созданных механизмом скаффолдинга, но при этом не требуется накладывать на поле какие-либо ограничения.

Несмотря на то что наши ограничения разрешают пользователям не назначать профиль, мы все же будем экспериментировать с профилем в следующих разделах, поэтому давайте пройдем процесс его назначения. Итак, чтобы связать объект `Profile` с объектом `User`, прежде всего необходимо создать объект `Profile`, воспользовавшись формой, сгенерированной механизмом скаффолдинга для класса `Profile`, которая выглядит далеко не идеально, но которой мы будем пользоваться. Откройте в броузере страницу *http://localhost: 8080/hubbub/profile/create* и создайте новый профиль пользователя.

4.1.2. Скаффолдинг и проверка

В ходе создания нового объекта `Profile` мы можем исследовать, как механизм скаффолдинга ведет себя в случае нарушения установленных ограничений. Взгляните еще раз на определение класса предметной области `Profile`, которое приводится в листинге 4.1.

Листинг 4.1. Класс Profile моделирует персональные свойства пользователя

```
package com.grailsinaction

class Profile {

    static belongsTo = User

    byte[] photo

    String fullName
    String bio
    String homepage
    String email
    String timezone
    String country
    String jabberAddress

    static constraints = {  ← Реализация проверки полей в виде блока ограничений

        fullName(nullable: true)
        bio(nullable: true, maxSize: 1000)
        homepage(url: true, nullable: true)
```

```
            email(email: true, nullable: true)
            photo(nullable: true)
            country(nullable: true)
            timezone(nullable: true)
            jabberAddress(email: true, nullable: true)
        }

    }
```

У нас определены ограничения для домашней страницы (необязательный адрес URL) и адреса электронной почты (также необязательный), и у нас имеется поле типа `byte[]` для хранения фотографии.

На рис. 4.2 показано, что произойдет, если попытаться создать профиль с недопустимыми адресами URL домашней страницы и электронной почты.

Обратите внимание, что поля, в которых обнаружены ошибки, выделены, а в верхней части страницы выведены соответствующие сообщения об ошибках. Описание ошибок, может быть, не совсем соответствует вашему вкусу (вскоре мы покажем, как их изменить), но количество функциональных возможностей интерфейса, которые мы получили за счет всего лишь нескольких строк кода, должно вас впечатлить. Здесь даже присутствует поле, позволяющее выгружать на сервер файл с фотографией.

Теперь исправим не понравившиеся нам сообщения об ошибках.

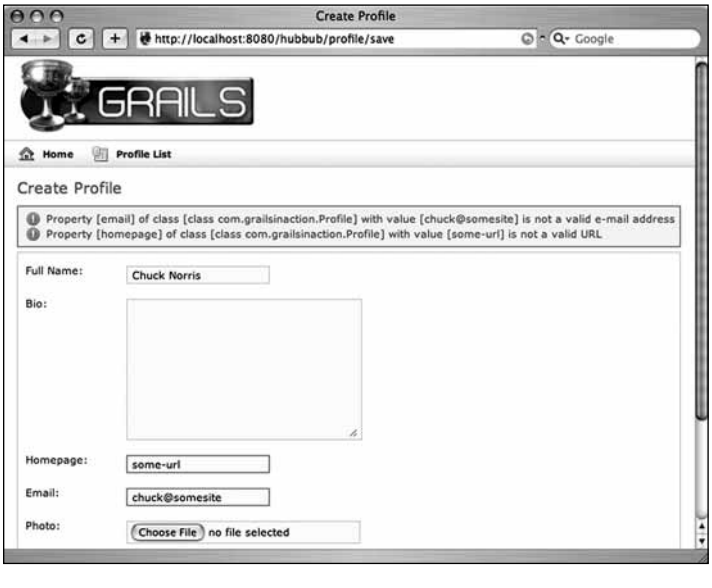


Рис. 4.2. *Механизм скаффолдинга обрабатывает ошибки, выявленные при проверке, и предоставляет элементы textareas для больших полей и поля для выгрузки файлов*

4.1.3. Настройка сообщений об ошибках

Механизм скаффолдинга поддерживает проверку содержимого полей, но сообщения об ошибках по умолчанию могут оказаться вам не по вкусу. Попробуем настроить эти сообщения. Одновременно мы коснемся проблемы интернационализации (i18n) нашего приложения Hubbub.

Наши исследования мы начнем с файла /grails-app/i18n/messages.properties, который содержит пакет ресурсов платформы Grails, используемых по умолчанию:

```
default.invalid.url.message=Property [{0}] of class [{1}] with value [{2}]
➥is not a valid URL
default.invalid.creditCard.message=Property [{0}] of class [{1}] with value
➥[{2}] is not a valid credit card number
default.invalid.email.message=Property [{0}] of class [{1}] with value
➥[{2}] is not a valid e-mail address
default.invalid.size.message=Property [{0}] of class [{1}] with value [{2}]
➥does not fall within the valid size range from [{3}] to [{4}]
```

Здесь вы найдете стандартные сообщения об ошибках, используемые платформой Grails. Среди них вы без труда обнаружите сообщения об ошибках из рис. 4.2. Эти сообщения можно настроить в соответствии со своими предпочтениями.

В текстах сообщений можно заметить специальные поля:

- {0} – для имени свойства
- {1} – для имени класса
- {2} – для значения

Существуют и другие поля, для максимального и минимального значений, используемых некоторыми ограничениями. Если вы захотите использовать поля, платформа Grails будет замещать их фактическими значениями во время выполнения.

Сообщения об ошибках, используемые по умолчанию, вполне пригодны, но обычно бывает необходимо приводить дополнительные пояснения, помогающие описать проблему. Рассмотрим наши поля с адресами домашней страницы и электронной почты. Чтобы ясно сообщить пользователю о проблеме, мы можем определить в файле messages.properties собственное сообщение, например:

```
profile.homepage.url.invalid=You must provide a valid web address for your
➥homepage. [{2}] simply does not cut it
profile.email.email.invalid=You must provide a valid email address. [{2}]
➥is just not the business
```

Записи в файле имеют формат {className}.{field}.{errorCode}, где errorCode – это те же коды ошибок, что были представлены в предыдущей главе (если вам требуется освежить их в памяти, загляните в раздел 3.3.1 «Стандартные валидаторы»).

Вы можете изменять файл ресурсов messages.properties прямо в процессе работы приложения, и эти изменения вступят в силу при следующей же проверке. На рис. 4.3 изображены наши новые, более уместные сообщения об ошибках.

Эти сообщения об ошибках все еще могут оставлять желать лучшего; важно понять, какая сила заключена в оформлении всех сообщений в виде отдельного файла. Если вам приходилось заниматься интернационализацией приложений, вы могли заметить, что файл messages.properties выглядит, как файл ресурсов, – потому что он и есть файл ресурсов! Это означает, что платформа Grails позволяет легко и просто выполнять интернационализацию ваших сообщений об ошибках (о чем подробнее рассказывается во врезке).

На этом наш экскурс в обработку ошибок и настройку сообщений можно считать завершенным, теперь давайте свяжем наш новый объект `Profile` с объектом `User`.

Рис. 4.3. *Вы можете создавать собственные сообщения об ошибках в файлах ресурсов*

Скаффолдинг и интернационализация (i18n)

Вы могли заметить, что каталог i18n содержит файлы с сообщениями об ошибках на нескольких языках, при этом каждый файл имеет собственное расширение. Платформа Grails автоматически определяет языковые настройки пользователя, опираясь на заголовок `Accept-Language`, отправляемый броузером, но для нужд тес-

тирования вы можете передавать параметр `?lang=ru` любым контроллерам, чтобы имитировать работу с пользователем, говорящим на русском языке. Платформа Grails способна даже сохранять языковые настройки в виде cookie, которые будут использоваться при последующих запросах. Ниже приводится пример формы с сообщениями об ошибках на французском языке:

***Рис. 4.4**. Форма с сообщениями об ошибках на французском языке*

Но нет большого смысла выполнять интернационализацию сообщений об ошибках, когда остальной интерфейс, созданный механизмом скаффолдинга, выводится на английском языке. По этой причине Марсель Овердик (Marcel Overdijk) разработал расширение i18n Templates, которое вы можете установить в свое приложение на платформе Grails. О том, как действует данное расширение, подробно рассказывается в главе 8, так что, если вам не терпится, можете сразу же перейти к ней.

Расширение i18n Templates реализует собственные версии используемых механизмом скаффолдинга шаблонов, которые поддерживают сменные пакеты сообщений, подобные пакетам с сообщениями об ошибках. Вы можете создавать интернационализированные версии надписей на кнопках, заголовков и сообщений для пользователей для любых элементов классов предметной области . За дополнительной информацией обращайтесь на страницу расширения на веб-сайте проекта Grails по адресу: *http://grails.org/Plugins*.

4.1.4. Управление отношениями с помощью механизма скаффолдинга

После того как с помощью формы будет создан корректный объект `Profile`, можно вернуться в форму Create User и выбрать этот объект `Profile` для создаваемого объекта `User`. На рис. 4.5 показана форма Create User с выбранным профилем.

***Рис. 4.5.** Форма создания нового объекта User позволяет выбрать объект Profile*

Если бы мы создали несколько профилей, выбрать нужный, опираясь на его идентификатор в базе данных, было бы довольно сложно, однако эту ситуацию можно упростить (о чем подробнее рассказывается во врезке ниже).

Совет

Платформа Grails позволяет вносить изменения в исходный программный код прямо в процессе выполнения приложения. Это фантастический способ повысить производительность труда, позволяющий вам не тратить время на остановку и запуск приложения для вступления изменений в силу. Когда платформа Grails обнаруживает какие-либо изменения, она автоматически перезапускает приложение. Однако будьте внимательны. Если в файле dataSource.groovy свойство `dbCreate` имеет значение `create-drop`, при перезапуске приложения база данных будет заново создана, и все имевшиеся в ней данные будут утеряны.

Механизм скаффолдинга в платформе Grails поддерживает все обычные типы отношений: 1:1, 1:m и m:n. Всего несколькими строчками программного кода вы можете автоматически реализовать интерфейс всего приложения, который позволяет вводить данные и задействовать отношения.

Настройка раскрывающихся списков, созданных механизмом скаффолдинга, с помощью метода toString()

Для заполнения раскрывающихся списков механизм скаффолдинга платформы Grails использует метод `toString()` объекта, поэтому сейчас попробуем настроить метод `toString()` класса `Profile` с целью немного упростить возможность выбора:

```
class Profile {

    // здесь находится существующий программный код класса
    // предметной области

    String toString() {
        "Profile for ${fullName} (${id})"
    }
}
```

Если теперь еще раз попытаться создать пользователя, платформа Grails заполнит раскрывающийся список Profile с помощью нового метода `toString()`, как показано на рис. 4.6:

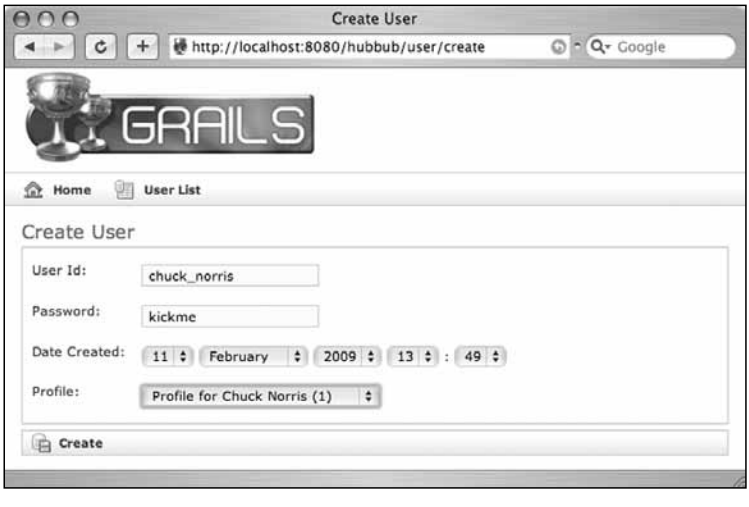


Рис. 4.6. *Новая реализация метода toString() раскрывающегося списка позволяет упростить выбор*

При наличии большого количества отношений механизм скаффолдинга становится трудноуправляемым, особенно в формах, реализующих отношения родитель–потомок. В таких ситуациях предпочтительнее создавать пользовательский интерфейс вручную.

Следует ли использовать механизм скаффолдинга в готовых приложениях?

Платформа Grails существенно упрощает автоматическое создание форм для выполнения простых операций CRUD, поэтому вы могли задуматься о возможности применения скаффолдинга платформы Grails в готовых приложениях. Подобные формы не имеют косметических украшений и они не обеспечивают удобство использования для простых пользователей, зато они прекрасно подходят для внутренних нужд. Они могут использоваться в административном разделе сайта для предоставления справочной информации и для реализации операций CRUD, где не требуется утонченность пользовательского интерфейса. В следующем разделе мы покажем, как можно украсить эти формы, чтобы они выглядели, как другие внутренние разделы вашего сайта. А в главе 10 мы продемонстрируем, как можно ограничить доступ к ним, чтобы только администраторы имели возможность ими пользоваться.

4.1.5. Настройка внешнего вида автоматически созданных форм с помощью CSS

Несмотря на то что формы, созданные с помощью механизма скаффолдинга, обладают значительной мощью, иногда они имеют слишком «простой» вид. Зачастую бывает достаточно добавить немного оформления, и встроенные шаблоны окажутся вполне пригодными для использования в готовых приложениях (особенно для внутренних административных нужд).

Если у вас появится желание изменить стили CSS для элементов форм, созданных с помощью механизма скаффолдинга, вы можете отредактировать файл CSS /web-app/css/main.css. Этот файл подключается всеми стандартными шаблонами платформы Grails, поэтому просто внесите необходимые изменения и обновите страницу в броузере. Если вы задумаете выполнить более глубокие настройки и захотите изменить сами макеты страниц (например, чтобы добавить баннер или нижний колонтитул), вам придется изменить стандартный макет платформы Grails, который используется механизмом скаффолдинга для создания любых форм.

В главе 1 мы познакомились с шаблонами страниц, и тогда мы говорили, что платформа Grails использует эти шаблоны (иногда их называют декораторами) для добавления в страницы приложения стандартных заголовков, нижних колонтитулов и боковых колонок. Подробнее

эти шаблоны мы будем изучать в главе 6, а пока вам достаточно знать, что вы можете изменить глобальный шаблон для своего сайта, отредактировав файл /grails-app/views/layouts/main.gsp. В листинге 4.2 представлена измененная версия данного файла с новым оформлением приложения Hubbub.

Листинг 4.2. Измененная версия стандартного декоратора для приложения Hubbub

```
<html>
    <head>
        <title>Hubbub &raquo; <g:layoutTitle default="Welcome" /></title>
        <link rel="stylesheet" href="
            <g:createLinkTo dir='css'
                    file='hubbub.css'/>"
        />
        <g:layoutHead />
    </head>
    <body>
        <div>
            <div id="hd">
                <a href="<g:createLinkTo dir="/"/>">
                    <img id="logo"
                        src="${createLinkTo(
                            dir: 'images',
                            file: 'headerlogo.png')}"

                        alt="hubbub logo"/>
                </a>
            </div>
            <div id="bd"><!-- начало тела страницы -->
                <g:layoutBody/>
            </div> <!-- конец тела страницы -->
            <div id="ft">
                <div id="footerText">
                    Hubbub - Social Networking on Grails
                </div>
            </div>
        </div>
    </body>
</html>
```

Добавленная таблица каскадных стилей

Использование собственного титульного изображения

В этой версии мы добавили в главный макет приложения Hubbub собственное изображение и файл CSS. В результате главная страница приложения выглядит так, как показано на рис. 4.7.

Используя описанный прием, можно быстро изменить схему размещения элементов для всего интерфейса, созданного с помощью механизма скаффолдинга. А когда вы больше узнаете о том, как действуют макеты (в главе 6), вы сможете переопределять макеты для отдельных кон-

троллеров (например, только для профилей) или даже для отдельных действий (например, только для формы редактирования профиля).

Рис. 4.7. *Используя собственные макеты, можно изменять интерфейс, созданный с помощью механизма скаффолдинга*

4.1.6. Что находится за пределами возможностей механизма динамического скаффолдинга?

Механизм динамического скаффолдинга обладает невероятной мощью; мы продемонстрировали вам некоторые способы его настройки, но существуют ситуации, когда не имеет смысла пытаться задействовать его:

- Сложные формы, предназначенные для работы с классами из разных предметных областей, плохо поддаются реализации с применением механизма динамического скаффолдинга (в результате вы получите пользовательский интерфейс, который примет только горячо любящая вас мать).
- Динамические формы не поддерживают технологию Ajax (хотя это ограничение можно обойти, реализовав собственные шаблоны для скаффолдинга, о которых мы будем говорить ниже).
- Динамические формы имеют весьма ограниченную поддержку возможности настройки взаимодействий. (Мы продемонстрировали некоторые манипуляции с CSS и шаблоном, но нет никакой возможности выполнить глубокую настройку пользовательского интерфейса без изменения самого генератора, о чем мы также будем говорить немного ниже).

Как видите, динамические формы можно использовать не во всех ситуациях. К счастью, динамические формы – не единственная разно-

видность форм, которые можно создавать с помощью механизма скаффолдинга. Платформа Grails позволяет генерировать статические формы, которые затем можно редактировать вручную с целью добавления в них своих особенностей. Давайте попробуем сгенерировать несколько статических файлов представлений и посмотрим, что дает нам механизм статического скаффолдинга.

4.1.7. Статический скаффолдинг: создание и настройка форм

Иногда динамический скаффолдинг плохо отвечает потребностям проекта. Возможно, у вас предусматривается реализация сложных форм ввода данных, связанных отношениями «многие ко многим» или «родитель-потомок», где требуются мастерство и умение. Или, может быть, вы хотите использовать совершенно иной подход к обработке представления.

В такого рода ситуациях платформа Grails позволяет с помощью механизма скаффолдинга создавать статические формы, которые могут использоваться в качестве отправной точки. Получив пакет файлов с реализацией контроллера и представления, которые обычно генерируются динамически, вы сможете быстро реализовать свои идеи.

Для различных ситуаций платформа Grails предлагает два способа создания статических файлов с помощью механизма скаффолдинга. При первом способе генерируются сразу все файлы контроллеров и представлений:

```
grails generate-all com.grailsinaction.User
```

Второй способ генерирует только файлы представлений, оставляя класс контроллера в покое:

```
grails generate-views com.grailsinaction.User
```

Если будут обнаружены существующие файлы контроллеров или представлений, у вас будет запрошено разрешение перезаписать их.

Наличие в платформе Grails двух отдельных команд для каждого вида операций очень удобно, потому что реализация контроллеров обычно весьма нечувствительна к изменениям в классах предметной области. Например, контроллер может выполнять такие операции, как `User.save(params)` или `User.get(params.id)`, вместо того, чтобы обращаться к отдельным свойствам объектов. От представлений же требуется отобразить и предоставить возможность редактировать каждое свойство класса предметной области, поэтому они весьма чувствительны к любым изменениям в классах предметной области. С другой стороны, представления очень чувствительны к изменениям в модели данных (таким, как добавление новых полей, переименование полей и другим), поэтому воз-

можность заново создавать только файлы представлений, когда это необходимо, весьма полезна.

Реализация стандартного контроллера – не слишком сложная наука. В предыдущих главах вы писали куда более мудреный код. Но вам стоит знать, какие методы генерируются для вас, – они перечислены в табл. 4.1.

Механизм статического скаффолдинга генерирует только четыре стандартных файла GSP (list, show, edit и create), все остальное реализуется логикой контроллера.

Таблица 4.1. Базовые методы контроллера, создаваемые механизмом скаффолдинга

Имя метода	Выполняемое действие	Отображается представлением
`index`	Вызывает метод `list()`	–
`list`	Отображает список объектов предметной области, разбитый на страницы	`list.gsp`
`show`	Отображает таблицу со всеми свойствами одного экземпляра класса предметной области	`show.gsp`
`delete`	Удаляет объект с указанным идентификатором, затем вызывает метод `list()`	–
`edit`	Отображает форму редактирования экземпляра класса предметной области и затем передает ее методу `update()`	`edit.gsp`
`update`	Обновляет указанный экземпляр класса предметной области, записывая в него новые значения свойств. В зависимости от результатов проверки передает управление либо методу `list()` либо методу `edit()`	нет
`create`	Отображает пустую форму редактирования нового экземпляра класса предметной области и затем передает управление методу `save()`	`create.gsp`
`save`	Сохраняет новый экземпляр класса предметной области. В зависимости от результатов проверки передает управление либо методу `list()`, либо методу `create()`	–

4.1.8. Настройка шаблонов механизма скаффолдинга: построение собственного генератора пользовательского интерфейса

Использование стандартных шаблонов платформы Grails экономит много времени, но они не всегда обеспечивают функциональность, необходимую проекту. Довольно просто можно изменять оформление шаблонов во время выполнения с помощью CSS, но иногда возникает потребность изменять их поведение, например убирать кнопку Delete (Удалить), когда форму просматривает пользователь, не обладающий правами администратора. В таких ситуациях необходимо вносить изменения в шаблоны, используемые механизмом скаффолдинга платформы Grails. Вы будете рады узнать, что это возможно!

В первую очередь необходимо установить шаблоны в приложение:

```
grails install-templates
```

Эта команда создаст группу файлов шаблонов в каталоге /src/templates, которые вы сможете редактировать. На рис. 4.8 показан полный список сгенерированных файлов.

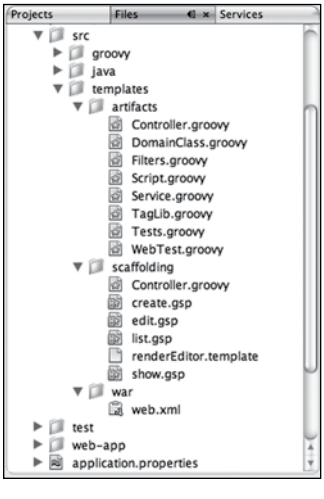


***Рис. 4.8.** Файлы шаблонов для механизма скаффолдинга, которые вы можете установить и изменить*

Нас интересуют файлы в каталоге /templates/scaffolding. В них присутствует программный код, динамически сгенерированный механизмом скаффолдинга. Вы можете изменять любые компоненты, полученные в процессе работы механизма скаффолдинга, от реализации контроллера до файлов GSP.

В каталоге artifacts находятся заготовки классов, которые генерируются командами Grails. Например, команда `grails generate-controller` создает файл /templates/artifacts/Controller.groovy. Внедрение в этот процесс может быть полезным, когда необходимо сгенерировать стандартные заголовки лицензий или применить к новым классам некоторые простые стандарты программирования.

Ниже, в главе 15, мы будем говорить о настройке файла web.xml. Помните, что платформа Grails не препятствует выполнению вами тонкой настройки шаблонов пользовательского интерфейса любыми способами, имеющими смысл для вашего приложения.

4.2. Извлечение данных динамическими методами и механизм Query by Example

Мы много говорили о скаффолдинге и включили в работу нашу модель данных, сгенерировав все формы, необходимые для выполнения базовых операций с данными (создания, редактирования и удаления). Но у нас осталась нереализованной еще одна важная операция: извлечение данных.

Оптимальный способ обратиться к модели данных в платформе Grails – через особенность, известную как *динамические методы поиска*. Эти методы обеспечивают лаконичность обращения к модели данных благодаря использованию естественного синтаксиса языка программирования. В данном разделе мы сначала подробно исследуем динамические методы поиска. Затем познакомимся с механизмом запроса по образцу (Query By Example, QBE), который предоставляет удобный способ запрашивать объекты предметной области, опираясь на частично заполненный эталонный объект. После этого мы рассмотрим некоторые интересные средства платформы Grails, позволяющие запрашивать списки объектов и определять их количество.

4.2.1. Реализация простой формы поиска

Один из лучших способов поближе познакомиться с динамическими методами поиска – реализовать простую форму поиска для приложения Hubbub. Допустим, что нам необходимо отыскать всех друзей, имеющих учетные записи в приложении Hubbub. Для этого расширим поддержку класса `User`.

Сначала необходимо создать форму поиска в виде файла /grails-app/views/user/search.gsp, содержимое которого показано в листинге 4.3.

Листинг 4.3. Простая форма поиска для приложения Hubbub

```
<html>
<head>
    <title>Search Hubbub</title>
```

```
    <meta name="layout" content="main"/>
</head>
<body>

    <formset>
        <legend>Search for Friends</legend>

    <g:form action="results">  ← ❶ Форма будет передаваться действию results
        <label for="userId">User Id</label>
        <g:textField name="userId" />  ← ❷ Поле ввода значения userId

        <g:submitButton name="search" value="Search"/>
    </g:form>

    </formset>

</body>
</html>
```

В листинге 4.3 используется несколько специальных тегов `<g:>`, с которыми вы еще не знакомы. Тег `<g:form>` ❶ определяет, что после щелчка на кнопке Submit (Отправить) значения формы должны передаваться действию `results` текущего контроллера, которым в данном случае является `User`. Тег `<g:textField>` ❷ определяет текстовое поле с именем `userId`; это означает, что внутри действия `results` оно будет представлено как `params.userId`.

Сохранив форму поиска, можно открыть в броузере страницу */hubbub/user/search* и ввести критерий поиска. На рис. 4.9 изображена форма поиска в действии. Мы уже видели подобную форму выше в этой же главе. Однако сейчас нам требуется реализовать поиск в контроллере. Давайте дополним класс `UserController` обработкой новой формы. В листинге 4.4 приводится реализация контроллера с двумя новыми действиями: `search` и `results`.

Листинг 4.4. Добавление логики поиска в файл UserController.groovy

```
class UserController {

    def scaffold = true

    def search = {  ← ❶ Переход к форме search.gsp
    }

    def results = {  ← ❷ Обрабатывает отправленную форму
                    параметров
        def users =

    User.findAllByUserIdLike(params.userId)  ← ❸ Поиск на основе формы
        return [ users: users, term : params.userId  ←
                                                      ❹ Передает представлению
                                                      пользователей и критерий поиска
```

```
        ]
      }
    }
```

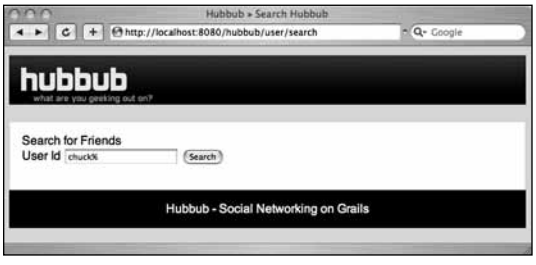


Рис. 4.9. *Простая форма поиска приложения Hubbub в действии*

Обратите внимание, что мы оставили скаффолдинг объекта `User` и дополнили его новой операцией поиска. Мы добавили пустое действие `search` ❶. Оно будет вызываться при обращении к форме поиска. Когда пользователь перейдет по адресу /user/search, будет вызвано это действие, и платформа Grails направит пользователя к файлу /grails-app/views/user/search.gsp, который действует как шаблон требуемой формы.

Когда пользователь щелкнет на кнопке Search (Найти), форма будет передана действию `results` ❷, которое вызовет динамический метод поиска объекта `User` ❸ и затем поместит критерий поиска и найденные объекты `User` в представление ❹.

Класс `User` не имеет метода `findByUserIdLike()` — платформа Grails создаст его динамически с помощью свойства `MetaClass`, которым обладают все классы в языке Groovy. Используя динамические возможности языка Groovy, платформа Grails позволяет создавать запросы на основе всех хранимых свойств, имеющихся в классе. Вы даже можете объединять два критерия с помощью `And` и `Or`. Действие `login` входа в систему могло бы быть реализовано примерно так:

```
def user = User.findByUserIdAndPassword(params.userId, params.password)
if (user) {
    // пользователь найден в базе данных..
}
```

Параметры динамических методов поиска мы исследуем в следующем разделе, а пока вернемся к вызову метода `findAllUserIdLike()`. На рис. 4.9 видно, что для всех пользователей, чьи имена начинаются с подстроки «chuck», мы использовали шаблонный символ языка SQL (`%`). Но было бы непрактично заставлять пользователя вводить символы `%` для обозначения допустимых вариантов совпадения. Давайте изменим реализацию контроллера так, чтобы он сам добавлял шаблонный символ:

```
def users = User.findAllByUserIdLike("%${params.userId}%")
```

Эта строка добавляет шаблонные символы с обеих сторон искомой строки для обеспечения полнотекстового поиска.

Теперь, когда у нас имеется реализация операции поиска, необходимо создать простую форму для отображения результатов. В листинге 4.5 представлена реализация страницы results.gsp (в том же каталоге, где находится файл search.gsp), так что наш контроллер может возвращать пользователю результаты поиска.

Листинг 4.5. Простая форма для отображения результатов поиска

```
<html>
    <head>
        <title>Search Results</title>
        <meta name="layout" content="main"/>
    </head>
    <body>

        <h1>Results</h1>
        <p>Searched ${com.grailsinaction.User.count()} records
        for items matching <em>${term}</em>.
        Found <strong>${users.size()}</strong> hits.
        </p>
        <ul>
            <g:each var="user" in="${users}">
                <li>${user.userId}</li>
            </g:each>
        </ul>

        <g:link action='search'>Search Again</g:link>

    </body>
</html>
```

❶ Отображает общее количество записей User

❷ Выполняет итерации по списку найденных пользователей

Форма представления результатов приведенная в листинге 4.5 использует еще один динамический метод, `User.count()`, для отображения общего количества объектов `User` в базе данных ❶. Для обхода списка всех найденных объектов `User` и вывода их имен используется тег платформы Grails `<g:each>` ❷. На рис. 4.10 показана форма с результатами поиска в действии.

Теперь у нас имеется полная реализация простой функции поиска, но нам предстоит еще многое узнать о запросах. Рассмотрим некоторые испытательные тесты, чтобы определить границы наших возможностей.

4.2.2. Многоликие динамические методы поиска

Динамические методы поиска могут использоваться для составления практически любых простых запросов, а их естественный предметно-ориентированный язык (Domain Specific Language, DSL) облегчает их

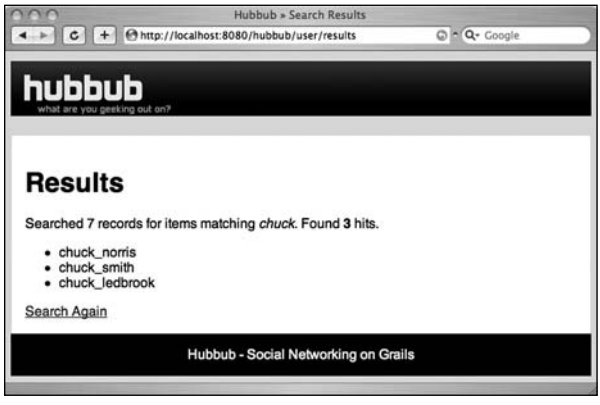


Рис. 4.10. *Форма отображения результатов поиска содержит несколько имен пользователей, начинающихся с подстроки «chuck»*

понимание и сопровождение. Создадим новый интеграционный тест, в котором мы сможем опробовать несколько запросов:

```
grails create-integration-test QueryIntegration
```

В листинге 4.6 демонстрируются различные способы использования динамических методов поиска для поиска определенных значений любых атрибутов класса предметной области. В данном тесте выполняется поиск по имени пользователя, по имени пользователя и паролю и по дате.

Листинг 4.6. Различные динамические методы поиска в действии

```
import grails.test.*
import com.grailsinaction.Profile
import com.grailsinaction.User

class QueryIntegrationTests extends GrailsUnitTestCase {

    void testBasicDynamicFinders() {

        new User(userId: 'glen', password: 'secret',
            profile: new Profile(email: 'glen@glensmith.com')).save()
        new User(userId: 'peter', password: 'sesame',
            profile: new Profile(homepage: 'http://www.peter.com/')).save()

        def user = User.findByPassword('sesame')   ❶ Поиск по одному атрибуту
        assertEquals 'peter', user.userId

        user = User.findByUserIdAndPassword('glen',
                                            'secret')   ❷ Поиск по двум атрибутам
        assertEquals 'glen', user.userId
```

```
        def now = new Date()
        def users =
            User.findAllByDateCreatedBetween(now-1, now)
        assertEquals 2, users.size()

        def profiles =
            Profile.findAllByEmailIsNotNull()
        assertEquals 1, profiles.size()
    }

}
```

❸ Поиск внутри диапазона

Поиск непустых значений

Метод `findBy*()` может принимать либо один ❶, либо два ❷ запрашиваемых атрибута и возвращает одно значение (или `null`, если искомое значение не найдено). Соответствующий ему метод `findAllBy*()` ❸ возвращает все найденные значения или пустой список, если не найдено ни одного соответствия.

Допустим, нам требуется определить число учетных записей, созданных за один или два последних дня. Сделать это можно, как показано в следующем фрагменте, с помощью метода `findAllBySignupDateBetween()`, который вернет список пользователей:

```
def now = new Date()
def users = User.findAllBySignupDateBetween(now-1, now)
assertEquals 2, users.size()
```

Динамические методы поиска обладают весьма широкими возможностями, и они имеют массу вариантов. В табл. 4.2 перечислены некоторые из основных динамических методов поиска. Вы можете объединять критерии поиска с помощью `And`, но выполнять запрос можно только к двум полям.

Таблица 4.2. Некоторые динамические методы поиска

Критерий поиска	Пример
LessThan	User.findAllByDateCreatedLessThan(lastWeek)
LessThanEquals	User.findAllByDateCreatedLessThanEquals(lastWeek)
GreaterThan	User.findAllByDateCreatedGreaterThan(lastWeek)
GreaterThanEquals	User.findAllByDateCreatedGreaterThanEquals(lastWeek)
Like	User.findAllByPasswordLike(“secret”)
Ilike	User.findAllByPasswordIlike(“Secret”)
NotEqual	User.findAllByPasswordNotEqual(“password”)
Between	User.findAllByDateCreatedBetween(lastWeek, now)
IsNotNull	User.findAllByPasswordIsNotNull()
IsNull	User.findAllByPasswordIsNull()

And	User.findAllByDateCreatedGreaterThanAndUserIdLike(lastWeek, 'glen')
Or	User.findAllByPasswordIsNullOrPasswordLike("secret")

Создание испытательных тестов – отличный способ убедиться в том, что программный код действует именно так, как ожидалось. Но что делать, когда, приступив к составлению запросов, вам требуется быстро смоделировать целый пакет различных запросов? Мы считаем, что удобнее всего воспользоваться консолью платформы Grails. Если запустить приложение командой `grails console`, платформа Grails откроет небольшое окно консоли с полным окружением, где вы сможете опробовать свои запросы и немедленно получить результат.

Как действуют динамические методы поиска

После того как мы понаблюдали за динамическими методами поиска в действии, вам, наверно, любопытно узнать, как они действуют. Методы с этими именами отсутствуют в определении класса , и для платформы Grails было бы неэффективно генерировать в каждом объекте отдельные методы для каждой возможной комбинации запросов, так?

Для реализации динамических методов поиска используется свойство `MetaClass`, с помощью которого производится перехват обращений к методам объектов предметной области. Используя метод `methodMissing`, которым обладают все классы в языке Groovy, платформа Grails перехватывает обращения к несуществующим методам и создает на их основе запрос Hibernate. Кроме того, она кэширует методы, благодаря чему последующие обращения к ним выполняются гораздо быстрее. Обязательно ознакомьтесь с описанием свойства `MetaClass`, чтобы повысить свое мастерство владения языком Groovy.

4.2.3. Использование стратегии немедленной и отложенной загрузки в динамических методах поиска

Когда механизм GORM выполняет поиск, такой как `findByUserId`, он запрашивает все поля класса предметной области `User`, однако извлечение вложенных коллекций (таких как `Post`) происходит по требованию. Иначе говоря, извлечение всех объектов `Post`, связанных с определенным пользователем, производится только при обращении к полю `posts` объекта `User` (за счет выполнения механизмом GORM второго запроса). Это называется стратегией *отложенной загрузки*. Такая стратегия используется механизмом GORM (Hibernate) по умолчанию, потому что

извлечение вложенных объектов было бы пустой тратой времени, если бы пользователь не предполагал обращаться к ним.

Однако, когда вы формируете свои собственные запросы, вам зачастую заранее известно, будет ли осуществляться доступ к вложенным коллекциям. В таких случаях есть смысл извлекать все вложенные объекты в одном запросе SQL – это называется стратегией *немедленной загрузки*. Реализовать ее можно, сообщив платформе Grails, какие коллекции было бы желательно извлекать из базы данных немедленно. Мы могли бы создать запрос, извлекающий информацию о пользователе и все его сообщения, как показано ниже:

```
def user = User.findByUserId('glen', [fetch:[posts:'eager']])
```

Если вам интересно, сколько запросов выполняется, вы можете установить параметр `logSql=true` в файле с настройками DataSource.groovy. В результате в окне консоли будут выводиться все запросы к базе данных, производимые библиотекой Hibernate. Это позволит определить те части программы, где применение стратегии немедленной загрузки может дать существенный прирост производительности. Если вы предполагаете всегда использовать стратегию немедленной загрузки вложенных коллекций, вы можете добавить параметр `mapping` в параметры настройки механизма GORM, обеспечивающий использование стратегии немедленной загрузки по умолчанию. О настройке источников данных и о параметрах доступа к ним мы поговорим в главе 13.

4.2.4. Когда динамические методы поиска не могут использоваться

Динамические методы поиска обладают широкими возможностями, но область их применения имеет некоторые ограничения. Самое существенное ограничение заключается в том, что вы можете использовать эти методы для поиска не более чем по двум полям. Если вам требуется включить в запрос более двух полей, следует использовать запросы с критериями или язык запросов библиотеки Hibernate (Hibernate Query Language, HQL).

То же относится и к случаю, когда требуется выполнить более сложный запрос, основанный на атрибутах объектов, связанных отношениями. В таких ситуациях вам также придется прибегнуть к использованию HQL. Но не стоит пугаться. Мы подробно исследуем обе указанные возможности в разделе 4.3.

4.2.5. Введение в механизм Query by Example (QBE)

Платформа Grails позволяет запросить объекты несколькими способами, мы начали с изучения динамических методов поиска. Однако платформа Grails обладает массой других возможностей, таких как механизм запроса по образцу (Query by Example, QBE). При использова-

нии механизма QBE создается объект предметной области и заполняется значениями свойств, которые требуется запросить (любые свойства, имеющие значение `null`, будут исключены из запроса). Затем выполняется поиск похожих объектов в базе данных.

Добавим новый испытательный тест в `QueryIntegrationTests` (из листинга 4.6), чтобы посмотреть, как действует механизм QBE. Реализация нового теста приводится в листинге 4.7.

Листинг 4.7. При использовании механизма QBE осуществляются заполнение образцового объекта и поиск похожих объектов в базе данных

```
void testQueryByExample() {

    new User(userId: 'glen', password: 'password').save()
    new User(userId: 'peter', password: 'password').save()
    new User(userId: 'cynthia', password: 'sesame').save()

    def userToFind = new User(userId: 'glen')
    def u1 = User.find(userToFind)                    ❶
    assertEquals('password', u1.password)

    userToFind = new User(userId: 'cynthia')
    def u2 = User.find(userToFind)
    assertEquals('cynthia', u2.userId)

    userToFind = new User(password: 'password')
    def u3 = User.findAll(userToFind)                 ❷
    assertEquals(['glen', 'peter'], u3*.userId)

}
```

❶ Отыщет единственного пользователя, соответствующего шаблону

❷ Отыщет всех пользователей, соответствующих шаблону

В этом тесте мы заполняем поля образцового объекта `User` требуемыми значениями и затем передаем его либо методу `User.find()`, чтобы отыскать первый соответствующий объект ❶, либо методу `User.findAll()`, чтобы получить все экземпляры, соответствующие заданным критериям ❷. Если метод `User.find()` обнаружит более одного объекта, соответствующего образцу, он все равно вернет только первый из них.

4.2.6. Динамические методы list(), listOrderBy() и countBy()

Мы детально изучили методы платформы Grails из семейства `find()`, но мы еще не исследовали некоторые другие удобные методы поиска, которые платформа Grails добавляет к классам предметной области, такие как `list()` и `count()`. Метод `User.list()` возвращает список всех объектов `User`, хранящихся в базе данных, а метод `User.count()` – общее количество объектов `User`, однако они могут использоваться для решения гораздо более замысловатых задач, чем эти.

Сначала рассмотрим более сложные случаи применения метода list(). Во-первых, метод list() может принимать дополнительные параметры, управляющие сортировкой, направлением сортировки и смещением, как показано в табл. 4.3.

Попробуем вызвать метод list() с некоторыми из этих параметров. Допустим, что нам требуется получить первые пять пользователей, упорядоченные по возрастанию значения поля userId, и задействовать стратегию немедленной загрузки сообщений этих пользователей, чтобы тут же выполнить итерации по ним. Сделать это достаточно просто:

```
def users = User.list([sort: 'userId', order: 'asc',
                       max: 5, fetch: [posts: 'eager']])
```

Таблица 4.3. Параметры метода list()

Параметр	Описание
max	Определяет максимальное число возвращаемых объектов
sort	Определяет поле, по которому должен быть отсортирован возвращаемый список
order	Определяет направление сортировки: 'desc' или 'asc' (значение по умолчанию: 'asc')
ignoreCase	Определяет, должна ли сортировка выполняться без учета регистра символов (значение по умолчанию: true)
fetch	Отображение параметров, определяющее стратегию немедленной или отложенной загрузки
offset	Определяет количество элементов в начале ResultSet, которые должны быть пропущены при формировании возвращаемого списка (удобно для организации постраничного просмотра)

Теперь, когда вы познакомились с некоторыми возможностями метода list(), настало время рассказать вам о методе listOrderBy*(). Этот метод позволяет определять поле для сортировки. Например, если необходимо выполнить сортировку по возрастанию значения в поле userId, можно использовать такой вызов:

```
def users = User.listOrderByUserId()
```

При этом вы по-прежнему можете применять параметры max, order и offset, если возникает необходимость управлять порядком сортировки или смещением при постраничном отображении результатов.

Теперь рассмотрим в действии соответствующий метод countBy*(). Попробовать найти всех пользователей с рискованным паролем можно таким способом:

```
def dodgyCount = User.countByPassword("password")
```

Этот вызов вернет количество пользователей с паролем «password».

4.3. Более сложные способы выполнения запросов

Динамические методы поиска выразительны и довольно мощны. Возможность записать такой запрос, как `User.findByUserId("joe")`, обеспечивает высокую производительность труда, но иногда подобной возможности бывает недостаточно. Платформа Grails не оставит вас наедине с проблемой: пришло время представить вам запросы с критериями и язык HQL.

В этом разделе рассказывается о достаточно сложных приемах выполнения запросов, так что не впадайте в уныние, если при первых нескольких прочтениях он покажется вам чересчур сложным. Когда при написании своих приложений вы не сможете реализовывать необходимые возможности с помощью динамических методов поиска, возвращайтесь сюда, чтобы еще раз рассмотреть эти продвинутые способы выполнения запросов. Пристегнитесь и поехали!

4.3.1. Больше возможностей: запросы с критериями

Динамические методы поиска прекрасно подходят для выполнения простых запросов, но иногда бывает необходимо реализовать нечто более сложное. Вам может потребоваться, например, организовать запрос с привлечением более двух полей или использовать сложные вложенные запросы. Или, может быть, вам понадобится сконструировать запрос динамически на основе данных, введенных пользователем в форме поиска. Пришло время поэкспериментировать с мощными запросами с критериями механизма GORM.

Вернемся к приложению Hubbub. Оно позволяет нам маркировать сообщения ключевыми словами; предположим теперь, что нам требуется отыскать все сообщения, созданные за прошедший день и отмеченные словом «Grails». В такого рода случаях запросы с критериями – это то, что вам нужно. В листинге 4.8 приводится реализация простого запроса с критериями.

Листинг 4.8. Запросы с критериями обладают удивительными возможностями

```
def user = User.findByUserId(params.user)

def entries = Post.createCriteria().list {          ← ❶ Создание объекта-запроса
    and {                                           ← ❷ Требует соблюдения всех условий
        eq('user', user)                                ← ❸ Искомые значения
        between('created', new Date()-1, new Date())
        tags {                                      ← ❹ Проверка объектов, связанных отношениями
            eq('name', 'Grails')

    }
    }
```

```
        maxResults(10)                    ← ❺ Ограничения для ResultSet
        order("created", "desc")
    }
```

Образовался целый лес фигурных скобок. Давайте разбираться по порядку. После создания объекта `Criteria` для запрашиваемого класса предметной области ❶ вызывается метод `list`, которому передаются критерии на языке GORM Criteria DSL. В данном случае мы использовали объединение условий по `and` ❷, потому что нам требуется соблюдение всех условий, заключенных в блок `{ }` (в зависимости от логики объединения данных мы могли бы здесь также использовать `or` или `not`).

Затем мы добавляем в запрос несколько параметров, которые называются «ограничениями». В данном случае мы использовали ограничения `eq`, требующее, чтобы объекты `Post` соответствовали существующему объекту `User`, и `between` ❸, ограничивающее поиск диапазоном дат. Заключительный элемент запроса – подзапрос по тегам, гарантирующий, что отбираться будут только сообщения, отмеченные тегом «Grails» ❹.

Закончив создание основного запроса, мы можем добавить другие ограничения. Здесь используются ограничения `maxResults()` и `order()` ❺, но в данном случае мы можем указать поле и направление сортировки в одной инструкции, вместо того чтобы вставлять отдельные ограничения `orderBy` и `sort`.

Где найти полный список доступных ограничений?

Язык GORM Criteria DSL – это лишь тонкая обертка для класса `Restrictions` из библиотеки Hibernate. Если заглянуть в описание класса `Restrictions`, можно найти все доступные варианты, включая `ilike` (для поиска без учета регистра символов), `ge` (больше или равно) и `isNotEmpty` (соответствует полям, имеющим некоторое значение).

Существует несколько сокращенных вариантов записи запросов с критериями. Во-первых, поскольку метод `list()` является наиболее часто используемой операцией, применяемой к экземпляру класса `Criteria`, его вообще можно опустить (хотя мы предпочитаем явно вызывать `list()`, чтобы программный код был более понятным). Запросы с критериями поддерживают также метод `count()`, возвращающий количество соответствующих объектов, и метод `get()`, возвращающий единственный результат.

Запросы с критериями настолько удобны, что со всеми классами предметной области можно даже использовать специальное замыкание `withCriteria`. Это часто обеспечивает более выразительный способ записи запросов в программном коде контроллера:

```
def entries = Post.withCriteria {
    and {
        eq('user', user)
        between('created', new Date()-1, new Date())
    }
}
```

В сокращенной форме записи запросов с помощью `withCriteria()` можно использовать все определения языка Criteria DSL, которые мы видели в более длинной версии `.createCriteria().list()`.

4.3.2. Динамические запросы с использованием запросов с критериями

Объекты критериев существенно упрощают реализацию сложных запросов, кроме того, они прекрасно подходят для создания динамических запросов. Представьте, что нам требуется создать в приложении Hubbub простую форму поиска по профилям пользователей. Мы должны предоставить пользователям форму с полями поиска (чтобы поиск мог производиться по полному имени, адресу электронной почты или домашней страницы) и позволить им выбирать способ объединения критериев поиска по AND, OR или NOT.

Сначала мы создадим простую форму с полями, имена которых соответствуют полям объекта `Profile`, как показано на рис. 4.11.

Рис. 4.11. *Более сложная форма поиска с логическими операторами*

После этого можно написать типичный метод поиска, который будет собирать объект `Criteria` «на лету», опираясь на заполненные пользователем поля формы. Благодаря гибкости объекта `Criteria` мы можем использовать логические операции (and, or, not) при его конструировании. В листинге 4.9 показано, как может выглядеть реализация динамического критерия поиска.

Листинг 4.9. Динамическая форма поиска

```
def advResults = {

    def profileProps =                                  ← ❶ Создает список доступных
        Profile.metaClass.properties*.name                свойств класса Profile
    def profiles = Profile.withCriteria {
        "${params.queryType}" {  ← ❷ Применяет логический оператор

            params.each { field, value ->  ← ❸ Итерации по входным параметрам

                if (profileProps.grep(field)  ← ❹ Ограничения для входных
                      && value) {                 параметров
                    ilike(field, value)
                }
            }
        }
    }
     [ profiles : profiles ]

}
```

Здесь нам пришлось немного потрудиться. Так как платформа Grails добавляет в объект `params` свои значения, помимо полученных из полей формы, нам необходимо извлечь из объекта `params` только те значения, которые имеют отношение к объекту `Profile`. С целью проведения такой фильтрации мы сначала отыскиваем все свойства класса `Profile` ❶, используя свойство `MetaClass`, которым обладают все объекты в языке Groovy. После этого конструируется объект `Criteria`, в котором определяется тип объединения критериев поиска, выбранный пользователем: `and`, `or` или `not` ❷. В заключение выполняются итерации по свойствам объекта `params` ❸, и к тем из них, которые соответствуют свойствам класса `Profile`, применяется оператор `ilike` ❹.

Это достаточно замысловатый случай использования запросов с критериями, поэтому не волнуйтесь, если он покажется вам чересчур сложным. Вам проще будет разобраться в нем, когда вы немного поэкспериментируете с исходным кодом. На рис. 4.12 демонстрируется попытка протестировать форму поиска посредством объединения пары значений ограничением `Or`. Для большей наглядности мы даже использовали шаблонные символы (`%`).

Результаты нашего опыта представлены на рис. 4.13, где использована простая форма вывода.

Несомненно, объекты `Criteria` предоставляют фантастические возможности для построения динамических запросов. Но они обладают еще более мощной особенностью, позволяющей создавать запросы в стиле отчетов. Теперь мы займемся исследованием функциональности `groupBy`.

Рис. 4.12. *Форма поиска с возможностью объединения критериев по AND или OR*

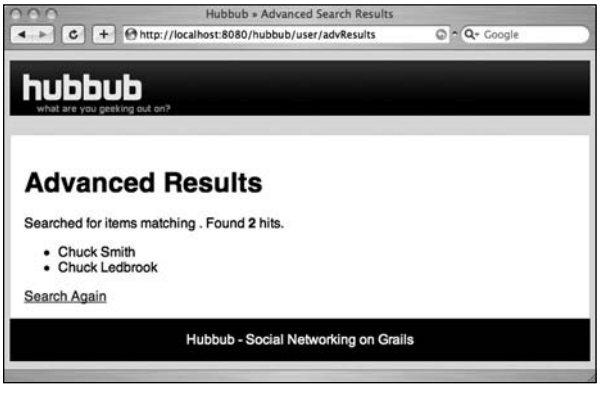


Рис. 4.13. *Простая форма вывода с двумя профилями, совпавшими с нашими критериями*

4.3.3 Создание облака тегов с использованием проекций запросов в виде отчетов

При создании сложных и динамических запросов запросы с критериями проявляются во всем своем блеске, но они могут также выполнять проекции. Проекции – это функции объединения, обобщения и фильтрации, которые могут применяться по окончании выполнения запроса.

Типичный пример использования проекций – вывод сводной информации о данных, полученных по запросу. Например, чтобы построить облако тегов пользователя, необходимо отыскать все его теги и подсчитать количество сообщений, отмеченных каждым из них. Это отличный случай для использования проекции `count()`, как показано в листинге 4.10.

Листинг 4.10. Использование проекций для группировки и подсчета результатов поиска

```
def tagList = Post.withCriteria {                    ← ❶ Создается объект запроса Criteria

    createAlias("user", "u")                         ← ❷ Определяются псевдонимы
    createAlias("tags", "t")                             для последующего использования
    eq("u.userId", "glen")                           ← ❸ Проверка, что объект Post принадлежит
                                                         корректному объекту User
    projections {                                    ← ❹ Применение проекций к объекту ResultSet
        groupProperty("t.name")                      ← ❺ Определяется,
        count("t.id")                                    какие проекции будут применяться
    }
}
```

Проекции являются обычными запросами с критериями, поэтому мы начали с того, что определили запрашиваемую информацию; в данном случае это коллекция объектов `Post` ❶. Поскольку мы выполняем группировку отбираемых сообщений по множеству объектов, нам необходимо определить псевдонимы для обоих полей, `User` и `Tag` ❷. Позднее это позволит нам ссылаться на эти поля в предложении `groupBy`.

Закончив подготовку инфраструктуры, мы приступаем к определению запроса. Первое ограничение `eq` отыскивает все сообщения, принадлежащие пользователю «glen» ❸. Получив все сообщения этого пользователя, мы можем сгруппировать их с помощью предложения `projections` ❹. Мы указали, что группировка должна выполняться по полю `t.name` и что для каждой группы должна выполняться операция `count()` ❺.

Результатом выполнения запроса является список пар ключ/значение, содержащих имена тегов и количество сообщений, отмеченных этими тегами. Все, что осталось сделать, – это преобразовать список в отображение, где имена тегов будут играть роль ключей, а числа – роль значений. Пришло время вспомнить свои навыки программирования на языке Groovy:

```
def tagcloudMap = tagList.inject([ : ]) { map, tag ->
    map[ tag[0] ] = tag[1]; map
}
```

Этот фрагмент создаст отображение имен тегов и их количества, которое можно превратить в облако тегов с помощью библиотеки тегов.

В механизме GORM существует множество проекций. Помимо `count()` можно также использовать `max()`, `min()`, `avg()`, `sum()` и другие разнообразные статистические методы. Проекции механизма GORM – это просто расширенные версии стандартных проекций библиотеки Hibernate. Ознакомьтесь с описанием класса `Projections` из библиотеки Hibernate, в нем вы найдете полный перечень доступных проекций.

Теперь пришло время рассмотреть последний способ выполнения запросов в платформе Grails: язык HQL. Вы не часто будете пользоваться им, но в некоторых крайних случаях он оказывается незаменимым.

Когда следует использовать запросы с критериями, а когда динамические методы поиска?

Выбор между запросами с критериями и динамическими методами поиска сводится к личным предпочтениям разработчика. Мы считаем, что динамические методы поиска обеспечивают фантастическое удобство в большинстве случаев и они применимы для решения многих задач. При их использовании за вами сохраняется возможность определять стратегию загрузки, порядок сортировки или смещение, и, кроме того, они обеспечивают простой и понятный способ определения запросов.

Однако если вам требуется выполнить запрос по трем или более атрибутам, использовать преимущества кэширования запросов или создать запрос, основанный на вложенных отношениях (например, отыскать все объекты `User`, для которых в объекте `Profile` определен адрес электронной почты *glen@glen.com*), выбор в пользу использования запроса с критериями является оптимальным.

Запросы с критериями отлично пригождаются также в случаях, когда необходимо динамически сгенерировать множество ограничений, основанных на данных, полученных от пользователя с формой HTML (например, когда пользователь указывает, по каким полям выполнять поиск и с какими ограничениями, а вы на основе этих данных должны сгенерировать запрос «на лету»). Мы показали пример такой ситуации в разделе 4.3.2.

4.3.4. Непосредственное использование HQL

Если у вас имеется опыт работы с библиотекой Hibernate, то, вероятно, вам уже случалось выражать свои замыслы на языке запросов библиотеки Hibernate (Hibernate Query Language, HQL). Если вам потребуются возможности HQL при создании приложений на платформе Grails, вы сможете напрямую использовать преимущества языка HQL в вызовах методов `find()` и `findAll()`. Язык HQL понадобится вам только для создания наиболее сложных запросов, причем при его использовании придется немного пожертвовать удобочитаемостью и простотой сопровождения программного кода. Но существуют ситуации, в которых он просто незаменим.

Ниже показано, как выглядит типичный запрос HQL:

```
println User.findAll("from User u where u.userId = ?", [ "joe" ])
```

Вы можете использовать знак вопроса (?) для подстановки различных элементов запроса, которые должны изменяться динамически. Последний аргумент в вызове метода – это список подставляемых значений, которыми будут замещаться знаки вопроса (количество элементов

в списке должно соответствовать количеству знаков вопроса). В предыдущем примере для подстановки мы использовали статическое значение «joe», но в действующих приложениях вы, скорее всего, будете применять динамические значения, такие как `params.userId`.

> **Защита от атак типа «инъекция SQL»**
>
> Одной из частых проблем безопасности в Интернете является обеспечение защиты от атак типа «инъекция SQL». При нападениях такого типа злоумышленник передает в запросе злонамеренные параметры с целью изменить запрос, направляемый базе данных (например, `...?userId=abc123;delete+from+users;`). По этой причине вы никогда не должны объединять строки запроса HQL со строками параметров. Если вы используете в запросах HQL символ подстановки (`?`), тогда вам всегда необходимо предусматривать меры защиты от подобных атак.

Однако вопросительные знаки вопроса в запросе не особенно удобочитаемы. Если вам когда-либо приходилось работать с HQL, возможно, вы уже знаете о существовании *именованных параметров*. Они позволяют заменить знаки вопроса более удобочитаемыми идентификаторами, начинающимися с символа двоеточия (`:`). При их использовании вы должны указывать соответствующее отображение пар ключ/значение в конце списка параметров в вызове метода. Попробуем воспроизвести тот же запрос с применением именованных параметров:

```
User.findAll("from User u where u.userId = :uid", [ uid: "joe" ])
```

Если запрос применяется внутри цикла постраничного вывода данных, возможно, с помощью тега `<g:paginate>`, в дополнение к обычным именованным параметрам можно использовать поля `max:` и `offset:`:

```
User.findAll("from User u where u.userId like :uid",
      [ uid: "j%", max: params.max, offset: params.offset ] )
```

Язык HQL весьма мощный, и в некоторых ситуациях он будет незаменим. Однако существует не так много вещей, которые под силу сделать средствами HQL, но невозможно с помощью запросов с критериями, поэтому HQL, вероятно, не должен быть вашим первым выбором.

На этом мы завершаем наш скоростной тур по средствам выполнения запросов в платформе Grails. Осталось исследовать лишь один аспект моделей предметной области – начальную загрузку исходных данных.

4.4. Начальная загрузка исходных данных

Иногда бывает полезно заполнять базу данных исходными данными при ее создании. Например, может быть желательно создавать учет-

ную запись администратора с предопределенным паролем для обеспечения возможности настройки системы. Но что делать, если таблицы базы данных создаются самой платформой Grails?

Ответ находится в файле /grails-app/conf/BootStrap.groovy. В этом файле имеются два замыкания, `init` и `destroy`, которые используются платформой Grails при запуске системы и при ее остановке соответственно.

Обычно в разных ситуациях процесс начальной загрузки выполняется немного по-разному (например, на этапе разработки могут создаваться учетные записи мнимых пользователей, а в действующей системе – нет). Платформа Grails позволяет адаптировать файл начальной загрузки к текущим потребностям.

В файле Bootstrap.groovy в листинге 4.11 мы создаем учетную запись администратора для использования на этапе разработки, но пропускаем эту операцию при запуске действующей системы.

Листинг 4.11. Заполнение базы данных исходной информацией в зависимости от ситуации

```
import com.grailsinaction.*
import grails.util.Environment

class BootStrap {

    def init = { servletContext ->

        switch (Environment.current) {  ❶ Проверка окружения

            case Environment.DEVELOPMENT:
                createAdminUserIfRequired()  ❷ Вызов методов в зависимости от окружения
                break;

            case Environment.PRODUCTION:
                println "No special configuration required"
                break;

        }
    }

    def destroy = {
    }

    void createAdminUserIfRequired() {
        if (!User.findByUserId("admin")) {  ❸ Проверить наличие учетной записи

            println "Fresh Database. Creating ADMIN user."
            def profile = new Profile(email: "admin@yourhost.com")
            def user = new User(userId: "admin",
                password: "secret", profile: profile).save()
        } else {
```

```
                println "Existing admin user, skipping creation"
            }
        }
    }
```

Обратите внимание, что выполняется проверка текущего окружения ❶ и учетная запись администратора создается, только если система запускается в среде разработки (то есть когда приложение запускается командой `grails run-app`) ❷. Кроме того, прежде чем создать учетную запись, мы проверяем ее наличие в базе данных ❸. Даже хотя стандартный драйвер базы данных HSQLDB, размещаемой в памяти, гарантирует, что при каждом запуске будет создаваться новая пустая база данных, есть вероятность, что позднее будет произведен переход на использование постоянной базы данных, такой как MySQL. Всегда выполняйте предварительные проверки, прежде чем вставлять образцовые данные.

И последнее, что следует заметить по поводу начальной загрузки, – у вас нет никаких гарантий, что будет вызван метод `destroy()`. Это замыкание не будет вызвано, если, например, сервер аварийно завершит свою работу или произойдут другие неприятности. Мы предупредили вас.

4.5. В заключение

В этой главе мы исследовали механизм скаффолдинга, реализованный в платформе Grails, и посмотрели, как с его помощью можно быстро создавать пользовательский интерфейс приложения. Мы также посмотрели, как изменять внешний вид приложения с помощью CSS и макетов и как настраивать шаблоны и сообщения об ошибках. Мы даже добавили свою собственную функцию поиска непосредственно в сгенерированные шаблоны, для того чтобы продемонстрировать гибкость механизма скаффолдинга.

Применяя наши знания механизма скаффолдинга, мы продемонстрировали множество приемов выполнения запросов в платформе Grails. Динамические методы поиска прекрасно подходят для создания запросов на основе двух полей и повышения удобочитаемости. Для случаев, когда возможностей динамических методов поиска оказывается недостаточно, мы рассмотрели улучшенные приемы выполнения запросов, использующие запросы с критериями, проекции и даже HQL.

Повторим некоторые из ключевых идей этой главы, которые вам стоит запомнить и применять в своих приложениях на платформе Grails:

- *Используйте механизм скаффолдинга для получения вознаграждения и дополнительного стимула*. Скаффолдинг даст вам ощущение движения вперед и позволит вам сконцентрироваться на дальнейшей разработке приложения. Кроме того, не бойтесь использовать

код, сгенерированный механизмом скаффолдинга, для реализации страниц административного раздела в готовом приложении.

- *Изучите свои параметры настройки*. Вы можете сделать достаточно много, чтобы интегрировать страницы, сгенерированные механизмом скаффолдинга, с вашими собственными страницами. Используйте возможности, предоставляемые CSS и макетами, для придания им внешнего вида, гармонирующего с внешним видом остальных страниц приложения.
- *Динамические методы поиска прекрасно подходят для создания запросов на основе двух полей*. Динамические методы поиска имеют говорящие имена, и их использование облегчит понимание логики работы приложения другими разработчиками. Если вам потребуется выполнить запрос по большему количеству полей, для этого лучше подойдут запросы с критериями.
- *Пользуйтесь консолью платформы Grails*. Консоль платформы Grails идеально подходит для экспериментов с динамическими методами поиска и позволяет опробовать множество вариантов без повторной компиляции.
- *Используйте широкие возможности запросов с критериями*. Запросы с критериями обладают более мощными возможностями, чем динамические методы поиска, но при этом они более сложны. Для группировки и обобщения данных в запросе используйте проекции.
- *Всегда используйте именованные параметры HQL*. Применение именованных параметров позволяет избежать опасности атак типа инъекции SQL. Никогда не собирайте строки запросов HQL динамически (используйте для этого запросы с критериями).
- *Выполняйте начальную загрузку данных по условию*. Grails позволяет выполнять различные подготовительные операции для каждого окружения, поэтому пользуйтесь этой возможностью. Не предполагайте, что база данных с учетными записями пользователей на этапе разработки находится в памяти; всегда проверяйте наличие образцовых данных, прежде чем создавать их.

В следующей главе мы применим наши умения, касающиеся модели предметной области, для работы с формами и контроллерами на уровне пользовательского интерфейса. Это будет перерыв в написании модульных тестов.

В этой главе:

- Взаимодействие контроллеров, моделей и представлений
- Собственные URL и постоянные ссылки
- Фильтры для перенаправления запросов
- Использование служб
- Привязка данных и обработка ошибок
- Использование управляющих объектов
- Выгрузка и отображение изображений

5

Управление потоком выполнения приложения

В главах 3 и 4 мы потратили немало времени на исследование того, как платформа Grails работает с объектами предметной области. Попутно мы пользовались контроллерами и реализовали несколько представлений GSP для отображения форм, но принцип действия контроллеров мы еще не рассматривали. Займемся этим прямо сейчас.

В этой главе мы сконцентрируем свое внимание на контроллерах. Мы познакомимся с более современными способами привязки данных для передачи переменных между формами и контроллерами, посмотрим, как действует механизм перенаправления, реализуем операцию выгрузки файлов, коснемся фильтров и даже попробуем отобразить содержимое своего собственного типа. По ходу дела мы добавим в приложение Hubbub возможность выгрузки и отображения фотографий в профилях и реализуем основную логику создания объектов `User` и `Post`. К концу главы вы будете готовы приступить к созданию в своих приложениях наиболее типичных, а также нескольких редко встречающихся контроллеров.

Но начнем мы главу с объединения всех уже полученных знаний о контроллерах, формах и классах предметной области. Мы дополним приложение Hubbub возможностью добавлять и отображать сообщения пользователей в хронологической последовательности в стиле сайта Twitter.

5.1. Основы контроллеров

До сих пор в наших исследованиях экосистемы Grails мы были сосредоточены в основном на модели (сохранении и извлечении объектов предметной области) и оставляли без внимания особенности взаимодействий между контроллерами, представлениями и библиотеками тегов.

Теперь мы более детально рассмотрим взаимодействия форма–контроллер с упором на контроллеры.

Представьте, что нам требуется отобразить последовательность последних сообщений пользователя. Для этого нам придется реализовать действие контроллера (например, метод `timeline` в контроллере `Post`), которое будет извлекать сообщения пользователя и передавать их представлению (/views/post/timeline.gsp) для отображения в броузере. Как мы уже знаем из главы 1, платформа Grails обеспечивает доступ к контроллерам как к адресам URL, используя соглашения об именовании. На рис. 5.1 показано, как платформа Grails преобразует имена методов действий контроллера в адреса URL.

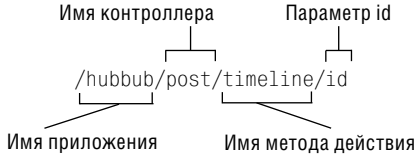


Рис. 5.1. *Порядок преобразования имен методов действий контроллеров в адреса URL*

Когда пользователь укажет в броузере адрес, например *http://localhost:8080/hubbub/post/timeline/chuck_norris*, платформа Grails вызовет действие `timeline` контроллера `Post` и при этом передаст значение `chuck_norris` в параметре `id`. Как правило, это действие будет извлекать некоторую информацию из базы данных и передавать ее представлению для отображения. В соответствии с соглашениями, принятыми в платформе Grails, имя представления совпадает с именем контроллера (но с расширением .gsp), таким образом, действие `timeline` будет извлекать данные и передавать их для отображения представлению /views/post/timeline.gsp. Данный процесс изображен на рис. 5.2.

Работу этого процесса в действии вы могли наблюдать в главе 1 и в примере формы поиска в главе 4. Давайте воспользуемся нашими знаниями о контроллерах, чтобы добавить в приложение Hubbub вывод сообщений.

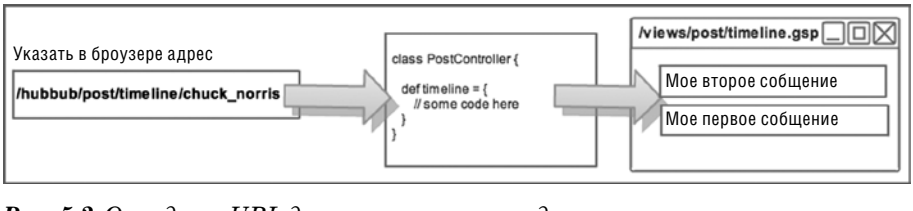


Рис. 5.2. *От адреса URL до контроллера и представления*

5.1.1. Реализация вывода сообщений в хронологическом порядке для приложения Hubbub

Последовательность сообщений должна включать в себя не только сообщения данного пользователя, но и сообщения других пользователей, участвующих в дискуссии. Но давайте начнем с малого и реализуем отображение всех сообщений одного пользователя.

Начнем с реализации метода действия `timeline()` в контроллере `PostController`. В листинге 5.1 приводится первая реализация вывода сообщений пользователя:

Листинг 5.1. Добавление метода действия timeline() в класс PostController

```
package com.grailsinaction

class PostController {

    def scaffold = true

    def timeline = {
        def user = User.findByUserId(params.id) ← ❶ Отыскать пользователя
                                                  по параметру id
        [ user : user ] ← ❷ Передать пользователя
                          в представление
    }
}
```

Как уже говорилось в предыдущих главах, методы действий обычно выполняют обработку параметров форм и используют их для взаимодействий с моделью данных. В данном случае необязательный параметр `id`, в котором хранится идентификатор пользователя, используется для извлечения объекта `User` из базы данных ❶. В окончательной версии приложения в этом месте уместно было бы выполнить проверку на пустое значение. Обработку ошибок мы добавим ниже в этой же главе.

После получения объекта `User` результат работы метода помещается в отображение, которое возвращается представлению ❷, оформляемому, как правило, в виде файла GSP. Представление может обращаться к любым объектам в отображении для вывода их в окне броузера. Мы могли бы явно указать инструкцию `return` в конце метода, но в данном случае в этом нет необходимости. Интерпретатор Groovy всегда воспринимает последнюю строку замыкания или метода как возвращаемое значение, поэтому программисты для платформы Grails обычно опускают инструкцию `return` в методах действий.

Теперь мы можем извлекать из базы данных информацию о пользователе, поэтому впору приступить к реализации представления, выполняющего отображение информации. Следуя соглашениям в платформе Grails, мы поместим реализацию представления в файл с тем же именем, что и у действия контроллера. В данном случае файл представле-

Необязательный параметр id

Вероятно, вам еще не приходилось наблюдать поле `id` в действии. В соответствии с соглашением, как было показано на рис. 5.1, действия контроллеров отображаются в адреса URL вида /контроллер/действие/id, где `id` передается методу действия в виде поля `params.id`. Поле `id` имеет свободный формат, что дает отличную возможность определения постоянных ссылок.

Например, обращение по адресу /user/profile/chuck_norris приведет к вызову действия `profile()` контроллера `User` со значением `chuck_norris` в поле `params.id`. Именно таким способом создавались постоянные ссылки до того, как в платформе Grails появилась возможность определять собственные отображения адресов URL, о чем мы будем рассказывать в разделе 5.7 этой главы.

В формах, созданных механизмом скаффолдинга платформы Grails, поле `id` используется для представления идентификатора в базе данных, поэтому вы часто будете видеть такие адреса URL, как /user/show/57, который в данном случае мог бы использоваться для отображения информации о пользователе со значением `id`, равным 57, в базе данных. Это практично, но выглядит не очень привлекательно.

ния будет называться /grails-app/views/post/timeline.gsp. В листинге 5.2 приводится первая реализация отображения сообщений пользователя в хронологическом порядке.

Листинг 5.2. Отображение сообщений пользователя

```
<html>
    <head>
        <title>
            Timeline for ${user.profile.fullName}
        </title>                                              ← ❶
        <meta name="layout" content="main"/>                    Доступ к вложенным
    </head>                                                     объектам предметной
    <body>                                                      области
        <h1>Timeline for ${user.profile.fullName}</h1>

        <div class="allPosts">
            <g:each in="${user.posts}" var="post"> ← ❷ Итерации по сообщениям
                <div class="postEntry">                  пользователя
                    <div class="postText">
                        ${post.content}
                    </div>                               ← ❸ Отображение каждого сообщения
                    <div class="postDate">
                        ${post.dateCreated}
                    </div>
```

```
                </div>
            </g:each>
        </div>
    </body>
</html>
```

Используя объект `User`, который был помещен контроллером в область видимости запроса (о чем будет рассказываться в разделе 5.1.3), мы можем читать значения полей объекта `User` и отображать данные в броузере. В данном примере мы извлекаем объект `Profile`, принадлежащий пользователю, и отображаем полное имя ❶.

В области видимости запроса мы можем обращаться к объектам, заключая их имена в конструкцию `${}` языка Groovy. Если представлению передается сложный объект (например, объект предметной области, связанный отношениями с другими объектами предметной области), к его элементам можно обращаться, как и в любом другом случае. В листинге 5.2 мы отображаем `${user.profile.fullName}` ❶.

Кроме того, мы выполняем итерации по всем объектам `Post`, принадлежащим объекту `User` ❷, отображая содержимое и значение поля `dateCreated` каждого сообщения ❸.

На рис. 5.3 показано новое представление `timeline` в действии.

Теперь, когда у нас имеется представление `timeline`, отображающее все сообщения пользователя, можно попробовать реализовать добавление новых сообщений. Для этого нам потребуется создавать и сохранять объекты предметной области.

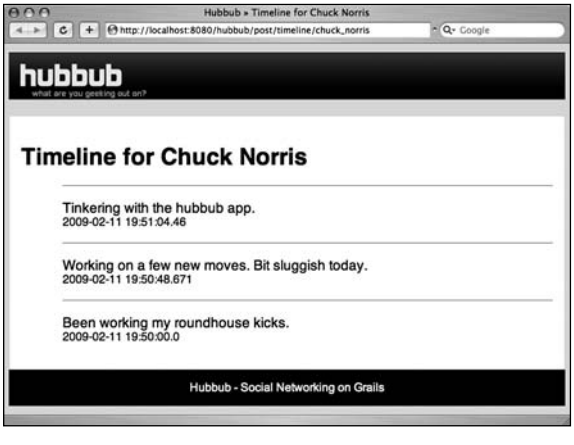


Рис. 5.3. *Первая версия нашего представления timeline в действии*

Нарушение соглашений об именовании представлений

В случаях, когда нежелательно, чтобы имя представления совпадало с именем действия, можно использовать метод `render()`. Например, если необходимо сохранить представление, действующее в паре с методом `timeline()`, в файле user_timeline.gsp, его имя можно передать методу `render()` в аргументе `view`:

```
render(view: "user_timeline", model: [ user: user ])
```

Обратите внимание, что при ссылке на представление расширение .gsp опускается.

5.1.2. Добавление новых сообщений

В главе 3 мы изучили, как создавать и сохранять объекты `Post` внутри модульного теста. Теперь, когда мы немного представляем, как действует контроллер, применим наши знания для создания пользовательского интерфейса.

Прежде всего расширим наше представление, предоставив пользователю возможность добавлять новые сообщения. Нам требуется получить нечто, напоминающее рис. 5.4.

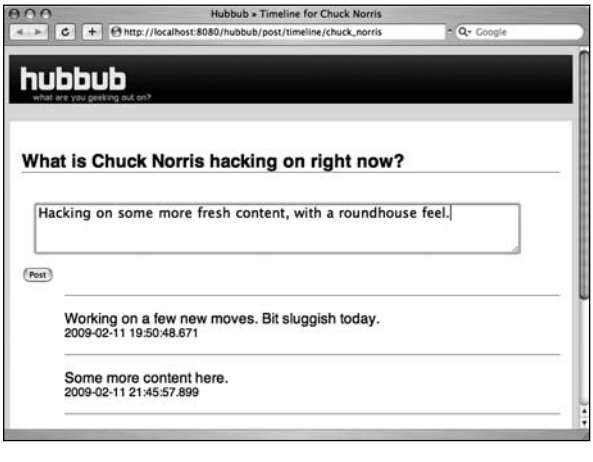


Рис. 5.4. *Возможность добавления новых сообщений в приложении Hubbub*

Для этого нам требуется добавить форму в файл timeline.gsp, куда можно будет вводить текст нового сообщения. И нам необходим компонент `textArea`, чтобы пользователь мог вводить текст сообщения. В листинге 5.3 приводится элемент `div` с формой ввода, который мы добавим в файл timeline.gsp.

Листинг 5.3. Добавление формы ввода нового сообщения

```
<div id="newPost">

  <h3>
    What is ${user.profile.fullName} hacking on right now?
  </h3>

  <p>
    <g:form action="addPost" id="${params.id}"> ← ❶ Сохраняет id из текущего URL
      <g:textArea id='postContent' name="content" ← ❷ Компонент textArea для
        rows="3" cols="50"/><br/>                    ввода текста сообщения
      <g:submitButton name="post" value="Post"/>
    </g:form>
  </p>
</div>
```

В листинге 5.3 мы добавили в страницу новую форму, использовав тег `<g:form>` и указав в качестве цели действие `addPost` ❶. Мы передаем форме текущее значение поля `id`, чтобы сообщить методу действия `addPost` идентификатор пользователя, вызвавшего ее. Например, при обращении к форме пользователем «chuck_norris» адрес URL приобретет вид /post/addPost/chuck_norris.

Мы также добавили тег `<g:textArea>`, чтобы позволить пользователю ввести текст сообщения ❷. Поскольку элемент управления имеет имя `content`, мы сможем использовать параметр `params.content` где-нибудь в логике контроллера.

В листинге 5.4 приводится дополненная реализация контроллера `PostController`, который теперь обрабатывает добавление новых сообщений в список.

Листинг 5.4. Измененная версия контроллера PostController, обрабатывающая добавление новых сообщений

```
package com.grailsinaction

class PostController {

    def scaffold = true

    def timeline = {
        def user = User.findByUserId(params.id)
        [ user : user ]
    }

    def addPost = {
        def user = User.findByUserId(params.id) ← ❶ Отыскивает пользователя
                                                  по значению id
        if (user) {
            def post = new Post(params) ← ❷ Запись данных в новый объект Post
```

```
            user.addToPosts(post) ← ❸ Связывает новое сообщение с пользователем
            if (user.save()) { ← ❹ save() вернет false, если проверка
                                   в объекте Post окончится неудачей
                flash.message = "Successfully created Post"   ← ❺ Информирует
            } else {                                              пользователя
                user.discard()                                    об успехе
                flash.message = "Invalid or empty post"           или неудаче
            }
        } else {
            flash.message = "Invalid User Id"
        }
        redirect(action: 'timeline', id: params.id) ← ❻ Возвращает пользователя
    }                                                   к списку сообщений

}
```

Когда в главе 3 мы исследовали возможность сохранения объектов предметной области, мы писали интеграционные тесты. Теперь мы выполняем те же операции в действующей форме. Как и действие `timeline`, действие `addPost` начинается с извлечения объекта `User` по его идентификатору ❶, но на этот раз мы добавили простейшую обработку ошибок, проверяющую существование пользователя.

Если объект `User` присутствует в базе данных, создается новый объект `Post`, при этом конструктору передается отображение `params` ❷. Как вы уже знаете из главы 2, когда конструктору передается отображение, интерпретатор Groovy связывает свойства отображения с полями объекта, пропуская те, для которых нет соответствия. В данном случае свойство `params.content` будет отображено в поле `content` объекта `Post`. На рис. 5.5 показано, как поле `content` объекта `params` отображается в одноименное поле объекта `Post`. На языке платформы Grails этот процесс называется *привязкой данных*.

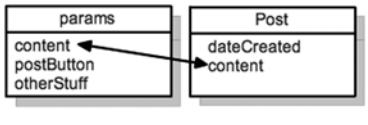


Рис. 5.5. Отображение объекта params в объект Post

После привязки данных формы к новому объекту `Post` он добавляется в объект `User` ❸. В заключение выполняется попытка сохранить его вызовом метода `save()`, который проверяет соответствие ограничениям, определенным в объекте `Post` ❹.

Независимо от того как пройдет операция сохранения, будет ли она выполнена успешно или потерпит неудачу, мы держим пользователя в курсе событий благодаря обратной связи, обеспечиваемой `flash.message` ❺. Мы рассмотрим область видимости кадра (flash scope) в следу-

ющем разделе, но в двух словах скажем, что это один из способов сообщить пользователю некоторую информацию. Ниже в этой же главе мы продемонстрируем более надежный вариант обработки ошибок, возникающих в ходе проверок, а пока будем использовать `flash.message`.

Когда все сделано, выполняется перенаправление пользователя к действию `timeline` ❻, которое отобразит список сообщений, включая и новое. Обратите внимание, что при перенаправлении передается параметр `id`. Это важно, потому что логика работы действия `timeline` опирается на идентификатор пользователя, и когда перенаправление выполняется программным способом (а не в результате ввода адреса URL в броузере), необходимо явно указывать значение идентификатора.

Теперь нам требуется внести изменения в файл timeline.gsp, чтобы обеспечить вывод сообщения об успехе или неудаче. В следующем фрагменте представлено дополнительное содержимое элемента `div`, предназначенного для вывода сообщения:

```
<g:if test="${flash.message}">
    <div class="flash">
        ${flash.message}
    </div>
</g:if>
```

Нам требуется добавить оформление элементу `div` с помощью CSS, чтобы он выглядел, как предупреждающее сообщение. Если вы загрузите исходные тексты примеров для этой главы, вы найдете необходимое оформление CSS, воспроизводящее эффект, изображенный на рис. 5.6.

Теперь, когда у нас предусмотрена обработка ошибок, можно попробовать добавить неправильное сообщение. На рис. 5.6 изображен результат попытки добавить пустое сообщение. Наша проверка сработала, и на этом реализацию функции добавления новых сообщений в приложении Hubbub можно считать законченной.

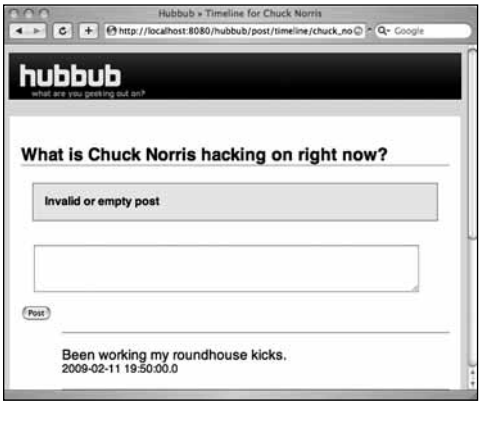


Рис. 5.6. *Попытка добавить пустое сообщение приводит к ошибке*

Итак, в нашем приложении мы реализовали две простые версии основных функций приложения Hubbub: отображение сообщений пользователя в хронологическом порядке и добавление новых сообщений. Кроме того, мы применили наши знания о том, как выполняется создание объектов предметной области (из главы 3) и как используются динамические методы поиска для их извлечения (из главы 4).

Но кое-что остается пока неисследованным. Откуда берутся все эти объекты `flash`, и что они делают? И почему мы не использовали стандартные сообщения об ошибках, которые мы видели в главе 4? Не волнуйтесь. К концу главы вы поймете все эти особенности реализации контроллеров.

Начнем с объекта `flash`, который имеет отношение к тому, как реализованы области видимости в платформе Grails.

5.1.3. Области видимости

Вы уже знаете, что методы действий контроллеров могут возвращать отображение для передачи информации представлению, и вы видели, что для передачи информации может также использоваться объект `flash`. Такая передача информации и срок жизни переменных, в которых она передается, называются *областью видимости* переменной.

Для хранения информации платформа Grails поддерживает различные области видимости, и каждая из них продолжает существовать на протяжении определенного отрезка времени. Когда производится передача данных из контроллера в представление в виде отображения, неявно используется *область видимости запроса*. Однако платформа Grails позволяет явно ссылаться на различные области видимости, чтобы обеспечить возможность сохранения данных на столь долгий срок, какой вам потребуется. В частности, платформа Grails предлагает четыре специальных области видимости в форме отображений, к которым можно обратиться по их именам из любого метода действия контроллера. Они перечислены в табл. 5.1.

Примечание

Имена `request`, `flash`, `session` и `servletContext` в действительности не обозначают настоящие отображения Java. Например, `request` – это Java-объект `HttpServletRequest`. Однако базовые объекты в платформе Grails были расширены посредством реализации через свойство `MetaClass` дополнительных операций, обеспечивающих сходство объектов с отображениями, что существенно упрощает работу программиста.

Если прежде вам приходилось создавать веб-приложения на языке Java, вы, вероятно, сталкивались с областями видимости запроса (`request`), сеанса (`session`) и сервлета (`servletContext`). Область видимости кадра (`flash`) в платформе Grails является уникальной, и во многих случаях она просто незаменима. На рис. 5.7 изображено относительное время

существования различных областей видимости. Далее мы рассмотрим каждую из них в отдельности.

Таблица 5.1. Платформа Grails поддерживает четыре контекста сохранения данных для их передачи между контроллерами и формами

Переменная области видимости	Время жизни данных в этой области видимости
`request`	Существует, пока текущее представление GSP не закончит отображение данных
`flash`	Возникает при переходе к следующей странице и существует до перехода к последующей
`session`	Существует, пока текущий пользователь не закроет броузер
`servletContext`	Существует, пока приложение не будет перезапущено (это отображение совместно используется всеми пользователями)

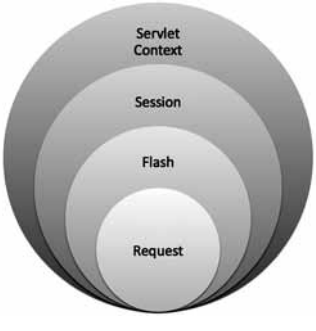


Рис. 5.7. *Срок жизни области видимости запроса самый короткий, области видимости servletContext – самый продолжительный*

Область видимости запроса

Область видимости запроса (request scope) отлично подходит для сохранения данных, совместно используемых контроллером и представлением (обычно – страницей GSP). В таком случае говорят, что сервер «отправил» запрос странице (в том смысле, что адрес URL в адресной строке броузера не изменяется, а все необходимые операции выполняются внутри сервера). Даже не зная об этом, мы использовали область видимости запроса всякий раз, когда возвращали отображение из методов действий контроллеров. Область видимости запроса сохраняет данные, пока представление не завершит их отображение.

Но как быть в случае с методом `addPost()`? Сначала мы выполняем тяжелую работу по добавлению нового сообщения, а затем нам необходимо перенаправить броузер, чтобы вывести последовательность сообщений пользователя и отобразить сообщение об успешном выполнении операции. В эту последовательность действий вовлекается броузер, а это означает прекращение существования области видимости запроса.

Для таких случаев платформа Grails предоставляет *область видимости кадра* (flash scope).

Область видимости кадра

Элементы отображения `flash` способны пережить одно (и только одно) перенаправление, после чего они удаляются. Это в точности соответствует потребностям действия `addPost()`. Мы можем безбоязненно поместить сообщение об успешном добавлении в отображение `flash` и быть уверенными, что оно переживет перенаправление в действие `timeline`.

Как и все остальные отображения областей видимости, отображение `flash` является универсальным отображением, но принятые соглашения требуют, чтобы подобные сообщения передавались в пользовательский интерфейс в виде атрибута `flash.message`. Как правило, вам придется видеть в файлах GSP обращения к области видимости кадра, обернутые в условные операторы, как в нашем примере выше. Ниже приводится фрагмент кода, который вы часто будете встречать в файлах GSP:

```
<g:if test="${flash.message}">
    <div class="flash">${flash.message}</div>
</g:if>
```

Для сохранения объектов, которые должны переживать не только одно перенаправление, вам потребуется исследовать область видимости сеанса (session scope).

Область видимости сеанса

Следующей по продолжительности жизни является *область видимости сеанса* (session scope). Объекты, сохраняемые в области видимости сеанса, продолжают существовать, пока пользователь не закроет свой броузер. За кулисами поддержка сеансов реализуется на сервере с использованием cookie с именем JSESSIONID, срок жизни которого истекает, когда пользователь закрывает окно броузера (или когда сервер закрывает сеанс после продолжительного бездействия пользователя).

Сеансы имеют свойство размножаться в кластерных средах, поэтому следует стараться свести к минимуму объем информации, сохраняемой в них. Типичным примером использования сеанса является помещение в сеанс объекта `User` после регистрации пользователя на сайте, поэтому обязательно ознакомьтесь с тем, как действуют обособленные объекты предметной области (смотрите врезку).

Проблемы с сеансами: тайна обособленных объектов

Если сохранить объект предметной области в области видимости сеанса (например, объект `User` после входа пользователя в систему), этот объект обретет независимость от сеанса Hibernate, что означает утрату возможности обращаться к любым объектам, связанным с ним отношениями, для которых применяется стратегия отложенной загрузки. Например, вы не сможете вызвать метод `session.user.following.each {}`. Чтобы вновь привязать объект (например, чтобы получить возможность навигации по связанным с ним объектам), следует использовать метод `attach()`, доступный во всех классах предметной области. Для нашего объекта `User` это выглядит так:

```
def user = session.user
if (!user.isAttached()) {
    user.attach()
}
user.following.each { nextFollowing -> /* выполнить какие-либо
операции */ }
```

Область видимости сервлета (приложения)

Последней из доступных областей видимости является *область видимости сервлета* (servletContext scope), которую иногда называют *областью видимости приложения*. Это отображение совместно используется всеми компонентами приложения и его удобно использовать для хранения информации, не зависящей от конкретного пользователя (например, для сохранения количества пользователей, зарегистрировавшихся с момента запуска приложения). Данная область видимости полезна также для загрузки ресурсов из самого веб-приложения (например, `servletContext.getResourceAsStream("/images/mylogo.gif")`). За дополнительной информацией обращайтесь к стандартной документации с описанием области видимости servletContext.

Примечание

Существуют также области видимости потока (flow scope) и диалога (conversation scope), но мы отложим их обсуждение до главы 9, где мы познакомимся с веб-потоками платформы Grails.

Теперь мы понимаем, как действуют области видимости, и разобрались с тайной объекта `flash`. Данные, которые передаются представлению для отображения, сохраняются в области видимости запроса; данные, которые должны преодолеть перенаправление, сохраняются в области видимости кадра; в области видимости сеанса сохраняются дол-

гоживущие данные о конкретном пользователе; а в области видимости приложения сохраняются долгоживущие данные, имеющие отношение к приложению в целом.

Пришло время заняться исследованиями взаимодействий между контроллерами, обеспечивающими возможность перемещения по приложению. В нашем действии `addPost()` мы вызывали метод `redirect()`, так что давайте познакомимся с потоками и с тем, как контроллеры выполняют перенаправление.

5.1.4. Методы действий по умолчанию

Платформа Grails позволяет реализовать в любом контроллере метод действия `index()`, который будет использоваться по умолчанию. Если пользователь обратится к контроллеру, не указывая имя действия (например, при обращении по адресу /hubbub/post), запрос будет обрабатываться методом `index()` (который обычно направляет пользователя к какому-то другому действию).

Давайте переопределим действие `index()` в контроллере `PostController` так, чтобы при попытках пользователей обратиться по адресу /hubbub/post, они немедленно перенаправлялись к действию `timeline()`. Дополненная реализация контроллера `PostController` приводится в листинге 5.5.

Листинг 5.5. Перехват действия index() и перенаправление пользователя

```
package com.grailsinaction

class PostController {

  def scaffold = true

  def index = {
    if (!params.id)
      params.id = "chuck_norris"
      redirect(action: 'timeline', params: params) ← Передача параметров через перенаправление
  }
  // здесь находится остальной программный код реализации контроллера
}
```

Одним из недостатков операции перенаправления является утрата отображения `params`, если оно явно не передается действию, перенаправление в которое производится. Если бы мы явно не передали объект `params` из действия `index()`, тогда действие `timeline()` могло бы получить произвольные значения в объекте `params`, в том числе и значения `null`.

Основное назначение действия `index()` заключается в перенаправлении, но мы пока не описали возможные варианты реализации перенаправления. Итак, пришло время их исследовать.

Мир альтернативных действий по умолчанию

Мы рассмотрели пример использования действия `index()` для выполнения операций по умолчанию, когда обращение происходит по адресу URL контроллера. Однако существует множество способов реализации действий по умолчанию.

Например, если в контроллере определен единственный метод действия, он будет интерпретироваться как действие по умолчанию. Если бы в листинге 5.5 метод `timeline()` был единственным методом контроллера, мы могли бы опустить реализацию метода `index()`, и при этом контроллер был бы способен обрабатывать обращения по адресу /hubbub/post.

Если в вашем контроллере определено несколько методов действий, вы можете явно объявить действие по умолчанию в начале определения контроллера:

```
def defaultAction = 'timeline'
```

Использование действия `index()` является довольно типичным, поэтому мы сначала продемонстрировали именно этот прием. Однако в большинстве подобных случаев предпочтительнее использовать объявление `defaultAction`, поскольку при этом вам не придется переупаковывать объект `params` и выполнять перенаправление клиента.

5.1.5. Способы перенаправления

Повсюду в нашем программном коде мы использовали метод `redirect()`. Например, после вызова метода `addPost()` мы отправляли пользователя обратно к действию `timeline`. При вызове действия по умолчанию `index()` мы выполняли перенаправление в другое действие.

Все наши обращения к методу `redirect()` следуют определенному шаблону, например:

```
redirect(action:'timeline')
```

При такой форме перенаправления действие с указанным именем должно быть определено в текущем контроллере.

Но что если нам требуется вызвать действие в другом контроллере? Допустим, после регистрации пользователя в контроллере `User` нам необходимо направить его в действие `timeline` контроллера `Post`. В таких ситуациях можно использовать расширенную версию метода `redirect()`:

```
redirect(controller: 'post', action:'timeline', id: newUser.userId)
```

Поле id является необязательным, как и имя действия. Если имя действия не указано, будет вызвано действие контроллера по умолчанию (как обсуждалось в предыдущем разделе).

Если при перенаправлении требуется передать дополнительные параметры, это можно сделать с помощью отображения params:

```
redirect(controller: 'post', action:'timeline',
    params: [fullName: newUser.profile.fullName,
        email: newUser.profile.email ]
        )
```

Наконец, имеется возможность выполнить перенаправление с использованием параметров uri: и url:, которые выполняют перенаправление по относительному и абсолютному адресам соответственно. Например, следующий фрагмент

```
redirect(controller: 'post', action:'timeline')
```

можно переписать с использованием параметра uri::

```
redirect(uri: '/post/timeline')
```

А если понадобится выполнить перенаправление на внешний ресурс, можно воспользоваться параметром url::

```
redirect(url: 'http://www.google.com?q=hubbub')
```

Это все, что вам следует знать о возможных способах перенаправления. Мы часто будем использовать их, поэтому вам следовало познакомиться с ними.

Наш следующий шаг заключается в том, чтобы взять все, что было реализовано в этой главе, и перенести в более удобную для сопровождения архитектуру. Одним из наиболее мощных инструментов, позволяющих сделать это, являются *службы Grails*.

5.2. Службы: повышаем надежность и удобство сопровождения приложения

В этой главе мы познакомились с большим количеством мощных приемов: со всеми видами привязки данных, с определением пути к контроллеру и с волшебством перенаправления. Но прежде чем столкнуться с трудностями, мы должны изучить некоторые приемы, которые обеспечат удобство сопровождения приложения на долгое время. В этом разделе мы узнаем, насколько сильно применение служб может упростить реализацию контроллеров.

К настоящему моменту в контроллере PostController мы реализовали действие addPost(), которое создает новые сообщения. Позднее нам потребуется создавать сообщения из множества других мест (в частности,

из службы REST и из шлюза Jabber). Это означает необходимость повторной реализации логики во всех этих местах. Но поскольку мы стремимся следовать принципу DRY (Don't Repeat Yourself – избегай повторений), такое повторение выглядит нежелательным.

В этом разделе мы переместим все функциональные возможности, связанные с созданием новых сообщений, в службу Grails, которую затем мы сможем вызывать откуда угодно. Это сделает приложение более чистым и более простым в сопровождении.

5.2.1. Реализация службы PostService

Пришло время выделить наши операции с объектами `Post` в отдельную службу `PostService`, к которой мы могли бы обращаться из любого места в приложении. Первая наша встреча со службами состоялась в главе 1, где мы реализовали простую службу цитат, чтобы отделить процедуру поиска цитат от их вывода. Службы обеспечивают простой, легко сопровождаемый и контролируемый способ инкапсуляции логики выполнения. Службы могут участвовать в транзакциях, внедряться практически в любом месте в приложении, и, кроме того, они просты в разработке.

Давайте создадим службу `PostService` для приложения Hubbub. Вас уже не будет удивлять, что процесс начинается в командной строке:

```
grails create-service com.grailsinaction.post
```

Эта команда создаст в каталоге /grails-app/services/com/grailsinaction начальную заготовку службы в файле PostService.groovy.

В листинге 5.6 мы добавили некоторую логику в новую службу, которая создает сообщения на основе передаваемых службе параметров `userId` и `content`.

Листинг 5.6. Определение службы PostService и связанного с ней класса исключения в файле PostService.groovy

```
package com.grailsinaction

class PostException extends RuntimeException {          ❶ Вынуждает транзакции
    String message                                        выполнять откат в случае
    Post post                                             ошибки
}

class PostService {          ❷ Логика обработки объектов Post помещается в службу

    boolean transactional = true          ❸ Обеспечивает возможность отката
                                            изменений в базе данных в случае ошибки

    Post createPost(String userId, String content) {          ❹ Оформление логики
        def user = User.findByUserId(userId)                    создания сообщения
        if (user) {                                             в виде метода
```

```
                def post = new Post(content: content)
                user.addToPosts(post)
                if (user.save()) {  ← ❺ Метод save() выполняет проверку объекта Post
                    return post
                } else {                                    ❻ При неудаче во время
                    throw new PostException(                проверки возбуждается
                        message: "Invalid or empty post", post: post)  ← исключение
                }
            }
            throw new PostException(message: "Invalid User Id")

        }
    }
```

В листинге 5.6 мы сначала определили новое исключение `PostException` с целью обработки любых ошибок, которые могут возникнуть при сохранении ❶. Мы будем использовать это исключение для сохранения текста сообщения об ошибке и любых объектов `Post`, не прошедших проверку. Язык Groovy позволяет определять в одном файле .groovy более одного класса, и мы воспользовались такой возможностью, чтобы сохранить исключение вместе со службой.

Далее следует определение службы, имя которой в соответствии с соглашениями, принятыми в платформе Grails, оканчивается словом «Service» (служба) ❷. С помощью признака `transactional` мы определили, что служба поддерживает транзакции ❸. (Семантика поддержки транзакций подробно будет рассматриваться в главе 14.) Метод `createPost()` принимает параметры `userId` (идентификатор пользователя) и `content` (текст сообщения) ❹ и либо возвращает объект `Post`, либо возбуждает исключение `PostException`, если что-то пошло не так.

Примечание

Поскольку класс `PostException` расширяет класс `RuntimeException`, как видно в листинге 5.6, платформа Grails будет автоматически откатывать любые транзакции при выполнении операций с базой данных внутри метода. С работой этого механизма вы поближе познакомитесь в главе 14.

Эта служба поддерживает механизм транзакций, поэтому мы можем попытаться вызвать метод `save()` ❺. В случае неудачи во время проверки этот метод вернет значение `null`, что приведет к возбуждению исключения `PostException` ❻. Вместе с исключением мы возвращаем сам объект `Post`, потому что вызывающему программному коду может потребоваться исследовать его и отобразить сообщение об ошибке с точной информацией. Если вызов метода `save()` выполняется благополучно, мы возвращаем сохраненный объект `Post`.

5.2.2. Вызов службы PostService в контроллере PostController

После создания службы PostService нам требуется связать ее с контроллером PostController. Как мы уже показывали в главе 1, это осуществляется за счет объявления в контроллере свойства, имя которого совпадает с именем службы (за исключением того, что первый символ в имени свойства употребляется в нижнем регистре).

В листинге 5.7 приводится измененная реализация контроллера PostController, который теперь все операции с сообщениями выполняет с помощью службы PostService.

Листинг 5.7. Версия контроллера PostController, использующая новую службу PostService

```
package com.grailsinaction

class PostController {

    def postService ←— ❶ Включение службы в контроллер
    def scaffold = true

    def timeline = {
        def user = User.findByUserId(params.id)
         [ user : user ]
    }

    def addPost = {
        try {
            def newPost =
                postService.createPost(params.id,  ←— ❷ Вызов методов службы
                                       params.content)
                flash.message = "Added new post: ${newPost.content}"
        } catch (PostException pe) { ←— ❸ Выполняется перехват исключений
            flash.message = pe.message      для отображения сообщений на странице
        }
        redirect(action: 'timeline', id: params.id)
    }
}
```

В листинге 5.7 мы изменили реализацию контроллера, внедрив в него службу PostService, которая теперь будет использоваться автоматически ❶. Теперь, когда у нас имеется служба, нам осталось только вызвать метод createPost() ❷ и предусмотреть обработку ошибок при добавлении недопустимых сообщений ❸.

Создание службы PostService повлекло за собой довольно много изменений в первоначальном программном коде, зато в итоге мы получили простую и понятную службу, которую позднее мы сможем исполь-

зовать для передачи сообщений через службы REST и очереди сообщений, так что результат стоил затраченных усилий.

Нам предстоит выяснить еще множество вопросов, касающихся служб (например, возможно ли использовать их как единичные объекты (singleton) или прототипы, как сохраняются транзакции и как тестировать службы); мы вернемся к ним в главе 14. Мы дали вам представление о службах здесь, поскольку наиболее часто они используются в контроллерах, а это означает, что они пригодятся нам ниже в этой же главе.

Журналирование: специальный случай внедрения

Одним из специальных случаев внедрения является использование механизма журналирования платформы Grails. Нам пока не приходилось использовать его, тем не менее в каждый контроллер, службу и класс предметной области автоматически внедряется объект `log` из библиотеки Log4j (что выполняется не с помощью классической процедуры внедрения, принятой в платформе Spring, а с привлечением некоторого волшебства свойства `MetaClass` на этапе начальной загрузки, хотя результат получается тот же).

Вы можете использовать объект `log` везде, где это потребуется:

```
log.debug "The value of user is: ${user}"
```

В частности, объект `log` удобно использовать в обработчиках исключений. Если вторым параметром методу объекта `log` передать экземпляр исключения, в журнал будет записана полная трассировка стека:

```
try {
    user.punchChuckNorris()
} catch (e) {
    log.error "Failed to punch Chuck Norris", e
}
```

Параметры журналирования определяются в файле /grails-app/conf/Config.groovy. Кроме того, имеется возможность использовать расширение runtimeLogging для изменения уровня журналирования прямо во время работы приложения.

Теперь, когда функциональность обработки сообщений выделена в отдельную службу, пришло время заняться исследованием других аспектов контроллеров, которые мы можем реализовать более простым и удобным способом.

Нам необходимо рассмотреть следующий жизненно важный вопрос, касающийся работы контроллеров, – *привязку данных*, то есть порядок получения и проверки данных формы и сохранение их в базе дан-

ных. До настоящего момента мы использовали упрощенные приемы работы с данными, но пришло время представить вам целый спектр более мощных возможностей.

5.3. Привязка данных

Теперь, когда у вас имеется представление о том, как действуют службы, можно вернуться к способу, которым мы получали значения параметров из формы и записывали их в объект предметной области. До настоящего момента мы либо манипулировали данными с помощью отображения `params` либо выполняли привязку входных параметров, передавая их конструктору `new Post(params)`, но как быть в случаях, когда желательно выполнить привязку только части параметров? Возможно, для вас будет нежелательно предоставлять пользователю возможность передавать свои собственные значения параметров в виде `/user/update?password=secret&userId=admin`.

Процесс преобразования параметров в свойства объекта называется *привязкой данных*. Мы пока не очень подробно касались этой темы, сейчас же мы займемся глубоким ее исследованием.

5.3.1. Привязка к существующему объекту

В большинстве случаев реализации привязки данных от нас требовалось создать новый объект предметной области на основе входных данных, хранящихся в объекте `params`. Однако изменение данных – это еще один часто встречающийся случай привязки данных. Возможно, вы предоставляете пользователям возможность переходить на использование другого профиля с измененным адресом электронной почты.

Собственно говоря, действие `update`, которое приводится в листинге 5.8, выполняет необходимые изменения.

Листинг 5.8. Привязка данных к свойствам может быть опасной

```
def update = {

    def user = session.user.attach() <-- ❶ Подключает объект User, существующий
                                          в области видимости сеанса
    user.properties = params <-- ❷ Изменяет свойства объекта в соответствии
                                  со значениями ключей отображения params
    if (user.validate()) { <-- ❸ Проверяет свойства объекта User,
                               прежде чем сделать его текущим
        flash.message = "Successfully updated user"
    } else {
        user.discard() <-- ❹ Отменяет изменения в случае неудачи при проверке
    }                      (в версии Grails 1.0)
    [ user : user ]

}
```

В этом примере мы повторно подключаем объект User, сохраненный в области видимости сеанса в процессе регистрации ❶, к сеансу Hibernate текущего потока выполнения. Затем мы выполняем привязку объекта params к свойствам объекта User ❷, поэтому значения параметров, имена которых совпадают с именами свойств объекта User, переписываются в объект User. В заключение выполняется проверка объекта на соответствие наложенным ограничениям ❸, а в случае неудачи выполняется откат изменений ❹.

Такое обновление действует именно так, как нам и требуется. Обратите внимание на явный вызов метода discard() в случае неудачной проверки ❹. Если в версии платформы Grails 1.0 опустить явный вызов метода discard() в случае неудачной проверки, механизм GORM сохранит его в базе данных с ошибочными значениями свойств. В версии Grails 1.1 явный вызов discard() больше не требуется – в случае ошибки при проверке платформа Grails сама пометит объект как доступный только для чтения, и он никогда не будет сохранен в базе данных. Если вам приходится работать с обеими версиями Grails 1.0 и 1.1, об этом важном различии следует помнить.

Теперь мы знаем, как выполняется изменение существующих объектов предметной области, но мы пока не знаем, как исключить или включить определенные параметры в процедуре привязки. Итак, нам следует познакомиться с методом bindData().

5.3.2. Белые и черные списки привязываемых параметров

Метод bindData(), присутствующий в контроллерах, позволяет исключать определенные параметры из процедуры привязки.

Версия метода bindData(), принимающая два аргумента, эквивалентна стандартной привязке свойств:

```
bindData(user, params)
```

Этот вызов даст тот же результат, что и более знакомая процедура привязки данных, имеющая форму операции присваивания, с которой вы уже встречались:

```
user.properties = params
```

Возможность исключения свойств обеспечивается версией метода bindData(), принимающей три аргумента, где в третьем аргументе передается список свойств, которые следует исключить из процедуры привязки. Если нам потребуется исключить возможность привязки значений свойств userId и password, мы сможем реализовать это примерно так:

```
bindData(user, params, ['userId', 'password'])
```

Данный прием решает лишь половину проблемы – он исключает определенные свойства. Но как быть, если нам требуется указать опреде-

ленные параметры, которые должны быть включены в процедуру привязки? Начиная с версии Grails 1.1, объект `properties` поддерживает оператор индексирования, который можно использовать для определения списка свойств, включаемых в процедуру привязки.

Например, если необходимо изменить только значения свойств `email` и `fullName`, это можно реализовать примерно так:

```
user.profile.properties['email', 'fullName'] = params
```

В таком случае будут изменены только свойства `email` и `fullName` объекта `Profile`, все остальные параметры, соответствующие свойствам объекта `User`, будут отвергнуты.

Теперь мы знаем, как выполняется привязка данных для единственного экземпляра класса предметной области; давайте рассмотрим, как выполняется привязка данных к целой иерархии объектов.

5.3.3. Сложные формы: привязка данных к множеству объектов

Во всех примерах, приводившихся до сих пор, привязка данных выполнялась для единственного объекта предметной области, однако платформа Grails позволяет обрабатывать формы, содержащие данные для вложенных объектов. Рассмотрим в качестве примера процедуру регистрации пользователя. Нам необходим объект `User` (в котором хранятся идентификатор и пароль) и объект `Profile` (где хранятся полное имя, биографические данные, адреса домашней страницы и электронной почты и другие атрибуты).

В листинге 5.9 мы реализовали форму, которая ссылается на поля объекта `User` и соответствующего ему объекта `Profile` способом, предусматривающим защиту от значений `null`. Отправка такой формы позволит создать сразу оба объекта, `User` и присоединенный к нему объект `Profile`.

Листинг 5.9. Форма, которая изменяет несколько объектов предметной области за одну операцию

```
<html>
    <head>
        <title>Register New User</title>
        <meta name="layout" content="main"/>
    </head>
    <body>

        <h1>Register New User</h1>

        <g:hasErrors>
            <div class="errors">
                <g:renderErrors bean="${user}" as="list" />
            </div>
        </g:hasErrors>
```

```
        <g:form action="register">
            <dl>
                <dt>User Id</dt>
                <dd><g:textField name="userId"
                    value="${user?.userId}"/></dd>
                <dt>Password</dt>
                <dd><g:passwordField name="password"
                    value="${user?.password}"/></dd>
                <dt>Full Name</dt>
                <dd><g:textField name="profile.fullName"
                    value="${user?.profile?.fullName}"/></dd>
                <dt>Bio</dt>
                <dd><g:textArea name="profile.bio"
                    value="${user?.profile?.bio}"/></dd>
                <dt>Email</dt>
                <dd><g:textField name="profile.email"
                    value="${user?.profile?.email}"/></dd>
                <dt><g:submitButton name="register" value="Register"/></dt>
            </dl>

        </g:form>

    </body>
</html>
```

Форма регистрации в листинге 5.9 содержит множество полей с информацией для обоих объектов `User` и `Profile`. Обратите внимание, что поля, имеющие отношение к профилю, объявлены в виде элементов формы с именами, включающими префикс «profile»: `profile.fullName`, `profile.bio`, `profile.email`. Платформа Grails использует этот префикс для привязки полей объекта, связанного отношением с сохраняемым объектом. На рис. 5.8 демонстрируется, как единый набор параметров распределяется по объектам `User` и `Profile`.

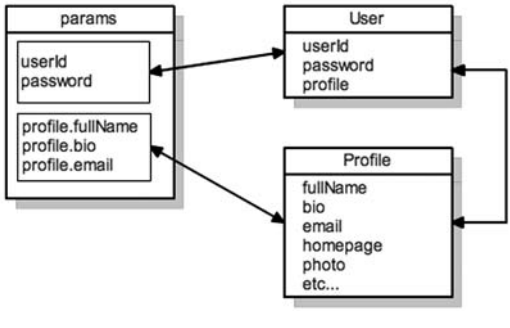


Рис. 5.8. *Разбиение привязываемых параметров производится в соответствии с префиксами в их именах.*

В листинге 5.10 приводится новая реализация действия `register()`, которую мы добавили в контроллер `UserController`. Создание и сохранение объекта `User` осуществляются на основе входного объекта `params`, одновременно с этим происходит и связывание полей объекта `Profile`.

Листинг 5.10. Реализация действия register() для контроллера UserController

```
def register = {

    def user = new User(params)
    if (user.validate()) {
        user.save()
        flash.message = "Successfully Created User"
            redirect(uri: '/')
        } else {
            flash.message = "Error Registering User"
            return [ user: user ]
        }
    }
}
```

Если вызов метода `user.validate()` терпит неудачу, мы возвращаем регистрационную форму, передавая в ней содержимое объекта `User`. Если форма получает объект, содержащий ошибки, мы отображаем их с помощью тегов `<g:hasErrors>` и `<g:renderErrors>`. Напомним, как выглядит элемент `div` с классом `errors` в форме регистрации (листинг 5.9):

```
<g:hasErrors>
    <div class="errors">
        <g:renderErrors bean="${user}" as="list" />
    </div>
</g:hasErrors>
```

Тег `<g:renderErrors>` отображает сообщения процедуры проверки для указанного объекта в виде маркированного списка HTML (`<ul>`), что удобно для информирования пользователя о возникших проблемах. Здесь для отображения сообщений об ошибках, выявленных во время проверки, используется тот же механизм скаффолдинга, который мы рассматривали в главе 4. На рис. 5.9 показана новая регистрационная форма в действии, отображающая соответствующие сообщения об ошибках.

Мы только что реализовали простую форму регистрации и даже предусмотрели обработку ошибок. Но теперь вас может беспокоить, как выполняется обработка ошибок за кулисами. Очень важно знать, как обрабатываются ошибки, на тот случай, если вам придется создавать приложения с собственными макетами страниц и с возможностью отображать сообщения об ошибках для определенных полей. Давайте рассмотрим этот вопрос.

Рис. 5.9. *Имеется возможность в одной форме выполнить привязку сразу нескольких объектов и включить обработку ошибок*

5.3.4. Обработка ошибок

В предыдущем разделе мы передавали представлению объект `User`, не прошедший процедуру проверки. С помощью тегов `<g:hasErrors>` и `<g:renderErrors>` представление отображало сообщения об ошибках (как показано на рис. 5.9). Вам, может быть, любопытно узнать, как эти теги определяют, что процедура проверки потерпела неудачу.

В главе 3 (в разделе 3.3) мы показывали, как вызов `user.validate()` заполняет объект `user.errors` информацией об обнаруженных ошибках. Этот объект используется методами `hasErrors()` и `renderErrors()` для итераций по ошибкам.

Но как быть, если вместо того, чтобы перечислять возникшие ошибки в верхней части страницы, необходимо визуально выделить отдельные значения полей, вызвавшие ошибки? Для этого можно воспользоваться специальной версией тега `hasErrors`, которому можно указать как сам объект в области видимости запроса, так и отображаемые поля. В листинге 5.11 приводится пример отображения сообщения об ошибке проверки адреса электронной почты рядом с полем `email`.

Листинг 5.11. Реализация вывода сообщений об ошибках рядом с отдельными полями пока выглядит достаточно сложной

```
<dt>Email</dt>
<dd>
    <g:textField name="profile.email" value="${user?.profile?.email}"/>
    <g:hasErrors bean="${user}" field="profile.email"> ← ❶ Отображает блок,
                                                         если поле содержит ошибку
```

```
        <g:eachError bean="${user}" field="profile.email">
            <p style="color: red;"><g:message error="${it}"/></p>
        </g:eachError>
    </g:hasErrors>
</dd>
```

❷ Выполняет итерации по ошибкам для поля email

❸ Отображает текст сообщения

В листинге 5.11 мы использовали тег <g:hasErrors>, чтобы определить наличие ошибок проверки для поля email в объекте Profile ❶. Если ошибки существуют, мы выполняем обход этих ошибок с помощью тега <g:eachError> ❷. Не забывайте, что данное поле может содержать несколько ошибок. В заключение мы извлекаем текст сообщения для каждой ошибки из пакета ресурсов с помощью тега <g:message> ❸.

Вероятно, после этого вариант с использованием тега <g:renderErrors> покажется вам более предпочтительным. Мы надеемся, что в будущих версиях платформы Grails реализация обработки ошибок на уровне отдельных полей будет менее трудоемкой.

Общее представление о разметке, генерируемой при использовании приема из листинга 5.11, дает рис. 5.10. Как видите, существует возможность определить местоположение ошибки, но для этого придется потрудиться, и красная звездочка рядом с полем, в котором обнаружена ошибка, вероятно, покажется более эффективной.

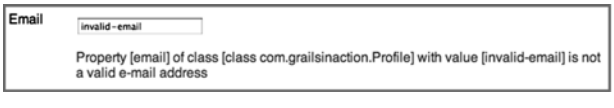


Рис. 5.10. *Вывод сообщений об ошибках рядом с каждым отдельным полем – труднодостижимая задача в версии Grails 1.1*

Теперь, когда вы познакомились с некоторыми возможностями привязки данных, реализованными в платформе Grails, и рассмотрели, как выполняется обработка ошибок, пришло время представить вам последний прием привязки данных, который обеспечивает еще большее удобство сопровождения всей операции в целом: использование *управляющих объектов*.

5.4. Управляющие объекты

Все описанные выше приемы манипулирования данными и их привязки выглядят замечательно, но как быть, если между данными формы и объектом предметной области нет прямого соответствия? Например, форма регистрации может иметь поля «password» (пароль) и «confirmPassword» (подтверждение пароля), значения которых необходимо про-

верить на совпадение, чтобы убедиться, что пользователь не допустил опечатку при вводе пароля.

В таких случаях платформа Grails предлагает использовать *управляющий объект*. Назначение управляющего объекта состоит в том, чтобы перенести данные из параметров формы в объект предметной области, попутно выполнив собственные проверки.

5.4.1. Обработка формы регистрации пользователя

Процедура регистрации пользователя предполагает выполнение некоторых проверок, которые имеют смысл только на этапе регистрации (вроде проверки совпадения паролей). Посмотрим на примере объекта `UserRegistrationCommand`, как можно совместить оба набора данных в единственном управляющем объекте.

Как правило, управляющие объекты – это объекты однократного использования, поэтому в соответствии с соглашениями они создаются в одном файле с контроллерами, использующими их. Для нашего случая дополним класс `UserController` новым управляющим классом, как показано в листинге 5.12.

Листинг 5.12. Класс UserRegistrationCommand

```
class UserRegistrationCommand {

    String userId
    String password
    String passwordRepeat ← Поле подтверждения пароля

    byte[] photo
    String fullName
    String bio
    String homepage
    String email
    String timezone
    String country
    String jabberAddress

    static constraints = {
        userId(size: 3..20)

        password(size: 6..8, blank: false,
            validator: { passwd, urc ->
                return passwd != urc.userId
            })
        passwordRepeat(nullable: false,  ← Проверяется, совпадает ли поле
            validator: { passwd2, urc ->     подтверждения пароля с полем
                                             пароля
                return passwd2 == urc.password
            })
        fullName(nullable: true)
```

```
            bio(nullable: true, maxSize: 1000)
            homepage(url: true, nullable: true)
            email(email: true, nullable: true)
            photo(nullable: true)
            country(nullable: true)
            timezone(nullable: true)
            jabberAddress(email: true, nullable: true)
        }

    }
```

Как много проверок здесь предусмотрено! Мы выполняем все проверки из обоих классов `User` и `Profile`, которые нам хорошо знакомы. Кроме того, мы добавили дополнительное поле и его проверку, необходимые только на этапе регистрации (если говорить точнее, значения полей `password` и `passwordRepeat` должны совпадать).

Управляющие объекты особенно удобны, когда необходимо выполнить дополнительные проверки только на этапе обработки формы и эти проверки не имеют значения для модели предметной области. Классическим примером такой ситуации является подтверждение пароля.

Самая приятная сторона управляющих объектов проявляется при создании действия контроллера, которое обрабатывает полученную форму. Достаточно просто передать управляющий объект первым аргументом в замыкании метода действия, и привязка будет выполнена автоматически. В листинге 5.13 приводится реализация действия `register2()`, выполняющего свою работу с помощью управляющего объекта.

Применение принципа DRY для валидаторов

Возможно, вам интересно узнать, имеется ли другие, более соответствующие принципу DRY (Don't Repeat Yourself – избегай повторений) способы избежать повторного использования в управляющих объектах ограничений, определяемых в классах предметной области. Встроенных механизмов для этого не существует, но многие используют творческий подход к решению этой проблемы. Существует три основных возможности. Применяя стандартную операцию присваивания между классами, можно совместно использовать:

- замыкание `constraints` целиком;
- отображения с ограничениями для отдельных полей;
- собственные замыкания валидаторов.

Ни один из этих подходов нельзя назвать элегантным, и мы надеемся на появление в будущих версиях платформы Grails механизма проверки, более соответствующего принципу DRY.

Листинг 5.13. Действие регистрации, использующее управляющий объект

```
def register2 = { UserRegistrationCommand urc ->          ❶ Привязка данных из params
                                                            к управляющему объекту
    if (urc.hasErrors()) {                                ❷ Метод hasErrors используется
                                                            для проверки допустимости объекта
        return [ user : urc ]
    } else {
        def user = new User(urc.properties)               ❸ Привязка данных
      user.profile = new Profile(urc.properties)            к новому объекту User

        if (user.save()) {                                ❹ Сохраняет и проверяет новый объект User
            flash.message =
                "Welcome aboard, ${urc.fullName ?: urc.userId}"
            redirect(uri: '/')
        } else {
            // возможно, неуникальное значение userId?
            return [ user : urc ]
        }
    }
}
```

В листинге 5.13 управляющий объект передается замыканию действия в первом аргументе ❶, вынуждая платформу Grails попытаться привязать свойства входного объекта `params` к полям управляющего объекта. В процессе привязки выполняется проверка полей, результаты которой можно получить вызовом метода `hasErrors()` самого управляющего объекта ❷.

Если проверки проходят успешно, можно выполнить привязку полей управляющего объекта к полям различных классов предметной области. В листинге 5.13 мы привязываем данные к обоим объектам `User` и `Profile` ❸, после чего пытаемся сохранить информацию о новом пользователе.

Мы проверяем успешность вызова метода `save()` ❹, потому что некоторые ограничения имеют смысл только в классе предметной области, а не в управляющем объекте. (Например, класс `User` проверяет уникальность значения поля `userId`.) Вы можете попытаться сымитировать проверку уникальности в управляющем объекте с помощью собственного валидатора, но даже тогда уникальность имени пользователя не гарантирована до тех пор, пока метод `save()` не подтвердит сохранение объекта в базе данных.

Управляющие объекты отлично подходят для обработки таких форм, когда отсутствует прямое соответствие с классом предметной области. Но они обладают и другими возможностями. Управляющие объекты могут, например, участвовать во внедрении.

5.4.2. Участие во внедрении

Управляющие объекты – это не просто объекты с данными, имеющие возможность выполнять проверки. На них распространяется действие того же механизма внедрения объектов, что и на контроллеры, а это означает, что они могут служить фантастическими контейнерами бизнес-логики.

В нашем примере с управляющим объектом `UserRegistrationCommand` пользователь вводит пароли в обычном текстовом поле, но представьте, что нам требуется хранить пароли в базе данных в зашифрованном виде. Если бы мы объявили службу `cryptoService`, мы могли бы внедрить ее непосредственно в управляющий объект. Сделать это можно примерно так:

```
class UserRegistrationCommand {

    def cryptoService

    String getEncryptedPassword() {
        return cryptoService.getEncryptedPassword(password)
    }
    // остальные свойства и валидаторы
}
```

После этого мы могли бы использовать службу `cryptoService` для выполнения всех необходимых операций по шифрованию и гарантировать, что только один класс знает, как реализовано шифрование паролей. Добавление вспомогательного метода, такого как `getEncryptedPassword()`, в класс управляющего объекта сделает программный код контроллера, использующий управляющий объект, гораздо более аккуратным.

Теперь, когда мы рассмотрели возможность привязки данных и логику работы контроллеров, нам осталось лишь самое сладкое в этой главе. Мы переходим к изучению возможности выгрузки фотографий, созданию простого файла безопасности и настройки адресов URL сайта.

Для начала рассмотрим, как реализовать выгрузку фотографий для профилей пользователей.

5.5. Работа с изображениями

К настоящему моменту мы познакомились почти со всеми особенностями контроллеров, которые могут вам потребоваться при создании следующего вашего проекта на платформе Grails. Однако имеется еще ряд особенностей – они необязательно будут присутствовать в каждом вашем приложении, но когда они потребуются, они действительно будут необходимы.

В этом разделе мы исследуем порядок выгрузки файлов (фотографий для наших профилей пользователей) и способы отображения собствен-

ных типов содержимого (в нашем случае – изображений). Несмотря на то что в следующем вашем приложении необязательно потребуется использовать фотографии, обсуждаемые приемы могут применяться для отображения содержимого любого типа.

5.5.1. Выгрузка файлов

Разве можно представить себе сайт социальных сетей без возможности выгружать фотографии и аватары? Однако, если ранее вам приходилось добавлять возможность выгрузки фотографий в веб-приложения на языке Java, вы, вероятно, знаете о трудностях, связанных с этим (нужно не только заниматься обработкой потоков байтов, но и помнить о проблемах безопасности, таких как ограничение размеров файлов для предотвращения атак типа «отказ в обслуживании»). Grails решает все сложные проблемы за вас. Давайте примемся за работу и реализуем возможность выгрузки фотографий для нашего приложения Hubbub.

Сначала создадим контроллер `ImageController`, который обеспечит выгрузку фотографий и их отображение:

```
grails create-controller com.grailsinaction.Image
```

Существует два способа реализации функции выгрузки файлов в контроллере; выбор в пользу одного из них зависит от того, чего вы хотите добиться. Если вы хотите сохранить изображение в классе предметной области, лучше использовать управляющий объект. В листинге 5.14 показано, как можно задействовать управляющий объект для выгрузки фотографии.

Листинг 5.14. Реализация выгрузки фотографии с помощью управляющего объекта

```
package com.grailsinaction

class PhotoUploadCommand {
    byte[] photo ← Поле для сохранения выгруженной фотографии
    String userId
}

class ImageController {

    def upload = { PhotoUploadCommand puc ->
        def user = User.findByUserId(puc.userId)
        user.profile.photo = puc.photo
        redirect(action: 'view', id: puc.userId)
    }

    def form = {
        // переход в форму выгрузки
        [ userList : User.list() ] ← Передача списка пользователей
    }                                  в представление
```

```
        def view = {
            // переход на страницу "просмотр фотографии"
        }
    }
```

Для выгрузки изображения используется управляющий объект, который выполняет привязку выгруженного изображения к массиву байтов.

Чтобы обеспечить фактическую возможность выгрузки фотографии в окне броузера, нам потребуется представление с элементом управления `Upload`. Кроме того, сама форма должна быть помечена, чтобы сообщить броузеру, что форма содержит поле выгрузки файла, поэтому мы будем использовать тег `<g:uploadForm>`, как показано в листинге 5.15.

Листинг 5.15. Форма выгрузки изображения

```
<g:uploadForm action="upload">
    User Id:
    <g:select name="userId" from="${userList}"
        optionKey="userId" optionValue="userId" />
    <p/>
    Photo: <input name="photo" type="file" />
    <g:submitButton name="upload" value="Upload"/>
</g:uploadForm>
```

Запомните, что для реализации формы выгрузки файла следует использовать тег `<g:uploadForm>`, а не `<g:form>`, и что в ней должно присутствовать поле ввода с атрибутом `type="file"` для информации о выгружаемом изображении.

Броузер отобразит форму из листинга 5.15 с полем выгрузки файла, как показано на рис. 5.11.

***Рис. 5.11.** Форма выгрузки изображения в действии*

Благодаря нашему управляющему объекту, обеспечивающему возможность выгрузки, пользователю достаточно будет сделать всего несколько щелчков мышью, чтобы сохранить свою картинку в базе данных.

5.5.2. Выгрузка в файловую систему

Если вы предполагаете сохранять выгружаемые изображения в файловой системе, а не в базе данных, вам необходимо использовать объект `MultipartHttpServletRequest` платформы Spring, который реализует процедуру выгрузки.

В таком случае у вас имеются более широкие возможности сохранения массива байтов:

```
def rawUpload = {

    // Объект MultipartHttpServletRequest платформы Spring
    def mhsr = request.getFile('photo')
    if (!mhsr?.empty && mhrs.size < 1024*200) {  ← Убедиться, что размер
        mhsr.transferTo(                            файла меньше 200 Кб
            new File(
                "/hubbub/images/${params.userId}/mugshot.gif"
            )
        )
    }
}
```

Класс `MultipartHttpServletRequest` имеет метод `transferTo()`, который позволяет переместить изображение непосредственно в файл, что очень удобно, если вы не склонны сохранять большие массивы байтов в базе данных. Подробное описание класса `MultipartHttpServletRequest` вы найдете в документации к платформе Spring.

5.5.3. Отображение фотографий из базы данных

Теперь, когда выгруженные изображения сохранены в базе данных или в файловой системе, нам необходимо обеспечить их отображение. Для этого в нашем приложении мы создадим теги `<img>` и заставим платформу Grails извлекать фотографии и отображать их.

Сначала мы создадим новое представление для отображения профиля, чтобы пользователь мог просматривать профиль, и добавим ссылку на изображение. Назовем представление /views/user/profile.gsp и добавим в него HTML-тег `<img>`:

```
<div class="profilePic">
    <g:if test="${profile.photo}">
        <img src="
            <g:createLink controller='image'           ← Ссылка для
                action='renderImage' id='${userId}'/>     отображения фотографии
            "/>
    </g:if>
    <p>Profile for <strong>${profile.fullName}</strong></p>
    <p>Bio: ${profile.bio}</p>
</div>
```

Этот фрагмент создает ссылку с адресом /image/renderImage/id, исходя из значения `userId` текущего пользователя. Как только мы реализуем действие `renderImage()` в контроллере `ImageController`, мы сможем связать тег `<img>` с адресом /image/renderImage/chuck_norris или с адресом любого другого пользователя.

В листинге 5.16 показано, как выполняется передача изображения клиенту.

Листинг 5.16. Передача изображения броузеру

```
def renderImage = {
    def user = User.findByUserId(params.id)
    if (user?.profile?.photo) {
        response.setContentLength(user.profile.photo.length)
        response.outputStream.write(user.profile.photo)
    } else {
        response.sendError(404) ← Если фотография не найдена, отправить
                                  сообщение с кодом ошибки 404
    }
}
```

Как показано в листинге 5.16, чтобы передать изображение броузеру, необходимо напрямую записать массив байтов в поток ответа, а также сообщить броузеру объем данных.

Наша реализация обеспечивает отображение фотографии, как показано на рис. 5.12, где приводится результат обращения по адресу /user/profile/chuck_norris.

***Рис. 5.12.** Отображение фотографии на странице просмотра профиля*

Теперь у нас имеется простая форма отображения содержимого профиля, которой мы завершаем наши исследования возможности отображения фотографий и всей функциональности пользовательского интерфейса в этой главе. Еще больше приемов реализации пользовательского интерфейса мы узнаем в главе 6, а пока исследуем еще две важные особенности контроллеров: фильтры и отображение адресов URL. Обе они влияют на то, как пользователь будет перемещаться по сайту, а их пол-

ное понимание является важным условием разработки мощных приложений на платформе Grails.

Начнем с фильтров.

5.6. Обработка запросов фильтрами

В этой главе мы потратили массу времени на исследование пользовательского интерфейса. Однако существуют не менее важные методики, которые могут потребоваться для реализации логики работы. В данном разделе мы рассмотрим порядок использования фильтров с целью добавить в конвейер запрос/ответ платформы Grails свою собственную обработку.

5.6.1. Ваш первый фильтр

Фильтры платформы Grails являются мощным инструментом обработки всех запросов (или подмножества запросов) и позволяют реализовывать операции, которые должны выполняться до или после вызова действия контроллера. Если ранее вам приходилось работать с фильтрами сервлетов языка Java, вы можете представлять себе фильтры платформы Grails как аналогичную возможность, хотя в отличие от фильтров сервлетов фильтры платформы Grails способны обрабатывать только запросы контроллеров, но не статические запросы на получение файлов JavaScript, CSS, изображений и так далее.

Классическим примером того, где может пригодиться применение фильтров, является обеспечение безопасности – вам может потребоваться проверить право пользователя на доступ к определенному ресурсу (некоторые популярные расширения обеспечения безопасности для платформы Grails для этих целей используют фильтры Grails). Создадим простой фильтр, чтобы показать, как он действует.

Сначала создадим заготовку фильтра:

```
grails create-filters LameSecurity
```

Эта команда создаст начальную заготовку фильтра в файле /grails-app/conf/LameSecurityFilters.groovy (обратите внимание на множественное число в слове «Filters» (фильтры) – если употребить его в единственном числе, фильтр не будет работать).

Теперь изменим фильтр, реализовав в нем выполнение некоторых простых проверок, как показано в листинге 5.17.

Листинг 5.17. Реализация простого фильтра безопасности

```
import com.grailsinaction.*

class LameSecurityFilters {

    def filters = {
```

```
        secureActions(controller:'post', ← ❶ Имя правила безопасности
                action:'(addPost|deletePost)') { ← ❷ Действие фильтра
            before = {                              распространяется
                                                    на два метода
                if (params.logonId) { ← ❸ Проверка наличия параметра logonId
                    session.user = User.findByUserId(params.logonId)
                }

                if (!session.user) { ← ❹ Проверка наличия пользователя в сеансе
                    redirect(controller: 'login', action: 'form')
                    return false ← ❺ Предотвращает вызовы последующих фильтров
                }

            }

            after = { model->
            }

            afterView = {
                log.debug "Finished running ${controllerName} -
                ↪${actionName}" ← ❻ Записывает диагностическую информацию
            }                       в журнал после завершения работы
                                    представления
        }
    }
}
```

Вы можете давать имена фильтрам, помещаемым в файл (исключительно в описательных целях), поэтому желательно выбирать имена, которые будут отражать логику работы фильтра. В листинге 5.17 мы назвали наш набор правил `secureActions` ❶.

Вы можете поместить в файл любое количество фильтров, и все соответствующие правила будут выполняться в порядке сверху-вниз, пока какое-нибудь из них не вернет значение `false` из своего замыкания `before()`. Как вы вскоре узнаете, фильтры допускают использование шаблонных символов (`controller: '*' action: '*'`), но в листинге 5.17 нам необходимо, чтобы фильтр срабатывал только при обращении к действиям `addPost` и `deletePost` ❷. Использование логических операторов поможет вам повысить избирательность применения фильтров.

Внутри фильтра описываются замыкания, содержащие операции до (`before`) или после (`after`) вызова действия контроллера, а также после отображения представления (`afterView`). В табл. 5.2 перечислены фазы жизненного цикла фильтра и их типичное предназначение.

Наше замыкание `before` проверяет, представил ли пользователь параметр `logonId` ❸; если представил и данный пользователь существует, объект `User` сохраняется в области видимости сеанса, что означает вход в систему. В противном случае пользователь перенаправляется в форму входа ❹.

Таблица 5.2. Фазы жизненного цикла фильтра и их назначение

Замыкание	Когда вызывается	Удобно для
`before`	Перед вызовом действия контроллера	Обеспечения безопасности, проверки заголовков Referrer
`after`	После вызова действия контроллера, но перед отображением представления	Изменения модели перед передачей ее представлению
`afterView`	После отображения представления	Измерения параметров производительности

Если какое-либо замыкание `before` вернет значение `false`, как это происходит в строке ❺, никакой другой фильтр, как и действие контроллера, вызываться не будет. Обычно этот прием используется в фильтрах (таких как фильтр безопасности), выполняющих перенаправление запроса.

Мы также добавили замыкание `afterView` ❻, чтобы продемонстрировать некоторые возможности диагностики и познакомить вас с некоторыми переменными, доступными в фильтрах платформы Grails. Несмотря на то, что фильтры не обладают функциональными возможностями контроллеров, они позволяют обращаться к переменным, доступным в контроллерах, которые мы рассматривали выше в этой главе (`request`, `response`, `params`, `session`, `flash` и некоторым другим переменным, доступным только в фильтрах). Кроме того, фильтры обладают двумя методами: `redirect()` и `render()`. В табл. 5.3 перечислены некоторые дополнительные переменные, доступные в фильтрах, но которые мы не использовали.

Существует особый (не используемый, правда, в листинге 5.17) случай замыкания `after`, принимающего модель в качестве аргумента. Это модель отображения, которое возвращается действием контроллера, обрабатывавшим запрос. В случае необходимости вы можете изменить содержимое этой модели в своем фильтре.

Таблица 5.3. Переменные, доступные в фильтрах

Переменная	Описание
`controllerName`	Имя текущего контроллера
`actionName`	Имя текущего действия
`applicationContext`	Контекст приложения на платформе Spring – удобно использовать для поиска объектов JavaBean платформы Spring, однако фильтры поддерживают шаблон внедрения зависимостей, что намного проще
`grailsApplication`	Текущий объект `grailsApplication` – удобно использовать для извлечения информации времени выполнения, такой как версия платформы Grails или имя приложения

Совет

Модификация существующих значений модели в фильтрах – не самая хорошая идея, так как это осложняет отладку программного кода и способствует появлению трудноуловимых ошибок в логике работы контроллера. Лучше добавьте в модель свои значения, но оставьте уже существующие значения в покое, в противном случае ожидайте ночных звонков с просьбами о поддержке и приготовьтесь к утомительной отладке.

Механизм внедрения и фильтры

Как быть, если появится потребность задействовать службу в фильтре? На фильтры, как и на другие компоненты платформы Grails, распространяется действие механизма внедрения объектов, поэтому вы легко можете объявлять службы (или другие компоненты) как свойства своего фильтра, и они будут внедряться в фильтр. Например:

```
class SecurityFilters {
    def authService
    def filters = {
        // где-нибудь в одном из ваших фильтров
        authService.checkAccess(params.userId,
                                controllerName, actionName)
    }
}
```

5.6.2. Возможности фильтрации адресов URL

Вы уже видели, что фильтры применяются к контроллерам и действиям, но существует еще масса возможностей управления избирательностью фильтров. Во-первых, вместо имен контроллеров и действий допускается использовать шаблонные символы (как вместо только имен контроллеров или только имен действий, так и вместо тех и других одновременно). Допускается также использовать логические операторы для отбора требуемых действий.

Рассмотрим несколько типичных примеров использования фильтров. Следующий набор фильтров применяется ко всем действиям всех контроллеров:

```
myGreedyFilter(controller:'*', action:'*') {
}
```

А этот набор фильтров применяется ко всем действиям контроллера `User`:

```
myControllerFilter(controller:'user', action:'*') {
}
```

Набор фильтров ниже применяется к отдельным действиям контроллера `User`:

```
mySelectiveFilter(controller:'user', action:'(update|edit|delete)') {
}
```

Однако, если вы мыслите не только в терминах контроллеров и действий (возможно, потому, что раньше вам приходилось иметь дело с магией отображения адресов URL для создания постоянных ссылок), вы можете описывать применение фильтров в стиле URI:

```
myGreedyUriFilter(uri:'/**') {
}

mySelectiveUriFilter(uri:'/post/**') {
}

myParticularUriFilter(uri:'/post/supersecret/list') {
}
```

Механизм сопоставления адресов URI позволяет использовать шаблонные символы в стиле Ant. Если прежде вам не доводилось сталкиваться с двойными звездочками, скажем, что они означают «все подкаталоги на любом уровне вложенности». Важно отметить, что отображение адресов URL выполняется перед обращением к фильтрам, поэтому, если вы пытаетесь возложить на фильтр задачу перехвата обращений к несуществующим адресам URL, вспомните об этом. Механизм отображения вернет ошибку с кодом 404 и ваш фильтр не будет вызван.

Это настоящий арсенал инструментов для выборочной фильтрации. Теперь вы приобрели навыки создания логики промежуточной обработки, которые пригодятся вам при создании вашего следующего приложения. Всякий раз, когда вам придется создавать собственный механизм безопасности, предусматривать фильтрацию на основе текущего состояния или заниматься отладкой и профилированием, фильтры платформы Grails обеспечат вам мощные средства избирательной обработки запросов.

В заключение мы рассмотрим еще одну особенность контроллеров: отображение адресов URL.

5.7. Создание собственных отображений URL

До сих пор мы следовали соглашению платформы Grails о преобразовании адресов URL по схеме /controllerName/actionName/id. Но даже это соглашение может быть настроено с использованием механизма отображения адресов URL.

В листинге 5.18 приводится содержимое файла /grails-app/conf/UrlMappings.groovy. В этом файле находятся правила, определяющие контроллеры и действия, куда должны направляться входящие запросы.

Листинг 5.18. В файле UrlMappings.groovy находится вся информация о маршрутизации для адресов URL

```
class UrlMappings {
    static mappings = {
        "/$controller/$action?/$id?"{
            constraints {
                // здесь определяются ограничения
            }
        }
        "/"(view:"/index")
        "500"(view:'/error')
    }
}
```

Переменные в файле UrlMappings, имена которых начинаются с символа $, могут вызывать недоумение, поэтому сейчас мы добавим в файл статическое отображение (постоянную ссылку) и посмотрим, как оно действует:

```
"/timeline/chuck_norris" {
    controller = "post"
    action = "timeline"
    id = "chuck_norris"
}
```

Эта постоянная ссылка обеспечивает возможность обращения по адресу URL /hubbub/timeline/chuck_norris и направляет запрос действию `timeline` контроллера PostController, передавая ему параметр `id` со значением `"chuck_norris"`. Обратите внимание: это не является перенаправлением, так как адрес URL в броузере остается неизменным, но вызываться будут действие и контроллер, указанные в постоянной ссылке.

Можно использовать также более краткую форму определения:

```
"/timeline/chuck_norris"(controller:"post",
           action:"timeline", id:"chuck_norris")
```

Мы считаем эту форму записи более удобочитаемой, и, кроме того, она обладает дополнительной гибкостью (в чем вы вскоре убедитесь).

Теперь, познакомившись с правилами отображения адресов, можно вернуться к тем переменным, что мы видели выше.

5.7.1. Hubbub: определение собственных постоянных ссылок

Существует возможность определять собственные переменные в отображении адресов URL, которые будут передаваться контроллерам как параметры. Например, было бы замечательно определить постоянные ссылки в приложении Hubbub, обеспечивающие доступ к страницам пользователей с последними сообщениями. Например, чтобы при об-

ращении по адресу URL /hubbub/users/glen возвращалась страница со всеми сообщениями пользователя Glen, а при обращении по адресу /hubbub/users/peter – страница с сообщениями пользователя Peter.

Постоянные ссылки такого рода можно создать с помощью следующего отображения URL:

```
"/users/$id" {
    controller = "post"
    action = "timeline"
}
```

При этом все еще будет вызываться действие `timeline` контроллера `PostController`, но в свойстве `params.id` будет передаваться значение «glen», «peter» или любое другое, образующее последнюю часть адреса URL.

5.7.2. Необязательные переменные и ограничения

При определении собственных переменных в строке отображения можно указать дополнительные ограничения, требующие совпадения значения с определенной строкой или списком.

Предположим, что нам необходимо создать постоянные ссылки на пользовательские ленты RSS или Atom. Реализовать такую постоянную ссылку можно с помощью следующего определения:

```
"/users/$userId/feed/$format?" {
    controller = "post"
    action = "feed"
    constraints {
        format(inList: ['rss', 'atom'])
    }
}
```

Здесь есть два важных момента, на которые следует обратить внимание. Мы сделали необязательной часть URL, определяющую формат (завершив ее символом `?`), благодаря этому пользователь может опустить эту часть. На тот случай, если пользователь все-таки укажет формат в строке URL, мы добавили раздел `constraints`, который требует, чтобы в качестве формата указывалось либо название «rss», либо «atom». Это означает, что адрес /hubbub/users/glen/feed/atom является допустимым, как и адрес /hubbub/users/glen/feed, а при обращении по адресу /hubbub/users/glen/feed/sneaky пользователю будет возвращено сообщение об отсутствии запрашиваемой страницы с кодом 404. Такие ограничения можно использовать для определения постоянных ссылок, допускающих возможность включения уточняющих фрагментов пути.

Правила в файле URLMapping применяются в порядке их следования, поэтому в начале можно указывать более определенные правила, а ближе к концу – более общие.

5.7.3. Обработка кодов ответа

Пока мы не ушли далеко от темы страниц с сообщением об ошибке 404, заметим, что существует возможность определять в файле UrlMappings отображения на собственные страницы для ответов с различными кодами. Если заглянуть в файл с правилами отображения по умолчанию, можно увидеть такое определение:

```
"500"(view:'/error')
```

Такое правило отображает ошибку «Внутренняя ошибка сервера с кодом 500» («Error 500: Internal Server Error») на адрес /views/error.gsp. Вы можете использовать этот механизм для отображения ответов HTTP с любыми кодами на одну из ваших страниц. Например, можно отобразить стандартную ошибку с кодом 500 на ставшую уже классической «страницу смерти» сайта Twitter:

```
"500"(view:'/failWhale')
```

Если необходимо, чтобы ошибка обрабатывалась действием контроллера, а не простым представлением GSP, это также возможно. Такую возможность удобно использовать для сохранения информации об адресе URL, вызвавшем появление ошибки с кодом 404. Ниже показано, как можно настроить обработку ответа с кодом 404 с помощью контроллера, предназначенного специально для этой цели:

```
"404"(controller: "errors", action: "notFound")
```

5.7.4. Прямое отображение в представление

Возможны ситуации, когда может потребоваться отобразить адрес URL непосредственно в представление, минуя логику контроллера. Классическим примером может служить домашняя страница, когда такое отображение можно реализовать следующим образом:

```
"/"(view:"/homepage/index")
```

Обратите внимание, что при определении правила мы опустили расширение .gsp.

5.7.5. Поддержка шаблонных символов

Механизм отображений адресов URL также поддерживает шаблонные символы в стиле утилиты Ant. Это особенно полезно для создания ссылок на свои собственные файлы.

Например, в случае динамического создания документов PDF, когда желательно определить постоянные ссылки, можно использовать такой прием:

```
"/documents/$docname**.pdf"(controller:"pdf", action:"render")
```

Применение шаблонных символов позволит обеспечить соответствие с файлом /documents/release-notes/myproject.pdf, а также с файлом /documents/manuals/myproject.pdf. В обоих случаях вы получите параметр `docname`, который можно использовать в действии контроллера. Параметр `docname` будет содержать относительный путь, соответствующий значению переменной `$docname` (например, release-notes/myproject и manuals/myproject). Данную возможность удобно использовать при разработке систем управления содержимым (Content Management Systems, CMS), когда документ PDF генерируется динамически на основе содержимого объекта предметной области.

На этом мы заканчиваем исследование файла UrlMappings. Теперь мы знаем о контроллерах все, что необходимо. Пришло время коротко вспомнить все, что мы рассмотрели.

5.8. В заключение

В этой главе мы исследовали массу приемов работы с контроллерами.

В начале главы мы реализовали вывод сообщений в хронологическом порядке и обеспечили возможность добавления новых сообщений в приложении Hubbub. Затем мы выделили операции с сообщениями в отдельную службу `PostService`, чтобы получить возможность пользоваться ею в последующих примерах. После приведения в порядок логики работы с сообщениями мы познакомились со всеми видами привязки данных, включая возможность определения белых и черных списков, обработку ошибок и управляющие объекты.

Затем мы немного поэкспериментировали с собственными типами содержимого, реализовав страницу просмотра профиля и возможность выгрузки фотографий. В заключение мы познакомились с логикой использования фильтров и механизмом отображения адресов URL.

Теперь повторим некоторые из ключевых идей, которые вы должны вынести из этой главы:

- *Используйте область видимости кадра (`flash`).* Область видимости кадра идеально подходит для передачи сообщений пользователю (когда выполняется перенаправление).
- *Используйте объект `errors` мудро.* Используйте объект `errors` в классах предметной области для отображения сообщений об ошибках, обнаруженных во время проверки. Используйте пакеты ресурсов, чтобы сделать сообщения об ошибках более подходящими для вашего приложения.
- *Используйте управляющие объекты.* Используйте управляющие объекты для обработки форм. Не нужно использовать их исключительно для проверки данных – они способны обеспечить выполнение более сложной логики обработки данных.

- *Разберитесь с механизмом привязки данных*. В платформе Grails имеется множество способов привязки данных, имеющих тонкие отличительные особенности. Изучите действие механизма привязки данных при работе с дочерними объектами, когда в именах параметров форм используются префиксы. Используйте механизм белых списков, чтобы при выполнении привязки данных не поставить под угрозу поля с важной информацией.
- *Будьте менее взыскательны к адресам URL*. Используйте действия по умолчанию, чтобы обеспечить большую терпимость к адресам URL; применяйте этот же прием для отображения собственных адресов URL. Адреса в виде постоянных ссылок проще запоминаются, а реализовать их не составляет большого труда.
- *Применяйте фильтры*. Используйте фильтры, когда необходимо выполнить некоторые операции для отдельных адресов URL или комбинаций контроллер-действие.

В этой главе мы многое узнали о внутренней организации контроллеров, а в следующей главе, опираясь на полученные знания, мы построим несколько представлений с целью придать приложению Hubbub неповторимый внешний вид.

В этой главе:

- Как действуют представления в файлах GSP
- Часто используемые библиотеки тегов GSP
- Отображение и обработка форм
- Создание собственных библиотек тегов
- Создание ошеломляющих макетов
- Визуальные и анимационные эффекты
- Интерактивные теги поддержки Ajax и удаленных взаимодействий

6

Разработка привлекательных представлений, форм и макетов

Большую часть этой книги мы занимались построением основы приложения Hubbub: обработкой входящих запросов, организацией взаимодействий с моделью данных, вызовом логики управления и созданием сообщений, пользователей и списков. Это было увлекательно, но не позволяло получить визуальное удовлетворение. Пришло время отложить реализацию функциональных возможностей и заняться пользовательским интерфейсом (UI) нашего веб-приложения.

Привлекательный интерфейс оказывает немалое воздействие на пользователей. Многие из них считают, что визуально привлекательное приложение обладает более высокой надежностью, более производительное и более удобное, чем приложение, имеющее непривлекательный внешний вид, хотя на самом деле ни одно из этих предположений может не соответствовать истине.

В этой главе мы сконцентрируемся на пользовательском интерфейсе. Мы рассмотрим основы составления форм и займемся исследованием поддержки различных броузеров с помощью макетов многократного использования. Кроме того, мы уделим внимание визуальным особенностям и реализуем несколько привлекательных анимационных эффектов, а также обеспечим некоторые взаимодействия с применением технологии Ajax. К концу главы приложение Hubbub будет искриться новым интерфейсом.

В первую очередь нам необходимо получить хорошее представление об основной технологии визуализации в платформе Grails – о серверных страницах Groovy (Groovy Server Pages, GSP), поэтому наше путешествие по UI начинается с тура по основным тегам форм GSP.

6.1. Основные теги форм

Мы уже с первой главы использовали файлы GSP для вывода разметки HTML, но не знакомились с ними достаточно подробно. Файлы GSP – это дальнейшее развитие технологии серверных страниц Java (Java Server Pages, JSP). Все, что вызывало сложности в применении JSP (особенно это относится к библиотекам тегов), было существенно упрощено в GSP. Когда вы обнаружите, что в состоянии вести разработку собственных библиотек тегов прямо во время работы приложения (когда отсутствует необходимость перезапускать его), вы влюбитесь в эту технологию.

В данном разделе мы рассмотрим все основные теги GSP, которые вы будете использовать изо дня в день при создании форм и реализации взаимодействий с пользователями. Будь то управление потоком выполнения, выполнение итераций или сложная обработка ошибок, в этих тегах вы найдете все необходимое. Начнем со знакомства с условной конструкцией «if».

6.1.1. Основные теги

Большую часть времени работы с представлениями платформы Grails вы будете тратить на основные теги GSP. Пришло время узнать о некоторых наиболее часто используемых из них.

Первые теги, с которыми вам необходимо познакомиться, – логические теги и теги для выполнения итераций. Все программисты просто обожают старый добрый условный оператор `if()`, поэтому сейчас мы представляем вам тег `<g:if>`, который в листинге 6.1 используется для проверки имени пользователя.

Листинг 6.1. Простой тег if

```
<g:if test="${user?.name == 'Chuck Norris'}">
    Roundhouse Kicks welcome here.
</g:if>
<g:elseif test="${user?.name == 'Jack Bauer'}">
    Lock up your Presidents.
</g:elseif>
<g:else>
    Take a number. We'll call you when we're ready.
</g:else>
```

Блоки `else` и `elseif` являются необязательными. Вы можете использовать конструкции `if`, `if ... else` или `if ... elseif ... else` в любых комбинациях по своему усмотрению.

Другим распространенным тегом является тег `<g:each>`, который выполняет итерации через коллекции или массивы. Он часто используется при выполнении операций над коллекциями объектов предметной об-

ласти, например для обхода всех пользователей, следящих за некоторыми событиями:

```
<g:each var="nextUser" in="${following}">
    <li>${nextUser.userId}</li>
</g:each>
```

Теги <g:if> и <g:each> – неотъемлемые составляющие операций выбора и итераций, и вы часто будете их использовать. Но вам придется также объединять их с некоторыми основными тегами ссылок для обеспечения пользователей возможностью перемещения по страницам приложения. Посмотрим, как можно организовать работу пользователя с помощью универсального тега <g:link>.

6.1.2. Полный комплект тегов ссылок

Еще одну группу часто используемых тегов составляют теги ссылок. Они обеспечивают удобный способ создания ссылок на действия контроллеров.

Например:

```
<g:link controller="post" action="global" >Back to Hubbub</g:link>
```

Вы можете опустить атрибут action, если ссылка указывает на действие по умолчанию.

Аналогично, если ссылка указывает на другое действие в текущем контроллере, можно опустить атрибут controller. Например, если бы предыдущим действием было /post/edit, можно было бы создать ссылку, указывающую на действие timeline (/post/timeline), как показано ниже:

```
<g:link action="timeline" >Back to Timeline</g:link>
```

Но иногда нежелательно создавать полный якорный тег – хочется лишь сослаться на некоторый адрес URL. Это может быть удобно при выполнении операций с применением технологии Ajax, когда приходится работать в терминах адресов URL, или при создании тегов <img>. При создании уменьшенных копий изображений со ссылками на пользователей, следящих за событиями, можно использовать такой прием:

```
<img src="<g:createLink action="renderImage" controller="image"
     id="${nextUser.userId}"/>" alt="${nextUser.userId}"/>
```

В результате будут создаваться ссылки на такие адреса URL, как /image/renderImage/glen. Не забывайте, что последняя часть адреса URL определяет содержимое атрибута params.id в вашем контроллере. Все теги ссылок поддерживают атрибуты action, controller и id, что обеспечивает значительную гибкость.

Несмотря на то что наибольшее распространение получили теги <g:link> и <g:createLink>, существует еще один тег, который вы будете использо-

вать не так часто: <g:createLinkTo> (да, имя тега может вводить в заблуждение). Тег <g:createLinkTo> удобно применять для создания ссылок на файлы, используемые приложением, и он часто используется для подключения файлов CSS и статических изображений. Например:

```
<link rel="stylesheet"
      href="${createLinkTo(dir:'css',file:'hubbub.css')}" />
```

Это даже не тег – это вызов метода!

Вы можете ссылаться на любые теги, используя синтаксис обращения к методам (с отображением в качестве аргумента) вместо вызовов в более классическом стиле JSP. Предыдущая строка кода эквивалентна следующему фрагменту:

```
<link rel="stylesheet" href="
    <g:createLinkTo dir='css' file='hubbub.css'/>
"/>
```

Некоторые находят стиль, напоминающий вызов метода, более удобочитаемым при определении вложенных тегов. В данном примере не сразу удается определить, какому тегу принадлежит та или иная закрывающая угловая скобка />, но хорошая интегрированная среда разработки помогает разобраться в этом.

Названия тегов ссылок похожи друг на друга, и это может приводить к путанице. Краткое их описание приводится в табл. 6.1.

Таблица 6.1. Теги ссылок

Тег	Описание
link	Генерирует тег <a href>, окружающий содержимое
createLink	Генерирует адрес URL для встраивания в другой тег
createLinkTo	Генерирует адрес URL локального файла

Совет

В главе 5 мы исследовали приемы отображения адресов URL с помощью файла UrlMappings.groovy. Вам будет интересно узнать, что теги ссылок действуют с учетом этих правил и будут соблюдать установленные отображения.

Следующий набор тегов, с которыми мы познакомимся, связан с формами. Посмотрим, насколько гибкие теги форм предоставляет платформа Grails.

6.1.3. Тур по тегам форм

Формы составляют основу интерактивности веб-приложений. Вам придется тратить массу времени на создание форм в файлах GSP и обработку их результатов в действиях контроллеров, поэтому очень важно по-

нимать, как использовать теги форм языка разметки HTML в файлах GSP. Вы легко можете смешивать и замещать HTML-теги форм тегами полей платформы Grails, но со временем вы поймете, что теги форм платформы Grails намного проще и удобнее.

В конце предыдущей главы мы показывали, как можно применять управляющие объекты для реализации процедуры регистрации пользователя, но мы не использовали какие-либо теги форм в форме регистрации. В листинге 6.2 представлена попытка создать форму регистрации на основе стандартных тегов форм платформы Grails. Мы добавили здесь несколько лишних полей, чтобы продемонстрировать имеющиеся возможности.

Листинг 6.2. Новая форма регистрации, демонстрирующая применение основных тегов форм

```
<dl>
    <g:uploadForm> ← Отмечает форму, поддерживающую выгрузку файлов
        <dt>User Id: </dt>
        <dd><g:textField name="userId" /></dd> ← Простое текстовое поле

        <dt>Password:</dt>
        <dd><g:passwordField name="password" /></dd> ← Поле ввода пароля,
        <dt>(repeat)</dt>                              скрывающее текст
        <dd><g:passwordField name="passwordRepeat" /></dd>

        <dt>Country:</dt>
        <dd><g:countrySelect name="country"
          noSelection="[‘’:’Choose your country...’]"/> ← Поле выбора страны

        <dt>Photo:</dt>
        <dd><input type="file" name="photo"/></dd> ← Поле выгрузки файла

        <dt>Timezone:</dt>
        <dd><g:timeZoneSelect name="timezone"/></dd> ← Поле выбора
                                                       часового пояса
        <dt>Who introduced you to Hubbub?</dt>
        <dd>

        <g:select name="referrer"
           from="${com.grailsinaction.Profile.list()}"
           optionKey="id"                                ← Заполняет список
           optionValue="fullName""                         выбора значениями
           noSelection="${[‘null’:’Please Choose...’]}" />  из базы данных

        </dd>

        <dt>Spam me forever:</dt>
        <dd>
            <g:checkBox name="spamMe" checked="true"/> ← Флажок
        </dd>
```

```
        <dt>Email Format:</dt>
        <dd>
        <g:radioGroup name="emailFormat"
            labels="['Plain','HTML']"
            values="['P', 'H']"
            value="H">
          ${it.label} ${it.radio}
        </g:radioGroup>
        </dd>

        <dt>
            <g:actionSubmit value="Register" />
        </dt>
        <dd>
            <g:actionSubmit value="Cancel" action="cancelRegister" />
        </dd>
    </g:uploadForm>
</dl>

<p>
    <g:link controller="post">Back to Hubbub</g:link>
</p>
```

Группа переключателей

Отправляет форму выбранному действию

Навигация в пределах приложения

Довольно большое количество тегов. Оставшуюся часть данного раздела мы посвятим их обсуждению, однако данный листинг позволяет вам увидеть их сначала в действии. На рис. 6.1 показано, как выглядит такая форма.

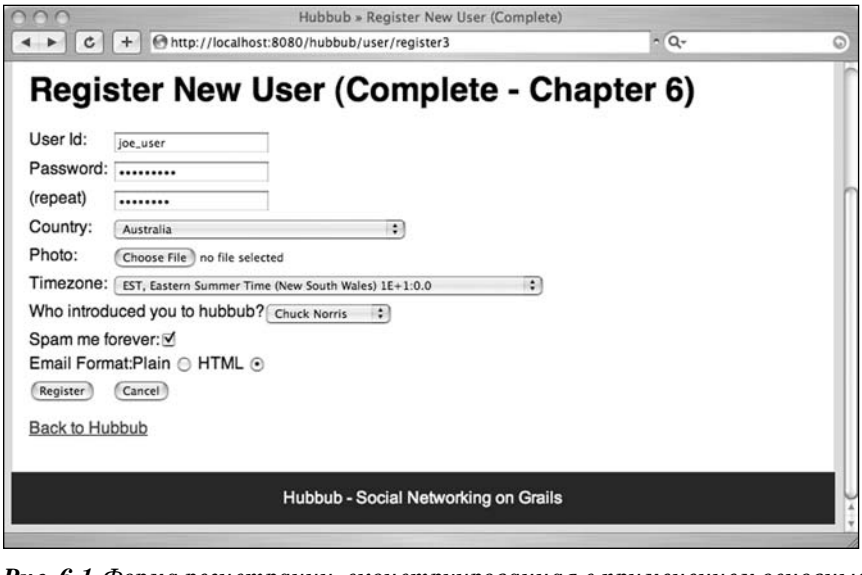


Рис. 6.1. *Форма регистрации, сконструированная с применением основных тегов форм*

Сама форма еще нуждается в некоторой доработке в смысле расположения компонентов, но с функциональной точки зрения она прекрасно удовлетворяет наши потребности. Форма дает достаточно полный обзор всех основных тегов форм. Давайте рассмотрим каждый из них по очереди.

Базовый тег form

Все формы в приложениях на платформе Grails начинаются с тега `<g:form>` или `<g:uploadForm>` в зависимости от того, необходимо или нет обеспечить поддержку возможности выгрузки файлов. Тег формы настраивается необязательными именами контроллера и действия:

```
<g:form controller="user" action="register">
```

Как и в случае с тегом `<g:link>`, обсуждавшимся выше, вы можете опираться на принятые соглашения. Например, внутри другого действия контроллера `User` (такого как /user/list) можно описать форму в терминах целевого действия, и по умолчанию будет использоваться текущий контроллер:

```
<g:form action="register">
```

Если необходимо, чтобы форма отправлялась текущему действию, можно опустить и имя действия.

Базовые теги textfield, textarea и passwordfield

Краеугольным камнем разработки веб-форм является использование текстовых полей. В число базовых вариантов входят теги `<g:textField>` (однострочное поле ввода), `<g:textArea>` (поле ввода многострочного текста) и `<g:passwordField>` (однострочное поле ввода, которое отображает вводимые символы, замещая их звездочками).

В нашей форме ввода используются однострочные версии полей ввода:

```
<g:textField name="userId" value="${newuser?.userId}"/>
<g:passwordField name="password" value="${newuser?.password}"/>
```

Атрибут `name` определяет имя поля в объекте `params` при передаче значений указанному действию. В данном примере значения полей будут передаваться в атрибутах `params.userId` и `params.password`.

Все поля форм поддерживают атрибут `value`, в котором определяется предустановленное значение поля. В предыдущем примере мы использовали оператор безопасного разыменования (`?.`) для заполнения формы первоначальными значениями.

Хотя мы и не использовали тег `textArea` в листинге 6.2, следует заметить, что этот тег имеет такой же формат, что и простые текстовые поля ввода:

```
<g:textArea name="bio" value="${newuser.bio}" rows="10" cols="60"/>
```

Такой тег приведет к появлению текстовой области с указанными размерами, заполненной содержимым свойства `newuser.bio`.

Как передать собственные атрибуты HTML в библиотеку тегов Grails?

Тег `textArea` явно поддерживает только атрибуты `name` и `value`, но он передает неизвестные ему атрибуты непосредственно генерируемому тегу HTML. Это означает, что можно использовать стандартные атрибуты `rows` и `cols` HTML-тега `TextArea`.

Такой механизм передачи атрибутов имеется во всех тегах платформы Grails, поэтому, если вам необходимо определить атрибуты для получающихся в результате тегов HTML (например, атрибут `class`, определяющий стиль CSS), просто добавьте их в число атрибутов используемого тега, и они автоматически будут переданы в окончательную разметку HTML.

Раскрывающийся список select

Одним из самых сложных (и изощренных) тегов форм платформы Grails является тег `<g:select>`. Тег `select` позволяет генерировать раскрывающиеся списки, заполненные информацией из источников данных или из внутренних списков и диапазонов, присутствующих в приложении.

Тег ожидает получить атрибут `from`, содержащий список объектов (или диапазон, если это потребуется). Вы к тому же должны определить атрибуты `optionKey` и `optionValue`, представляющие свойства каждого объекта, которые должны использоваться для создания раскрывающегося списка.

Обратимся еще раз к примеру формы регистрации из листинга 6.2:

```
<g:select name="referrer"
    from="${com.grailsinaction.Profile.list()}"
    optionKey="id"
    optionValue="fullName"
    noSelection="${['null':'Please Choose...']}" />
```

Атрибут `from` извлекает список профилей из базы данных. Атрибут `optionKey` является представлением поля `id` объекта `Profile`, значение которого будет возвращено в виде параметра `referrer`. Атрибут `optionValue` является представлением строки, которая будет отображаться непосредственно в раскрывающемся списке, – она будет содержать значение поля `fullName` объекта `Profile`.

Особый случай, когда в списке не выбран ни один из пунктов, обрабатывается атрибутом `noSelection`, в котором передается отображение с ключом, равным `'null'` (обратите внимание: это строка, а не значение).

В атрибуте `from` передается фрагмент программного кода на языке Groovy, поэтому вы можете использовать здесь любой динамический ме-

тод поиска, возвращающий список, или предоставлять свой статический список:

```
<g:select from="['Groovy', 'Java', 'Python']" name="preferredLanguage"/>
```

Если атрибуты `optionKey` и `optionValue` не определены, тег вызовет метод `toString()` для каждого элемента списка для получения ключа и отображаемого значения, что прекрасно подходит для случая использования статического списка.

Флажки и переключатели

Поддержка флажков реализована в виде простого тега, содержащего имя и значение, причем последнее определяет состояние флажка:

```
<g:checkBox name="spamMe" value="${newuser.spamMe}"/>
```

Переключатели обладают большей гибкостью. Наиболее типичный подход к их использованию заключается в применении тега `radioGroup`. В листинге 6.2 мы реализовали выбор формата для электронных писем (`emailFormat`) с помощью тега `radioGroup`:

```
<g:radioGroup name="emailFormat"
        labels="['Plain','HTML']"
        values="['P', 'H']"
        value="H">
    ${it.label} ${it.radio}
</g:radioGroup>
```

Обратите внимание, что для переключателей необходимо определить оба значения атрибутов `labels` и `values`.

Тег `radioGroup` отличается от тега `checkBox` еще и тем, что выполняет итерации по содержимому всех элементов. Именно поэтому мы должны указать элементы `${it.label}` и `${it.radio}`, которые используются для отображения каждого переключателя. Такой итеративный подход обеспечивает значительную гибкость в том, как будут отмечены переключатели, но это уникальный случай среди тегов форм платформы Grails.

Платформа Grails предоставляет также тег `<g:radio>` для случаев, когда требуется создать переключатели без выполнения итераций, но при этом вам придется установить атрибут `name` для каждого переключателя, входящего в группу, в одно и то же значение. По этой причине часто бывает надежнее использовать подход на основе тега `radioGroup`.

Скрытые поля

Хотя мы не демонстрировали использование скрытых полей в листинге 6.2, они являются всего лишь частным случаем тега `<g:textField>`, не отображаемого броузером. Например:

```
<g:hiddenField name="dateRendered" value="${new Date()}" />
```

Вы безо всяких ограничений можете помещать любые текстовые значения в теги hiddenField, и они будут возвращаться в составе отображения params вместе с формой.

Обработка нескольких кнопок submit

Наличие нескольких кнопок в одной форме может осложнить ее обработку. Традиционно было принято передавать форму одному контроллеру, а в контроллере предусматривать конструкцию switch, вызывающую то или иное действие в зависимости от того, на какой кнопке щелкнул пользователь.

Платформа Grails предоставляет тег кнопки <g:actionSubmit>, который упрощает передачу формы требуемому действию, решая все проблемы, связанные с выбором действия, за вас. Например:

```
<g:actionSubmit value="Register" />
<g:actionSubmit value="Cancel" action="cancelRegister" />
```

Обратите внимание, что в этом примере мы определили атрибут action только для одной кнопки. Если имя действия не указано, платформа Grails будет полагать, что оно соответствует имени кнопки, в котором используются только символы нижнего регистра (для первой кнопки это будет действие /user/register). Если вы используете иные соглашения, вы можете прямо указывать имя действия (как это сделано во второй строке, где вызывается действие /user/cancelRegister).

Поля выбора даты

Одним из самых слабо документированных тегов платформы Grails является тег datePicker. Этот тег создает последовательность элементов управления, дающих пользователю возможность выбирать значение даты.

Тег отображается как последовательность раскрывающихся списков, с помощью которых пользователь может выбрать число месяца, название месяца и год (дополнительно можно обеспечить возможность выбора часов, минут и секунд), что идеально подходит для реализации приложений, в которых особое внимание уделяется датам, таких как приложения бронирования мест в гостиницах или билетов на самолет. Когда пользователь отправит такую форму, механизм привязки данных платформы Grails преобразует коллекцию элементов управления в объект Date языка Java, обеспечив простоту дальнейшей обработки этого значения.

Конструируется тег datePicker очень просто:

```
<g:datePicker name="arrivalDate"
    value="${myBooking.arrivalDate}"
    precision="day"/>
```

Поле precision определяет, какие поля будут доступны пользователю для изменения. В этом примере значение day обеспечит появление

в форме полей для ввода числа месяца, месяца и года. Если в поле `precision` указать значение `minute`, будут созданы дополнительные поля выбора часов и минут. Полный список параметров тега вы найдете в справочном руководстве к платформе Grails.

Одним общим недостатком программного обеспечения при работе с тегом `datePicker` является необходимость использования в контроллере механизма привязки данных платформы Grails (либо в виде вызова метода `bindData()`, либо в виде операции присваивания свойств) для преобразования элементов управления тега `datePicker`, задействованных в работе . На практике это означает, что использовать привязку свойств в стиле Groovy для сохранения нельзя. Например, инструкция `new Booking(params).save()` не может использоваться для обработки формы, содержащей тег `datePickers`. Кроме того, не следует ожидать, что параметр `params.arrivalDate` будет содержать дату – фактическое значение даты конструируется при привязке данных из нескольких элементов управления HTML, генерируемых тегом `datePicker`.

Краткий обзор тегов

К настоящему моменту мы узнали массу базовой информации и познакомились со многими распространенными тегами, которые вам, вероятно, придется использовать. В табл. 6.2 и табл. 6.3 перечислены соответственно наиболее часто и реже используемые теги, с которыми вы будете сталкиваться при разработке приложений на платформе Grails.

Таблица 6.2. Основные теги форм

Тег	Описание
`g:form`	Создает стандартный элемент `form` разметки HTML
`g:uploadForm`	Создает форму, поддерживающую вложенные элементы выгрузки файлов
`g:textField`	Создает однострочное текстовое поле ввода
`g:passwordField`	Создает однострочное текстовое поле ввода, скрывающее вводимые символы
`g:textArea`	Создает многострочное текстовое поле ввода
`g:checkbox`	Создает флажок, возвращающий значение `true` или `false`
`g:radio` и `g:radioGroup`	Создает переключатель, который может быть добавлена в группу, для случаев, когда может быть выбран только один из вариантов
`g:select`	Создает раскрывающийся список с предустановленными элементами
`g:hiddenField`	Создает поле, невидимое в окне броузера
`g:actionSubmit`	Создает кнопку, вызывающую отправку формы определенному действию контроллера

Таблица 6.3. Реже используемые теги форм

Тег	Описание
g:datePicker	Создает последовательность раскрывающихся списков для выбора даты и времени
g:currencySelect	Отображает список валют, доступных для выбора
g:paginate	Отображает последовательность ссылок для навигации по страницам многостраничного набора данных
g:timezoneSelect	Отображает список часовых поясов Java, доступных для выбора
g:countrySelect	Отображает список стран, доступных для выбора
g:message	Отображает сообщение из пакета сообщений
g:hasErrors	Проверяет наличие ошибок, выявленных при проверке ограничений объекта
g:renderErrors	Отображает в виде списка все сообщения об ошибках, выявленных при проверке ограничений объекта
g:javascript	Встраивает программный код на языке JavaScript, или ссылку на файл, или библиотеку JavaScript

Совет

Платформа Grails позволяет использовать в файлах GSP библиотеки тегов JSP. Если у вас уже имеются какие-либо библиотеки тегов, нет никакой необходимости переписывать их – просто добавьте директиву `<% @taglib %>` в файл GSP. Примеры использования директивы можно найти в электронной документации по адресу: *http://grails.org/doc/1.1.x/*.

На этом мы завершаем блиц-тур по основным тегам форм платформы Grails. На протяжении всей оставшейся части книги мы будем использовать эти теги, поэтому не чувствуйте себя брошенными на произвол судьбы – вам еще не раз представится возможность ознакомиться с практическими примерами!

Существует еще один тег, заслуживающий особого внимания, поскольку он реализует возможность постраничного просмотра наборов данных, разбитых на страницы. В следующем разделе мы добавим в приложение Hubbub возможность постраничного просмотра списка сообщений.

6.1.4. Добавление возможности постраничного просмотра списка сообщений

Одним из самых необычных и интересных основных тегов является тег `<g:paginate>`. Он отображает последовательность навигационных ссылок, позволяющих перемещаться по страницам с результатами.

Глобальный список сообщений в нашем приложении Hubbub будет включать все сообщения, имеющиеся в системе, поэтому он будет расти чрезвычайно быстро. Добавление тега `<g:paginate>` позволит нам добавить навигационные ссылки, указывающие на страницы с более старыми сообщениями, как показано на рис. 6.2.

При условии, что контроллер определил количество записей на одной странице, использование тега становится простым делом – достаточно передать ему общее число записей. Вы можете также сообщить тегу количество результатов, которое требуется отображать на одной странице, с помощью параметра `max` (параметр `max` является необязательным и по умолчанию имеет значение 10).

```
<g:paginate total="${postCount}" max="25" />
```

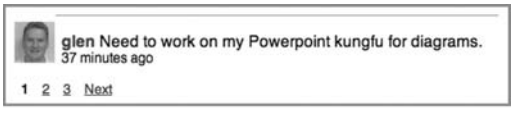


Рис. 6.2. *Тег <g:paginate> в действии*

Тег делит общее количество записей на максимальное количество записей на одной странице и создает ссылки, указывающие на отдельные страницы. Каждая ссылка указывает на текущий контроллер и передает ему значения `max` и `offset` (например, `max = 25, offset = 50`). Когда пользователь щелкает на ссылке, указывающей на третью страницу в списке сообщений, выполняется переход по адресу `/post/global?max=25&offset=50`.

Динамические методы поиска знают о существовании переменных `max` и `offset`, поэтому, когда вызывается действие, подобное тому, что приводится ниже, оно передает методу `Post.list()` коллекцию `params`, содержащую значения `max` и `offset`:

```
def posts = Post.list(params)
def postCount = Post.count()
[ posts, postCount ]
```

Это означает, что метод `list()` учтет данные значения в запросе, сместится в списке результатов на указанное значение и вернет следующую «страницу» сообщений. Попробуйте задействовать тег постраничного просмотра в глобальном списке сообщений приложения Hubbub, и вы сможете увидеть волшебство в действии (а заодно и понять, что именно происходит).

Теперь у вас должно быть четкое представление обо всех основных тегах форм платформы Grails. Но иногда могут возникать ситуации, когда ни один из стандартных тегов не будет удовлетворять вашим потребностям, тогда вам придется писать свой собственный тег. Как это делается, мы рассмотрим в следующем разделе.

6.2. Расширение представлений собственными тегами

Мы использовали множество предоставляемых платформой Grails стандартных тегов форм, условных тегов, тегов обработки ошибок и даже выполнения итераций. Но как быть, когда сил стандартных тегов оказывается недостаточно? Именно для таких ситуаций предоставляется возможность создавать свои собственные теги, поэтому сейчас мы попробуем написать несколько таких тегов.

Платформа Grails поддерживает три типа нестандартных тегов, перечисленные в табл. 6.4.

Таблица 6.4. Типы нестандартных тегов, поддерживаемые платформой Grails

Тип	Описание	Пример
Простые теги	Выводят разметку HTML	`<g:button>`
Логические теги	Выполняют вывод, опираясь на проверку условия	`<g:if test="">`
Итеративные теги	Выполняют итерации через коллекцию, отображают содержимое каждого элемента	`<g:each in="">`

Приятно узнать, что имеется возможность писать любые простые, логические и итеративные теги. Мы напишем по одному тегу каждого типа, чтобы получить некоторое представление о том, как это делается.

6.2.1. Простые теги

В нашем списке сообщений в приложении Hubbub можно было бы использовать относительное время. Большинство приложений в стиле Web 2.0 поддерживают возможность вывода времени в формате «5 минут назад», что достаточно удобно и снимает проблему разницы часовых поясов. Давайте создадим тег `dateFromNow`, который будет получать дату и выводить ее в новом формате.

Вас уже не должно удивлять, что все начинается с создания шаблона:

```
grails create-tag-lib date
```

Эта команда создаст заготовку библиотеки тегов в файле /grails-app/taglib/DateTagLib.groovy. Реализация простого тега связана с обработкой параметра `attrs` и отображением разметки HTML в поток вывода.

Мы будем вызывать наш тег, как показано ниже:

```
<g:dateFromNow date="${post.created}"/>
```

Реализация тега dateFromNow в классе DateTagLib заключается в создании замыкания с именем dateFromNow и обработке атрибута date. Первая черновая реализация приводится в листинге 6.3.

Листинг 6.3. Создание собственного тега dateFromNow

```
class DateTagLib {

    def dateFromNow = { attrs ->

        def date = attrs.date ◄— Обращение к атрибуту date
        def niceDate = getNiceDate(date) // какая-либо реализация этого метода

        out << niceDate ◄— Вывод отформатированной даты в страницу

    }
}
```

Мы извлекаем значение даты из атрибута attrs.date и передаем его нашему методу getNiceDate(), который превращает это значение в строку формата «x минут назад», после чего мы выводим результат в выходной поток. В языке Groovy перегруженный оператор << для потоковых объектов вызывает метод write(), поэтому здесь нет ничего, на чем следовало бы остановиться, кроме метода getNiceDate(), реализация которого приводится в листинге 6.4.

Листинг 6.4. Реализация преобразования даты в относительный формат для приложения Hubbub

```
static String getNiceDate(Date date) {

    def now = new Date()

    def diff = Math.abs(now.time - date.time)
    static final long second = 1000
    static final long minute = second * 60
    static final long hour = minute * 60
    static final long day = hour * 24

    def niceTime = ""

    long calc = 0;

    calc = Math.floor(diff / day)
    if (calc) {
        niceTime += calc + " day" + (calc > 1 ? "s " : " ")
        diff %= day
    }

    calc = Math.floor(diff / hour)
    if (calc) {
```

```
            niceTime += calc + " hour" + (calc > 1 ? "s " : " ")
            diff %= hour
        }

        calc = Math.floor(diff / minute)
        if (calc) {
            niceTime += calc + " minute" + (calc > 1 ? "s " : " ")
            diff %= minute
        }

        if (!niceTime) {
            niceTime = "Right now"
        } else {
            niceTime += (date.time > now.time) ? "from now" : "ago"
        }

        return niceTime

    }
```

Теперь у нас имеется тег многократного использования, который мы можем применять везде, где потребуется вывести относительное время в формате «x минут назад». В примере тега мы использовали всего один атрибут (`date`), но вообще теги могут иметь множество атрибутов.

Что такое g:? Использование собственных пространств имен

Во всех наших примерах имена тегов имеют вид `<g:tagName>`. Префикс `g:`, используемый в именах тегов, называется *пространством имен* тега. Если в своей библиотеке тегов вы не определили собственное пространство имен, по умолчанию будет использоваться префикс `<g:>`. Можно ли его изменить? Для этого требуется добавить объявление нового пространства имен в файл DateTagLib.groovy:

```
static namespace = "hubbub"
```

После этого вы сможете обращаться к своим тегам, используя формат `<hubbub:dateFromNow ...>`. На практике лучше задавать короткие названия пространств имен с целью уменьшения объема ввода с клавиатуры; для приложения Hubbub мы определим пространство имен «h» и, значит, сможем использовать такие теги, как `<h:dateFromNow>`.

Вы всегда должны объявлять пространство имен для своих тегов, чтобы уменьшить вероятность конфликта имен с другими тегами, – в частности с новыми встроенными тегами платформы Grails, которые могут появиться в будущем.

Теперь, когда у нас имеется собственный простой тег, займемся поддержкой платформой Grails, обеспечивающей возможность создавать логические теги.

6.2.2. Логические теги

Иногда бывает необходимо отобразить то или иное содержимое в зависимости от соблюдения какого-либо условия. Предположим, что нам требуется отобразить определенное содержимое только для пользователей броузера Internet Explorer, например такое, как показано ниже:

```
<h:lameBrowser userAgent="MSIE">
    <p>Dude, Firefox really is better. No, really. </p>
</h:lameBrowser>
```

Нам необходимо, чтобы текст сообщения могли видеть только пользователи IE, а все остальные броузеры пропускали его. Для этого реализуем тег `UtilTagLib`, как показано в листинге 6.5.

Листинг 6.5. Тег lameBrowser, демонстрирующий логические теги в действии

```
class UtilTagLib {

    static namespace = "h"

    def lameBrowser = { attrs, body ->

        if(request.getHeader('User-Agent')
        ↪          =~ attrs.userAgent ) {
            out << body()
        }
    }

}
```

← Проверка заголовка User-Agent на соответствие указанному значению

← Отображает содержимое тега

Обратите внимание, что логические теги принимают два параметра: параметр `attrs`, который уже встречался нам выше, и `body`, который заключает в себе содержимое тега. Если проверка возвращает значение `true`, мы отображаем значение `body()` в поток вывода; в противном случае ничего не выводится.

Теги такого рода часто используются для обеспечения безопасности. Например, при использовании расширения JSecurity вам часто придется пользоваться таким удобным тегом, как `isLoggedIn`, для вывода информации по условию:

```
<jsec:isLoggedIn>
    <div>Logged in as: <jsec:principal/>
        (<g:link controller="auth" action="signOut">
          sign out
        </g:link>)</div>
</jsec:isLoggedIn>
```

Обратите внимание, что содержимым логического тега может быть достаточно сложный фрагмент GSP, вызывающий другие теги из других библиотек тегов.

Теперь, закончив изучение логических тегов, обратимся к последнему типу нестандартных тегов, поддерживаемому платформой Grails: итеративным тегам.

6.2.3. Итеративные теги

Итеративные теги являются самыми сложными из всех типов нестандартных тегов. Теги этого типа выполняют множество вызовов метода `body()` с различными входными значениями на каждой итерации.

Боковая колонка в приложении Hubbub должна содержать изображения пользователей, за сообщениями которых мы пытаемся следить. Мы могли бы реализовать это с помощью стандартного тега `<g:each>`, как показано ниже:

```
<!-- Пользователи, за сообщениями которых я слежу -->
<div id="friendsThumbnails">
    <g:each var="followUser" in="${following}">
        <img src="
            <g:createLink action="tiny" controller="image"
                id="${followUser.userId}"/>
            "alt="${followUser.userId}"/>
    </g:each>
</div>
```

С другой стороны, тег `eachFollower` может обеспечить более наглядное решение и позволяет реализовать эту возможность, как показано ниже:

```
<h:eachFollower in="${following}">
    <img src="
        <g:createLink action="tiny" controller="image"
            id="${followUser.userId}"/>
        " alt="${followUser.userId}"/>
</h:eachFollower>
```

Тег `eachFollower` можно реализовать с применением итераций, внутри которых вызывается метод `body`:

```
def eachFollower = { attrs, body ->

    def followers = attrs.followers
    followers?.each { follower ->

        body(followUser: follower)

    }

}
```

Но это лишь простое копирование функциональности стандартного тега <g:each>. Честно говоря, мы с трудом можем представить себе ситуации, где использование более наглядной семантики тега <g:each> только ухудшает реализацию. Разумнее потратить свое время на реализацию тега для вывода уменьшенных фотографий пользователей, чем мы и займемся ниже.

6.2.4. Вызов одного тега из другого

При разработке собственных тегов часто возникает необходимость задействовать в своей реализации стандартные теги платформы Grails. Например, при создании тега, включающего ссылки на стандартные действия контроллера, будет, вероятно, желательно воспользоваться стандартным тегом createLink.

Попробуем применить эту идею в нашем новом теге, чтобы посмотреть, как это делается. Возьмем для примера создание адресов URL для уменьшенных изображений пользователей, следящих за появлением новых сообщений. Как было показано в предыдущем разделе, мы теперь делаем это, используя разметку HTML с динамическим конструированием URL:

```
<img src="
    <g:createLink action="tiny" controller="image"
        id="${followUser.userId}"/>
    " alt="${followUser.userId}"/>
```

Вероятно, было бы гораздо удобнее спрятать все это в собственный тег tinyThumbnail, как показано ниже:

```
<h:tinyThumbnail userId="${followUser.userId}"/>
```

В своей реализации тега tinyThumbnail нам хотелось бы воспользоваться функциональностью стандартного тега g:createLink. И это вполне возможно! Ниже приводится реализация нашего тега:

```
def tinyThumbnail = { attrs ->

    def userId = attrs.userId
    out << "<img src='"
    out << g.createLink(action: "tiny",
        controller: "image", id: userId)
    out << "' alt='${userId}'"

}
```

← Использование существующего тега платформы Grails

Как мы уже видели выше, имеется возможность вызывать существующие теги, используя форму обращения к методам. При этом префикс пространства имен должен использоваться как объект, относительно которого вызывается метод («g» – для тегов из стандартной библиотеки тегов, как уже было показано выше).

Теперь, когда вы знаете, как создавать свои собственные теги, можно перейти к улучшению макетов размещения элементов приложения.

6.3. Добавление привлекательных макетов

Массу времени мы потратили на реализацию функциональных возможностей приложения Hubbub, но еще ни минутки – на его внешний вид. Платформа Grails позволяет не только быстро разрабатывать функциональные возможности, но и улучшать внешний вид приложения. В основе оформления внешнего вида приложений на платформе Grails лежат макеты.

Мы используем макеты, начиная с главы 4 (и касались их даже в главе 1), но еще не рассматривали, как они действуют. Вся основная функциональность макетов платформы Grails реализована в виде популярной Java-библиотеки макетов SiteMesh (*http://www.opensymphony.com/sitemesh/*). Возможно, раньше вам не приходилось сталкиваться с библиотекой SiteMesh, поэтому сейчас мы познакомимся с ней поближе.

6.3.1. Введение в SiteMesh

Библиотека SiteMesh действует как декоратор страниц. Вы реализуете отображение простой разметки HTML своих страниц, после чего она передается декоратору SiteMesh (который добавляет заголовок, нижний колонтитул, боковые колонки и другие элементы), и окончательный результат отображается броузером. Этот процесс схематически изображен на рис. 6.3.

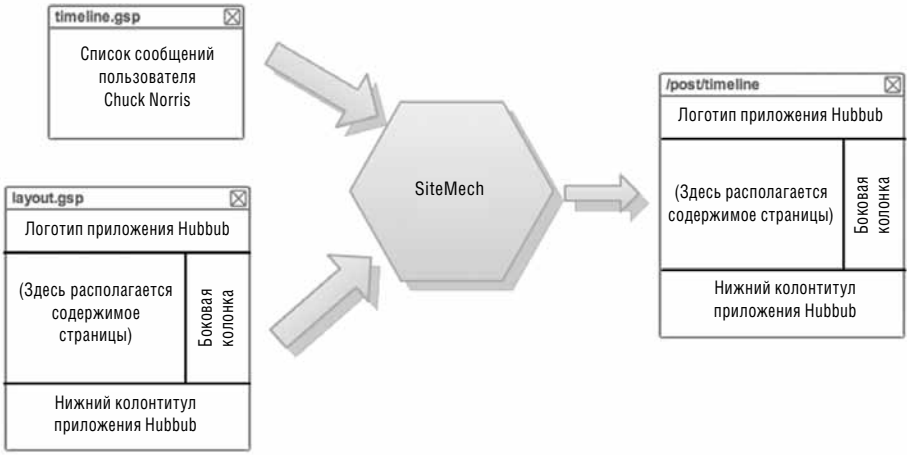


Рис. 6.3. *Создание страниц с помощью библиотеки SiteMesh*

Одной из самых мощных и сложных возможностей библиотеки SiteMesh является ее способность объединять элементы страницы с вашими де-

кораторами. Вам будет понятнее, если объяснить это на примере, поэтому сейчас мы посмотрим, как можно воспользоваться библиотекой SiteMesh для реализации внешнего вида приложения Hubbub.

Процесс объединения

Вернемся к нашему шаблону из главы 4 – простому макету страницы, добавляющему поле заголовка и простой нижний колонтитул. Мы поместили шаблон в файл /grails-app/views/layouts, но так как нам требуется, чтобы шаблон применялся и к динамическим формам, создаваемым механизмом скаффолдинга, мы отредактировали шаблон и сохранили измененную версию в файле /grails-app/views/layouts/main.gsp. Этот шаблон приводится в листинге 6.6.

Листинг 6.6. Простой шаблон для приложения Hubbub

```
<html>
    <head>
        <title>Hubbub &raquo;
            <g:layoutTitle default="Welcome" /> ← ❶ Добавляет элемент title
        </title>                                    из страницы с содержимым
        <link rel="stylesheet" href="
            <g:createLinkTo dir='css' file='hubbub.css'/>
        "/>
        <g:javascript library="application" />
        <g:layoutHead /> ← ❷ Добавляет элемент head из страницы с содержимым
    </head>
    <body>
        <div>
            <div id="hd">
                <a href="<g:createLinkTo dir="/"/>">
                    <img id="logo" src="
                        ${createLinkTo(dir: 'images',
                            file: 'headerlogo.png')}
                    " alt="hubbub logo"/>
                </a>
            </div>
            <div id="bd"><!-- начало тела страницы -->
                <g:layoutBody/> ← ❸ Добавляет элемент body из страницы с содержимым
            </div> <!-- конец тела страницы -->
            <div id="ft">
                <div id="footerText">
                    Hubbub - Social Networking on Grails
                </div>
            </div>
        </div>
    </body>
</html>
```

Макет начинается с размещения заголовка. Все элементы заголовка предваряются префиксом «Hubbub >>», за которым следует заголовок

целевой страницы ❶. Если целевая страница не имеет собственного элемента <title>, отображается текст «Welcome» («Добро пожаловать»). Это позволяет нам изменить все заголовки в одном месте.

Затем мы используем тег <g:layoutHead> из библиотеки SiteMesh ❷ для добавления содержимого элемента <head> целевой страницы и тег <g:layoutBody> ❸ для добавления содержимого элемента <body> целевой страницы.

После того как отработают все эти теги, страница представления будет объединена с шаблоном макета, что обеспечит единый внешний вид всего приложения. Такой подход к созданию разметки дает двойную выгоду. Страницы содержимого становятся проще и прозрачнее, а страницы макета упрощают изменение внешнего вида всего приложения за счет изменения единственного файла.

Отображение версии приложения в нижнем колонтитуле

Тег <g:meta> обеспечивает доступ ко всем записям в файле application.properties приложения на платформе Grails, находящемся в корневом каталоге проекта. Часто бывает удобно выводить информацию о версии приложения в нижнем колонтитуле страницы, чтобы ее можно было указать в сообщениях о найденных ошибках и чтобы убедиться, что новая версия была установлена благополучно. Отобразить версию приложения и платформы Grails можно с помощью следующего фрагмента программного кода:

```
Version <g:meta name="app.version"/>
on Grails <g:meta name="app.grails.version"/>
```

Вы можете изменить версию приложения в любой момент, выполнив команду grails setversion в командной строке или вручную отредактировав файл application.properties. Номер версии приложения оказывает влияние на имя, присваиваемое целевому файлу WAR, поэтому лучше всего изменять номер версии при каждой сборке дистрибутива, что позволит вам сохранять более старые копии файлов WAR и выполнять откат, если что-то пойдет не так. (Это не устраняет необходимость пользоваться системой управления версиями, но позволяет быстро вернуться к прежней версии.)

Применение макетов по соглашению

В главе 1 мы познакомили вас с простым способом применения макетов по соглашению. Теперь мы рассмотрим их возможные варианты и поближе познакомимся с областью их применения.

Например, при работе с контроллером Post мы могли бы обеспечить его декорирование с помощью приемов, перечисленных в табл. 6.5.

Таблица 6.5. Соглашения по применению макетов

Применить шаблон...	Пример соглашения или приема
Ко всем действиям контроллера	Создать макет в файле с именем /layouts/post.gsp
К определенному действию контроллера	Создать макет в файле /layouts/post/list.gsp
К части целевой страницы	Включить в целевую страницу тег: `<g:applyLayout name="postFragment">Hi</g:applyLayout>`
К отдельной странице, преодолев принятые соглашения	Включить в целевую страницу тег: `<meta name="layout" content="vanilla"/>`

Использование соглашений устраняет необходимость определять тег `<meta>` и одновременно облегчает сопровождение приложения. Когда мы открываем файл GSP и видим, что в нем отсутствуют теги `meta`, мы точно знаем, где искать декоратор (/views/layouts/<controllerName>), что делает сопровождение приложения более удобным.

Пока мы занимаемся изучением макетов страниц, исследуем некоторые способы, которые позволят упростить их.

6.3.2. Стандартизация макетов страниц

Если вам приходилось когда-нибудь разрабатывать веб-приложения с многоколоночными макетами страниц, вы знаете, насколько сложными могут быть таблицы стилей CSS макетов. Некоторые броузеры не совсем корректно отображают плавающие элементы `div`, что осложняет создание колонки вдоль правого края. Порой очень сложно обеспечить растягивание некоторых разделов страницы при сохранении неизменными других. Существует также масса проблем, связанных с размером шрифтов, что существенно снижает привлекательность внешнего вида страниц в Internet Explorer.

Все эти проблемы необходимо решать с помощью макетов приложения, и сейчас мы покажем вам некоторые наиболее удачные приемы работы с ними. Сначала мы схематически изобразим, как мы хотим, чтобы приложение Hubbub выглядело. На рис. 6.4 приводится грубый набросок внешнего вида приложения.

Как видите, мы предусмотрели область заголовка, ниже которой располагаются область вкладок и правая боковая колонка, а еще ниже – нижний колонтитул, занимающий всю ширину страницы. Если попытаться воспроизвести такую страницу вручную, вас ожидает шок. Чтобы воспроизвести подобный макет, придется проделать огромный объ-

ем работы с таблицами стилей CSS и преодолеть невероятные трудности обеспечения одинакового внешнего вида во всех броузерах. В платформе Grails выбран более разумный способ создания макетов на основе CSS с помощью библиотеки YUI Grids.

Рис. 6.4. *Макет пользовательского интерфейса приложения Hubbub*

Библиотека YUI Grids предоставляет простой механизм управления расположением компонентов страницы, основанный на применении таблиц стилей CSS. Это файл CSS размером 4 Кбайт, который можно добавить в приложение и навсегда забыть о проблемах совместимости броузеров.

Вы можете загрузить файл CSS и ознакомиться с исчерпывающей документацией на странице проекта YUI Grids (*http://developer.yahoo.com/yui/grids/*), там же вы найдете программный продукт YUI Grids Builder. Вы даже можете воспользоваться веб-версией построителя для создания своего простого макета. На рис. 6.5 изображено приложение YUI Grids Builder в процессе создания макета, представленного выше.

Создав с помощью YUI Grids Builder макет страницы, щелкните на кнопке Show Code (Показать код), и вы получите шаблон, готовый к дальнейшей настройке. В листинге 6.7 приводится код разметки, сгенерированный программой YUI для макета, изображенного на рис. 6.5.

Листинг 6.7. Код разметки, сгенерированный программой YUI

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
 "http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
  <title>YUI Base Page</title>
  <link rel="stylesheet"
    href="http://yui.yahooapis.com/2.5.1/build/
    ➥reset-fonts-grids/reset-fonts-grids.css" type="text/css">
</head>
```

```
<body>
<div id="doc3" class="yui-t4">
  <div id="hd"><h1>Hubbub Logo</h1></div>
  <div id="bd">
    <div id="yui-main">
      <div class="yui-b"><div class="yui-g">
        <!-- ЗДЕСЬ РАСПОЛАГАЮТСЯ ВАШИ ДАННЫЕ -->
      </div>
    </div>
  </div>
  <div class="yui-b">
    <!-- ЗДЕСЬ РАСПОЛАГАЮТСЯ ВАШИ ЭЛЕМЕНТЫ НАВИГАЦИИ --></div>
  </div>
  <div id="ft">Hubbub - Social Networking on Grails</div>
</div>
</body>
</html>
```

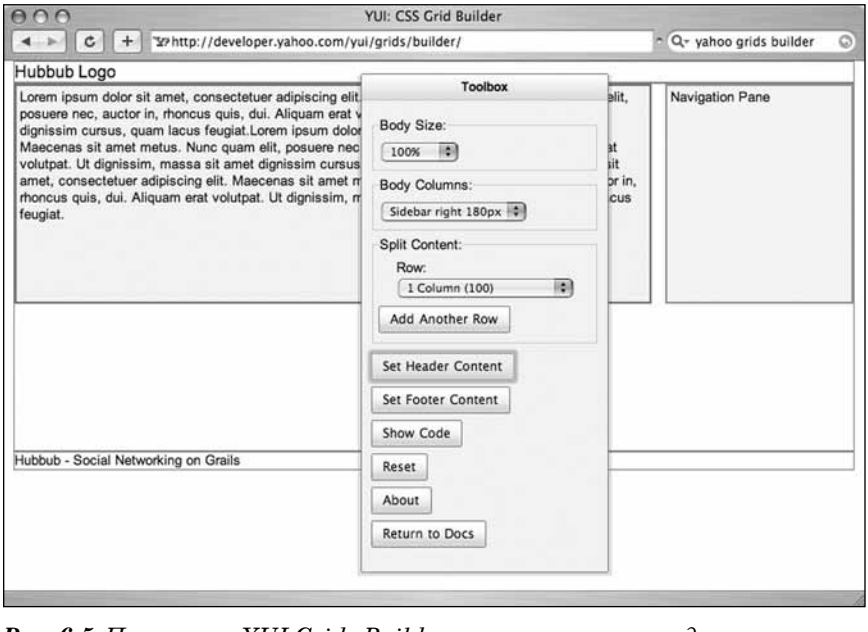


Рис. 6.5. *Программа YUI Grids Builder позволяет легко создавать макеты на основе CSS, совместимые со всеми броузерами*

Благодаря нескольким элементам `div`, добавленным программой YUI, мы получили макет на основе CSS, который будет выглядеть одинаково во всех броузерах. Все, что осталось теперь сделать, – это добавить теги `<g:layout>`, чтобы объединить макет с заголовком и телом целевой страницы, и дополнить макет некоторыми стилями CSS, реализующими наши представления о внешнем виде страниц.

Обратите внимание, что в макете имеется несколько фрагментов, которые мы сможем использовать в разных местах. Например, колонка справа должна отображать несколько вкладок, и такая возможность вполне может пригодиться еще где-нибудь в приложении. Поэтому, вместо того чтобы помещать весь код разметки в макет, мы можем воспользоваться особенностями шаблонов платформы Grails.

6.3.3. Фрагментирование макетов с помощью шаблонов

Когда возникает необходимость многократного использования фрагментов макета приложения, их можно оформить в виде шаблонов. Шаблоны формируют фрагменты файлов GSP и могут встраиваться в страницы с помощью тега `<g:render>`.

Возьмем в качестве примера статический фрагмент формы регистрации пользователя в системе, представленный в листинге 6.8.

Листинг 6.8. Фрагмент для боковой колонки

```
<div id="sidebar">

    <h3>Login</h3>
    <div id="loginForm">
        <g:form controller="login">

            <ul>
                <li>User Id: <g:textField name="userId" size="12"/></li>
                <li>Password:
                    <g:passwordField name="password" size="12"/>
                </li>
                <li><g:submitButton name="login" value="Login"/></li>
            </ul>

        </g:form>
    </div>

</div>
```

В приложении может иметься несколько страниц, требующих наличия этого маленького фрагмента. Если сохранить фрагмент в файле /views/_sidebar.gsp (обратите внимание на символ подчеркивания), вы сможете ссылаться на него в своих страницах GSP или в макетах с помощью такого тега:

```
<g:render template="/sidebar"/>
```

Или, что еще лучше, шаблоны сами могут содержать ссылки на модель, которые можно передавать с помощью атрибута `model`:

```
<g:render template="/sidebar" model="[defaultUser: 'name-here']"/>
```

Это делает шаблоны самодостаточными фрагментами представлений. Шаблоны можно использовать даже в контроллерах для отображения фрагментов при помощи технологии Ajax, о чем мы будем говорить в разделе 6.4.

Создав шаблоны, можно попробовать обеспечить возможность, позволяющую пользователям самим выбирать внешний вид приложения Hubbub. Эта возможность называется *поддержкой тем оформления*; платформа Grails значительно упрощает ее реализацию.

6.3.4. Добавление поддержки тем оформления

Приложение Hubbub прекрасно смотрится в своей серо-голубой гамме цветов. Но обычно сайты социальных сетей предоставляют пользователям возможность выбирать собственное оформление для своих списков сообщений. Давайте добавим поддержку тем в приложение Hubbub.

Сначала нам необходимо предусмотреть способ выбора темы оформления, а для этого нам нужно добавить новое поле в объект `Profile`:

```
package com.grailsinaction

class Profile {

    static belongsTo = User

    String fullName
    // ... остальные поля опущены
    String skin ← Добавляет поддержку тем оформления для каждого пользователя

    static constraints = {
        fullName(nullable: true)
        // ... остальные ограничения опущены
        skin(nullable: true, blank: true, inList: ['blues', 'nighttime'])
    }
```

Примечание

Вам придется дополнить формы редактирования профиля возможностью выбора темы оформления, но мы оставим это вам в качестве самостоятельного упражнения (у вас имеется, правда, возможность получить исходные файлы с примерами из репозитария github).

После добавления поддержки тем оформления нам необходимо настроить страницу со списком сообщений, воспользовавшись новшеством. Если у пользователя имеется своя тема оформления списка сообщений, мы можем применить ее, используя CSS. В листинге 6.9 приводится измененный фрагмент файла timeline.gsp.

Листинг 6.9. Применение темы оформления с помощью CSS

```
<html>
    <head>
        <title>Timeline for ${user.profile.fullName}</title>
        <meta name="layout" content="main"/>
        <g:if test="${user.profile.skin}"> ← Проверка наличия темы оформления
            <link rel="stylesheet" href="
                <g:createLinkTo dir='css'                    ← Применение темы
                    file='${user.profile.skin}.css'/>          оформления пользователя
            "/>
        </g:if>
    </head>
    <body>
```

Измененное действие `timeline` проверяет, была ли настроена тема оформления пользователем, и если была, добавляет ссылку на соответствующую таблицу стилей в каталоге CSS. Файлы стилей могут переопределять цвет фона тела страницы и ее заголовка. Для надлежащей работы темы оформления важно правильно использовать классы CSS и атрибуты `id` в коде разметки страниц представлений.

На рис. 6.6 изображена страница, оформленная темой «nighttime» (ночь). Если вам необходимо предусмотреть возможность применения тем оформления ко всему сайту, а не только к отдельным страницам, лучше всего для этих целей использовать фильтры, которые были представлены в главе 5. Определите имя темы в фильтре и передайте его в виде переменной области видимости сеанса. Затем настройте тему оформления CSS в главном макете, и все готово.

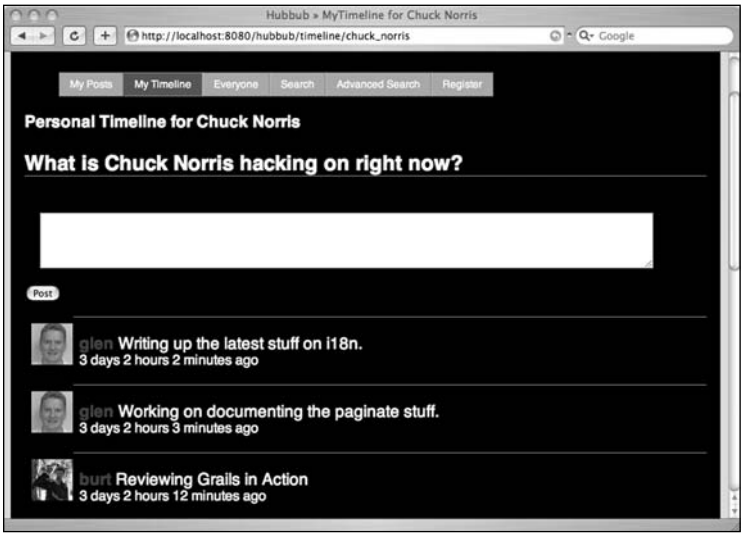


Рис. 6.6. *Тема nighttime (ночь) в действии*

6.3.5. Реализация навигационных вкладок

Реализовав поддержку тем оформления, можно обратить внимание на механизм навигации. Какие элементы пользовательского интерфейса можно задействовать, чтобы дать пользователю возможность перемещаться по приложению? Наиболее привычной и узнаваемой является последовательность навигационных вкладок в верхней части страницы.

На ранних этапах развития платформы Grails все эти навигационные элементы в виде вкладок приходилось создавать вручную. Как правило, для этого создавался нестандартный тег, генерирующий элементы меню, и выполнялась тяжелая работа для выделения подходящей вкладки в зависимости от текущего действия контроллера. Однако такая реализация быстро становилась трудноуправляемой.

С момента появления расширения Navigation, созданного Марком Палмером (Marc Palmer), создание навигационных вкладок стало простым делом. Подробнее со сторонними расширениями мы познакомимся в главе 8, а пока рассмотрим лишь самые основные моменты, связанные с установкой и использованием расширения Navigation.

Чтобы установить расширение Navigation, перейдите в корневой каталог проекта и выполните следующую команду:

```
grails install-plugin navigation
```

Платформа Grails сообщит, что она загрузила расширение и установила его в ваше приложение.

После установки расширения можно добавить в свои классы контроллеров новое свойство `navigation` и сообщить расширению, для каких действий контроллера следует создать вкладки в меню. Мы хотим создать для приложения Hubbub меню, которое выглядело бы так, как показано на рис. 6.7.

Рис. 6.7. *Навигационное меню, которое требуется получить*

Хотя кажется, что для этого придется немало потрудиться, расширение Navigation сильно упрощает создание меню. В листинге 6.10 приводится фрагмент измененных классов `UserController` и `PostController` с новыми свойствами `navigation`, которые определяют, что должно включаться в меню.

Листинг 6.10. Определение пунктов меню в контроллерах

```
class PostController {

    static navigation = [
        [group:'tabs', action:'timeline', title: 'My Timeline', order: 0],
        [action: 'global', title: 'Global Timeline', order: 1]
```

```
        ]
        // ... остальной код опущен
    }

    class UserController {

        static navigation = [
            [group:'tabs', action:'search', order: 90],
            [action: 'advSearch', title: 'Advanced Search', order: 95],
            [action: 'register', order: 99, isVisible: { true }]
        ]
        // ... остальной код опущен
    }
```

В определении каждого свойства `navigation` указывается последовательность отображений, где каждое отображение представляет один пункт меню. В каждом отображении указываются действие, которое должно вызываться в результате щелчка на пункте меню, и необязательный заголовок. Если заголовок не указывается, расширение будет использовать отформатированное имя действия (имя `register` превратится в «Register», а имя `advSearch` — в «Adv Search»).

Для позиционирования пунктов меню допускается использовать свойство `order`. (В будущем сторонние расширения, возможно, будут использовать это поле для автоматического включения своих элементов в ваше меню.) Вы можете даже использовать свойство `isVisible`, которое проверяется перед отображением меню. Например, можно обеспечить отображение пункта меню, вызывающего появление формы регистрации, только если пользователь еще не зарегистрировался.

Теперь, когда мы определили пункты меню, нам требуется добавить в шаблон два тега, которые обеспечат отображение меню. В элемент `head` основного макета мы добавим тег `<nav:resources/>`:

```
<head>
    <nav:resources/>
</head>
```

Он обеспечивает включение таблицы стилей CSS для меню. (Вы можете определить собственные стили, но оформление по умолчанию выглядит достаточно привлекательно.)

В элемент `body` основного макета в место, где должно находиться меню, мы добавим тег `<nav:render/>` с атрибутом `group`, который мы определили в свойствах `navigation`:

```
<div id="bd"><!-- start body -->
    <nav:render group="tabs"/>
    <g:layoutBody/>
</div>
```

После внесения этих двух дополнений наше меню готово к работе. На рис. 6.8 изображено наше новое меню в действии.

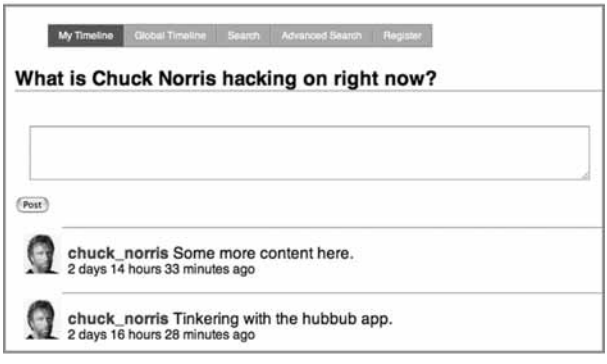


Рис. 6.8. *Расширение Navigation позволяет легко и просто добавлять меню*

Теперь, когда у нас появилось привлекательное меню, пришло время заняться тем, как можно вывести привлекательность пользовательского интерфейса на новый уровень. Один самых очевидных способов заключается в использовании технологии Ajax для обновления списка сообщений без полной перезагрузки страницы при добавлении нового сообщения. Но прежде чем приняться за него, нам необходимо немного узнать о том, как платформа Grails обеспечивает поддержку Ajax.

6.4. Применение тегов Ajax

К настоящему моменту мы уже немало потрудились над пользовательским интерфейсом, но самое интересное мы оставили напоследок. В этом разделе мы задействуем возможности технологии Ajax для дальнейшего улучшения пользовательского интерфейса приложения Hubbub. Мы получим представление о том, как выполняется динамическое обновление страниц приложения Hubbub за счет выполнения запросов в программном коде. Затем мы обеспечим воспроизведение некоторых анимационных эффектов с помощью JavaScript.

Но сначала мы должны заложить основы, выбрав библиотеку JavaScript для выполнения всей тяжелой работы.

6.4.1. Выбор библиотеки JavaScript

Платформа Grails реализует поддержку функциональности Ajax с помощью *адаптивных* библиотек тегов. Это означает, что все стандартные теги поддержки технологии Ajax могут использовать любую библиотеку JavaScript по вашему выбору и обеспечивать при этом один и тот же прикладной интерфейс как на стороне клиента, так и на стороне сервера. «Из коробки» поддерживается библиотека Prototype, а Dojo, YUI и jQuery поддерживаются с помощью дополнительных расширений. Следовательно, вы можете начать разработку приложения, используя одну библиотеку, а затем переключиться на другую.

Не все библиотеки хорошо уживаются друг с другом, поэтому, если какое-то ваше представление GSP использует определенную библиотеку, есть смысл задействовать ее и в реализации взаимодействий с применением технологии Ajax. Например, если вы планируете воспроизводить какие-либо анимационные эффекты с помощью библиотеки Scriptaculous, логично выбрать библиотеку Prototype (потому что она использует библиотеку Scriptaculous). Если вы планируете использовать механизм автодополнения из библиотеки YUI, вы можете использовать ее и для организации удаленных взаимодействий Ajax (потому что вы уже заложили затраты времени на загрузку страниц и развертывание инфраструктуры YUI).

Наиболее широко разработчиками приложений на платформе Grails применяется библиотека Prototype (хотя все большую популярность начинает приобретать библиотека jQuery, благодаря своей ненавязчивости и богатой библиотеке эффектов), поэтому в наших примерах мы тоже будем использовать Prototype.

6.4.2. Основы удаленных взаимодействий Ajax

Пришло время задействовать возможности технологии Ajax для обновления списка сообщений пользователя. Мы переделаем форму отправки нового сообщения и будем отправлять ее содержимое, а затем обновлять список с последними сообщениями с помощью приемов Ajax.

Первый шаг на пути к использованию Ajax состоит в том, чтобы импортировать выбранную библиотеку. Для начала мы изменим элемент `head` в представлении timeline.gsp, чтобы сообщить платформе Grails, что для организации удаленных взаимодействий мы собираемся использовать библиотеку Prototype. Тег `javascript` с атрибутом `library` сгенерирует теги разметки HTML для подключения исходных файлов JavaScript, входящих в состав указанной библиотеки:

```
<head>
    <title>Timeline for ${user.profile.fullName}</title>
    <meta name="layout" content="main"/>
    <g:javascript library="prototype"/>
</head>
```

Подключив библиотеку, мы теперь можем изменить форму так, чтобы она отправляла содержимое объекта `Post` с применением механизмов удаленных взаимодействий. Для извлечения содержимого формы и отправки его удаленной службе с помощью технологии Ajax платформа Grails предоставляет тег `<g:submitToRemote>`. Измененное определение формы приводится в листинге 6.11.

Листинг 6.11. Добавление нового сообщения с помощью технологии Ajax

```
<g:form
        action="ajaxAdd">
    <g:textArea id='postContent' name="content" rows="3" cols="50"/><br/>
```

```
    <g:submitToRemote value="Post" ← ❶ Создается ссылка Ajax
        url="[controller: 'post', action: 'addPostAjax']" ← ❷ Параметры
                                                            передачи
        update="allPosts" ← ❸ Идентификатор обновляемого элемента HTML
        onSuccess="clearPost(e)"          ← ❹ Обработчики событий,
        onLoading="showSpinner(true)"        генерируемых тегом
        onComplete="showSpinner(false)"/>
    <img id="spinner" style="display: none"
        src="<g:createLinkTo dir='/images' file='spinner.gif'/>"/> ←┐
                                          ❺ Изображение-индикатор выполнения операции
</g:form>
```

Теги `<g:form>` и `<g:textField>` остались без изменений. В форму был добавлен только тег `<g:submitToRemote>` ❶. Единственным обязательным атрибутом этого тега является `url`, который определяет значения `controller` и `action` ❷. Для обработки формы мы добавим в контроллер `PostController` новое действие `addPostAjax()`.

Если определен атрибут `update` ❸, он должен содержать значение атрибута `id` элемента `div`, который будет обновляться по завершении запроса. В нашем случае действие `addPostAjax()` будет возвращать обновленную разметку HTML списка сообщений, которая должна помещаться в элемент `div` со значением `allPosts` в атрибуте `id`.

Если определен атрибут `onSuccess` ❹, он должен содержать функцию JavaScript, которая будет вызываться сразу же после успешного выполнения запроса (но не в случае ошибки с кодом 404 или какой-либо другой). Независимо от того, какая функция будет указана в атрибуте `onSuccess`, она должна принимать аргумент (`e`), который является объектом `XMLHTTPResponse`, представляющим запрос. В нашем случае мы использовали функцию `clearPost`, которая очищает элемент `textArea` после успешной отправки нового сообщения.

Наконец, функции, указанные в атрибутах `onLoading` и `onComplete`, вызываются до и после выполнения запроса Ajax. Мы используем их, чтобы отобразить и скрыть анимированное изображение-*индикатор*, показывающее, что идет выполнение операции ❺.

Если вам любопытно узнать, что делают функции `clearPost` и `showSpinner` на языке JavaScript, заметим, что в них нет ничего необычного, – лишь несколько строк программного кода, использующего библиотеку Prototype и обеспечивающего отзывчивость пользовательского интерфейса. При выполнении операций Ajax всегда желательно показывать и скрывать изображение-индикатор, чтобы пользователь видел, что что-то происходит. В листинге 6.12 приводятся функции JavaScript, которые мы использовали, чтобы держать пользователя в курсе событий.

Листинг 6.12. Информирование пользователя о происходящем с помощью JavaScript

```
<g:javascript>
    function clearPost(e) {
```

```
            $('postContent').value='';
        }
        function showSpinner(visible) {
            $('spinner').style.display = visible ? "inline" : "none";
        }
    </g:javascript>
```

Закончив реализацию пользовательского интерфейса, нам нужно определить новое действие `addPostAjax()` в контроллере `PostController`. Реализация этого действия приводится в листинге 6.13.

Листинг 6.13. Реализация действия addPostAjax()

```
def addPostAjax = {
    try {
        def newPost = postService.createPost(                    ❶ Создает сообщение с помощью службы
            session.user.userId, params.content)
        def recentPosts = Post.findAllByUser(                    ❷ Запрашивает 20 последних сообщений
            session.user,
            [sort: 'dateCreated', order: 'desc', max: 20])
        render(template:"postentries",                           ❸ Отображает шаблон postentries для каждого результата запроса
            collection: recentPosts,
            var: 'post')
    } catch (PostException pe) {
        render {                                                 ❹ Выводит сообщение об ошибке в случае неудачи
            div(class:"errors", pe.message)
        }
    }
}
```

Благодаря тому что в главе 5 всю основную работу мы реализовали в виде службы `PostService`, мы теперь можем использовать ее в реализации операций Ajax ❶. После создания нового объекта `Post` мы извлекаем последние сообщения для отправки клиенту ❷. Затем мы передаем шаблон `postentries`, созданный нами ранее, и коллекцию с последними сообщениями методу `render()` ❸, который подготовит и отправит разметку HTML со списком сообщений клиенту для обновления содержимого элемента `div` с идентификатором `allPosts`.

Если в ходе этого процесса произойдет какая-либо ошибка, мы, используя построитель разметки (обсуждался в разделе 2.4.4 главы 2), с помощью версии метода `render()` ❹ отправляем обратно элемент `div` с сообщением об ошибке (со специальным оформлением, в результате чего на странице это сообщение отображается в красной рамке).

На рис. 6.9 изображена форма с поддержкой Ajax, находящаяся в процессе отправки нового сообщения, она дополнена стилизованным изображением-индикатором, демонстрирующим, что выполняется некоторая операция.

Тег `submitToRemote`, примененный в листинге 6.11, имеет еще несколько атрибутов, которые мы пока не использовали, включая атрибуты, обе-

спечивающие возможность обработки ответов сервера с кодами ошибок, такие как on404 и другие. Кроме того, данный тег в состоянии обрабатывать возвращаемые значения в формате JSON, и мы будем использовать эту возможность в примерах ниже в этой же главе. За дополнительной информацией о возможностях тега обращайтесь к электронной документации Grails, по адресу: *http://grails.org/doc/1.1.x/*.

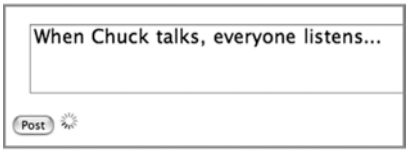


***Рис. 6.9.** Изображение-индикатор, которое выводится при выполнении запроса Ajax*

Закончив реализацию простейшей поддержки Ajax, можно заняться исследованием возможностей библиотек воспроизведения анимационных эффектов, которые позволят сделать наше приложение визуально еще более эффектным.

6.4.3. Эффекты++: вперед с визуальными и анимационными эффектами

В этой главе мы уже придали приложению Hubbub весьма привлекательный внешний вид, но пришло время совершить большой скачок и добавить анимационные эффекты в стиле Web 2.0. Если в последний раз вам приходилось воспроизводить визуальные эффекты с помощью библиотек JavaScript где-то в конце 90-х, можем вас обрадовать: с тех пор возможности ушли далеко вперед.

Благодаря таким библиотекам, как Scriptaculous, YUI и jQuery, визуальная эффектность и совместимость JavaScript с разными броузерами стали реальностью. На совершенно новом техническом уровне были созданы разнообразные библиотеки, обеспечивающие возможность плавной деградации и простоту интеграции с приложением.

Для воспроизведения визуальных эффектов мы будем использовать библиотеку Scriptaculous. Она поставляется в составе платформы Grails, поэтому вам не придется ничего устанавливать и вы сможете сразу же приступить к ее использованию. Она поддерживает множество эффектов, воспроизведение которых обычно можно вызвать одной строкой кода.

Введение в библиотеку Scriptaculous

Scriptaculous – это библиотека JavaScript, позволяющая добавлять в приложение весьма выразительные визуальные эффекты. Растворение, появление, мерцание, изменение размеров и анимация – типичные

эффекты пользовательского интерфейса, предлагаемые библиотекой Scriptaculous.

Если ранее вам не приходилось использовать библиотеку анимационных эффектов JavaScript, вас может устрашать сама мысль о сложности программирования анимационных эффектов. Успокойтесь: библиотека Scriptaculous содержит хорошие настройки по умолчанию для невероятного количества эффектов и обеспечивает визуальную привлекательность даже для взыскательных особ.

Первый шаг на пути к использованию библиотеки состоит в том, чтобы подключить ее в заголовке страницы:

```
<head>
    <title>Timeline for ${session.user.userId}</title>
    <g:javascript library="scriptaculous" />
</head>
```

Благодаря тому что Scriptaculous уже включает библиотеку Prototype, нет необходимости импортировать обе библиотеки.

После подключения библиотеки воспроизведение анимационных эффектов сводится к простому вызову одного из множества методов объекта `Effect` из библиотеки Scriptaculous (например, `Effect.Fade('myDiv')`). Давайте проведем испытания и добавим в приложение Hubbub полосу TinyURL.

Плавное изменение прозрачности полосы TinyURL

Если когда-либо вам приходилось пользоваться сайтом Twitter, вы могли заметить, что все адреса URL, помещаемые в сообщения, автоматически *сжимаются* с помощью специальных сайтов сжатия адресов URL, таких как TinyURL.com. Так как адреса URL могут иметь значительную длину, применение такого сжатия URL позволяет вставлять адреса URL в сообщения, длина которых на сайте Twitter ограничена 140 символами. Сейчас самое время добавить такую возможность в приложение Hubbub. Одновременно мы привнесем некоторый визуальный эффект JavaScript для обеспечения дополнительной привлекательности.

От нас требуется облегчить для пользователей возможность добавления адресов URL в их сообщения, поэтому мы дополним форму отправки сообщения полем ввода TinyURL. Однако это поле ввода нужно пользователям не всегда – оно будет необходимо им, только когда они захотят вставить адрес URL в сообщение. Поэтому мы добавим элемент `div`, содержащий поле ввода TinyURL, который будет постепенно появляться и исчезать, когда это потребуется.

В первую очередь, сделаем элемент `div` невидимым по умолчанию с помощью свойства CSS `display: none;`:

```
<div id="tinyUrl" style="display:none;">
    <g:form>
```

```
        TinyUrl: <g:textField name="fullUrl"/>
        <g:submitToRemote action="tinyurl" onSuccess="addTinyUrl(e)"
            value="Make Tiny"/>
    </g:form>
</div>
```

Затем добавим в форму ссылку с программным кодом JavaScript, которая обеспечит постепенное проявление и растворение элемента `div`. Простая ссылка рядом с кнопкой Post (Отправить) будет смотреться очень неплохо:

```
<a href="#" id="showHideUrl" onClick="return toggleTinyUrl()">
    Show TinyURL
</a>
```

Теперь нам нужно добавить реализацию появления и скрытия на языке JavaScript:

```
<g:javascript>

function toggleTinyUrl() {
    var tinyUrlDisplay = $('tinyUrl').style.display;
    var toggleText = $('showHideUrl');
    if (tinyUrlDisplay == 'none' ) {
        new Effect.BlindDown('tinyUrl', { duration: 0.5 });
        toggleText.innerText = 'Hide TinyURL';
    } else {
        new Effect.BlindUp('tinyUrl', { duration: 0.5 });
        toggleText.innerText = 'Show TinyURL';
    }
}
</g:javascript>
```

Обращения к библиотеке Scriptaculous занимают всего по одной строке, `Effect.BlindDown()` и `Effect.BlindUp()`. Этим методам передается значение атрибута `id` элемента `div` с полем ввода TinyURL. Данная пара эффектов вызывает постепенное появление и исчезновение поля TinyURL.

Вы можете изменить продолжительность воспроизведения эффекта. Мы предпочитаем воспроизводить эффекты достаточно быстро, чтобы не отвлекать пользователя от работы. Например, мы настроили вызов `BlindDown()` так, что теперь он воспроизводит эффект в два раза быстрее, чем принято по умолчанию:

```
new Effect.BlindDown('tinyUrl', { duration: 0.5 })
```

За дополнительной информацией о параметрах эффектов обращайтесь к документации по библиотеке Scriptaculous (*http://script.aculo.us/*). Кроме того, существует страница с фантастическими демонстрационными примерами, которая позволит вам поиграть с каждым из имеющихся эффектов и посмотреть, как они будут выглядеть в броузере (*http://wiki.github.com/madrobby/scriptaculous/combination-effects-demo*).

В библиотеке Scriptaculous имеются сокращенные версии для большинства методов воспроизведения эффектов в пользовательском интерфейсе. Например, теперь, когда вы знаете, как действует эффект, мы можем упростить наш пример:

```
<a href="#" id="showHideUrl"
    onclick="Effect.toggle('tinyUrl', 'blind',
    ➥ { duration: 0.5 }); return false;">
    TinyURL Bar
</a>
```

Метод `Effect.toggle()` выполняет плавное появление и сокрытие, реализованные нами выше, и даже предоставляет различные механизмы переключения (скольжение, затенение, проявление). Обратите внимание, что мы явно возвращаем значение `false` из обработчика события `onclick`, чтобы предотвратить переход по ссылке «#».

Обработка возвращаемых значений в формате JSON

Это, конечно, здорово, что наше поле ввода TinyURL появляется и исчезает с воспроизведением анимационного эффекта, но что нужно сделать, чтобы пользователь мог пользоваться им? Нам необходимо, чтобы после ввода пользователем адреса URL в поле ввода и щелчка на кнопке Make Tiny (Сжать) сжатая версия URL автоматически добавлялась к содержимому поля `textArea` с текстом сообщения.

Чтобы реализовать это, нам необходимо:

1. Извлечь полный адрес URL из текстового поля и отправить его серверу с привлечением технологии Ajax.
2. Создать сжатую версию URL (TinyURL) с помощью некоторого действия контроллера и вернуть полученное значение.
3. Получить возвращаемое значение в представлении и добавить его в элемент `textArea` с текстом сообщения.

Самый простой способ реализовать первый шаг заключается в том, чтобы определить кнопку `submitToRemote`. Она обернет значения формы и отправит их контроллеру с применением технологии Ajax. Ниже показано, как в этом случае может выглядеть содержимое файла /views/posts/list.gsp:

```
<g:form>
    TinyUrl: <g:textField name="fullUrl"/>
    <g:submitToRemote action="tinyurl"
        onSuccess="addTinyUrl(e)" value="Make Tiny"/>
</g:form>
```

Тег `<g:submitToRemote>` вызовет действие `tinyurl()` контроллера `PostController` и передаст ему объект `params` со всеми полями формы в нем. В нашем случае имеется только один параметр (`fullUrl`), содержимое кото-

рого требуется сжать. После успешного завершения запроса будет вызвана функция JavaScript, определяемая атрибутом `onSuccess`.

Самое потрясающее во всем этом, что функция, указанная в атрибуте `onSuccess`, получит результат работы удаленного действия. Если действие вернет текст, фрагмент XML или JSON, мы сможем использовать его в функции JavaScript на стороне клиента.

Теперь создадим действие контроллера. В листинге 6.14 приводится реализация функции TinyURL. Мы принимаем параметр `fullUrl`, переданный пользователем, и отправляем его на веб-сайт TinyURL для сжатия.

Листинг 6.14. Действие TinyURL, выполняющее сжатие адресов URL и возвращающее результат в формате JSON

```
def tinyurl = {
    def origUrl = params.fullUrl?.encodeAsURL()
    def tinyUrl =
        new URL("http://tinyurl.com/api-create.php?url=${origUrl}").text
    render(contentType:"application/json") {
        urls(small: tinyUrl, full: params.fullUrl)
    }
}
```

Улучшенный класс `URL` в языке Groovy позволяет обращаться к содержимому по адресу URL с помощью свойства `text`. В листинге 6.14 мы вызываем сценарий на сайте TinyURL, который возвращает сжатую версию исходного значения `${origUrl}`. Получив сжатую версию адреса URL, мы, благодаря универсальности метода `render()`, возвращаем небольшой фрагмент в формате JSON. Возвращаемый фрагмент JSON будет содержать свойства с исходной и сжатой версией URL. Для отображения нам достаточно иметь только сжатую версию, но мы хотели продемонстрировать, как возвращать несколько значений.

Теперь дополним картину реализацией функции обратного вызова `addTinyUrl(e)`, обрабатывающей возвращаемый фрагмент в формате JSON. В листинге 6.15 приводится программный код JavaScript с реализацией нашего обработчика.

Листинг 6.15. Функция обратного вызова, реализующая обработку возвращаемого значения в формате JSON

```
<g:javascript>

function addTinyUrl(e) {
    var tinyUrl = e.responseJSON.urls.small;
    $("postContent").value += tinyUrl;
    updateCounter();
    toggleTinyUrl();
}

</g:javascript>
```

Когда действие контроллера возвращает содержимое типа `application/JSON`, библиотека Prototype автоматически применяет функцию `evals()` к возвращаемому значению для преобразования его в объект на языке JavaScript. С этого момента мы можем обратиться к свойству `small`, значение которого было установлено действием контроллера из листинга 6.14, и отобразить сжатую версию URL. На рис. 6.10 изображена новая особенность приложения в действии.

Теперь у нас имеется превосходный визуальный эффект, реализованный средствами библиотеки Scriptaculous, и мы даже объединили его с незамысловатым волшебством Ajax и JSON.

На этом мы завершаем обзор поддержки технологии Ajax, имеющейся в платформе Grails. В этой главе мы рассмотрели невероятный объем сведений об особенностях реализации пользовательского интерфейса в приложениях на платформе Grails, так что на этом мы и остановимся.

***Рис. 6.10.** Функция TinyURL в действии*

6.5. В заключение

Мы начали эту главу с обзора основных тегов форм платформы Grails и познакомились с таким количеством тегов, которое вы едва ли будете использовать в одном приложении. Мы также посмотрели, как создавать собственные теги для ситуаций, когда стандартные теги оказываются недостаточно гибкими.

Затем мы изучили различные возможности, предоставляемые платформой Grails для реализации макетов и шаблонов, и даже обеспечили поддержку своих тем оформления и реализовали навигационное меню.

В заключение мы исследовали некоторые передовые концепции поддержки Ajax в платформе Grails, реализовав динамическое обновление списка сообщений и функцию TinyURL, использующую формат JSON для взаимодействия со службами. Попутно мы приобрели некоторые навыки создания визуальных эффектов на JavaScript, позволяющих сделать пользовательский интерфейс более эффектным.

Мы многое узнали и выработали несколько оптимальных приемов:

- *Применяйте возможность постраничного просмотра*. Возможность постраничного просмотра огромных наборов данных обеспечивает пользователю дополнительное удобство, и к тому же она проста в реализации.
- *Создавайте собственные теги*. Уделите время разработке своих собственных тегов для наиболее типичных элементов пользовательского интерфейса. Это поможет вам сэкономить время, упростит сопровождение приложения и даст возможность повторно использовать их в будущих проектах.
- *Используйте макеты, основанные на соглашениях*. Старайтесь создавать макеты, опираясь на соглашения, а не на явные объявления в метатегах. Часто макет, предназначенный для определенного действия, может обеспечить больше удобства в сопровождении, чем привлечение магии метатегов. Используйте преимущества метатегов, когда возникает необходимость определить подмножество страниц для контроллера, но во всех остальных случаях старайтесь использовать макеты по соглашению.
- *Используйте интеллектуальные макеты*. Реализуйте отображение сообщений из объекта `flash` в одном макете, а не в каждом представлении. Используйте шаблоны для определения часто используемых фрагментов HTML, явно передавайте им элементы модели. При использовании технологии Ajax сопротивляйтесь искушению прямого отображения разметки HTML и реализуйте отображение любой информации через обращение к шаблону.
- *Выберите библиотеку JavaScript*. Ознакомьтесь с достоинствами и недостатками различных библиотек JavaScript. Все они используют различные подходы к решению одних и тех же задач. Выберите библиотеку поддержки Ajax, которая лучше всего будет соответствовать остальной части вашего приложения. Загрузка библиотеки занимает некоторое время, поэтому старайтесь минимизировать число библиотек, используемых в приложении.
- *Используйте библиотеку макетов*. При разработке сложных макетов на основе CSS используйте библиотеку YUI Grids. Она поможет вам сэкономить время и обеспечит одинаковое отображение макета во всех броузерах.

К настоящему моменту мы изучили большое количество базовых понятий платформы Grails, но существует еще одна важная область, о которой мы знаем пока совсем немного: тестирование. В следующей главе мы подробнее рассмотрим поддержку тестирования контроллеров, валидаторов, библиотек тегов и других элементов, имеющихся в платформе Grails.

7

В этой главе:
- Зачем необходимо тестирование
- Как тестировать различные части приложения
- Выбор стратегии тестирования

Создание надежных приложений

Мы представили все основные компоненты приложений на платформе Grails, и теперь вы знаете более чем достаточно, чтобы приступить к созданию полезных и удобных приложений. Вероятно, вы страстно желаете поскорее начать программировать и экспериментировать! И пока у вас такая высокая мотивация, есть смысл сделать шаг назад и посмотреть, как можно повысить качество и надежность приложений с помощью тестирования.

7.1. Зачем необходимо тестировать программное обеспечение?

Существует два подхода к разработке программного обеспечения: *надежность определяется проектом* и *надежность достигается через тестирование*. Первый из них предполагает, что вы в состоянии спроектировать изначально надежное программное обеспечение. Этой философии еще предстоит проникнуть в коммерческий сектор, чтобы занять более или менее значимое положение, а вот вторая философия – достижение надежности через тестирование – стала весьма популярной в последние годы.

Принцип очень прост: для каждой строки программного кода, написанной нами, необходимо создать тест, который позволит убедиться, что код действует, как ожидалось. Если все тесты выполняются без ошибок, значит, программный продукт работает правильно, и это вселяет в нас уверенность при выпуске программы. Ключевым моментом здесь является автоматизация выполнения максимального количества тестов, с тем чтобы их можно было запускать часто и достоверно. Кроме того, тестирование позволяет выполнять реорганизацию программного кода с большей уверенностью, чем без его проведения. Одна из многих проблем, возникающих при разработке программного обеспечения, за-

ключается в том, что изменения в программном коде могут вызывать непреднамеренные побочные эффекты, поэтому все, что позволяет выявить их на ранних стадиях, является существенной выгодой.

В платформе Grails тесты приобретают еще большую значимость. Как вы уже знаете, компилятор Groovy не в состоянии определить несоответствия типов, как это делает компилятор Java, потому что Groovy – динамический язык. Это означает, что тесты являются первой линией обороны на пути исключений, возникающих при приведении типов, и обращений к несуществующим свойствам или методам.

Некоторые утверждают, что за высокую скорость разработки программного кода приходится платить тяжелым бременем тестирования. Мы не согласимся с этим. Если следовать стратегии упрощенного тестирования в языке Java, вы определенно заметите, что объем тестирования при разработке приложений на платформе Grails увеличился. Но это некорректное сравнение – вы должны стремиться к тому, чтобы охватить тестами весь программный код приложения, независимо от того написан ли он на языке Java или Groovy. В противном случае возникает риск внести в программное обеспечение больше ошибок, чем может быть оправдано.

Тестирование имеет множество преимуществ, но в конечном итоге все сводится к простому вопросу: «Вы хотите писать надежные приложения?» Если ответ – да (мы искренне надеемся, что он таков), тогда нет никаких причин отказываться от тестирования. Но прежде чем вы почувствуете дрожь в коленях от одной мысли о тяжелом бремени написания тестов, мы хотели бы вас успокоить. Даже если вы еще не привыкли писать тесты, вы обнаружите, что чем больше вы занимаетесь этим процессом, тем легче он становится. Но это еще не все, вы, безусловно, заметите, как растет ваша уверенность в своем программном коде, и постепенно тесты станут для вас обязательной подстраховкой.

Мы будем подходить к тестированию в направлении снизу вверх – от модульных тестов к функциональным. Существует и другой, не менее правильный подход к тестированию – сверху вниз, – при котором тестирование начинается с проверки функциональных возможностей, и затем выполняется спуск до уровня модульного тестирования по мере реализации каждой отдельной особенности. Какой подход лучше подходит для вас или вашей команды, следует выяснить на практике, но какой бы подход вы ни выбрали, приемы, описываемые здесь, одинаково пригодны для использования. Если вы еще не знакомы с терминами «модульное тестирование» и «функциональное тестирование», не беспокойтесь; мы объясним их по ходу дела.

Разработка, управляемая тестированием

В последние годы существенно изменилось отношение к тестированию. Раньше тесты создавались после того, как был написан программный код, если вообще создавались. Теперь же даже существует движение

под лозунгом: «Тесты важны. В действительности они настолько важны, что должны создаваться до программного кода!» Процесс, отстаиваемый этим движением, известен, как «разработка, управляемая тестированием» (test-driven development, TDD). Принцип прост: тесты должны создавать еще до того, как будет написан соответствующий программный код.

Соблюдение принципа TDD имеет несколько преимуществ. Прежде всего такой подход гарантирует наличие тестов для всех классов. Во-вторых, разработка программного кода идет быстрее, чем раньше, поскольку тесты позволяют сразу же проверить работоспособность программного кода – цикл «написали фрагмент кода – проверили его» становится более эффективным. В-третьих, когда в первую очередь создается тест, вы, скорее всего, будете проектировать свои классы и системы так, чтобы их легко было тестировать. Прикручивать тесты к уже написанному программному коду может оказаться невозможным без его изменения. Наконец, принцип TDD заставляет с самого начала задумываться о том, какое поведение вы хотели бы получить от своего класса. Сосредоточившись на вопросе «какой», а не «как», вы сможете улучшить архитектуру своих классов и повысить скорость разработки программного кода. Если вы знаете, куда идти, вы скорее достигнете пункта назначения.

Исправление ошибок

Мы верим, что вы в совершенстве владеете искусством устранения ошибок, но мы призываем вас в ходе этого процесса следовать определенным порядком. На первом этапе исправления ошибки необходимо написать тест, выявляющий ее. Этот тест должен быть написан так, чтобы он терпел неудачу до тех пор, пока не будет исправлена ошибка.

Вы можете сначала исправить ошибку, а затем написать тест, но как тогда убедиться, что тест будет терпеть неудачу при наличии ошибки? Для этого вам придется отменить изменения и запустить тест еще раз. Гораздо проще сначала написать тест и воспользоваться преимуществами принципа TDD.

Из-за наличия этих преимуществ мы полагаем, что следование принципу TDD следует всячески поощрять. Мы можем даже распространить принцип TDD на философию тестирования сверху вниз, согласно которой сначала пишутся функциональные тесты, затем модульные тесты и только потом программный код. Если вы выберете этот путь, создавайте функциональные тесты постепенно, только для тестирования части особенности (или особенностей), имеющей отношение к определенным классам, над которыми вы работаете. Попытка написать функциональный тест, охватывающий всю особенность целиком, – бесплодная затея.

7.2. Модульное тестирование

Тестирование приложения на платформе Grails сначала может представляться весьма отпугивающей перспективой. Платформа Grails сама предоставляет водопроводную систему и, по-видимому, опрыскивает ваши классы волшебством, поэтому на первый взгляд может показаться, что протестировать классы по отдельности невозможно. Но не надо пугаться! Платформа Grails предоставляет собственную систему тестирования, чтобы помочь вам плавать в этих водах и упростить модульное тестирование.

Почему мы начинаем с модульного тестирования? Одна из причин состоит в том, что мы избрали философию тестирования снизу вверх; следовательно, вполне логично начинать изучение снизу. Но, как будет показано ниже, при желании вы сможете использовать интеграционные тесты платформы Grails для «модульного тестирования». Модульные тесты обладают двумя основными преимуществами перед интеграционными:

- Они быстрее выполняются.
- Их можно запускать из интегрированной среды разработки.

Уменьшение протяженности цикла «написали фрагмент код – проверили его» крайне важно для сохранения концентрации внимания, поэтому модульные тесты идеально подходят для изучения на данной стадии.

Достаточно теории – перейдем к практике. В главе 3 мы начинали изучение с модели предметной области, поэтому и сейчас мы начнем с тестирования классов предметной области.

7.2.1. Тестирование классов предметной области

Необходимость тестирования классов предметной области может показаться странной – что там тестировать? Этот вопрос был бы правомерен, если бы классы предметной области не обладали собственным поведением (методами). Однако все классы предметной области содержат логику, которую можно проверить: ограничения. По этой причине платформа Grails обеспечивает явную поддержку тестирования ограничений посредством класса `GrailsUnitTestCase`.

Создание модульного теста

Посмотрим, как выполняется тестирование, на примере создания теста для класса `User` в приложении Hubbub, который имеет несколько интересных ограничений. Все модульные тесты сохраняются в каталоге test/unit, поэтому создайте файл test/unit/com/grailsinaction/UserUnitTests.groovy, если он не был создан ранее. Реализация испытательного теста приводится в листинге 7.1.

Листинг 7.1. Наш первый модульный тест для класса предметной области

```
package com.grailsinaction

class UserUnitTests extends grails.test.GrailsUnitTestCase {
    void testConstraints() {
        def will = new User(userId: "william")
        mockForConstraintsTests(User, [ will ])
        def testUser = new User()
        assertFalse testUser.validate()
        assertEquals "nullable",
            testUser.errors["userId"]
        assertEquals "nullable",
            testUser.errors["password"]

        testUser = new User(userId: "william", password: "william")
        assertFalse testUser.validate()
        assertEquals "unique", testUser.errors["userId"]
        assertEquals "validator", testUser.errors["password"]
        ...
        testUser = new User(userId: "glen", password: "passwd")
        assertTrue testUser.validate()
    }
}
```

❶ Имитация класса предметной области

❷ Проверка испытуемого объекта

❸ Проверка наличия ошибок, выявленных при наложении ограничений

Самое первое, на что можно обратить внимание, – это объем программного кода в тесте. Полные тесты часто содержат больше программного кода, чем тестируемые классы, но выгода перевешивает цену. Как видно из примера, программный код теста весьма прост, несмотря на то что содержит много повторений. Обратите также внимание на имя класса `UserUnitTests`: суффикс `UnitTests` не является обязательным требованием платформы Grails, однако следование этому соглашению позволит легко отличать модульные и интеграционные тесты в интегрированной среде разработки.

Что такое имитация?

Процесс имитации заключается в создании фальшивых версий классов или методов, которые могут использоваться вместо действующих реализаций. Самое интересное, что вы можете управлять поведением сымитированного объекта или метода, то есть вы можете передавать известные данные испытуемому объекту.

Имитации могут создаваться на любом этапе тестирования, но обычно это делается при модульном тестировании. Класс `GrailsUnitTestCase` предоставляет множество методов, которые можно использовать для моделирования динамических свойств и методов, которые обычно добавляются платформой Grails.

Первое, что происходит в листинге 7.1, – это расширение испытательным тестом вспомогательного класса `GrailsUnitTestCase`, предоставляемого платформой Grails. Это самый старший класс в своем семействе, и вы, скорее всего, будете использовать его прямо или косвенно во всех своих модульных тестах. Он не только делает вашу жизнь проще, но и позволяет создавать имитации объектов, предотвращая появление побочных эффектов, которые могут приводить к неудаче в процессе тестирования по непонятным причинам.

Далее вызывается специальный метод, предоставляемый классом `GrailsUnitTestCase`, для создания имитации нашего класса предметной области, что даст нам возможность протестировать работу его ограничений ❶. Не забывайте, что сами классы предметной области ничего не знают об ограничениях, поскольку их интерпретацией и обработкой занимается сама платформа Grails.

После применения метода `mockForConstraintsTests()` к классу предметной области можно приступать к тестированию ограничений. Для этого необходимо создать экземпляр смоделированного класса предметной области и вызвать метод `validate()` ❷. Если все поля объекта предметной области соответствуют ограничениям, возвращается значение `true`, в противном случае – `false`.

За кулисами

Все методы `mock*()`, предоставляемые классом `GrailsUnitTestCase`, используют приемы работы со свойством `MetaClass` для добавления имитаций свойств, методов и объектов. Это означает, что вы можете добавлять или переопределять существующие компоненты для тонкой настройки их поведения.

Кроме того, сымитированные ограничения выполняют настоящую привязку данных, поэтому свойство `errors` является действующим экземпляром интерфейса `Errors` платформы Spring. Это, например, позволяет проверять нарушение сразу нескольких ограничений, накладываемых на одно поле.

Теперь вы можете обнаружить, что ваш объект предметной области обладает свойством `errors`, которое можно использовать для выявления полей, нарушивших ограничения, и причин этих нарушений. Данное свойство, созданное системой тестирования, ведет себя практически так же, как свойство, создаваемое платформой Grails, благодаря этому у вас не будет возникать недоразумений при переходе от теста к прикладному программному коду. Единственное отличие состоит в том, что в программном коде приложения вы не сможете интерпретировать свойство `errors` как отображение. Самое большое преимущество такого расширенного поведения свойства в тестах заключается в том, что

вы легко можете проверить, какие ограничения были нарушены, передав имя поля в качестве ключа отображения и проверив возвращаемое значение ❸. Значением в этом случае будет имя первого нарушенного ограничения, соответствующего полю, или `null`, если поле соответствует ограничениям. Если поля нашего экземпляра класса `User` содержат значения `null`, мы можем выполнить проверку для строки `"nullable"` (то есть имя ограничения).

Запуск теста

Попробуем теперь запустить наш тест. Один из способов сделать это заключается в том, чтобы запустить тест непосредственно из интегрированной среды разработки, если она поддерживает такую возможность, но сейчас мы воспользуемся командой платформы Grails:

```
grails test-app -unit
```

Эта команда запустит модульные тесты приложения (то есть тесты, находящиеся в каталоге test/unit) и сгенерирует отчеты. Сейчас у нас имеется единственный модульный тест, поэтому результат работы команды будет выглядеть, как показано ниже:

```
...
Running 1 Unit Test...
Running test UserUnitTests...
SUCCESS
Unit Tests Completed in 1514ms ...
...
```

Мало того что результаты выводятся в консоль, платформа Grails к тому же генерирует отчет в формате HTML и сохраняет его в каталоге test/reports/html. Запустите свой броузер и откройте файл index.html в этом каталоге, чтобы ознакомиться с результатами тестирования пакета и класса.

Как платформа Grails узнает, какой метод следует вызывать для проведения тестирования? Допустим, вы могли бы написать метод `checkErrors()`, который вызывается из нескольких методов теста, но для вас было бы нежелательно, чтобы он вызывался как самостоятельный тест. Это не проблема: платформа Grails будет вызывать только общедоступные методы, которые объявлены как возвращающие значение `void`, имена которых начинаются со слова «test» и которые не принимают аргументов. Примером такого метода является метод `testConstraints()` в листинге 7.1.

Пока все довольно просто. Теперь поближе рассмотрим строку ❶, потому что мы все должны объяснить важность метода `mockForConstraintsTests()`. Как видно из листинга, во втором аргументе методу передается список экземпляров класса `User`, хотя в данном случае этот список содержит всего один элемент. Для проверки ограничения `unique` поля `userId` нам необходим хотя бы один экземпляр класса предметной об-

ласти, чтобы у нас была возможность сравнить его с новым значением, и список, который передается методу `mockForConstraintsTests()` как множество «существующих» записей. Как видно из испытательного теста, поле `userId` не проходит проверку ограничений, когда в него записывается значение `"william"`, потому что список уже содержит экземпляр предметной области с таким значением этого поля.

Чтобы убедиться в этом, измените четвертую строку модульного теста, как показано ниже:

```
mockForConstraintsTests(User)
```

Теперь снова запустите тест. На этот раз вы увидите следующую строку среди результатов, выводимых в консоль:

```
testConstraints...FAILURE
```

Если заглянуть в отчет в формате HTML, вы увидите, что тест потерпел неудачу из-за того, что ожидалось уникальное значение, а отыскалось значение `null`. Ограничение `unique` *не было* нарушено.

Сейчас у вас может возникнуть вопрос, стоит ли таких усилий проверка ограничений. Если вы еще не убедились в необходимости этого, представьте себе такую ситуацию: опечатка в имени ограничения может вызвать трудноуловимую ошибку, на поиск которой могут уйти часы. Разве 5–10 минут, потраченных на написание теста, не стоят того?

Осталось ответить всего на один вопрос: откуда взялись все эти методы `assert*()`? Пользователям библиотеки JUnit они покажутся знакомыми, и на то есть свои причины.

За кулисами: JUnit

Библиотека JUnit де-факто стала стандартной платформой тестирования в языке Java, поэтому она не только используется платформой Grails как фундамент собственной платформы тестирования, но и сам язык Groovy обеспечивает прямую поддержку этой библиотеки. На рис. 7.1 изображена иерархия классов от JUnit через Groovy к платформе Grails. Не волнуйтесь о самых верхних трех классах на диаграмме – мы доберемся до них в свое время.

Все классы модульных тестов, которые вы будете создавать, будут наследовать один из этих классов JUnit, поэтому вы можете использовать те же приемы, что использовались при работе с платформой тестирования JUnit. Однако эти классы не единственное, что использует платформа Grails: для создания отчетов команда `test-app` использует утилиту Ant платформы JUnit.

Большая часть методов `assert*()` предоставляется классом `TestCase` платформы JUnit, но помимо них класс `GroovyTestCase` содержит несколько дополнительных методов. В частности, в нем присутствует семейство методов `shouldFail()`, позволяющих убедиться, что некоторый

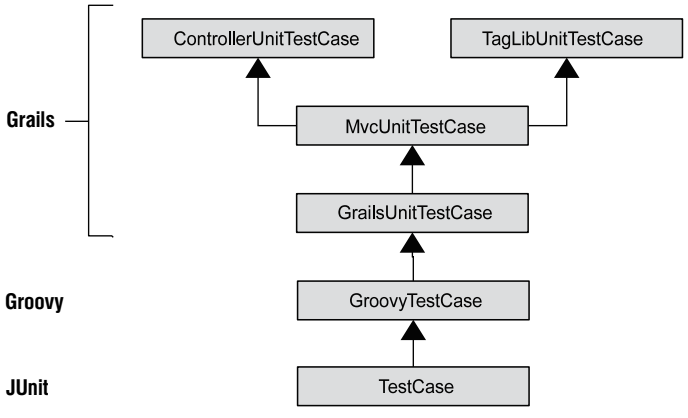


***Рис. 7.1.** Иерархия классов тестов в платформе Grails – все тесты в платформе Grails в конечном итоге наследуют класс TestCase платформы JUnit*

программный блок возбуждает исключение. Например, следующий фрагмент проверяет, возбуждает ли метод исключение, когда ему передается пустая строка:

```
shouldFail(IllegalArgumentException) {
    myService.testMethod("")
}
```

Здесь мы определили тип ожидаемого исключения, но вы можете опустить этот аргумент, если тип возбуждаемого исключения не имеет значения.

В табл. 7.1 перечислены некоторые из наиболее часто используемых и полезных проверок, имеющихся в вашем распоряжении, когда ваши тесты наследуют класс `GroovyTestCase`. Обратите внимание, что все методы в таблице имеют также версии, принимающие строку в первом аргументе. Это дает возможность указать текст сообщения, которое будет выводиться в случае неудачи, что, в свою очередь, позволит легко определить по отчетам причину ошибки прохождения того или иного теста. Например, если следующая проверка окончится неудачей,

```
assertEquals "Unique constraint should have triggered, but didn't",
             "unique",
             testUser.errors["userId"]
```

в отчете появится сообщение «Unique constraint should have...» («Должно было сработать ограничение уникальности, но этого не произошло»).

Это был краткий обзор платформы JUnit, дополнительную информацию вы можете найти в печатных и электронных изданиях. Хорошей отправной точкой является сайт *http://www.junit.org*. Большая часть

информации, приводимой там, в равной степени применима и к тестам на платформе Grails – это просто здорово, что имеется возможность использовать другие платформы!

Таблица 7.1. Полезные методы проверок для ваших испытательных тестов

Метод	Описание
`assertEquals(expected, actual)`	Сравнивает два значения или два объекта, используя метод `equals()`
`assertTrue(value)`	Проверяет значение `value` на равенство `true` после приведения его к типу Boolean
`assertFalse(value)`	Проверяет значение `value` на равенство `false` после приведения его к типу Boolean
`assertNull(value)`	Проверяет значение `value` на равенство значению `null`
`assertNotNull(value)`	Проверяет значение `value` на *неравенство* значению `null`
`assertSame(expected, actual)`	Проверяет, являются ли два объекта одним и тем же экземпляром
`assertArrayEquals(expected, actual)`	Сравнивает два массива на равенство, проверяя, содержат ли массивы одни и те же элементы
`shouldFail(type, closure)`	Проверяет, возбуждает ли замыкание `closure` исключение типа `type` или любого другого типа, если тип исключения не указан

Многие классы предметной области включают не только поля и ограничения; программисты часто снабжают свои модели данных дополнительными методами, которые, вне всяких сомнений, также требуется тестировать. Методы классов предметной области не сильно отличаются от методов любых других классов, поэтому они могут тестироваться обычным способом. Мы продемонстрируем доступные приемы при изучении вопроса тестирования служб; помните, что эти приемы могут применяться к любым классам, включая и классы предметной области.

7.2.2. Тестирование служб

Выделение логики приложения в отдельные службы – обычная (и рекомендуемая) практика. Мы видели множество служб в предыдущих главах, но они представляют интерес с точки зрения тестирования, потому что являются самыми обычными классами. Платформа Grails не внедряет в службы ни динамические свойства, ни динамические мето-

ды (за исключением свойства log), поэтому приемы, используемые для их тестирования, могут применяться к любым классам.

Рассмотрим пример тестирования простой службы. Следующий класс во многом является искусственным (он никак не связан с приложением Hubbub), но он позволяет продемонстрировать различные возможности модульного тестирования, имеющиеся в платформе Grails:

```
class BasketService {
    def userService

    def listItems() {
        def user = userService.getCurrentUser()
        if (!user) { throw new NoUserException() }
        else {
            log.info "Retrieving basket items for ${user.name}"
            return Item.findAllByUser(user)
        }
    }
}
```

Основная идея состоит в том, что метод listItems() возвращает все элементы, находящиеся в покупательской корзине текущего пользователя, или возбуждает исключение, если пользователь не указан.

При тестировании такого класса возникает несколько неприятных препятствий в виде запроса, свойства log и поля. На этапе выполнения все три элемента предоставляются самой платформой Grails, поэтому вам не приходится беспокоиться об их установке. Но модульный тест – это совсем другое дело, поэтому нам придется сымитировать их либо средствами метапрограммирования (для имитации динамических свойств и методов), либо за счет создания имитационных объектов (для удовлетворения зависимостей службы). Для единственного теста это слишком большой объем работы, причем не самой простой.

К счастью, механизм тестирования платформы Grails предоставляет все необходимые инструменты, позволяющие легко и просто решить эти проблемы. В листинге 7.2 приводится модульный тест для данной службы. Обратите внимание, что испытательный тест наследует класс GrailsUnitTestCase. Это обеспечивает возможность имитации методов, о которой мы только что говорили.

Листинг 7.2. Тестирование службы Grails

```
package com.grailsinaction

class BasketServiceUnitTests extends grails.test.GrailsUnitTestCase {
    void testListItems() {
        mockLogging(BasketService) ← ❶ Создание имитации свойства log

        def currUser = new User(name: "lucy")
        def otherUser = new User(name: "alan")
```

```
        mockDomain(Item, [
            new Item(user: currUser, name: "orange"),
            new Item(user: otherUser, name: "lemon"),
            new Item(user: currUser, name: "apple") ])

        def userControl = mockFor(UserService)
        userControl.demand.getCurrentUser(1..1) = {->
            currUser
        }

        def testService = new BasketService()
        testService.userService = userControl.createMock()

        def items = testService.listItems()
        assertEquals 2, items.size()
        assertEquals "orange", items[0].name
        assertEquals "apple", items[1].name

        userControl.verify()
    }
}
```

← ❷ Имитация класса предметной области

← Имитация зависимости

← Проверка вызова сымитированных методов

Рассмотрим сначала свойство `log`: после вызова метода `mockLogging(Class)` ❶ все создаваемые экземпляры данного класса будут чудесным образом снабжаться действующим свойством `log`, способным выводить в консоль все журналируемые сообщения. По умолчанию отладочные и трассировочные сообщения не выводятся во избежание вывода лишней информации, но вы можете разрешить вывод отладочных сообщений, передав методу `mockLogging()` значение `true` во втором необязательном аргументе.

Существует похожий метод для имитации классов предметной области ❷. Это более мощный и потому более сложный метод. Прежде всего метод `mockDomain(Class, List)` добавляет фиктивную реализацию всех динамических методов классов предметной области. Наиболее значимыми исключениями являются поддержка запросов с критериями и языка запросов Hibernate (Hibernate Query Language, HQL). Поддержка динамических методов поиска обеспечивается полностью, и их поведение полностью соответствует поведению действительных методов.

Большинство статических методов, динамически добавляемых платформой Grails, необходимы только для поиска и обработки записей в базе данных, но мы имеем дело с модульным тестом, поэтому база данных просто отсутствует. К счастью, эта проблема решается с помощью второго аргумента метода `mockDomain()`. Как и в случае с методом `mockForConstraintsTests()`, в этом аргументе (который допускается оставлять пустым) передается список экземпляров класса предметной области, представляющих записи в базе данных, причем этот список играет роль хранилища данных для динамических методов поиска и других аналогичных методов. Например, вызов `Item.list()` вернет все эле-

менты списка, переданного методу `mockDomain()`. Более того, методы поиска всегда будут возвращать экземпляры класса предметной области в том же порядке, в каком они определяются в списке, что гарантирует определенный порядок следования записей в тестах. Нет ничего хуже, чем неудача теста в результате изменения порядка следования элементов коллекции.

Пока мы говорим о втором аргументе, хотелось бы выяснить, что происходит со значениями поля `id`. При желании вы можете явно присвоить значения полям `id` всех объектов предметной области в списке, но это совершенно необязательно. По умолчанию каждый экземпляр класса предметной области получит значение идентификатора, соответствующее его позиции в списке, начиная с 1.

Поля id и version и интегрированные среды разработки

Как вы уже знаете, классы предметной области не объявляют явно поля `id` и `version`, но вы можете определять их значения в программном коде. Это возможно благодаря тому, что компилятор платформы Grails сам добавляет эти поля в классы.

К сожалению, не все интегрированные среды разработки используют компилятор Grails для компиляции классов предметной области приложения, поэтому эти поля могут не добавляться. В таком случае при попытке присвоить этим полям некоторые значения интегрированная среда разработки сообщит об ошибке обращения к несуществующему полю во время выполнения теста. Что неудивительно, поскольку данные поля действительно не существуют.

Эту проблему можно решить, явно объявив поля `Long id` и `version` в классах предметной области. Возможно, было бы неплохо, если бы вы намекнули разработчикам интегрированной среды разработки о необходимости использовать компилятор платформы Grails или внедрять необходимые поля.

Даже если тестируемый метод не использует методы поиска или извлечения объектов, по-прежнему может иметь смысл передавать список методу `mockDomain()`. Вызов метода `save()` добавит объект предметной области в список, а вызов метода `delete()` удалит его. Это позволит проверить, корректна ли работа методов, использующих их.

Мы выяснили, как работать со свойством `log` и с классами предметной области, теперь нам нужно выяснить, как имитировать зависимости. Ни один из рассматривавшихся до сих пор методов не поможет нам в этом, потому что они слишком узкоспециализированы. Нам необходима более универсальная система имитации; она обеспечивается методом `mockFor()`.

7.2.3. Универсальная имитация в платформе Grails

Имитация объекта должна выглядеть и вести себя как класс, который она имитирует. Например, если класс реализует некоторый интерфейс, вы можете создать класс, который тоже реализует этот интерфейс, но обладает фиктивными реализациями методов. Однако если вам необходимо сымитировать много классов, причем не все из них реализуют интерфейсы, потребуется приложить значительные усилия. Кроме того, вы можете столкнуться с проблемами свойств и методов, которые обычно добавляются динамически.

Нам необходим простой способ, который дал бы возможность добавлять имитацию реализаций любых методов, а также позволил бы имитировать только вызываемые при тестировании методы, облегчая бремя тестирования. Метод `mockFor()` – это именно то, что нам необходимо.

Первый шаг к созданию имитаций зависимостей (или *вспомогательных* объектов, если говорить языком тестирования) заключается в вызове метода `mockFor()` с классом, который требуется сымитировать. Вы уже видели его в листинге 7.2, но мы приведем соответствующую строку еще раз:

```
def mockControl = mockFor(UserService)
```

Почему мы присвоили переменной `mockControl` значение, возвращаемое методом? Потому что возвращаемый объект не является имитацией экземпляра класса `UserService`, но он позволяет добавлять фиктивные методы и создавать имитации экземпляров. Своим поведением он напоминает класс `MockControl` из библиотеки EasyMock 1.x, что объясняет имя переменной.

После создания управляющего объекта можно приступать к определению методов, которые предполагается вызывать, и созданию их реализаций. В листинге 7.2 ожидается, что тестируемая служба будет вызывать метод `getCurrentUser()` вспомогательного объекта `userService`, поэтому мы объявляем ожидаемый метод с помощью свойства `demand` управляющего объекта. На рис. 7.2 показана схема разбиения вызова на составляющие его части.

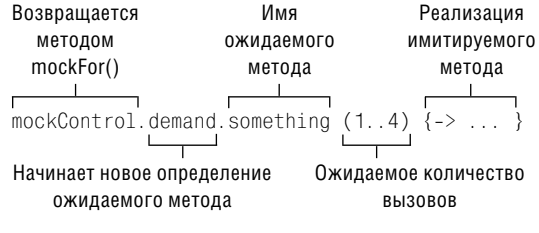


***Рис. 7.2.** Определение ожидаемого метода для объекта, возвращаемого методом mockFor()*

Данная синтаксическая конструкция должна быть знакома тем, кому уже приходилось использовать классы `MockFor` и `StubFor` в языке Groovy. Определение ожидаемого метода выглядит, как обращение к свойству `demand` с диапазоном значений в качестве аргумента. Этот диапазон определяет количество ожидаемых вызовов метода. Например, диапазон `2..2` свидетельствует о том, что метод будет вызываться точно два раза, тогда как диапазон `1..4` устанавливает, что метод будет вызываться как минимум один раз, но не более четырех раз. Если диапазон не определен, неявно подразумевается значение `1..1`, которое означает, что метод должен вызываться точно один раз.

Почему бы не использовать класс MockFor?

Существует несколько причин, объясняющих, почему не используется класс `MockFor`, но главная заключается в том, что этот класс недостаточно хорошо соответствует механизму метапрограммирования, используемому платформой Grails. Кроме того, синтаксис использования этого класса становится неудобным, когда в тесте требуется сымитировать более одного вспомогательного объекта.

Это не означает, что вы не можете использовать класс `MockFor` в модульных тестах Grails, но мы полагаем, что вы найдете метод `mockFor()` более простым и немного более гибким.

Замыкание – это реализация имитации. Вы можете использовать его для проверки значений аргументов и возвращать определенное значение или объект тестируемому программному коду. Количество и типы аргументов замыкания должны соответствовать количеству и типам аргументов имитируемого метода, в противном случае имитация не будет вызываться.

А что если метод при тестировании попытается обратиться к свойству одного из вспомогательных объектов? Все просто. Как вы уже знаете, обращения к свойствам в языке Groovy транслируются в вызовы соответствующих методов доступа, поэтому все, что от вас требуется, это создать имитации этих методов. Например, допустим, что вспомогательный объект имеет свойство с именем `length`, тогда сымитировать чтение этого свойства можно следующим способом:

```
mockControl.demand.getLength(1..2) {-> ... }
```

Точно так же можно реализовать метод записи.

Итак, мы определили ожидания для управляющего объекта, что дальше? Это зависит от требований. Если класс при тестировании создает вспомогательный объект, тогда больше ничего не нужно делать. Вспомогательный объект автоматически получает имитации реализаций

методов, которые мы определили. В листинге 7.2 нам потребовался вспомогательный объект сам по себе, поэтому мы использовали метод `createMock()`:

```
testService.userService = userControl.createMock()
```

Все это было лишь подготовкой. Нам по-прежнему необходимо вызвать тестируемый метод и убедиться, что он действует, как ожидается. Основная идея состоит в том, чтобы проверить соответствие поведения и значений, возвращаемых имитациями методов. Во многих случаях имеет также смысл убедиться, что тестируемый метод передает имитациям методов соответствующие значения в аргументах.

Но и после вызова метода и проверки его поведения тестирование нельзя считать законченным. Как узнать, вызывались ли имитации методов требуемое число раз, если вообще вызывались? Вполне возможно, что вас это не интересует, и в некоторых случаях это вполне оправданно. Но что, если действительная реализация одного из имитируемых методов сохраняет некоторые данные или отправляет сообщение по электронной почте? В этом случае хотелось бы иметь уверенность, что метод вызывается.

За кулисами управляющий объект подсчитывает количество вызовов каждого метода. Как только какой-либо метод будет вызван большее количество раз, чем ожидалось, будет возбуждена ошибка проверки. Это никак не поможет нам, если метод не вызывался ни разу, но и эта проблема легко разрешима. После вызова тестируемого метода можно вызвать метод `mockControl.verify()`. Этот метод возбудит ошибку проверки, если какой-либо метод вызывался меньше ожидаемого числа раз.

На рис. 7.3 приводится типичная последовательность шагов, обычно выполняемых при модульном тестировании. Это общий шаблон эффективного использования системы имитации, поэтому практически все ваши модульные тесты будут выглядеть похожими друг на друга.

Прежде чем двинуться дальше, отметим еще две особенности метода `mockFor()`, которые могут заинтересовать вас. Первая связана с имитацией статических методов. Такая имитация может быть выполнена добавлением слова `static` после свойства `demand`:

```
mockControl.demand.static.list() { Map args -> ... }
```

Вот и все. Ничего другого не требуется для получения действующей имитации статического метода.

Вторая особенность имеет отношение к порядку, в котором вызываются имитации методов. В некоторых случаях может быть важно, чтобы тестируемый метод вызывал различные методы в определенном порядке, и любые отклонения должны вызывать ошибку при тестировании. Эта особенность известна как *строгий* режим и соответствует поведению

метода `mockFor()` по умолчанию. С другой стороны, порядок вызова методов вообще может не иметь никакого значения, и в этом случае было бы желательно предотвратить неудачу теста только из-за того, что вызовы методов производились в ином порядке. К счастью, метод `mockFor()` предусматривает возможность устанавливать *свободный* режим для таких случаев путем передачи значения `true` во втором аргументе:

```
def mockControl = mockFor(OtherService, true)
```

Как видите, метод `mockFor()` является мощным и простым в применении инструментом тестирования, который может использоваться для имитации практически всего, что угодно. Однако когда дело касается контроллеров, желательно было бы иметь инструмент, которые позволил бы получить все стандартные динамические свойства и методы. Создавать их вручную с помощью `mockFor()` было бы слишком утомительно.

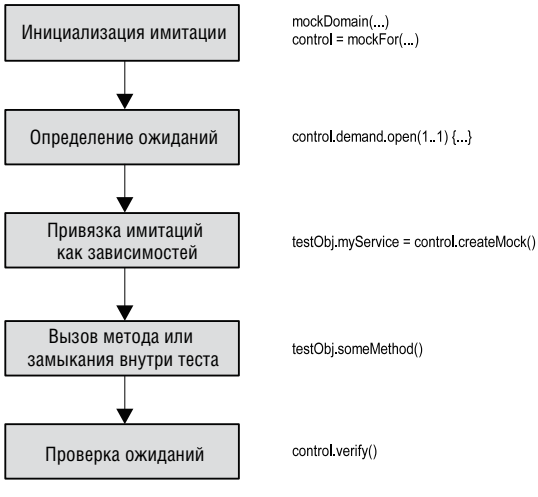


Рис. 7.3. *Структура модульного теста, использующего механизм имитации платформы Grails, с примерами для каждого шага*

7.2.4. Тестирование контроллеров

Контроллеры весьма непросты для тестирования, потому что они обладают значительным количеством динамических свойств и методов и способны отображать страницы самыми разными способами. Они могут возвращать модель, отображать представление, отображать произвольный текст или перенаправлять на другой адрес URL. По этой причине платформа Grails включает поддержку класса испытательного теста для контроллеров: `ControllerUnitTestCase`.

Посмотрим, как он действует, на примере создания теста для одного из контроллеров приложения Hubbub. Начнем с действия, возвращающе-

го модель. Все, что нам требуется, – убедиться, что модель содержит данные, соответствующие параметрам, переданным действию. Для этого нам нужно создать тестовые параметры, выполнить действие и проверить содержимое возвращаемого отображения. В листинге 7.3 показано, как мы реализуем вышесказанное.

Листинг 7.3. Тестирование действия контроллера, возвращающего модель

```
package com.grailsinaction

class PostControllerUnitTests extends grails.test.ControllerUnitTestCase {
    void testShow() {
        mockDomain(User, [
            new User(userId: "glen"),
            new User(userId: "peter") ]
        this.controller.params.id = "peter" <-- ❶ Установка параметров запроса

        def model = this.controller.show()

        assertEquals "peter", model["viewUser"]?.userId <-- ❷ Проверка модели, возвращаемой действием
    }
}
```

В самом начале каждого теста создается испытательный тест контроллера на основе класса `ControllerUnitTestCase`, поэтому мы можем сразу же приступить к его использованию ❶. Но как тест определяет, экземпляр какого класса контроллера следует создать? По соглашению. Платформа тестирования просматривает имя класса модульного теста и использует для определения часть имени до слова «Controller» включительно, таким образом имя `PostControllerUnitTests` превращается в `PostController`. Если это соглашение не действует в вашем случае, можно передать имя класса конструктору `super`:

```
PostControllerUnitTests() {
    super(PostController)
}
```

Часто действия контроллера принимают параметры, поэтому вы должны обеспечить их передачу. Делается это просто – добавьте необходимые параметры в отображение `params` ❶. Данное отображение является одним из специальных свойств, доступных в тестах. Все такие свойства перечислены в табл. 7.2.

Затем вызывается действие контроллера, и возвращаемое значение присваивается переменной, чтобы потом можно было проверить элементы, содержащиеся в нем ❷.

Все просто и безболезненно. Но реализации контроллеров редко отличаются подобной простотой, и действия, возвращающие модель, являются самыми простыми для тестирования. А что если контроллер выполняет перенаправление или вызывает метод `render()`?

Таблица 7.2. Имитации свойств контроллеров, доступные в модульных тестах

Имя свойства	Описание
request	Имитация запроса (GrailsMockHttpServletRequest)
response	Имитация ответа (GrailsMockHttpServletResponse)
session	Имитация сеанса HTTP (MockHttpSession платформы Spring)
flash	Отображение, представляющее область видимости кадра (flash)
params	Отображение, содержащее параметры действия
redirectArgs	Отображение, содержащее аргументы, переданные тестируемым действием методу redirect()
renderArgs	Отображение, содержащее аргументы, переданные тестируемым действием методу render()
template	Отображение, содержащее аргументы шаблона, если метод render() вызывал шаблон; имя шаблона содержится в значении ключа name и может иметь элементы с ключами bean, collection и model (в зависимости от того, какой именованный аргумент передавался методу render())
modelAndView	Экземпляр класса ModelAndView платформы Spring, к которому можно обратиться за получением имени представления и модели представления

Порядок тестирования этих и других случаев зависит от того, какой механизм используется контроллером. Рассмотрим пример перенаправления:

```
redirect(controller: "post", action: "list")
```

Чтобы убедиться в корректности вызова метода, можно проверить содержимое отображения redirectArgs:

```
assertEquals "post", this.controller.redirectArgs["controller"]
assertEquals "list", this.controller.redirectArgs["action"]
```

Аналогичный подход можно использовать и для метода render(), только при этом следует проверять содержимое свойства renderArgs, а не redirectArgs. Но не все формы метода render() можно протестировать таким способом – некоторые из них записывают информацию непосредственно в поток вывода, включая следующие:

- render ... as XML/JSON
- render { ... }
- render «...»
- render text: «...»

Вместо проверки отображения renderArgs можно проверить содержимое, сгенерированное этими методами, обратившись к свойству contentAsString объекта response. В примере ниже мы проверяем, начинается ли сгенерированный фрагмент XML с элемента post:

```
assertTrue response.contentAsString.startsWith("<post>")
```

При непосредственной проверке сгенерированного содержимого не следует полагаться на предположения о количестве пробельных символов и о порядке следования текстовых элементов. Например, в приложении Hubbub при выводе сообщений в формате XML нельзя заранее точно предсказать, какой текст будет сгенерирован. Не забывайте, что большая часть пробелов может просто игнорироваться парсерами XML, а порядок следования атрибутов не имеет значения. Так, сначала вы можете использовать реализацию, которая генерирует хорошо отформатированную разметку XML, а позднее заменить ее реализацией, которая вообще не использует пробельные символы. Тогда простые операции сравнения строк будут приводить к неудаче при выполнении теста.

За кулисами

Как вы уже знаете, контроллеры обладают некоторыми динамическими свойствами и методами, доступными во время выполнения, что осложняет их тестирование. Класс ControllerUnitTestCase облегчает вам жизнь, используя метод mockController() класса GrailsUnitTestCase для добавления методов и свойств при подготовке теста. Эти фиктивные реализации как раз и занимаются заполнением отображений ...Args и выводят отображаемое содержимое в имитацию ответа, позволяя проверить его с помощью свойства contentAsString.

Контроллеры обладают достаточно большим количеством уникальных особенностей, и мы могли бы посвятить целую главу различным приемам их тестирования. К счастью, с полной палитрой возможностей можно познакомиться в руководстве пользователя платформы Grails. Однако два объекта тестирования заслуживают особого упоминания: управляющие объекты и библиотеки тегов.

Управляющие объекты

Тестирование объектов, непосредственно использующих объект params, реализуется достаточно легко, но применение тех же приемов для тестирования действий, использующих управляющие объекты, просто невозможно. Проблема состоит в том, что механизм связывания параметров в управляющих объектах, используемый во время выполнения, недоступен в модульных тестах. Но не стоит беспокоиться, решение оказывается на удивление простым.

Допустим, что имеется действие `login`, которое принимает управляющий объект, содержащий имя пользователя и пароль:

```
def login = { LoginCommand cmd ->
    ...
}
```

Все, что необходимо сделать в тесте, – создать экземпляр управляющего объекта и передать его действию:

```
def testLogin() {
    def testCmd = new LoginCommand(username: "peter", password: "pass")
    this.controller.login(cmd)
    ...
}
```

Это так просто. Единственная ложка дегтя в бочке меда – ошибки, выявляемые при проверке ограничений. Поскольку вы создаете управляющий объект самостоятельно, никаких проверок ограничений к данным не применяется. Это осложняет тестирование реализации обработки ошибок проверки данных в действиях. Но не волнуйтесь – разработчики платформы Grails как всегда оказались на шаг впереди, и в подобных ситуациях вы можете вызывать метод `validate()` управляющего объекта непосредственно:

```
def testLogin() {
    mockCommandObject(LoginCommand)

    def testCmd = new LoginCommand(username: "peter", password: "pass")
    testCmd.validate()
    this.controller.login(cmd)
    ...
}
```

Если какие-либо поля управляющего объекта будут содержать недопустимые значения, метод `validate()` заполнит свойство `errors` управляющего объекта соответствующими значениями. Тестируемое действие не заметит разницы: оно будет видеть управляющий объект с ошибками, выявленными при проверке ограничений.

Библиотеки тегов

Тестирование при использовании библиотек тегов реализуется немного сложнее, чем при использовании управляющих объектов. В отличие от управляющего объекта, библиотеку тегов нельзя передать действию контроллера, а это означает, что необходимо использовать иной подход.

Рассмотрим некоторый произвольный код контроллера, в котором используется тег `link`:

```
def show = {
    ...
```

```
        render link(controller: "post", action: "list") { "Home" }
    }
```

Заметили, что обращение к тегу выглядит, как вызов метода? Это и есть наш способ! Мы можем сымитировать «метод» тега в контроллере с помощью вызова `mockFor()`, как показано ниже:

```
def mock = mockFor(MyController)
mock.demand.link(1..1) { Map args, Closure c ->
    ...
}
```

Теперь в процессе тестирования вместо тега контроллер будет вызывать эту имитацию. Прием действует независимо от того, как выполняется обращение к тегу, как к `link(...)` или `g.link(...)`. Самый заметный недостаток этого решения заключается в отсутствии поддержки других пространств имен тегов, так что, если контроллер вызывает теги, находящиеся в нестандартном пространстве имен, следует использовать интеграционные тесты.

Здорово, что есть возможность имитировать теги при тестировании контроллеров, но было бы еще лучше, если бы имелась возможность тестировать сами теги. В конце концов, они тоже содержат программный код.

7.2.5. Тестирование библиотек тегов

Сначала перспектива тестирования библиотек тегов может показаться пугающей. Обычно библиотеки тегов используются в файлах серверных страниц Groovy (Groovy Server Pages, GSP), как же их можно проверить из простого класса? Легко. Библиотека тегов – это класс, содержащий замыкания, и вы можете интерпретировать эти замыкания как методы. В действительности библиотеки тегов имеют много общего с контроллерами, от соблюдения принципа внедрения зависимостей до обеспечения доступа к динамическим свойствам.

Для вас уже не будет сюрпризом известие о том, что платформа Grails обеспечивает класс поддержки модульного тестирования для библиотек тегов: `TagLibUnitTestCase`. Этот класс ведет себя практически точно так же, как его собрат для тестирования контроллеров, используя соглашение по определению имени библиотеки тегов, экземпляр которой требуется создать. Кроме того, он обеспечивает доступ к имитациям объектов запроса, ответа и сеанса.

Для демонстрации того, как писать модульные тесты для библиотек тегов, возьмем простой тег, который выводит свое содержимое в объект ответа, если пользователь зарегистрировался в приложении Hubbub:

```
class SecurityTagLib {
    def isLoggedIn = { attrs, body ->
        if (session.user) {
            out << body()
```

```
        }
    }
}
```

Этот тег может использоваться примерно так:

```
<g:isLoggedIn>You are logged in!</g:isLoggedIn>
```

Это очень простой тег, но он поможет нам показать важные приемы, применяемые при тестировании тегов.

В листинге 7.4 приводится реализация нашего модульного теста для библиотеки тегов. Он проверяет работу тега при обоих условиях: когда пользователь зарегистрирован и когда пользователь не зарегистрирован.

Листинг 7.4. Пример модульного теста для библиотеки тегов

```
package com.grailsinaction

class SecurityTagLibUnitTests extends grails.test.TagLibUnitTestCase {
    void testIsLoggedIn() {
        String testContent = "You are logged in!"
        mockSession["user"] = "me"
        tagLib.isLoggedIn([:]) {-> testContent } ← ❶ Вызов тега как метода

        assertEquals testContent, tagLib.out.toString() ← ❷ Проверка вывода
    }

    void testIsLoggedInNoUser() {
        String testContent = "You are not logged in!"
        tagLib.isLoggedIn([:]) {-> testContent }

        assertEquals "", tagLib.out.toString()
    }
}
```

Здесь нет ничего удивительного. Мы добавляем пользователя в имитацию сеанса, к которой библиотека тегов может обратиться с помощью динамического свойства `session`. Самое интересное в тесте – это то, как вызывается тег и как проверяется вывод, сгенерированный им.

Прежде всего класс поддержки модульного тестирования автоматически создает экземпляр класса библиотеки тегов и присваивает его свойству `tagLib` ❷. Затем производится вызов тега, как если бы это был метод. Одна из странностей вызова – пустое отображение (`[:]`) в виде аргумента. Оно необходимо, потому что тег ожидает получить в первом аргументе отображение с атрибутами. Если бы тег требовал хотя бы один атрибут, мы могли бы использовать менее явный синтаксис:

```
tagLib.isLoggedIn(attr1: "value") { ... }
```

Так выглядит немного лучше, но такой подход неприменим в случае отсутствия атрибутов.

Самое важное для этого модульного теста заключено далее в строке ❶, где мы передаем тегу простое замыкание, не имеющее аргументов. Как вы уже видели в предыдущей главе, содержимое тега GSP передается его реализации в виде замыкания, поэтому здесь мы подставляем свою имитацию, которая возвращает простую строку. Если тег выведет содержимое, что должно произойти, когда область видимости сеанса будет содержать объект `user`, тогда эта строка попадет в волшебное свойство `out`.

Чтобы проверить, был ли соответствующий текст сгенерирован тегом, мы вызываем метод `toString()` свойства `out` и сравниваем результат с ожидаемым значением ❷. Вот, собственно, и все, хотя, возможно, стоит отметить, что свойство `out` является экземпляром класса `StringWriter`.

Не все теги такие же простые, как `isLoggedIn`, но все возможности, которые мы до сих пор применяли в этой главе, могут использоваться и для тестирования библиотек тегов. Вы можете использовать метод `mockFor()` для имитации зависимостей и вызовов других тегов, и вы можете использовать метод `mockLogging()`, если тестируемый тег использует свойство `log`.

Мы рассмотрели практически все, что вам необходимо знать о модульном тестировании. Такой объем информации непросто осилить за один раз, но вы неплохо потрудились и можете гордиться собой! Как вы могли заметить, в модульных тестах возникает совсем немного ситуаций, которые нельзя было бы разрешить с помощью классов, предоставляемых платформой Grails. Однако иногда вы можете обнаружить, что приходится создавать слишком сложные имитации, чтобы протестировать простые методы. Это может быть признаком того, что сам метод нуждается в реорганизации, однако во многих случаях это обусловлено самой его природой. В такой момент стоит подумать о возможности применения альтернативного подхода.

7.3. Интеграционное тестирование

Модульные тесты предназначены для тестирования отдельных, неделимых *модулей* функциональности (таких как классы и их методы), а интеграционные тесты – для проверки крупных частей приложения, таких как контроллеры вместе с элементами, от которых они зависят. Основное внимание уделяется проверке совместной работы модулей, то есть их корректной *интеграции*.

На первый взгляд интеграционные тесты в платформе Grails напоминают модульные тесты. На самом деле, когда создается интеграционный тест командой

```
grails create-integration-test Book
```

единственное очевидное его отличие от модульного теста заключается в том, что он сохраняется в каталоге test/integration, а не test/unit. Но, как все мы знаем, внешность бывает обманчива.

Главное отличие состоит в том, что интеграционные тесты выполняются внутри полноценного окружения платформы Grails с действующей базой данных и при наличии всех необходимых динамических свойств и методов. В интеграционных тестах их не требуется имитировать. Каковы бы ни были намерения и цели, тесты применяются к действующему экземпляру приложения. Основной недостаток заключается в отсутствии контейнера сервлета, что имеет определенные последствия.

Ниже коротко перечислено, что работает, а что – нет:

- Загружаются все расширения, которые допускают возможность работы в испытательной среде.
- Выполняется `BootStrap`. Контекст приложения на платформе Spring заполняется полностью.
- Классы предметной области оперируют с действующей базой данных (по умолчанию – с базой данных HSQLDB, располагаемой в памяти).
- Доступны все динамические свойства и методы.
- Используются имитационные версии объектов запроса к сервлету, ответа и сеанса.
- Фильтры платформы Grails или сервлета не выполняются.
- Отображение адресов URL не оказывает влияния.
- Представления недоступны для тестирования.

Возможно, некоторые из пунктов этого списка пока не имеют для вас большого значения, поэтому мы рекомендуем вам вернуться к нему, когда вы поближе познакомитесь с такими понятиями, как службы и контекст приложения на платформе Spring. Более полное представление об интеграционных тестах вы получите, когда мы рассмотрим некоторые примеры.

За кулисами

Каждый интеграционный тест выполняется в пределах транзакции, которая откатывается по окончании теста. Это означает, что никакие изменения не сохраняются в базе данных. Кроме того, тестовому окружению платформы Grails по умолчанию присваивается имя «test».

Возможно, вас теперь интересуют вопросы, когда следует использовать интеграционные тесты и как их писать. Было бы здорово, если бы можно было привлечь успешный опыт сообщества Grails, но пока имеется

совсем немного руководств по этой теме. Тем не менее мы попытаемся обеспечить вас полезной начальной информацией.

Существуют три ситуации, в которых вам потребуется прибегнуть к интеграционным тестам:

- Когда невозможно протестировать какой-либо компонент с помощью модульного теста.
- Когда необходимо протестировать взаимодействия между объектами.
- Когда проще выполнить тестирование на фактических данных, чем заниматься реализацией имитации всего и вся.

Ситуации, в которых модульные тесты не могут использоваться, мы рассмотрим в разделе 7.3.2, а сейчас мы подробнее остановимся на втором и третьем пунктах списка.

7.3.1. Восполнение недостатков

Вероятно, вы уже много раз видели знак «смотрите под ноги» , напоминающий об опасности падения в проем. Вас может это удивить, но данное предупреждение обретает особый смысл при тестировании приложений. Модульные тесты чрезвычайно полезны, они помогают убедиться в том, что классы и методы *сами по себе* действуют именно так, как требуется; однако их работа может нарушиться, как только они начнут взаимодействовать друг с другом. Эти взаимодействия являются теми самыми «проемами», которые не охватываются модульными тестами и о которых следует помнить.

Рассмотрим конкретный пример. Допустим, что у нас имеется контроллер, использующий службу для получения списка некоторых элементов, ассоциированных с пользователем. Он извлекает названия элементов, составляя из них новый список, и передает их соответствующему представлению. Реализация этой операции может выглядеть, как показано ниже:

```
class MyController {
    def myService

    def show = {
        def items = myService.fetchItemsFor(params.userId)
        [ itemNames: items.collect { it.name } ]
    }
}
```

Предположим также, что модульное тестирование контроллера и службы дает положительный результат, то есть нет никаких очевидных признаков неправильной работы контроллера. Означает ли это, что программный код безупречен?

Существуют ли здесь проблемы, зависит от поведения метода службы: если он возвращает пустой список, не содержащий ни одного элемен-

та, то все работает замечательно. Но что если метод службы вернет значение `null` вместо пустого списка? Нет никаких причин, препятствующих ему в этом, однако в таком случае вызов `items.collect {...}` в контроллере возбудит исключение `NullPointerException`. Эта ошибка появилась из-за сделанного на стороне контроллера неверного предположения, что метод службы будет возвращать пустой список, а не значение `null` (именно поэтому методы должны возвращать пустые коллекции вместо значения `null`).

Совет

Ошибки часто проявляют себя в пограничных условиях, когда передаются пустые коллекции, отсутствуют запрашиваемые значения или числовые значения равны нулю или отрицательны, поэтому важно проверить как можно больше таких условий, Это особенно важно потому, что они обычно игнорируются при тестировании вручную.

Этот пример демонстрирует разновидность проблем, которые могут возникнуть, когда классы, прошедшие модульное тестирование, начинают взаимодействовать между собой. Когда вы добавите динамические свойства и методы, вы обнаружите множество ситуаций, когда что-то может пойти не так. Простой интеграционный тест, пример которого показан в листинге 7.5, поможет быстро выявить проблему. Приведенный тест не только охватывает ситуацию с отсутствующими элементами в списке, но и ситуацию, когда в списке присутствует несколько элементов.

Листинг 7.5. Ваш первый интеграционный тест, включающий объекты предметной области и службу

```
import grails.test.GroovyPagesTestCase

class MyControllerTests extends GroovyTestCase {
    def myService ← ❶ Внедрение экземпляра службы MyService

    void testShowNoItems() {
        def myController = new MyController()
        myController.myService = myService ← ❷ Создание объекта для тестирования

        myController.params.userId = "glen" ← ❸ Установка параметров действия
        def model = myController.show()

        assertEquals 0, model["itemNames"].size()
    }

    void testShow() {
        new Item(userId: "glen", name: "chair").save()
        new Item(userId: "peter", name: "desk").save()
        new Item(userId: "glen", name: "table").save(
            flush: true) ← ❹ Инициализация данных
```

```
        def myController = ...
        ...
        def names = model["itemNames"]
        assertEquals 2, names.size()
        assertNotNull names.find { it == "chair" }
        assertNotNull names.find { it == "table" }
    }
}
```

❺ Проверка результатов

На первый взгляд этот тест похож на модульные тесты, которые мы видели выше. Тем не менее он имеет глубокие отличия, поэтому рассмотрим его поближе. Первая «диковинка», которую можно заметить, – это свойство `myService` ❶, подобное свойству в контроллере `MyController`. При запуске теста платформа Grails запишет в это свойство полностью настроенный экземпляр службы `MyService`. Такое внедрение зависимостей в тест может показаться странным, но причина этого прояснится, как только мы начнем рассматривать сам метод теста.

Как и в модульных тестах, когда необходимо протестировать определенный класс, будь то контроллер, служба или что-то другое, вы создаете экземпляр класса с помощью ключевого слова `new` ❷. Проблема такого подхода состоит в том, что любые зависимости, которые обычно внедряются во время выполнения (в данном случае свойство `myService`), остаются пустыми. Как вы увидите в главе 14, зависимости заполняются, только если объект управляется платформой Spring. (Интеграционные тесты и класс `BootStrap` являются исключениями из этого правила, но их экземпляры создаются самой платформой Grails.) Чтобы обойти эту проблему, мы вручную устанавливаем свойство `myService` контроллера, записывая в него значение поля `myService` теста. Это может выглядеть, как запрещенный прием, тем не менее это на 100 процентов официальный способ удовлетворения зависимостей в интеграционных тестах.

Надоели имитации?

Порой необходимость имитации класса за классом в модульных тестах может показаться слишком утомительной, и тогда интеграционные тесты могут стать неплохим решением. Но не спешите: усилия, которые потребуется приложить, чтобы подготовить данные для проверки всех пограничных условий в интеграционном тесте, могут превосходить усилия, необходимые для создания имитаций вспомогательных объектов. Хотя иногда реализация интеграционного теста может оказаться значительно проще.

К сожалению, строгих правил не существует, поэтому есть смысл поэкспериментировать. Если интеграционный тест оказывается проще в реализации, тогда используйте его! Чем сложнее тест в реализации, тем выше вероятность, что его не следует писать.

После создания и инициализации контроллера можно определить некоторые испытательные параметры ❸. Обратите внимание, что мы используем те же синтаксические конструкции, что и в модульных тестах, поэтому вы не должны испытывать затруднений при переключении между модульными и интеграционными тестами. Одно важное отличие касается управляющих объектов: в интеграционных тестах вы заполняете объект `params`, как показано в этом примере. Платформа Grails автоматически создает экземпляры любых управляющих объектов, объявленных в действии контроллера, и выполняет привязку параметров. Это означает, что добавление управляющих объектов в реализацию действия не оказывает влияния на его тест.

Наконец мы подходим к проблеме испытательных данных. В модульных тестах данные либо напрямую передаются испытуемому объекту или методу, либо поставляются имитациями вспомогательных объектов. В интеграционных тестах используется иной подход: тесты проводятся при наличии действующей базы данных, поэтому данные создаются с помощью классов предметной области ❹, как это делается во время работы приложения. Но поскольку операции выполняются с действующей базой данных, нет никаких гарантий, что данные будут возвращаться в каком-то определенном порядке, если только явно не используется их сортировка. Примите это во внимание и при проверке результатов старайтесь использовать приемы, не зависящие от порядка следования данных ❺.

Мы надеемся, что теперь вы получили представление о том, что собой представляют интеграционные тесты и в чем заключается их удобство. Теперь вы знаете достаточно, чтобы самостоятельно написать интеграционный тест для любого контроллера или службы. За дополнительной информацией обращайтесь к руководству пользователя платформы Grails, где приводится множество сведений о тестировании контроллеров, библиотек тегов и других компонентов приложений на платформе Grails.

При тестировании контроллеров и служб вы можете выбирать, какой дорогой пойти, но в некоторых ситуациях модульные тесты оказываются бесполезными.

7.3.2. Когда могут применяться только интеграционные тесты

Поддержка модульного тестирования в платформе Grails поможет вам пройти долгий путь тестирования различных компонентов, но на этом пути существуют провалы, о которых вы должны знать. В частности, модульные тесты не помогут вам проверить запросы с критериями или запросы на языке запросов библиотеки Hibernate (Hibernate Query Language, HQL), а также работу механизма отображения адресов URL. Кроме того, в главе 9 вы увидите, что и для проверки работы веб-потоков требуются интеграционные тесты.

Когда дело доходит до запросов с критериями или до языка HQL, можно использовать прием интеграционного тестирования, представленный в листинге 7.5. Платформа Grails сама позаботится обо всем необходимом. Вы даже можете переключаться между использованием динамических методов поиска, запросов с критериями и HQL, не прикасаясь к испытательному тесту.

Тестирование отображения адресов URL – это лишь вопрос использования класса `GrailsUrlMappingsTestCase`. Для примера допустим, что у нас определено следующее отображение:

```
class UrlMappings {
    static mappings = {
        "/basket/$username"(controller: "basket", action: "show")
        "/$controller/$action?/$id?"()
        "500"(view: "error")
    }
}
```

Тогда наш испытательный тест, позволяющий убедиться в правильности отображения адресов URL, может выглядеть так, как показано в листинге 7.6. Этот тест проверяет также обратное отображение – от контроллеров и действий к URL.

Листинг 7.6. Тестирование отображений адресов URL

```
class UrlMappingsTestCase extends grails.test.GrailsUrlMappingsTestCase {
    void testMappings() {
        assertUrlMapping("/item/edit/123",
                controller: "item",
                action: "edit") {
            id = 123
        }

        assertUrlMapping("/basket/fred",
                controller: "basket",
                action: "show") {
            username = "fred"
        }

        assertForwardUrlMapping(500, view: "error")
    }
}
```

← ❶ Проверка отображения адреса URL

← Проверка отображения параметра

← ❷ Проверка отображения кода состояния

Как видите, тест имеет очень простую реализацию. Суперкласс автоматически загружает все отображения адресов URL приложения и выполняет все необходимые подготовительные действия в методе `setUp()`, поэтому мы сразу же можем приступить к тестированию. Основным методом тестирования отображений адресов URL является метод `assertUrlMapping()` ❶, который принимает адрес URL в первом аргументе, а имена контроллера и действия – во втором. В первом тесте проверяется отображение адреса URL `"/item/edit/123"` на контроллер `item` и дейст-

вие `edit` со значением `123` в параметре `id`. Кроме того, этот метод проверяет правильность обратного отображения.

Второй тест похож на первый, но во втором мы проверяем работу более специфичного правила отображения адреса URL (`"/basket/$username"`), которое отменяет действие более общего правила.

Наконец мы проверяем отображение кода состояния `500` ❷. Здесь мы используем метод `assertForwardUrlMapping()`, потому что класс поддержки не позволяет проверять обратное отображение для кодов состояний.

Существует также парный ему метод `assertReverseUrlMapping()`, который принимает те же аргументы, что и другие методы проверки.

Нам больше нечего добавить к вышесказанному об интеграционных тестах. Они удивительно просты в реализации. Вся хитрость заключается лишь в том, чтобы определить, когда их следует использовать. Надеемся, что мы дали вам достаточно информации, чтобы вы могли принимать обоснованные решения.

Следующий этап тестирования заключается в том, чтобы проверить правильную работу приложения в целом и особенно пользовательского интерфейса. Для этого необходимо использовать механизм тестирования, который запускает приложение в контейнере сервлета и взаимодействует с ним. Нам необходимы функциональные тесты.

7.4. Функциональное тестирование

Комбинация модульных и интеграционных тестов может вселить в вас значительную долю уверенности, что приложение действует именно так, как надо, и они достаточно просты в разработке и в использовании. Но все же в их области существуют провалы: представления, фильтры и перехватчики контроллеров. Как нам закрыть эти провалы?

Ответ таков: протестировать работу приложения в целом, включая пользовательский интерфейс. Это как раз то, что видят и с чем взаимодействуют конечные пользователи, поэтому важно убедиться, что он действует, как ожидается. Мы называем это «функциональным тестированием», и существует несколько различных способов его реализации. Например, можно посадить человека перед компьютером и предложить ему проверить работу приложения, следуя некоторым инструкциям: щелкнуть на эту ссылку, ввести эти данные, щелкнуть на этой кнопке и так далее. Это совершенно правильный подход, и он по-прежнему используется достаточно широко.

Причина, по которой компании часто прибегают к ручному тестированию, заключается в высокой сложности разработки функциональных тестов, а тестировать пользовательский интерфейс особенно сложно. Но при ручном тестировании вы теряете преимущества автоматизации процесса: регулярность выполнения, воспроизводимость и так далее.

Мы не хотим терять эти преимущества, поэтому для нас будет приятно узнать, что существует несколько расширений для платформы Grails, способных выручить нас. Для начала мы рассмотрим способ проверки пользовательского интерфейса без привлечения броузера.

7.4.1. Расширение Functional Test

Естественной средой выполнения для приложений на платформе Grails является контейнер сервлета. Проверить работу приложения в такой среде можно единственный способом – обеспечив взаимодействия с сервером по протоколу HTTP. Некоторые расширения для платформы Grails позволяют сделать это, и самым зрелым из них является расширение WebTest от компании Canoo. Но мы дадим шанс проявить себя относительно новому расширению Functional Test.

Для начала установим расширение:

```
grails install-plugin functional-test
```

Несмотря на не очень броское название, расширение Functional Test с лихвой оправдывает его.

Прежде чем приступить к созданию теста, использующего свежее установленное расширение, нужно определить, что именно мы собираемся тестировать. В качестве примера мы возьмем страницу приложения Hubbub, где отображаются сообщения пользователя, потому что на этой странице имеется значительный объем содержимого, которое можно проверить, и к тому же она включает в себя форму. Создание заготовки теста для этой страницы выполняется очень просто:

```
grails create-functional-test PostList
```

Эта команда создаст файл test/functional/PostListTests.groovy, который вы найдете по-спартански простым.

Наш следующий шаг заключается в том, чтобы наполнить его некоторым программным кодом. В листинге 7.7 приводятся два теста: первый проверяет, создается ли сообщение об ошибке при попытке пользователя получить доступ к странице без регистрации в приложении, а второй проверяет корректность отображения страницы после того, как пользователь зарегистрируется.

Листинг 7.7. Функциональный тест для страницы приложения Hubbub со списком сообщений

```
package com.grailsinaction

class PostListTests extends functionaltestplugin.FunctionalTestCase {
    void testTimelineNotLoggedIn() {
        get("/post/timeline") ← ❶ Передача запроса GET по адресу URL

        assertStatus 500 ← Проверка кода состояния HTTP
    }
```

```
    void testTimeline() {
        post("/login/index") {
            userId = "peter"
            password = "password"
        }

        assertTitle "Hubbub » New Post"
        assertContentContains(
            "What are you hacking on right now?")
    }
}
```

❷ Передача запроса POST с данными

❸ Проверка содержимого страницы

Тест может показаться не очень внушительным, но он выполняет удивительно большое количество проверок. Сначала мы отправляем запрос GET по адресу страницы со списком сообщений ❶. В данном случае адрес определяется относительно контекста сервлета, потому что начинается с символа слэша (/), но вы можете также указать любой из следующих адресов:

- абсолютный адрес URL (в виде «http://...»)
- относительный адрес URL (такой как «post/timeline»)

Адреса URL второго вида интерпретируются как относительные применительно к *последнему загруженному адресу URL* или, в случае самого первого запроса, применительно к значению по умолчанию (определяемому параметром настройки `grails.functional.test.baseURL`).

За кулисами

Внутренняя реализация расширения Functional Test использует библиотеку HtmlUnit, поэтому, если вы знакомы с этой библиотекой, вы можете обращаться к объекту `Page` этой библиотеки через свойство `page`:

```
assertTrue page.forms["userDetails"]
```

Обратите также внимание, что методы `get()`, `post()`, `put()` и подобные им ничего не возвращают. Это обусловлено тем, что расширение (или, если быть более точными, библиотека HtmlUnit) сохраняет ответ в памяти – все последующие операции и проверки применяются к этому ответу, пока не будет выполнен следующий запрос.

Нам известно, что пользователь не может напрямую обратиться к странице, предварительно не зарегистрировавшись, поэтому мы ожидаем, что ответ будет содержать код состояния 500, который мы и проверяем. Это все, что делается в первом тесте.

Второй тест начинается с регистрации пользователя в приложении посредством отправки запроса POST контроллеру `login` ❷. Это напоминает отправку запроса GET, которая выполняется выше, но на этот раз мы передаем методу `post()` замыкание, заполненное данными для запроса POST. Каждое свойство, устанавливаемое в этом замыкании, соответствует определенному параметру в массиве данных запроса POST. Здесь все просто и понятно – как раз как мы любим. Кроме того, вы не ограничены запросами GET и POST – расширение поддерживает также методы `put()` и `delete()`.

Вернемся к нашему тесту. Действие `login` выполняет перенаправление на страницу со списком сообщений пользователя, поэтому далее мы проверяем соответствие заголовка и содержимого этой странице ❸. Убедиться, что страница является результатом перенаправления, можно следующим способом:

```
redirectEnabled = false
post("/login/index") { ... }

assertRedirectUrl("/post/timeline")
followRedirect()
redirectEnabled = true
```

Свойство `redirectEnabled` дает возможность определять, должно ли расширение «следовать» за перенаправлением. Если установить его в значение `false`, можно будет проверить адрес URL перенаправления и позволить расширению загрузить его с помощью метода `followRedirect()`. Однако в данном случае нам нужно убедиться лишь в том, что была отображена страница со списком сообщений, а не в том, что это произошло в результате перенаправления.

Теперь, когда тест написан, его можно запустить:

```
grails functional-tests
```

На выполнение теста требуется некоторое время, потому что команда сначала должна запустить приложение в контейнере сервлета. По окончании вы обнаружите, что она сгенерировала те же типы отчетов в том же месте (test/reports) , что и команда `test-app`.

В этом коротком примере мы продемонстрировали лишь некоторые из проверок, которые вы можете использовать, но расширение поддерживает значительную их коллекцию; мы перечислили многие из них в табл. 7.3. Полный перечень вы найдете в документации, по адресу: *http://www.grails.org/Grails+Functional+Testing*.

Многие проверки также имеют версии со словами «Contains» (содержит) и «Strict» (строго) в их именах, что обеспечивает дополнительную гибкость.

Таблица 7.3. Часто используемые методы проверки, предоставляемые расширением Functional Test

Метод	Описание
`assertStatus(int)`	Проверяет код состояния ответа HTTP
`assertContent(String)`	Проверяет содержимое страницы (без учета регистра символов и пробельных символов)
`assertTitle(String)`	Проверяет заголовок страницы
`assertHeader(String, String)`	Проверяет значение именованного заголовка HTTP
`assertContentType(String)`	Проверяет тип содержимого ответа

К настоящему моменту у вас может сложиться мнение, что все это требует немалых усилий и что реализация теста выполняется на слишком низком уровне. Действительно, набор имеющихся возможностей может быть и невелик, но они весьма мощны и обладают достаточной гибкостью. Вы легко можете группировать фрагменты тестов в виде своих собственных методов и повторно использовать их. Например, предположим, что нам необходимо проверить страницу со списком сообщений в нескольких тестах. Никому не захочется повторять один и тот же программный код снова и снова, поэтому мы можем реализовать проверки в виде метода, как показано ниже:

```
def checkTimelinePage() {
    assertTitle "Hubbub » New Post"
    assertContentContains "What are you hacking on right now?"
    ...
}
```

Этот метод может вызываться из любого места в наших тестах для выполнения полной проверки страницы. Создавая промежуточные фрагменты программного кода, подобные этому, мы можем создавать комплекты полных и надежных тестов.

Это все хорошо, но хотелось бы сказать несколько слов предупреждения: тесты пользовательского интерфейса – вещь достаточно хрупкая, потому что пользовательский интерфейс имеет тенденцию часто изменяться, пусть даже и незначительно. Преодолеть проблему недолговечности можно, если найти баланс между проверкой достаточно большого числа аспектов пользовательского интерфейса, позволяющей убедиться в правильной его работе, и ее специфичностью, из-за которой приходится регулярно переписывать тесты. Опыт имеет здесь большое значение, но мы полагаем, что лучше быть с тестами построже, ведь вы легко сможете ослабить их требования, если они будут приходить в негодность слишком часто. В противном же случае вы откроете дорогу для проникновения ошибок.

Это был (почти) полный обзор возможностей тестирования (почти) всех аспектов приложения. Функциональные тесты обеспечивают последний элемент уверенности, что все работает, как надо. Вы можете использовать их также для тестирования на уровне интеграции с системой, то есть вы можете запускать комплект функциональных тестов при развертывании приложения в период подготовки или во время эксплуатации, чтобы убедиться в работоспособности приложения.

Мы «почти» закончили изучение приемов тестирования, что еще осталось? Осталось рассмотреть приемы тестирования с применением JavaScript и других клиентских технологий.

7.4.2. Другие инструменты тестирования

Оба расширения, WebTest и Functional Test, используют интерпретатор Rhino JavaScript, и хотя данный интерпретатор постоянно улучшается, он по-прежнему несовместим с некоторыми библиотеками JavaScript. Это оставляет брешь, которая может быть заполнена с помощью других инструментов.

Selenium

Если для отправки запросов HTTP и обработки ответов расширение Functional Test использует Java-библиотеку, то расширение Selenium располагается на стороне броузера. У него имеются два существенных преимущества перед другими методиками:

- Вы можете проверять работу приложения с помощью броузеров, предпочитаемых вашими пользователями.
- Вы бесплатно получаете в свое распоряжение интерпретатор языка JavaScript, реализованный в броузере!

Различия, существующие между броузерами, отравляют жизнь веб-разработчика, поэтому не стоит пренебрегать возможностью запускать полный комплект автоматизированных тестов в разных броузерах. И когда другие расширения поддержки функционального тестирования спотыкаются о ваш замечательный динамический пользовательский интерфейс, из-за того что они не в состоянии обработать программный код JavaScript, использование броузера для выполнения тестов становится очевидным решением.

Замечательная особенность этого инструмента заключается в наличии интегрированной среды разработки Selenium IDE – в виде расширения для броузера Firefox. Она позволяет создавать тесты с помощью мыши. У вас практически всегда будет появляться необходимость корректировать тесты, сгенерированные этой средой, но начинать не на пустом месте всегда проще. Дополнительную информацию о расширении вы можете найти на странице проекта, по адресу: *http://www.grails.org/Selenium+plugin*.

Возникает интересный вопрос: какое расширение следует использовать, Functional Test (или WebTest) или Selenium? Может быть, оба? Это сложный вопрос, и ответ на него во многом зависит от личных предпочтений. Возможно, интерфейс Selenium IDE покажется вам более удобным, но вообще расширения Functional Test и WebTest работают существенно быстрее.

Как отпетым программистам нам больше нравятся выразительность расширения Functional Test и тот уровень контроля, который оно обеспечивает при создании тестов. Мы предпочитаем задумываться о применении Selenium, только когда необходимо протестировать совместимость приложения с разными броузерами или когда использование JavaScript начинает вызывать проблемы.

JUnit

Поскольку мы заговорили о JavaScript – разве это не программный код? Должны ли мы тестировать и его? Да, и это возможно. Хотя это программный код, при желании его можно протестировать с помощью Selenium. Но тогда нужно говорить о функциональном тестировании JavaScript, а не о модульном.

Если в вашем приложении имеется какой-то нетривиальный программный код JavaScript, есть смысл создать для него модульный тест. В этом вам поможет инструмент JsUnit, который, как можно предположить из названия, является членом семейства платформ тестирования xUnit. Установите расширение JsUnit для платформы Grails и вперед!

Документацию вы найдете на веб-сайте платформы Grails, а описание платформы JsUnit – на сайте *http://jsunit.berlios.de/*.

Fixtures

Как известно, платформа Grails уже имеет механизм начальной подготовки данных при запуске приложения: класс `BootStrap`. Возможно, вы полагаете, что этого более чем достаточно для удовлетворения ваших потребностей; если это так, то мы хотели бы, чтобы вы пересмотрели свое мнение.

Основная проблема имеет отношение к функциональным тестам: хотя класс `BootStrap` вызывается при запуске сервера, нет никакого способа вернуть данные в исходное состояние во время выполнения тестов. Это означает, что база данных начинает заполняться записями, и все последующие тесты должны учитывать наличие дополнительных данных, добавленных предыдущими тестами. Кошмар!

Для примера представьте, что имеется тест, ожидающий, что на странице со списком будет отображаться пять сообщений. Затем приходит другой разработчик и создает тест, добавляющий еще одно сообщение в этот список. Происходит поломка нашего теста, потому что теперь страница со списком отображает *шесть* сообщений.

Решение проблемы заключается еще в одном замечательном расширении для платформы Grails: Fixtures. Он позволяет определять наборы данных (с помощью классов предметной области, а не SQL), которые можно загрузить в любой момент времени. Пример использования этого расширения в интеграционных тестах можно увидеть на веб-странице: *http://www.grails.org/Fixtures+Plugin*.

Использование расширения Fixtures в функциональных тестах сопряжено с определенными трудностями, потому что тесты выполняются за пределами сервера. Один из вариантов решения этой проблемы состоит в создании специализированного контроллера, который будет удалять данные из базы данных и затем загружать подготовленные наборы данных. Только будьте внимательны и делайте этот контроллер недоступным в действующих приложениях.

На этом мы завершаем краткий обзор альтернативных инструментов тестирования и описание приемов тестирования в платформе Grails. Теперь вы должны четко себе представлять, как выполняется модульное и функциональное тестирование, а также уметь пользоваться этими знаниями. Вы обнаружите массу возможностей облегчить тестирование и сделать его более увлекательным, включая такие инструменты, как платформа easyb для ведения разработки, управляемой поведением приложения (*http://www.easyb.org/*), и расширение Code Coverage, позволяющее оценить степень охвата программного кода тестами (*http://www.piragua.com/* или *http://www.grails.org/Test+Code+Coverage+Plugin*). Поэтому продолжайте тестировать!

7.5. В заключение

Тестирование – один из важнейших способов убедиться в высоком качестве продукта. Как без него можно быть уверенным в безупречной работе приложения? Тестирование оказывает также существенное влияние на реорганизацию программного кода, потому что без него невозможно понять, не была ли нарушена работа какой-либо части приложения.

Основная цель этой главы состояла в том, чтобы привить вам привычку тестирования. Как только тестирование войдет у вас в привычку, вы обнаружите, насколько просто ей следовать. Вторая цель состояла в том, чтобы показать вам, как писать тесты всех уровней: модульные, интеграционные и функциональные. Вы должны использовать тесты всех уровней, чтобы гарантировать максимальную надежность. Существует множество инструментов, способных помочь в этом, так почему бы не использовать их?

Следующие рекомендации помогут вам получить максимум выгоды от тестирования:

- *Пишите тесты перед тем, как писать программный код, и тестируйте его постоянно*. Если вы будете писать тесты в первую очередь,

то вы сможете не только гарантировать полный охват ими программного кода, но и намного отчетливее себе представить, что должен делать ваш программный код. При постоянном тестировании вы сразу же будете замечать нарушения работоспособности, и вам проще будет их устранять. Это поможет повысить скорость разработки.

- *Упрощайте возможность тестирования*. Вы могли подумать, что простота тестирования достигается за счет использования инструментов тестирования, и в этом есть доля правды. Но немало усилий необходимо приложить и для того, чтобы написать программный код, простой для тестирования, с целью обеспечить его охват тестами. Возможность внедрять в испытательные тесты такие элементы, как вспомогательные объекты, использование интерфейсов и отказ от статических переменных и методов, может существенно помочь в этом.

  Когда обнаруживаются сложности в тестировании, поищите решение для похожего программного кода, которым можно было бы воспользоваться. Иначе вы рискуете оставить свой код неохваченным тестами.

- *Поддерживайте полноту охвата тестами*. Достаточно просто пропустить или отложить создание теста, но это станет началом скользкого пути. Существует теория, что в районах, где встречаются разбитые окна, уровень преступности выше, потому что в таких районах люди обычно думают, что раз никто ни о чем не беспокоится, то и им это делать необязательно. Нечто похожее происходит и с тестами: если в программном коде появляются фрагменты, не охваченные тестами, у разработчика создается впечатление, что они не имеют большого значения. Старайтесь избегать появления участков, неохваченных тестами, насколько это возможно.

- *Используйте приемы непрерывной интеграции*. Работая в команде, вы быстро обнаружите, что программный код, добавленный одним человеком, легко становится неработоспособным после внесения изменений другим. Чаще пользуйтесь системой непрерывной интеграции (Continuous Integration, CI), чтобы гарантировать выявление нарушений на самых ранних этапах. В действительности использование системы непрерывной интеграции может быть полезным даже для группы, состоящей из одного человека, например для выявления файлов которые не были созданы им.

- *Выполняйте тестирование на всех уровнях*. Мы не можем не подчеркнуть важность этого еще раз: обязательно проводите как модульное, так и функциональное тестирование. Даже если тесты где-то перекрываются и в чем-то избыточны, это гораздо лучше, чем наличие пробелов в охвате тестами.

Лекция окончена. А в качестве награды за упорство вы получаете возможность приятно провести время с интересными расширениями.

III

Grails на каждый день

Во второй части этой книги вы познакомились с основными блоками, из которых строятся приложения на платформе Grails. В третьей части вы будете использовать эти знания для создания всех необходимых элементов действующих приложений.

В главе 8 будут представлены расширения для платформы Grails – способ увеличения возможностей приложений с помощью сторонних модулей. Мы будем использовать расширения для придания приложению Hubbub потрясающего внешнего вида и познакомим вас с расширениями, применяемыми для вывода диаграмм, графиков, всплывающих подсказок, редакторов текста с возможностью форматирования, и множеством других. Кроме того, мы добавим некоторые важные функциональные расширения, такие как интеграция с электронной почтой и возможность полнотекстового поиска.

В главе 9 будет дан обзор Grails Webflow – легковесного механизма определения состояний, который упрощает возможность добавления в приложение последовательностей действий в стиле мастера. На примере создания покупательской корзины для интернет-магазина в приложении Hubbub вы узнаете, как платформа Grails способна упростить для пользователя процедуру выполнения последовательности операций, обеспечивая возможность определения специальных подпоследовательностей в случае необходимости. Кроме того, вы узнаете, как протестировать все это.

Безопасность – жизненно важная тема в развертывании любого приложения в Интернете, поэтому в главе 10 мы дадим вам исчерпывающие сведения касательно ее обеспечения в сети. После знакомства с основными слабыми местами в веб-приложениях вы научитесь устранять их и сохранять все защищенным. Кроме того, мы представим вам популярное расширение обеспечения безопасности для платформы Grails,

которое позволит вам добавлять самые разные способы управления доступом к особенностям приложения Hubbub.

Предоставление функциональных возможностей приложений посредством прикладного интерфейса REST стало распространенной практикой для большинства успешных приложений Web 2.0. В главе 11 мы поговорим об идеях, лежащих в основе прикладных интерфейсов REST, и расскажем вам обо всех инструментах, с помощью которых вы сможете быстро встраивать поддержку особенностей технологии REST в свои приложения. Однако поддержка REST – не единственное развлечение среди имеющихся возможностей организации удаленных взаимодействий, поэтому мы также рассмотрим многие другие популярные технологии удаленных взаимодействий, включая SOAP и RMI.

Архитектуры, ориентированные на передачу сообщений, стали чрезвычайно широко использоваться для разработки высоко масштабируемых приложений. В главе 12 мы познакомим вас с механизмом обмена сообщениями, добавив в приложение Hubbub шлюз обмена мгновенными сообщениями. Вы изучите очереди сообщений и темы и выясните, как расширение JMS обеспечивает высокую производительность при взаимодействии с системами обмена сообщениями. После этих легковесных механизмов обмена сообщениями мы рассмотрим также возможности планирования выполнения заданий с помощью расширения Quartz.

К концу третьей части вы будете обладать всеми знаниями, необходимыми для разработки надежных, многофункциональных веб-приложений на платформе Grails.

В этой главе:

- Основы расширений для платформы Grails
- Добавление графиков и диаграмм
- Встраивание поддержки электронной почты
- Добавление возможности полнотекстового поиска
- Улучшение пользовательского интерфейса

8

Расширения: добавляем поддержку Web 2.0 за 60 минут

Не так много программных продуктов обладает полным набором необходимых возможностей, и совершенно неудивительно, что платформа Grails не является исключением. К счастью, существует множество инструментов и библиотек на языке Java, которые помогут вам реализовать практически любые необходимые вам особенности. Платформа Grails представляет собой платформу, в основе которой лежит язык Java, поэтому вы можете использовать почти все библиотеки Java. Многие из них весьма надежны и достаточно продуманные, так зачем повторно изобретать колесо?

Вы можете использовать библиотеки Java как есть, но значительные преимущества можно получить при применении «устройства», обеспечивающего сопряжение платформы Grails и библиотек, а также упрощающего использование и настройку последних. Эти задачи решает система расширений платформы Grails. Идея состоит в том, что *функциональные возможности* связываются в модули, которые могут загружаться платформой и встраиваться в систему. Такими функциональными возможностями могут быть полнотекстовый поиск, реализация облаков тегов или новые технологии создания пользовательского интерфейса. В действительности многие особенности платформы Grails реализованы в виде расширений, включая механизм GORM.

На рис. 8.1 показано, как обеспечивается взаимосвязь между расширениями и платформой Grails. Для тех, кто знаком с Eclipse или любой другой интегрированной средой разработки на языке Java, заметим, что они обладают системами расширений, аналогичными системе, имеющейся в платформе Grails.

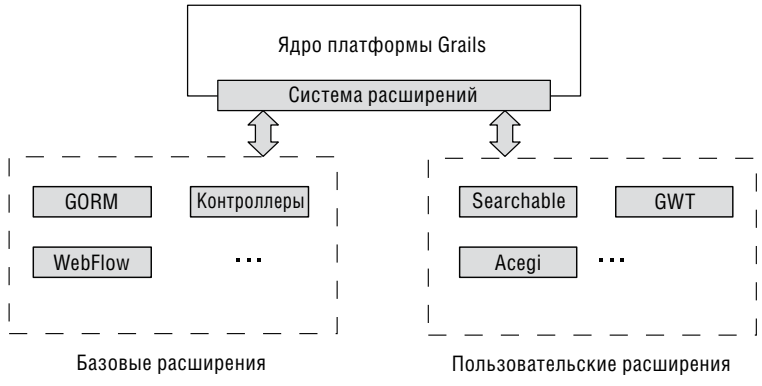


Рис. 8.1. *Расширяемая архитектура*

Благодаря этому любой желающий может реализовать дополнительные функциональные возможности, написав расширение, и этим занимаются многие. Вы увидите, насколько просто устанавливать расширения в приложения, фактически накачивая их стероидами. Начиная с раздела 8.2, мы будем знакомиться с некоторыми популярными расширениями.

8.1. Использование чужого труда в своих интересах

Когда может потребоваться установить расширение? Если вы обнаруживаете потребность в некоторой особенности, которая не является характерной только для вашего приложения, то есть в той, которая не связана непосредственно с его логикой работы, есть смысл поискать расширения, которые реализуют эту особенность. Там, где в простом проекте на языке Java вы применили бы отдельную библиотеку или инструмент, в приложении на платформе Grails, вероятно, лучшим решением будет использование расширения. Помните, в повторном изобретении колеса обычно мало проку.

Возьмем в качестве примера приложение Hubbub. Нам необходимо организовать отправку сообщений по электронной почте из различных частей приложения, таких как модуль регистрации нового пользователя или модуль рассылки ежедневных новостей, но рассылка по электронной почте не является прямой обязанностью приложения Hubbub. Данная особенность лежит за пределами основных обязанностей Hubbub и часто используется самыми разными приложениями. Это делает поддержку рассылки по электронной почте отличным кандидатом на установку расширения, и вы не удивитесь, узнав, что такое расширение уже существует.

Как только вы решите, что поддержка некоторой особенности может быть реализована с помощью расширения, можно приступать к поиску наиболее подходящего.

8.1.1. Поиск расширений

Существует два основных источника информации о расширениях: репозиторий расширений для платформы Grails и веб-сайт проекта Grails.

Репозиторий расширений для платформы Grails

В платформе Grails имеется пара команд, которые могут использоваться для получения информации о расширениях, при этом они обе оперируют с так называемым *репозиторием расширений для платформы Grails*. Этот репозиторий представляет собой централизованное хранилище, где находится множество расширений для платформы Grails, что упрощает для Grails получение информации о них, установку и выгрузку. Как будет показано ниже, вы даже можете настраивать поддержку своих локальных репозиториев.

А теперь запросим информацию об интересующих нас возможностях:

```
grails list-plugins
```

Результат работы команды будет выглядеть примерно так, как показано ниже:

```
Plug-ins available in the Grails repository are listed below:
------------------------------------------------------------
acegi              <0.5.1>          -- Grails Spring Security 2.0 Plugin
aop                <no releases>    -- No description available
audit-logging      <0.4>            -- adds hibernate audit logging
...
```

В полном соответствии со своим именем команда `list-plugins` выводит список расширений, доступных в настоящее время в репозитории. Для каждого из них отображаются имя, номер последней версии и краткое описание. Обратите внимание, что для некоторых расширений выводятся «<no releases>» (<нет номера версии>) и «No description available» (Описание недоступно). Во многих случаях это признак того, что расширение было добавлено в репозиторий еще до выхода версии Grails 1.0 и не обновлялось. Будьте осторожны с такими расширениями.

> **За кулисами**
>
> Репозиторий расширений для платформы Grails реализован как репозиторий Subversion и доступен по протоколу HTTP, поэтому чтобы получить доступ к нему, ваш компьютер должен быть подключен к Интернету. Структура каталогов и файлов в репозито-

рии соответствует соглашениям, позволяющим платформе Grails загружать определенные версии расширений.

Если вам требуется настроить прокси-сервер HTTP, запустите следующую команду:

```
grails set-proxy
```

Введите запрашиваемую информацию. Список расширений будет сохранен в локальном кэше, в файле с именем plugins-list.xml, который обычно сохраняется в каталоге $HOME/.grails/1.1/plugins.

Список расширений достаточно длинный. Если вы пользуетесь UNIX-подобной системой, такой как Mac OS X или Linux, мы рекомендуем пропустить вывод команды через фильтр утилиты grep, например так:

```
grails list-plugins | grep "mail"
```

В операционной системе Windows это невозможно (если только вы не пользуетесь пакетом программ Cygwin), однако расширения следуют в списке в алфавитном порядке, поэтому для вас не составит большого труда отыскать требуемое расширение в списке.

Теперь у нас имеется список, что дальше? Нам требуется отыскать расширение, которое облегчит отправку сообщений по электронной почте из приложения Hubbub, поэтому поищем что-нибудь со словом «mail» в названии. Поиск расширения со словом «mail» в названии, которое согласно описанию «обеспечивает поддержку операций с электронной почтой в действующих приложениях Grails», не должен занять много времени. Это похоже на поиск по картотеке.

Как только будет найдено искомое расширение, вы можете получить дополнительные сведения о нем с помощью следующей команды:

```
grails plugin-info mail
```

Вы получите результаты, подобные показанным ниже:

```
---------------------------------------------------------------------------
Information about Grails plugin
(Перевод: Информация о расширении для платформы Grails)
---------------------------------------------------------------------------
Name: mail | Latest release: 0.5
(Перевод: Название: mail | Последняя версия: 0.5)
---------------------------------------------------------------------------
Provides Mail support to a running Grails application
(Перевод: Обеспечивает поддержку операций с электронной почтой в действующих
приложениях Grails)
---------------------------------------------------------------------------
```

```
Author: Graeme Rocher
(Перевод: Автор: Греем Рочер (Graeme Rocher))
---------------------------------------------------------------------------
Author's e-mail: graeme@g2one.com
(Перевод: Адрес электронной почты автора: graeme@g2one.com)
---------------------------------------------------------------------------
Find more info here: http://grails.org/Mail+Plugin
(Перевод: Дополнительную информацию можно найти по адресу: http://grails.
org/Mail+Plugin)
---------------------------------------------------------------------------
This plug-in provides a MailService class as well as configuring the
necessary beans within the Spring ApplicationContext.
...
(Перевод: Это расширение предоставляет класс MailService, а также выполняет
настройку всех необходимых компонентов Spring ApplicationContext)
---------------------------------------------------------------------------
Available full releases: 0.1 0.1-ALPHA 0.3 0.4 0.5 0.6-SNAPSHOT
...
(Перевод: Доступные версии: 0.1 0.1-ALPHA 0.3 0.4 0.5 0.6-SNAPSHOT)
```

Здесь приводятся важная информация о местоположении электронной документации (*http://grails.org/Mail+Plugin* в данном примере) и подробное описание, содержащее достаточно сведений, чтобы вы могли решить, подходит вам это расширение или нет.

Несмотря на то что репозиторий расширений для платформы Grails упрощает жизнь пользователям, он содержит далеко не все доступные расширения. Расширения, отсутствующие в репозитории, часто можно найти на веб-сайте проекта Grails.

Веб-сайт проекта Grails

Помимо массы другой информации, полезной для разработчиков приложений на платформе Grails, на основном веб-сайте проекта Grails имеется страница, содержащая список расширений, разбитых по категориям. Здесь находится также документация ко многим из них, это означает, что поиск расширений по веб-сайту может быть весьма плодотворным. Однако его недостаток заключается в том, что в предоставлении самой свежей информации веб-сайт полагается на пользователей и авторов расширений, поэтому здесь могут приводиться устаревшие, ошибочные или относящиеся к несуществующим расширениям сведения.

Список расширений по категориям можно найти по адресу: *http://grails.org/plugin/home*. Этот раздел сайта известен как «Портал расширений» (Plugin Portal), здесь вы можете вести поиск расширений и справочной информации для них. Сайт поддерживает возможность рассылки новостей в виде ленты RSS, позволяющей получать самую свежую инфор-

мацию об обновлениях существующих расширений и появлении новых. Вы можете даже голосовать за понравившиеся вам расширения.

Что делать, если поиск с помощью команд Grails или на веб-сайте проекта не увенчался успехом? У вас есть еще один шанс – задать вопрос в списке рассылки (*http://grails.org/Mailing+lists*). Сообщество Grails очень дружественно и отзывчиво.

Отыскав требуемое расширение, его можно установить в свое приложение.

8.1.2. Установка расширений

Как и с любым другим программным обеспечением, прежде чем использовать расширение, его нужно установить. К счастью, платформа Grails предоставляет для этого простую команду:

```
grails install-plugin <название расширения>
```

Подставьте название расширения, которое требуется установить, и платформа Grails загрузит и установит последнюю его версию из репозитория расширений для платформы Grails. При установке следует использовать название, которое отображается командой `list-plugins`; в нашем предыдущем примере расширение имеет название «mail».

Во время установки расширения платформа Grails сохраняет локальную копию архива zip с расширением. Это означает, что при установке расширения в другой проект платформа Grails сможет использовать версию из кэша вместо того, чтобы загружать расширение из репозитория повторно. В случае выхода новой версии расширения команда `install-plugin` загрузит ее из репозитория. Логика работы платформы Grails при установке расширения приводится на рис. 8.2.

А как быть, если потребуется использовать одну и ту же версию расширения во всех ваших проектах? Постоянное обновление большого числа проектов может занимать достаточно много времени, особенно когда после установки новой версии требуется вносить изменения в программный код. Или, возможно, вы тщательно протестировали определенную версию, и обновление влечет за собой неоправданный риск. Каковы бы ни были причины, часто бывает необходимо явно указывать версию устанавливаемого расширения. Добиться этого можно разными способами, но самый простой заключается в передаче номера желаемой версии команде `install-plugin`, например так:

```
grails install-plugin mail 0.1
```

Другой вариант – указать местоположение файла пакета с расширением, который традиционно имеет имя в формате grails-<название>-<версия>.zip, такое как grails-mail-0.1.zip. В качестве местоположения можно указывать адрес URL, например:

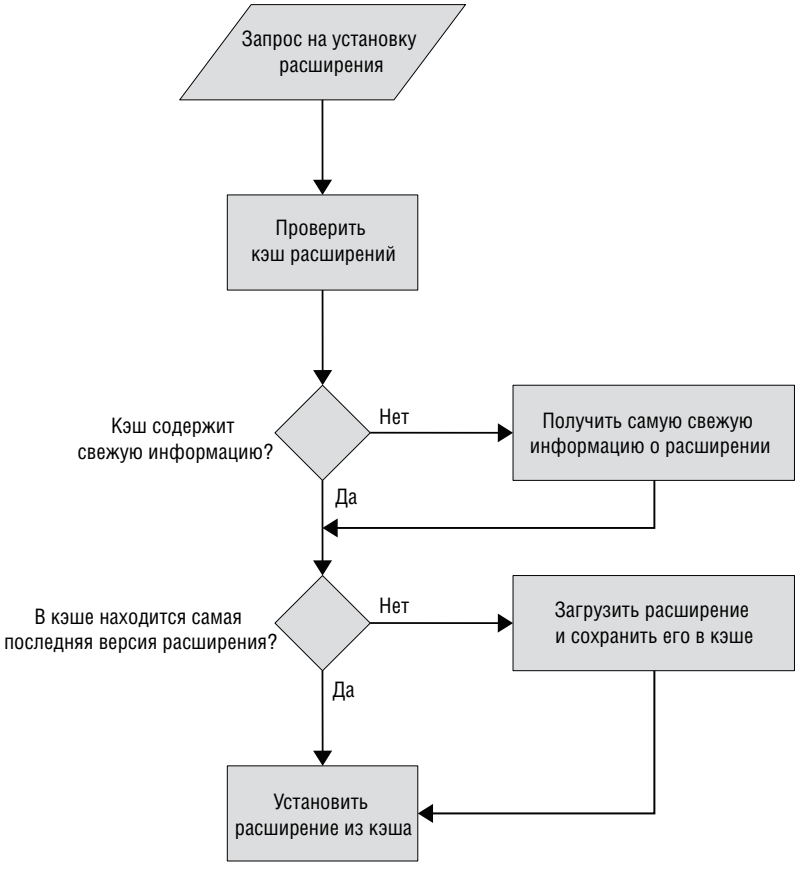


Рис. 8.2. *Логика работы платформы Grails при установке расширения*

```
grails install-plugin http://somewhere.org/plugins/grails-mail-0.1.zip
```

или путь к локальному файлу, если файл был загружен вами вручную или получен от третьих лиц:

```
grails install-plugin /home/ben/plugins/grails-mail-0.1.zip
```

Путь к файлу может быть абсолютным, как в предыдущем примере, или относительным. Некоторые расширения распространяются исключительно таким способом, и в этом случае они не появляются в выводе команды `list-plugins`.

В главе 16 мы рассмотрим еще один способ, который связан с настройкой локального репозитория, но сначала нам требуется поближе познакомиться с процедурой установки расширений.

Где сохраняются мои расширения?

Итак, вы установили пару расширений, но где платформа Grails сохранила их? Хотя вам совсем не обязательно знать, где сохраняются расширения, эта информация может пригодиться вам как источник дополнительных сведений и вдохновения, когда вы решите написать свое собственное расширение.

По умолчанию расширения сохраняются в каталоге

```
$HOME/.grails/<grailsVersion>/projects/<project>/plugins
```

где `$HOME` – домашний каталог текущего пользователя, `<grailsVersion>` – версия платформы Grails, использовавшейся при установке расширения, и `<project>` – название проекта, в который устанавливается расширение.

При глобальной установке расширения сохраняются в другом каталоге, поскольку они не связаны с каким-то определенным проектом:

```
$HOME/.grails/<grailsVersion>/global-plugins
```

В главе 16 мы покажем, как управлять местоположением, куда платформа Grails сохраняет расширения.

Проекты запоминают установленные расширения

После установки расширения в проект этот проект запоминает факт установки. Если вы являетесь единственным разработчиком приложения, данный факт не имеет большого значения, но представьте, что вы работаете в составе команды, и кто-то другой добавил расширение в проект и отправил изменения в используемую вами систему управления версиями (Version Control System, VCS). Поскольку проект запоминает, какие расширения были установлены, платформа Grails автоматически загрузит и установит все, что отсутствует в вашей локальной копии. Вам ничего не придется делать. Когда вы попытаетесь запустить приложение, оно будет работать, по крайней мере это касается расширений.

Глобальные расширения

Некоторые расширения настолько универсальны, что у вас может появиться желание установить их во все свои проекты. Рассмотрим расширение Debug, которое позволяет отслеживать потоки информации в приложении на платформе Grails. Оно не добавляет новых особенностей в приложение, однако обеспечивает дополнительную поддержку процесса разработки. И это делает данное расширение идеальным кандидатом для установки во *все* проекты.

Установка одного и того же расширения в каждый проект – довольно утомительное занятие, поэтому платформа Grails позволяет устанавливать такие расширения глобально, для чего достаточно добавить ключ `-global` в команду `install-plugin`:

```
grails install-plugin -global debug
```

Как уже говорилось выше, проекты обычно сохраняют информацию об установленных расширениях. Однако в случае глобальной установки расширения связь проекта с расширением не сохраняется, поэтому если установить расширение глобально и отправить изменения проекта в VCS, платформа Grails не будет автоматически загружать и устанавливать это расширение у ваших коллег.

Совет

Когда вы решаете, устанавливать ли расширение глобально, спросите себя: это расширение действительно *требуется* для проектов? Если ответ утвердительный, тогда вам *не* следует устанавливать его глобально.

Зависимости расширений

Многие расширения являются самодостаточными. Это означает, что все, что им необходимо, либо доступно в самой платформе Grails, либо находится в пакете расширения. Но некоторые расширения имеют более сложную организацию и требуют наличия особенностей, предоставляемых другими расширениями, – они *зависят* от других расширений, которые в свою очередь могут зависеть от других расширений.

На рис. 8.3 показан (достаточно искусственный) пример зависимостей расширения для приложения на платформе Grails. Как видно на рисунке, управление расширениями и их зависимостями может быть довольно трудоемким. По этой причине платформа Grails берет на себя все тяготы управления зависимостями. Когда выполняется установка расширения, она автоматически проверяет его зависимости и устанавливает все необходимые расширения, если они не были установлены ранее.

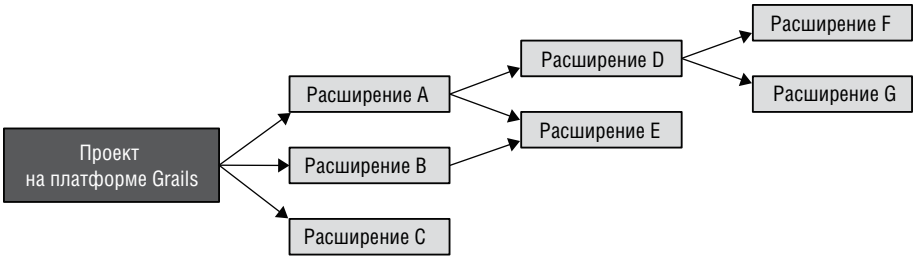


Рис. 8.3. *Пример зависимостей расширения*

Взгляните на рис. 8.3 и представьте, что вам необходимо установить в проект расширение A. Когда вы запустите команду `install-plugin`, платформа Grails автоматически загрузит и установит расширения D, E, F и G, потому что расширение A прямо или косвенно зависит от них. Если ранее вы уже установили расширение B, тогда платформа Grails установит только расширения D, F и G – расширение E уже будет установлено, так как оно требуется расширению B.

Для вас, как для пользователя расширений, все эти операции будут выполняться незаметно. Даже если на данном этапе вы не вполне понимаете, что происходит, все в порядке. Единственное, что вам следует запомнить, – даже при установке расширения из архива zip, а не из репозитория расширений платформа Grails по-прежнему будет пытаться устанавливать зависимости из репозитория. Возможно, это не является проблемой для вас, но вам стоит знать об этом на всякий случай.

Удаление расширений

Отсутствие легкого способа удалить нечто установленное может оказаться весьма досадным обстоятельством. Платформа Grails для удаления из системы установленного расширения предоставляет команду с соответствующим именем `uninstall-plugin`. Передайте команде имя расширения, и она сделает все остальное:

```
grails uninstall-plugin mail
```

Напомним, что проекты запоминают информацию об установленных расширениях. Значит, платформа Grails автоматически удалит расширение, если оно установлено в проект, в котором оно не используется. Это также существенно упрощает жизнь при работе в команде.

Применяем наши знания: начинаем улучшать внешний вид приложения Hubbub

Теперь, когда вы получили представление о том, как выполняются установка и удаление расширений, а также другие сопутствующие операции, можно попробовать применить теоретические знания к реальному проекту. В оставшейся части этой главы мы познакомим вас с несколькими популярными расширениями для платформы Grails и придадим дополнительную привлекательность внешнему виду приложения Hubbub, добавив такие популярные особенности в стиле Web 2.0, как вывод графиков и диаграмм, интеграция с электронной почтой, полнотекстовый поиск и другие улучшения в пользовательском интерфейсе.

Как только вы узнаете, как пользоваться расширениями для быстрого добавления новых особенностей в ваше приложение, вы почувствуете прилив сил от практически мгновенного вознаграждения, которое дает такой способ применения функциональных возможностей многократного использования. Начнем с создания диаграмм.

8.2. Добавление графиков и диаграмм

Задача построения диаграмм в веб-приложениях традиционно воспринималась как головная боль. Большинство библиотек генерируют изображения прямо в ходе выполнения приложения или заранее, что требует некоторого времени и страдает от недостаточной гибкости. А если говорить об эстетике, то только ваша мама сможет положительно отозваться о таких диаграммах. Если вы хотите придать им потрясающий внешний вид, вам придется использовать средства создания Flash-изображений, которые имеют собственные проблемы с производительностью и доступностью.

Когда появилась служба Google Chart, мир веб-диаграмм внезапно стал гораздо проще и элегантнее. Принцип действия службы Google Chart концептуально прост. Вы создаете на своей странице тег `<IMG>`, который ссылается на адрес URL службы Google Chart, и добавляете в адрес URL параметры, определяющие тип создаваемой диаграммы, данные для отображения и размер диаграммы. Служба Google генерирует диаграмму с типичной для Google скоростью, к которой уже все привыкли, и вы получаете привлекательную диаграмму.

8.2.1. Установка расширения Google Chart

Для разработчиков приложений на платформе Grails проблем с созданием диаграмм еще меньше. Расширение Google Chart предлагает набор простых тегов, упрощающих добавление диаграмм в страницы. Установим расширение и создадим несколько диаграмм:

```
grails install-plugin google-chart 0.4.8
```

Мы определили здесь конкретную версию, но вы можете опустить номер версии, чтобы установить самую свежую. Расширение Google Chart распространяется вместе с примером страницы, демонстрирующей доступные типы диаграмм. Если вы захотите узнать, что вам доступно, запустите свое приложение и откройте адрес *http://localhost:8080/hubbub/plugins/google-chart-0.4.8/*. Обратите внимание, что вы должны изменить номер версии в соответствии с версией расширения Google Chart, установленной у вас.

Теперь познакомимся с основными тегами диаграмм и посмотрим, как их можно использовать.

8.2.2. Создание первой диаграммы

В нашей первой диаграмме мы реализуем представление некоторой простой статистической информации о приложении Hubbub. Мы создадим страницу со статистической информацией для каждого пользователя, где будут отображаться некоторые сведения о частоте отправки сообщений.

Сначала мы определим правило отображения адресов URL, чтобы обеспечить наших пользователей дружественным способом получения страницы со статистическими сведениями:

```
"/users/$userId/stats" {
    controller = "user"
    action = "stats"
}
```

Теперь, когда у нас имеется постоянная ссылка `stats`, добавим действие `stats` в контроллер `UserController`. Для начала создадим круговую и столбчатую диаграммы, отображающие дни недели, когда пользователь наиболее активно отправлял сообщения.

В нашем действии нам необходимо определить отображение, содержащее названия дней недели (Mon, Tues и так далее) и количество сообщений, отправленных в эти дни. Реализация этого действия приводится в листинге 8.1.

Листинг 8.1. Добавление действия stats в контроллер UserController

```
def stats = {
    User user = User.findByUserId(params.userId)
    if (user) {
        def sdf = new java.text.SimpleDateFormat('E')
        def postsOnDay = [:]
        user.posts.each { post ->
            def dayOfWeek = sdf.format(post.dateCreated)
            if (postsOnDay[dayOfWeek]) {
                postsOnDay[dayOfWeek]++
            } else {
                postsOnDay[dayOfWeek] = 1
            }
        }
        return [ userId: params.userId, postsOnDay: postsOnDay ]
    } else {
        flash.message = "No stats available for that user"
        redirect(uri: "/")
    }
}
```

Преобразует дату в день недели

Преобразует дату сообщения в день недели

Отображает день недели и количество сообщений, отправленных в этот день

Единственное, что здесь необычно, – использование класса `SimpleDataFormat` для форматирования даты сообщения. Это довольно хитрый способ передачи даты и преобразования ее в день недели. Затем создается отображение, роль ключей в котором играют названия дней недели, а роль значений – количество сообщений в эти дни. Мы могли бы реализовать данные вычисления более эффективно с помощью проекций Hibernate (о которых рассказывалось в главе 4 в разделе 4.3.3), но пока будем использовать простое решение.

Получив отображение, можно приступать к выводу диаграммы в файле GSP. Расширение Google Chart предоставляет библиотеку тегов, содержащую удобные теги для вывода всех распространенных типов диаграмм. Начнем с круговой диаграммы, как показано в листинге 8.2.

Листинг 8.2. Создание трехмерной круговой диаграммы

```
<g:pieChart type="3d"
        title='Posts Per Day'
        size="${[600,200]}"
        labels="${postsOnDay.keySet()}"
        dataType='text'
        data='${postsOnDay.values().asList()}' />
```

Мы использовали тег `<g:pieChart>` и передали ему текст заголовка, названия меток и данные. Как и следовало ожидать, атрибут `labels` – это список строк, а атрибут `data` – это список чисел. Все остальное берет на себя расширение Google Chart. Поле `dataType` имеет большое значение, и мы вернемся к нему вскоре, но сначала взгляните на рис. 8.4, где изображено то, что получилось.

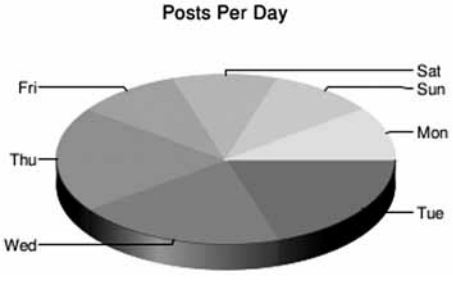


***Рис. 8.4.** Трехмерная круговая диаграмма в действии*

Получилась весьма привлекательная круговая диаграмма, которую мы реализовали всего одной строкой! Здесь используются цвета по умолчанию, но вы можете выбрать свои цвета, о чем мы тоже скоро поговорим. Но прежде чем продолжить, разберемся, какое влияние оказывает тип данных.

8.2.3. Что это за история с атрибутом dataType?

Расширение Google Chart поддерживает три типа данных, которые можно указывать в атрибуте `dataType`: `text`, `simple` и `extended`. То, какой из них вам нужно использовать, определяется типом и размером данных, на основе которых требуется построить диаграмму. В зависимости от значения атрибута `dataType` расширение Google Chart выполняет нормализацию данных и кодирование результата для отправки на сервер Google.

Доступные типы данных описываются в табл. 8.1.

Для наборов данных, содержащих больше, чем несколько элементов, лучшим выбором чаще всего является тип `simple`, поскольку тип `text` в результате дает длинный адрес URL.

Таблица 8.1. Типы данных, поддерживаемые расширением Google Chart

Тип данных	Дискретные значения	Достоинства/недостатки
`text`	100	Создает простой в отладке, но достаточно длинный адрес URL. Каждое значение в адресе присутствует в простом текстовом виде.
`simple`	63	Предпочтительно использовать для большинства небольших диаграмм. Создает самые короткие адреса URL. Значения данных сжаты с использованием специальной системы кодирования: 1=A, 2=B и так далее.
`extended`	4096	Удобно использовать только для очень больших диаграмм с мелкой шкалой. Для этого типа данных используется похожая на `simple` система кодирования: 0=AA, 1=AB, 2=AC и так далее.

8.2.4. Гистограммы: настройка цветов и сетки

Закончив реализацию круговой диаграммы и познакомившись с используемыми типами данных, посмотрим, как можно реализовать отображение тех же данных в виде гистограммы (столбчатой диаграммы). Как и в случае с круговой диаграммой, все начинается с тега диаграммы, как показано в листинге 8.3.

Листинг 8.3. Гистограмма, создаваемая расширением Google Chart

```
<g:barChart type="bvs"
        title="Posts Per Date (bar)"
        size="${[200,200]}"
        colors="${['00FF00']}"
        fill="${'bg,s,efefef'}"
        axes="x,y"
        axesLabels="${
            [0:postsOnDay.keySet(),
             1:[0,postsOnDay.values().max()/2,
                 postsOnDay.values().max()]
            ]
        }"
        dataType="simple"
        data="${postsOnDay.values().asList()}" />
```

Теперь тег выглядит немного более сложным, потому что мы начали использовать некоторые дополнительные возможности библиотеки. Прежде всего посмотрим на получившийся результат, изображенный на рис. 8.5.

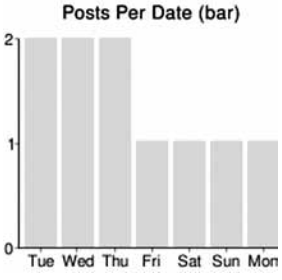


Рис. 8.5. *Гистограмма Google Chart в действии*

Большинство новых атрибутов, которые мы добавили, необходимы, потому что мы работаем с диаграммой, имеющей оси. Нам нужно указать значения меток для оси Y в формате `[ минимальное, среднее, максимальное ]`. Значения меток определяются программно, чтобы показать, что они могут определяться при построении диаграмм для любых значений данных. В нашем примере максимальное число сообщений в день равно двум, поэтому минимальному, среднему и максимальному значениям меток соответствуют 0, 1 и 2.

Кроме того, метки осей имеют свой собственный формат, который может приводить в замешательство. Метки определяются как `0: (список меток)`, `1: (список меток)`, где 0 соответствует оси X, а 1 – оси Y.

Значение `bvs` в атрибуте `type` сообщает расширению Google Chart, что нам требуется стиль «Barchart Vertical Stacked» (вертикальная столбчатая диаграмма с наложением). С помощью этого атрибута можно выбирать между горизонтальным и вертикальным расположением столбцов, определять режим наложения или группировки нескольких наборов данных. Ознакомиться с доступными разновидностями диаграмм можно в руководстве разработчика «Google Chart Developer's Guide» (*http://code.google.com/apis/chart/*).

Мы добавили также несколько дополнительных настроек. Сразу бросается в глаза, что мы предусмотрели настройку цветов. В качестве фона мы выбрали светло-серый цвет (`fill="${'bg,s,efefef'}"`), а для столбцов поменяли желтый цвет, используемый по умолчанию, на зеленый (`colors="${['00FF00']}"`). Обратите внимание, что расширение Google Chart использует шестнадцатеричные значения для представления цветов, которые могут быть вам знакомы по CSS. Цвет фона определяется в примере как `bg,s,efefef`, что переводится как «background, solid fill, light gray» (фон, сплошная заливка, светло-серый цвет).

Расширение Google Chart полно таких магических кодов; их слишком много, чтобы рассказывать о каждом здесь, поэтому за дополнительной информацией обращайтесь к электронной документации, где описывается весь набор параметров. В вашей власти настроить практически любой аспект внешнего вида диаграммы.

8.2.5. Линейные диаграммы: работа с несколькими наборами данных и стилями линий

Третьим распространенным типом диаграмм является линейная диаграмма. Ее создание даст нам возможность исследовать приемы отображения графиков для нескольких наборов данных.

Сейчас в нашем распоряжении имеется всего одна статистическая характеристика, поэтому давайте сфабрикуем вторую. Мы скопируем значения из отображения `postPerDay` в список `doubleData`, содержащий удвоенные значения. В листинге 8.4 приводится измененная версия файла stats.gsp с новым тегом `lineChart`.

Листинг 8.4. Создание линейной диаграммы для отображения нескольких наборов данных

```
<%
def doubleData = postsOnDay.values().collect { it * 2 }
def seriesData = [ postsOnDay.values().asList(), doubleData ]
%>

<g:lineChart type="lc"
        title="Posts Per Date (line)"
        size="${[600,200]}"
        colors="${['FF0000','0000FF']}"
        axes="x,y"
        lineStyles="${[[2,2,2],[2,8,4]]}"
        legend="${[ 'Original', 'Doubler' ]}"
        gridLines="16.6,25"
        axesLabels="${[0:postsOnDay.keySet(),
                       1:[0,doubleData.max()/2,
                       doubleData.max()]]}"
        dataType="simple"
        data="${ seriesData }" />
```

В этом примере мы передаем данные в атрибут `data` в виде списка списков, где каждый из вложенных списков представляет отдельную последовательность данных. Мы также настроили стили линий и цвета – обратите внимание, что и в том, и в другом случае передается список, в котором каждое значение соответствует своему набору данных. Для большей привлекательности мы добавили легенду с описанием наборов данных и координатную сетку, что обеспечивает более высокую удобочитаемость. Результат изображен на рис. 8.6.

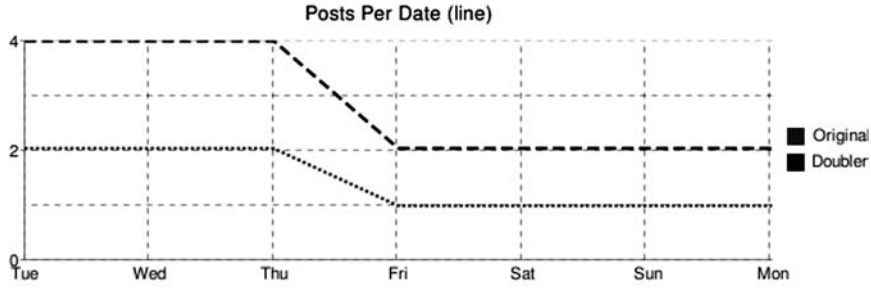


Рис. 8.6. *Линейная диаграмма, построенная на основе нескольких наборов данных*

Помимо передачи данных в виде списка списков здесь больше нет ничего, о чем вам следовало бы знать, чтобы уметь работать с несколькими наборами данных. Вы можете передавать списки цветов и стилей линий (по одному для каждой последовательности данных) и добавлять собственную легенду, передавая список строк (опять же по одной для каждой последовательности данных).

Создание координатной сетки на первых порах может вызывать некоторые трудности. Два значения (16.5, 25) определяют шаги точек на графике, в которых должны выводиться линии координатной сетки, в процентах. В нашем примере вертикальные линии должны появляться с интервалом 16.5 процентов по оси X (то есть разбивая диаграмму на семь частей), а горизонтальные линии – с интервалом 25 процентов по оси Y (разбивая диаграмму на четыре части). При наличии переменного числа категорий вам может потребоваться вычислять значения по оси X динамически.

Последнее, что вы можете пожелать настроить, – это стили линий. Тройки значений (2,2,2 и 2,8,4) определяют толщину линии, длину пунктира и величину промежутка между пунктирами. В первом примере определена линия толщиной два пикселя, с длиной пунктира два пикселя и промежутком между пунктирами два пикселя, что в результате дает замечательную пунктирную линию.

Ограниченный объем книги не позволяет нам описать все мыслимые параметры диаграмм, а Google предлагает великое их множество, но эти примеры помогут вам начать их использовать. У вас появятся свои первые диаграммы, появляющиеся в мгновение ока.

Теперь посмотрим, что можно предложить пользователям, не имеющим доступа к Интернету.

8.2.6. Диаграммы в Интранете: использование Google Chart без Google

Google Chart – замечательное расширение, но как быть, если приложение разворачивается в Интранете, где ваши пользователи не имеют вы-

хода в Интернет? Диаграммы от Google оказываются недоступными, верно? Нет, не верно.

Замечательные парни, занимающиеся JFreeChart (отличной автономной библиотекой построения диаграмм на языке Java), создали сервлет, который реализует функциональность прикладного интерфейса Google Chart. Он называется Eastwood; существует даже расширение для платформы Grails, обеспечивающее интеграцию приложений на платформе Grails с этим сервлетом:

```
grails install-plugin eastwood-chart
```

После установки расширения вы можете сообщить расширению Google Chart, чтобы оно использовало локальный сервлет `eastwood` вместо удаленной службы Google Chart. Выполнить эту настройку можно посредством статического свойства `apiString` библиотеки тегов Google Chart, определить которое можно в файле BootStrap.groovy:

```
GoogleChartTagLib.apiString = "http://localhost:8080/hubbub/chart?"
```

Обладая определенными навыками, можно даже реализовать переключение между использованием службы Google Chart и сервлета Eastwood по условию, в зависимости от того, где запускается приложение, – на внутреннем сервере или на внешнем.

На этом мы заканчиваем знакомство с механизмом построения диаграмм. Следующая особенность, которую мы добавим в приложение, – это поддержка отправки общений по электронной почте; и у нас уже имеется расширение mail, ожидающее своего часа.

8.3. Добавление поддержки электронной почты

Если ваше приложение предусматривает возможность подписки на рассылку какой-либо информации, ее реализация будет возможна не раньше, чем будет обеспечена поддержка электронной почты. Будь то приветственные письма, отсылаемые вновь зарегистрировавшимся пользователям, ответы на сообщения о забытых паролях или ежедневные обзоры новостей, поддержка электронной почты в настоящее время рассматривается как вполне стандартное требование.

Платформа Grails предлагает расширение Mail, обеспечивающее простоту отсылки сообщений по электронной почте. Вы можете отправлять электронную почту из контроллеров или служб и даже использовать сложные представления GSP для создания электронных писем в формате HTML.

Установим расширение и начнем:

```
grails install-plugin mail
```

После установки расширения необходимо добавить в настройки две строки, чтобы сообщить расширению информацию о почтовом серве-

ре и адрес отправителя по умолчанию. Изменим наш файл /grails-app/conf/Config.groovy:

```
grails.mail.host="192.168.1.9"
grails.mail.default.from="hubbub@grailsinaction.com"
```

Настраивать адрес отправителя по умолчанию необязательно. Вы можете указывать свой адрес отправителя при каждом обращении к расширению Mail, но если определить адрес отправителя по умолчанию, вам будет проще изменить его позднее, если это потребуется.

Если в разных окружениях у вас используются разные почтовые серверы, вы можете определить эти параметры настройки внутри разделов development/test/production в файле Config.groovy, чтобы в каждом окружении использовались свои значения.

Добавление параметров настройки в программном коде

Возможно, вам будет интересно узнать, как расширение Mail читает значения параметров настройки из файла Config.groovy и как вы сами можете добавлять свои собственные настройки в программном коде приложения.

Представим, что для приложения Hubbub можно определить параметр настройки, включающий использование прокси-сервера. Мы могли бы добавить в файл Config.groovy значение этого параметра, как показано ниже:

```
hubbub.proxy.enabled = true
```

Если вам потребуется получить значение этого параметра в контроллере, вы сможете это сделать, обратившись к неявно существующему объекту `grailsApplication`, например так:

```
if (grailsApplication.config.hubbub.proxy.enabled) {
    /* выполнить операции с использованием прокси-сервера */
}
```

Но объект `grailsApplication` недоступен внутри служб, поэтому чтобы получить доступ к настройкам из службы, придется немного потрудиться. В первую очередь необходимо импортировать объект `ConfigurationHolder`, а затем обратиться к программному коду статически:

```
import org.codehaus.groovy.grails.commons.ConfigurationHolder
class MyService {
    def doStuff {
        if (ConfigurationHolder.config.hubbub.proxy.enabled) {
            /* выполнить операции с использованием прокси-сервера */
        }
    }
}
```

Совет

Возможно, вас беспокоит вопрос, сможет ли расширение Mail взаимодействовать с почтовым сервером, имеющим нестандартную конфигурацию. Не волнуйтесь, расширение обладает всеми необходимыми настройками, включая возможность выбора нестандартных портов и поддержку соединений SSL. Полный перечень параметров настройки вы найдете на домашней странице расширения Mail.

Закончив настройку, можно попробовать отправить наше первое электронное письмо.

8.3.1. Отправка электронной почты из программного кода

Расширение Mail обеспечивает две основные возможности вызова почтовой службы:

- С помощью метода `sendMail()`, динамически добавляемого ко всем контроллерам
- С помощью вызова метода `mailService.sendMail()` из службы, где определено внедренное свойство `mailService`

Совет

Служба `MailService` автоматически внедряется во все контроллеры, поэтому в контроллерах нет необходимости добавлять инструкцию `def mailService`.

Начнем с механизма, основанного на контроллерах, потому что он предоставляет некоторые дополнительные возможности, недоступные в службах. В качестве первого письма отправим сообщение «добро пожаловать к нам» с подтверждением регистрации, как показано в листинге 8.5.

Листинг 8.5. Отправка электронного письма с текстом приветствия

```
def welcomeEmail = {

    if (params.email) {
        sendMail { ← Добавляется расширением Mail
            to params.email
            subject "Welcome to Hubbub!"
            body """
            Hi, ${params.email}. Great to have you on board.

            The Hubbub Team.
            """
        }
        flash.message = "Welcome aboard"
    }
    redirect(uri: "/")

}
```

Тело сообщения определяется как многострочное значение

Использование расширения Mail в программном коде заключается в передаче соответствующих значений to, subject и body в виде замыкания. Вы можете также определить поля cc и bcc и передавать списки адресов, разделенных запятыми. В листинге 8.5 мы применили способ определения многострочного текста, предоставляемый языком Groovy (“””), благодаря чему мы можем встраивать текст сообщения непосредственно в программный код контроллера.

Рассмотренный способ отлично подходит для простых случаев, но для нас нежелательно встраивать логику работы макета в контроллер. Пришло время перенести логику макета в файл GSP для облегчения сопровождения приложения в будущем.

8.3.2. Использование представления в качестве тела письма

Встраивание макета письма в класс контроллера означает, что вы лишаете себя возможности использовать преимущества редактора HTML в вашей интегрированной среде разработки. Кроме того, такой подход осложняет вовлечение дизайнера в процесс разработки. К счастью, расширение Mail позволяет доверить вывод макета письма представлению. Рассмотрим, как это делается.

Сначала создадим стандартное представление, в котором будет находиться содержимое. Мы добавим также некоторые стили CSS, поскольку они доступны, и определим тип содержимого как text/html, чтобы иметь возможность отправлять электронные письма в формате HTML. Шаблон нашего электронного письма приводится в листинге 8.6.

Листинг 8.6. Представление с шаблоном электронного письма, включающего стили CSS

```
<%@ page contentType="text/html"%>
<html>
<head>
    <title>Welcome Aboard</title>
    <style type="text/css"> ← Стили CSS для письма
        body {
            font-family: “Trebuchet MS"
        }
    </style>
</head>
<body>
    <h1>Howdy!</h1>
    <p>
        Hi, ${email}. Great to have you on board. ← Используется переменная,
    </p>                                             предоставленная контроллером
   <p>
        <strong>The Hubbub Team.</strong>
    </p>
```

```
</body>
</html>
```

Теперь, когда у нас имеется содержимое письма, нам необходимо связать его с действием контроллера. Реализуется это с помощью собственных атрибутов тега `sendMail`. В листинге 8.7 приводится измененная версия действия.

Листинг 8.7. Измененная версия действия welcomeEmail, используемого представлением

```
def welcomeEmail = {

    if (params.email) {
        sendMail {
            to params.email
            subject "Welcome to Hubbub!"
            body(view: "welcomeEmail", model: [ email: params.email ])
        }
        flash.message = "Welcome aboard"
    }
    redirect(uri: "/")

}
```

В теле тега мы передаем атрибуты `view` и `model`. Так как мы уже находимся внутри контроллера `UserController`, можно использовать относительное имя файла GSP, соответствующего представлению (без расширения .gsp). При отправке электронного письма из службы или из задания Quartz необходимо указывать полный путь к представлению (в данном примере это был бы путь `"/user/welcomeEmail"`), потому что в платформе Grails отсутствует запрос сервлета.

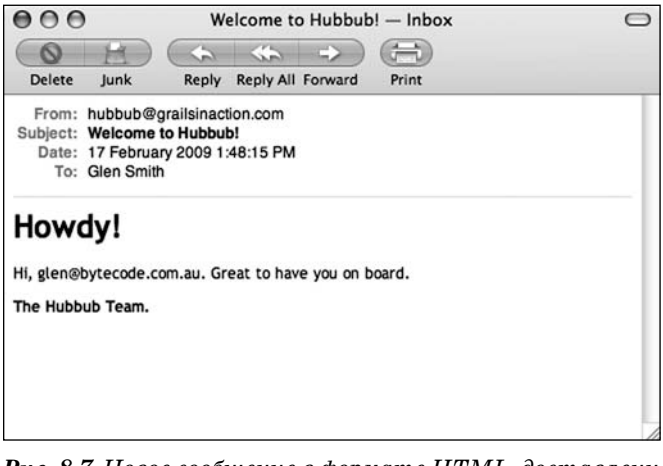


Рис. 8.7. *Новое сообщение в формате HTML, доставленное пользователю*

На рис. 8.7 изображено наше приветственное письмо в формате HTML, отправленное пользователю.

На этом мы завершаем знакомство с расширением Mail. Далее по списку следует добавление возможности полнотекстового поиска. С целью ее реализации рассмотрим расширение Searchable.

8.4. Полнотекстовый поиск: реализуем свою процедуру поиска

Ежедневно в социальные сети стекается множество пользователей, поэтому количество сообщений, оставляемых в приложении Hubbub, будет расти с космической скоростью. И у пользователей появится потребность в процедуре поиска.

В старые добрые времена поиск по сайту реализовывался с использованием запросов SQL (таких как: `post.content like '%grails%'`), но по сегодняшним меркам этого недостаточно. Применение запросов SQL с оператором `like` может быть крайне неэффективным, и оно становится все более и более затруднительным по мере увеличения числа полей, участвующих в поиске. В большинстве ситуаций пользователю требуется выполнять поиск по множеству полей, что исключает использование SQL.

К счастью, нашлись умные люди, которые реализовали возможность полнотекстовой индексации и поисковые механизмы, индексирующие содержимое базы данных и предоставляющие удобные способы поиска. Одним из наиболее популярных решений полнотекстового поиска на языке Java является Lucene, а также библиотека его высокоуровневой абстракции – Compass.

Расширение Searchable для платформы Grails служит оберткой для библиотек полнотекстового поиска и обеспечивает простой и прозрачный способ реализации поиска. Всякий раз, когда в базе данных сохраняется объект предметной области (например, вызовом метода `post.save()`), расширение добавляет объект в полнотекстовый индекс. Когда выполняется изменение или удаление экземпляра, расширение изменяет индекс соответственно. Когда возникает потребность выполнить поиск, вы просто вызываете метод `Post.search("искомые ключевые слова")` и получаете список найденных соответствий. Кроме того, с помощью полностью настраиваемого предметно-ориентированного языка (Domain-Specific Language, DSL) можно определить, какие свойства классов предметной области следует индексировать.

Начнем работу с установки расширения:

```
grails install-plugin searchable
```

После установки расширения можно начинать настройку объектов поиска. Это требует некоторого размышления.

8.4.1. Определение доступности объектов для поиска

Первый шаг к использованию расширения Searchable состоит в определении объектов для индексирования. Пользователям приложения Hubbub потребуется возможность выполнять поиск по содержимому объектов `Post` (чтобы отыскать сообщения, соответствующие критериям поиска) и объектов `User` (имеет ли учетную запись мой друг «Joe Cool»?).

Самый простой способ добавить объекты `Post` и `User` в индекс для поиска заключается в добавлении в них свойства `searchable`, как показано ниже:

```
class User {

    static searchable = true
    ...
}

class Post {

    static searchable = true
    ...
}
```

Если класс предметной области имеет свойство `searchable`, расширение будет индексировать его поля элементарного типа (строки, даты, числа и их коллекции). Позднее мы уточним, по каким полям будет производиться индексирование, а пока будем придерживаться настроек, используемых по умолчанию.

Итак, простейшая возможность поиска реализована. Если теперь мы запустим приложение Hubbub, то сможем использовать страницу, предоставляемую расширением Searchable, для поиска по нашему индексу. Откройте страницу *http://localhost:8080/hubbub/searchable* и попробуйте воспользоваться ею, как показано на рис. 8.8.

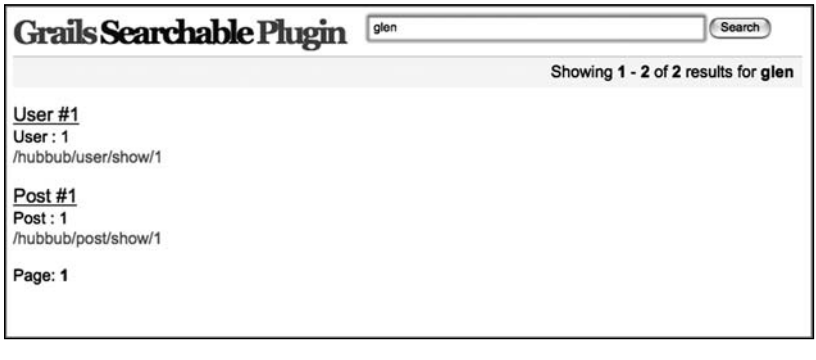


Рис. 8.8. *Интерфейс по умолчанию расширения Searchable вполне пригоден к использованию*

Не так плохо, особенно если учесть, что все это было достигнуто за пять минут, но нам осталось сделать кое-что еще. Поиск по данным был выполнен, но результаты выглядят не совсем так, как нам хотелось бы. Были обнаружены пользователь и сообщение, в которых присутствует слово «glen», но нам требуется оформить результаты в стиле приложения Hubbub, и было бы неплохо отформатировать ссылки на результаты, применив формат представления постоянных ссылок (/user/profile/glen для ссылки на профиль и /users/glen для ссылки на список сообщений). Кроме того, пользователю может потребоваться выполнить поиск только по сообщениям (Find a Post, Поиск в сообщениях) или только по пользователям (Find a Friend, Поиск друзей).

Давайте выйдем за рамки страницы поиска по умолчанию и создадим свою страницу поиска. Прежде всего создадим собственный контроллер поиска, а затем перейдем к реализации представления:

```
grails create-controller com.grailsinaction.search
```

Свою реализацию мы начнем с предоставления возможности выполнять поиск по всем сообщениям, потому что она наиболее востребована. В листинге 8.8 приводится наша первая попытка реализовать собственный контроллер поиска.

Листинг 8.8. Наш собственный контроллер поиска

```
package com.grailsinaction

class SearchController {

    def search = {
        def query = params.q

        if (!query) {
            return [:]
        }

        try {
            def searchResult = Post.search(query, params) ← ❶ Вызов нового метода поиска
            return [searchResult: searchResult]
        } catch (e) {
            return [searchError: true]
        }
    }
}
```

Как мы уже видели ранее, расширение Searchable добавляет в каждый класс предметной области динамический метод `search()` ❶. Он принимает два аргумента: строку запроса и отображение с параметрами. Обычно отображение с параметрами содержит значение максимального числа совпадений, выводимых на одной странице, смещение для постраничного просмотра и порядок сортировки. Пока мы просто будем

передавать объект `params`, а о явной передаче параметров побеспокоимся позже.

Метод `search()` возвращает отображение, содержащее информацию о результатах поиска, а также список классов предметной области, соответствующих критерию. В табл. 8.2 приводится перечень доступных значений.

Обычно в представлении выполняется обход поля `results` и отображение каждого объекта, соответствующего критерию (возможно, с выделением найденного ключевого слова). Вы можете использовать также значения `total` и `max` для отображения дополнительной информации (например, «получено 10 из 326 совпадений»).

Теперь, когда у нас имеется готовый к работе контроллер, осталось только объединить его с небольшим файлом /views/search/search.gsp, чтобы дать пользователю возможность вводить значения и просматривать результаты поиска. Мы пока не будем заниматься реализацией постраничного просмотра и начнем с самого простого, как показано в листинге 8.9.

Таблица 8.2. Возвращаемое значение searchResult позволяет получить информацию о запросе

Поле	Описание
`total`	Общее число совпадений в индексе
`results`	Список экземпляров классов предметной области, соответствующих запросу
`max`	Максимальное число возвращаемых совпадений (обычно используется для организации постраничного просмотра; значение по умолчанию: 10)
`offset`	Количество элементов в начале множества результатов, которые должны быть пропущены при формировании возвращаемого списка; используется для организации постраничного просмотра
`scores`	Список значений, определяющих степень соответствия каждого найденного совпадения (число с плавающей точкой в диапазоне от 0.0 до 1.0)

Листинг 8.9. Первая версия нашей собственной формы поиска

```
<html>
    <head>
        <title>Find A Post</title>
        <meta name="layout" content="main"/>
    </head>
    <body>

        <h1>Search</h1>
```

```
        <g:form>
            <g:textField name="q" value="${params.q}"/>
            <g:submitButton name="search" value="Search"/>
        </g:form>
        <hr/>

        <g:if test="${searchResult?.results}">
            <g:each var="result" in="${searchResult.results}">

                <div class="searchPost">
                    <div class="searchFrom">
                        From
                    <g:link controller="users"
                        action="${result.user.userId}">
                            ${result.user.userId}
                    </g:link>
                        ...
                        </div>
                        <div class="searchContent">
                            ${result.content}
                        </div>
                    </div>

            </g:each>
        </g:if>

    </body>
</html>
```

Обход результатов поиска

Создание ссылок на профили пользователей

Отображение содержимого найденных сообщений

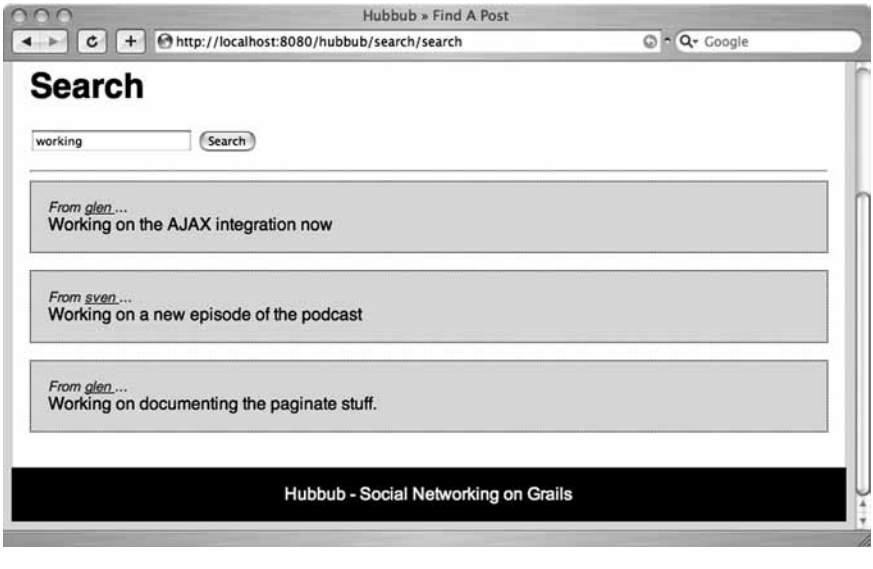


Рис. 8.9. *Наша первая форма поиска в действии*

Если поиск увенчался успехом, мы выполняем итерации по результатам и отображаем их в элементах `div`. Это дает нам возможность применять стили CSS. На рис. 8.9 (см. стр. 327) изображена наша первая версия формы поиска в действии.

Теперь страница с результатами выглядит намного лучше, но ключевые слова здесь никак не выделяются. Посмотрим, что для решения этой проблемы может предложить нам расширение Searchable.

8.4.2. Выделение найденных совпадений

При просмотре результатов поиска пользователь, вероятно, предпочел бы видеть ключевые слова выделенными. Расширение Searchable дает нам возможность реализовать такую особенность, но для этого придется немного поработать с замыканиями. В листинге 8.10 приводится измененная версия контроллера, которая выделяет обнаруженные ключевые слова.

Листинг 8.10. Измененная версия контроллера поиска, которая выделяет обнаруженные совпадения

```
package com.grailsinaction

class SearchController {

    def search = {
        def query = params.q

        if (!query) {
            return [:]
        }

        try {
            params.withHighlighter = {highlighter, index, sr -> ❶
                // инициализация списка выделенных результатов поиска
                if (!sr.highlights) { ❷
                    sr.highlights = []
                }

                // сохранить выделенный текст;
                // "content" - это свойство класса Post, участвующее
                // в поиске
                def matchedFragment = highlighter.fragment("content") ❸
                sr.highlights[index] = "..." +
                     (matchedFragment ?: "") + "..." ❹
            }

            def searchResult = Post.search(query, params)
            return [searchResult: searchResult]

        } catch (e) {
```

```
                return [searchError: true]
            }
        }
    }
```

Представленная измененная реализация поиска использует замыкание `withHighlighter` ❶, которое получает объект `highlighter` (используемый для хранения выделенного слова с окружающим его текстом), счетчик `index` (используемый для слежения за номером совпадения) и сам объект с результатами поиска.

Для каждого результата мы создаем новый объект `highlights`, если таковой еще не был создан ❷, и используем его для хранения версии результата поиска с выделением (версии, содержащей выделенное ключевое слово).

Из каждого результата мы извлекаем фрагмент содержимого поля `content` объекта `Post`, соответствующего критериям поиска ❸, и окружаем его многоточиями ❹, чтобы показать, что это именно фрагмент. Фрагмент будет содержать совпавшее слово и несколько окружающих его слов для обеспечения контекста. Совпавшее слово будет заключено расширением в теги `<b>`.

В листинге 8.11 приводится измененный фрагмент представления, который извлекает найденные фразы и отображает их в броузере.

Листинг 8.11. Измененный фрагмент представления, обрабатывающий совпадения с ключевыми словами

```
<g:each var="result" in="${searchResult.results}" status="hitNum">

    <div class="searchPost">
        <div class="searchFrom">
            From
            <g:link controller="users" action="${result.user.userId}">
                ${result.user.userId}
            </g:link>
            ...
        </div>
        <div class="searchContent">
            ${searchResult.highlights[hitNum]}
        </div>
    </div>

</g:each>
```

В этой реализации используется атрибут `status` тега `<g:each>`. Затем применяется значение `hitNum`, чтобы окружить соответствующее совпадение тегами `<b>`. На рис. 8.10 показаны результаты поиска по слову «work».

После добавления выделения ключевого слова страница поиска начинает выглядеть более полезной. Но мы не хотим, чтобы на странице по-

иска отображались тысячи совпадений, поэтому сейчас мы перейдем к реализации постраничного просмотра.

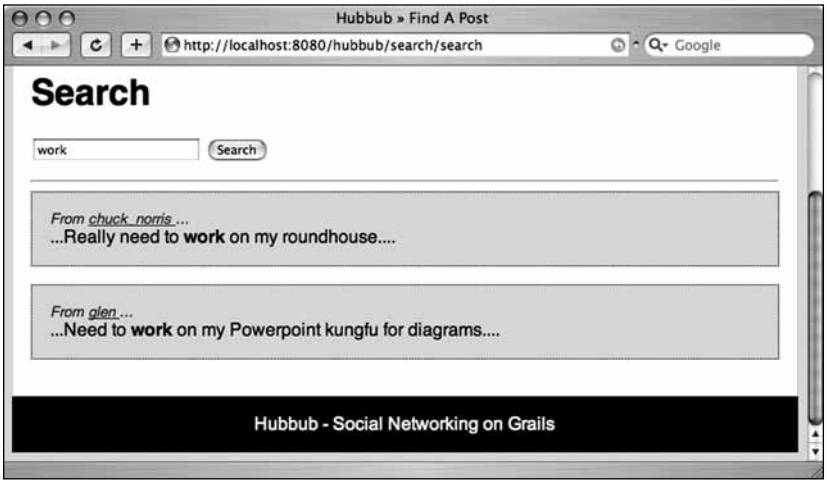


Рис. 8.10. *Выделение совпадений с ключевым словом в действии*

8.4.3. Реализация постраничного просмотра

До сих пор мы исследовали только первую страницу результатов. Мы пока не определяли значение свойства `max` при реализации поиска, и по умолчанию расширение Searchable возвращало первые десять совпадений. Пришло время дать пользователям возможность управлять количеством результатов на одной странице. Самое замечательное, что мы можем использовать тот элемент управления постраничным просмотром, с которым мы познакомились в главе 6.

Сначала добавим в страницу раскрывающийся список, с помощью которого пользователи будут выбирать количество результатов поиска, отображаемых на одной странице. Если мы присвоим этому элементу имя `max`, расширение Searchable само обнаружит и будет использовать его, то есть нам не придется вносить изменения в программный код контроллера. Ниже приводится измененная реализация формы:

```
<g:form>
    <g:textField name="q" value="${params.q}"/>
    <g:select name="max" from="${[1, 5, 10, 50]}"
            value="${params.max ?: 10}" />
    <g:submitButton name="search" value="Search"/>
</g:form>
```

Наш контроллер останется прежним, но нам требуется внести кое-какие изменения в пользовательский интерфейс, в раздел отображения результатов, чтобы использовать некоторую информацию, касающую-

ся постраничного просмотра (общее число результатов поиска, порядковый номер просматриваемой страницы и так далее).

Обработка случая, когда все результаты умещаются на одной странице, осложняет реализацию немного больше, чем хотелось бы. В листинге 8.12 приводится измененная часть файла GSP.

Листинг 8.12. Отображение выделенных ключевых слов в результатах поиска

```
Displaying hits
    <b>${searchResult.offset+1}-
        ${searchResult.offset + searchResult.max}</b> of
    <b>${searchResult.total}</b>:

    <g:set var="totalPages"
        value="${Math.ceil(searchResult.total / searchResult.max)}"/>
    <g:if test="${totalPages == 1}">
        <span class="currentStep">1</span>
    </g:if>
    <g:else>
        <g:paginate controller="search" action="search"
            params="[q: params.q]"
            total="${searchResult.total}"
            prev="&lt; previous" next="next &gt;"/>
    </g:else>
    <p/>
```

Если все результаты поиска умещаются на одной странице, нам вообще не нужно вызывать тег `paginate`. В противном случае мы должны сообщить тегу общий размер списка с результатами. Как уже говорилось в главе 6, тег сам управляет своим состоянием и заботится о создании необходимых ссылок для перехода на следующую и предыдущую страницы.

На рис. 8.11 показан элемент управления постраничным просмотром в действии.

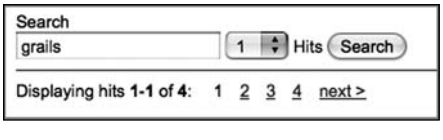


***Рис. 8.11.** Реализация постраничного просмотра результатов поиска*

На этом реализацию постраничного просмотра результатов поиска можно считать оконченной. Теперь мы знаем достаточно много, чтобы суметь реализовать большую часть особенностей функции поиска. Но мы исследовали еще не все возможности расширения Searchable: займемся некоторыми дополнительными особенностями.

8.4.4. Настройка индексирования

До сих в наших исследованиях возможностей реализации функции поиска мы полагались на правила индексирования, принятые по умолчанию: мы просто добавляли в классы предметной области свойство `searchable`. Но при наличии большого количества данных полезно контролировать, что именно сохранять в индексе, – нет смысла индексировать миллион полей с паролями, потому что вы едва ли когда-нибудь пожелаете предоставить возможность поиска по ним. Следует также проявлять осторожность и избегать индексирования данных, изменяющихся одновременно (например, поля `clickCounter`, которое имеется в каждом объекте предметной области), потому что при этом вам придется столкнуться со сложными блокировками и обработкой исключений, связанных с обновлением индекса.

Настройка места хранения индекса

При установке расширения Searchable создается новая команда, используемая для установки файла с настройками. Если вы не устанавливали собственную версию файла, будут применяться значения по умолчанию. Большинство из этих значений вполне подходят для эксплуатации, за исключением местоположения файла индекса, в качестве которого по умолчанию используется домашний каталог текущего пользователя. Это явно непригодно для действующего приложения.

Чтобы установить собственный файл с настройками, воспользуйтесь следующей командой:

```
grails install-searchable-config
```

Она создаст файл /grails-app/conf/Searchable.groovy, в котором вы сможете определить местоположение файлов индекса (и другие параметры – смотрите комментарии в созданном файле). Например, на своих действующих серверах вы можете определить специальное место для файлов индекса:

```
compassConnection = new File(
        "/var/indexes/${appName}/${grailsEnv}"
    ).absolutePath
```

Вы можете определить местоположение файлов отдельно для стадии разработки, тестирования и эксплуатации, как в файле Config.groovy.

К счастью, расширение Searchable предоставляет более полный язык DSL, обеспечивающий возможность создавать избирательные индексы, благодаря чему вы можете определять поля, которые будут доступ-

ны для поиска. Модернизируем наш класс User так, чтобы исключить пароли из индексирования:

```
class User {

    static searchable = {
        except = [‘password’]
    }
    // остальная реализация...
}
```

Наиболее часто используемыми операциями языка DSL в расширении Searchable являются except (индексировать все поля, за исключением перечисленных) и only (индексировать только перечисленные поля).

8.4.5. Подсказка по запросу: возможно, вы имели в виду «Grails»?

Одна из самых интересных особенностей сайта Google – предоставление подсказок по запросу. Не смогли вспомнить, как правильно пишется «zucchini», и выполнили поиск по слову «zukini»? Google покажет вам результаты поиска, но при этом спросит: «Возможно, вы имели в виду zucchini?» Расширение Searchable может предоставить аналогичную возможность, если сообщить ему, какие классы предметной области являются объектом проверки.

Включение данной возможности производится с помощью языка Searchable DSL, с которым мы познакомились в предыдущем разделе. Изменим определение класса предметной области Post так, чтобы сделать его доступным для поиска подсказок по запросу, что достигается с помощью параметра spellCheck:

```
static searchable = {
    spellCheck “include”
}
```

Как только класс предметной области будет объявлен как доступный для поиска подсказок по запросу, вы сможете передавать программному коду реализации поиска новый параметр suggestQuery. Ниже приводятся изменения в действии, выполняющем поиск, в действие добавлено использование нового параметра:

```
params.suggestQuery = true
def searchResult = Post.search(query, params)
```

Несмотря на незначительность изменений в контроллере, нам придется внести существенные изменения в представление, чтобы реализовать отображение подсказок. Для этого мы будем использовать внутренние классы расширения Searchable. Возможно, впоследствии эта возможность будет реализована в библиотеке тегов Searchable, но пока подсказки необходимо отображать вручную:

```
<g:if test="${searchResult?.suggestedQuery}">
<%@ page import="org.codehaus.groovy.grails.plugins.
            ➥ searchable.util.StringQueryUtils" %>
<p>Did you mean
<g:link controller="search" action="search"
    params="[q: searchResult.suggestedQuery]">
        ${StringQueryUtils.highlightTermDiffs(
            params.q.trim(), searchResult.suggestedQuery)
        }
</g:link>?
</g:if>
```

Теперь у нас имеется все, что необходимо, поэтому можно запустить приложение и проверить его работу. На рис. 8.12 изображены результаты попытки отыскать название популярной конкурирующей веб-платформы.

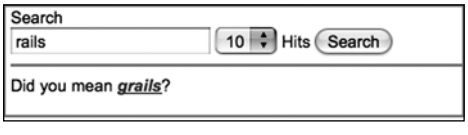


Рис. 8.12. *Возможность вывода подсказок по запросу в действии*

Параметр `suggestQuery` отлично подходит для демонстрации возможностей, но у нас осталась неисследованной еще одна базовая возможность поиска. Что если нам потребуется ограничить круг поиска сообщениями только текущего пользователя? Чтобы реализовать это, нам необходимо понять, как действуют компоненты поиска.

8.4.6. Поиск с учетом отношений

Мы уже рассмотрели, как выполняется индексация классов предметной области, находящихся в определенных отношениях друг с другом. Мы узнали, что если один класс предметной области связан отношением с другим классом (объект `User` может быть связан с множеством объектов `Posts`) и связанный класс предметной области также помечен как доступный для функции поиска, индекс будет содержать данные для этого отношения. Это замечательно, поскольку позволяет без лишних хлопот обращаться к потомку через родителя.

Но как быть, если необходимо выполнить поиск по всем объектам `Posts`, принадлежащим определенному пользователю? Нам требуется некоторый способ хранения полей объекта `User` в индексе, относящемся к данным типа `Post`. Расширение Searchable учитывает такие отношения, если определен параметр `component`.

Сначала мы продемонстрируем вам реализацию, а затем подробно разберем, что происходит за кулисами. В листинге 8.13 приводится версия класса `Post` с измененным отображением `searchable`.

Листинг 8.13. Версия класса Post с более тонкой настройкой поддержки функции поиска

```
class Post {

    static searchable = {
        user(component:true) ← Разрешает поиск объектов Post с учетом объекта User
        spellCheck “include”
    }

    static belongsTo = [ user : User ]
    // остальная реализация...
}
```

Мы сообщили расширению Searchable, что требуется сохранять все поля объекта `User` вместе с индексом каждого объекта `Post`. Когда индекс сохраняется таким образом, мы можем выполнять поиск объектов `Post`, используя ограничения, основанные на свойствах объекта `User` (например, чтобы отыскать все сообщения, где поле `userId` имеет значение «glen»).

Чтобы реализовать такую возможность, добавим в пользовательский интерфейс обработку нового параметра:

```
<g:if test="${session.user}">
    Just My Stuff:
    <g:checkBox name="justMine" value="${params.justMine}"/>
</g:if>
```

Мы также изменили контроллер поиска, добавив обработку флажка. Проще всего реализовать новую особенность можно за счет добавления ограничения к отправляемому запросу. Для этого можно, например, воспользоваться синтаксисом определения ограничений в стиле Google и добавить `+userId:glen` в конец строки запроса.

Ниже приводится измененный фрагмент нашего контроллера, который автоматически добавляет ограничение, когда отмечен флажок `justMine`:

```
if (params.justMine) {
    query += “ +userId:${session.user.userId}”
}
def searchResult = Post.search(query, params)
```

Теперь у нас имеется возможность ограничивать круг поиска одним пользователем. Использование строк прекрасно подходит для простых случаев ограничения диапазона поиска, но при попытке задействовать в ограничениях конфиденциальные данные это может стать источником проблем, связанных с безопасностью. Злонамеренный пользователь может сделать все, что угодно с отправляемым значением `queryString` строки запроса, и вам следует сделать все возможное, чтобы устранить все опасности, связанные с регулярными выражениями, созданными вручную.

Для таких случаев лучше всего использовать язык построения запросов SearchBuilder DSL, поддерживаемый расширением Searchable. Вы могли бы, например, переделать предыдущую реализацию поиска, добавив в нее спецификаторы `must()`, которые ликвидируют возможность подделки строки запроса. Допустим, так:

```
def searchResult = Post.search params, {
    must(queryString(query))
    must(term("userId", session.user.userId))
}
```

Отладка индексов с помощью Luke

Luke (*http://www.getopt.org/luke/*) – это инструмент просмотра файлов индексов, позволяющий увидеть, что в них хранится. Если у вас начинают появляться проблемы, связанные с тем, как настройки параметра `component` влияют на создание индекса, загрузите инструмент Luke (или используйте версию Web Start) и укажите ему каталог с файлами индексов.

Вкладку `Documents` (Документы) удобно использовать для отладки конструкций поиска на языке DSL – здесь вы сможете в пошаговом режиме пройти по всем элементам индекса и точно увидеть, какие ключевые слова сохраняются, как показано ниже:

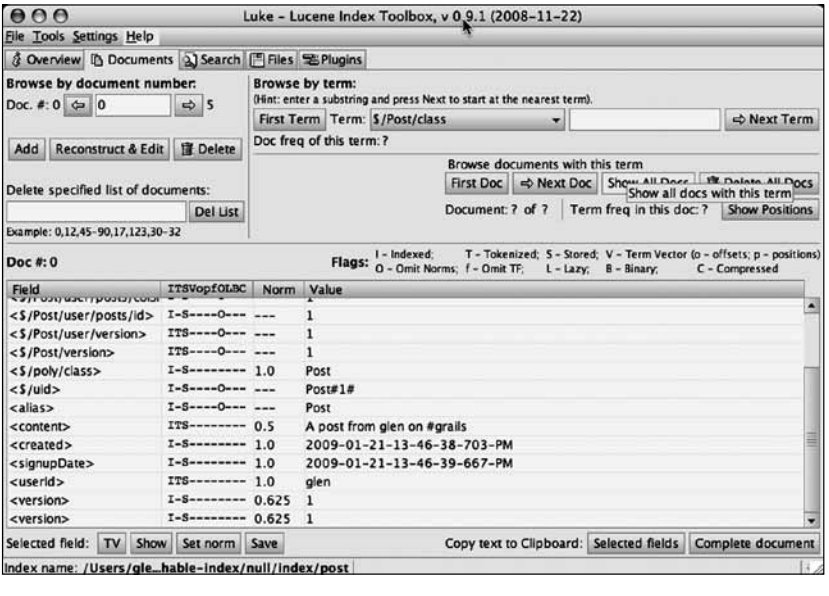


Рис. 8.13. *Внешний вид приложения Luke*

Использование конструкций языка SearchBuilder DSL обеспечивает намного более «безопасный» подход, гарантирующий применение ограничений только к окончательному запросу. Дополнительную информацию о поддержке языка SearchBuilder DSL в расширении Searchable можно найти на странице Grails wiki (*http://www.grails.org/Searchable+Plugin++Searching+-+Query+Builder*). Однако будьте внимательны, язык DSL может быть достаточно сложным в использовании, поэтому для начала вам желательно будет познакомиться с Compass API, на котором он основан.

Расширение Searchable – сложный, но мощный инструмент. Найдите время, чтобы ознакомиться с электронной документацией и посмотреть, какие параметры настройки доступны.

Теперь пришло время заняться последним расширением: GrailsUI.

8.5. Улучшение интерфейса с помощью GrailsUI

Закончив реализацию функции полнотекстового поиска, можно приступить к улучшению пользовательского интерфейса приложения. Для платформы Grails существует множество расширений, добавляющих различные эффекты в пользовательский интерфейс приложения, но наиболее популярным (и простым в использовании) является расширение GrailsUI.

GrailsUI – это обертка для YahooUI (YUI), популярной библиотеки JavaScript от компании Yahoo!; интеграция с платформой Grails через расширение обеспечивает ей большую привлекательность. Она включает в себя все, что только может потребоваться: от всплывающих подсказок до компонентов выбора даты, редакторов с богатыми возможностями форматирования, диалоговых окон, таблиц данных и даже полей с функцией автодополнения, реализованной на базе технологии Ajax, – причем все это имеет фантастическую интеграцию с платформой Grails, упрощающей их использование. Наши цели улучшения интерфейса приложения Hubbub весьма скромны: мы добавим всплывающие подсказки, возможность редактирования форматированного текста, элемент выбора даты и несколько полей с функцией автодополнения на основе технологии Ajax.

Сначала нам необходимо установить расширение:

```
grails install-plugin grails-ui
```

Теперь, когда расширение установлено и готово к работе, можно приступать к реализации новых возможностей.

8.5.1. Добавление всплывающих подсказок

Всплывающие подсказки – одна из самых простых в реализации особенностей пользовательского интерфейса, поэтому мы начнем именно с нее. Всплывающие подсказки нам нужны, чтобы дать пользователю

возможность навести указатель мыши на изображение любого пользователя и увидеть его имя.

Применение всех компонентов расширения GrailsUI реализуется по одному и тому же шаблону:

1. В разделе заголовка страницы объявляется, какие компоненты будут использоваться.
2. Создание компонента производится с помощью тега GrailsUI.
3. Для выполнения более тонких настроек применяются специальные теги YahooUI, которые передаются механизму управления отображением.

Сначала объявим в странице /post/timeline.gsp, что будет использоваться тег наведения указателя мыши. Для этого в раздел `head` страницы мы добавим специальный тег GrailsUI `gui:resources`:

```
<gui:resources components="['tooltip']"/>
```

После объявления ресурса добавим в боковую колонку элемент всплывающей подсказки в виде тега `<gui:tooltip text="текст подсказки">`. У нас уже имеется собственный тег для отображения миниатюрных изображений пользователей (`h:tinyThumbnail`), теперь мы добавим в него возможность отображения имени пользователя при наведении указателя мыши:

```
<div id="friendsThumbnails">
    <g:each var="followUser" in="${following}">
        <gui:toolTip text="${followUser.userId}">
            <h:tinyThumbnail userId="${followUser.userId}"/>
        </gui:toolTip>
    </g:each>
</div>
```

В качестве текста подсказки мы использовали имя пользователя, этого должно быть достаточно, чтобы сообщить пользователю, кто является его сторонником. На рис. 8.14 показана всплывающая подсказка в действии.

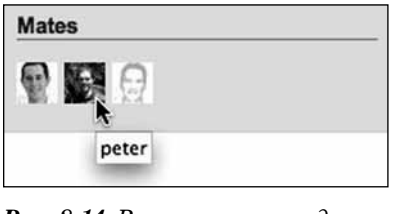


Рис. 8.14. *Всплывающие подсказки в разделе со списком сторонников*

Тег `tooltip` позволил нам получить представление о том, как действуют компоненты расширения GrailsUI. Теперь можно заглянуть чуть даль-

ше и посмотреть, как можно использовать новоприобретенное знание для реализации редактора форматированного текста на странице регистрации.

8.5.2. Реализация возможности редактирования форматированного текста

Компонент редактора форматированного текста, входящий в состав расширения GrailsUI, используется для редактирования полей, содержащих текст в формате HTML, и он будет полезен для добавления в приложение базовой функциональности, свойственной системам управления содержимым (Content Management System, CMS).

В приложении Hubbub в форме регистрации нам хотелось бы предоставить пользователям возможность добавлять свою краткую биографию в формате HTML, и мы хотели бы позволить им форматировать свой текст, как им заблагорассудится. Однако пользователи не особенно любят редактировать разметку HTML вручную, поэтому мы дадим им полноценный редактор HTML.

Как и в случае с всплывающими подсказками, сначала необходимо добавить объявление ресурса. Следующее объявление подключает редактор форматированного текста к странице /user/register.gsp:

```
<gui:resources components="['richEditor']"/>
```

После добавления определения использование редактора – это лишь вопрос добавления тега со значениями атрибутов, которые вы хотели бы иметь по умолчанию. Обычно значения извлекаются из существующего поля формы. В листинге 8.14 приводится измененное определение поля `bio`.

Листинг 8.14. Реализация поля bio с возможностью форматирования текста

```
<dt>Bio:</dt>
<dd>
    <p style="margin-top: 3em"/>
    <gui:richEditor id="bio"
                height="100px"
                width="600px"/>
    <script type="text/javascript">
        GRAILSUI.bio.defaultToolbar.titlebar =
            'Enter your bio';
    </script>
</dd>
```

← Изменение значения свойства, лежащего в основе объекта YUI

Такое определение обеспечивает пользователя полноценным редактором с богатыми возможностями. Он обладает поддержкой всех возможностей форматирования, какие только можно ожидать: поддерж-

кой жирного и курсивного шрифтов, шрифта с подчеркиванием и возможности добавления гиперссылок. При использовании тега `richEditor` ему достаточно указать значения атрибутов `id` и `value`, но мы также добавили атрибуты `height` и `width`. Доступ к дополнительным атрибутам осуществляется с помощью программного кода JavaScript; так, мы воспользовались возможностью изменять параметры настройки для помещения своего текста в заголовок редактора.

Теперь можно добавить в биографию дополнительную информацию, как показано на рис. 8.15. Этот текстовый редактор обладает всеми возможностями, какие только могут вам потребоваться. Библиотека YUI обладает большим разнообразием параметров настройки (включая возможность удалять и добавлять свои собственные кнопки, изменять внешний вид и определять обработчики событий всех видов). Мы не будем углубляться в подробности, но вы можете обратиться к обширнейшей документации по библиотеке YUI (*http://developer.yahoo.com/yui/*), чтобы познакомиться со всеми имеющимися атрибутами.

Теперь, когда у нас имеется действующий редактор форматированного текста, можно обратить внимание на способ выбора дат.

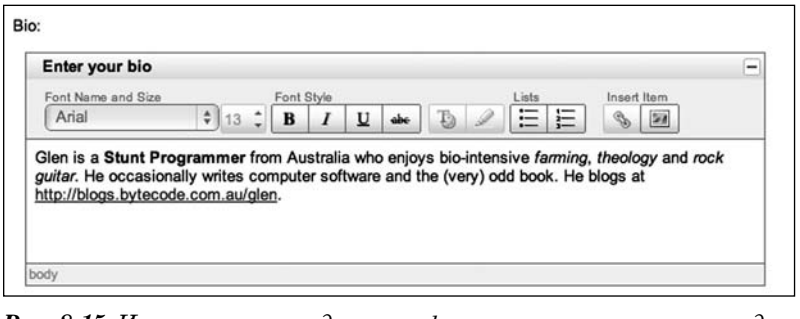


Рис. 8.15. *Использование редактора форматированного текста для ввода биографии*

8.5.3. Реализация выбора даты с помощью календаря

Стандартный элемент выбора даты, имеющийся в платформе Grails, вполне функционален, но в отношении внешней привлекательности и удобства в использовании скажем так: «он имеет некоторые недостатки». Он позволяет указывать дату и время, но при этом выбор приходится делать сразу в нескольких раскрывающихся списках, что может вызывать у пользователя неудовольствие. В большинстве случаев, когда необходимо указать дату, пользователь ожидает увидеть элемент, имеющий вид календаря, который стал популярным благодаря сайтам бронирования авиабилетов и мест в гостиницах.

Расширение GrailsUI включает в себя один такой элемент, и чтобы перейти от использования стандартного элемента выбора даты к элементу из расширения GrailsUI, достаточно изменить всего одну строку. Давайте применим этот элемент, чтобы дать пользователям приложения Hubbub возможность выбирать дату публикации сообщения – сообщение не будет отображаться, пока не будет достигнута указанная дата.

Сначала нам нужно дополнить раздел заголовка страницы /user/post.gsp и добавить в него новый элемент управления. Обратите внимание, что в теге можно указать список названий элементов управления, если требуется включить в страницу более одного элемента:

```
<gui:resources components="['tooltip','datePicker']"/>
```

После того как определение будет добавлено должным образом, можно включить в представление сам тег:

```
Schedule Post:
<gui:datePicker id='postOn' value="${new Date()}" includeTime="true"/>
```

В данном примере мы включили возможность выбора времени, благодаря этому пользователь сможет точно определять момент публикации сообщения. Элемент управления лучше выглядит, когда в нем отключена возможность выбора времени, но при прямой замене элемента выбора даты платформы Grails, когда необходимо обеспечить высокую степень детализации, такая возможность будет совсем не лишней.

Однако здесь нас подстерегает одна ловушка. Компоненты из библиотеки YUI могут быть весьма требовательны к своему оформлению. При использовании некоторых компонентов необходимо изменять стиль тела страницы, указывая один из классов CSS, входящих в состав библиотеки YUI, и компонент YUI Calendar, на основе которого создан компонент GrailsUI Calendar, является одним из них.

Мы изменим класс раздела `body` непосредственно в шаблоне `Post` (/layouts/user/post.gsp), а не в файле GSP страницы (/views/user/post.gsp). Это позволит нам избежать использования специального программного кода SiteMesh вида `${pageProperty(name: "body.style")}` для объединения атрибутов тегов макета и представления. Ниже приводится измененный фрагмент макета:

```
<body class="yui-skin-sam">
```

После добавления класса CSS календарь готов к использованию. Внешний вид представления можно видеть на рис. 8.16.

Текстовое поле все еще нуждается в дополнительном оформлении с помощью CSS, но раскрывающийся календарь выглядит просто замечательно. Теперь, когда мы добавили новый элемент выбора даты, можно перейти к последнему элементу управления – к полю ввода с функцией автодополнения на основе применения технологии Ajax.

Рис. 8.16. *Использование элемента календаря для выбора даты*

8.5.4. Введение в автодополнение

Мы собираемся закончить главу одним из самых сложных элементов управления в наборе GrailsUI – полем с функцией автодополнения. Мы хотим добавить в пользовательский интерфейс приложения Hubbub поле, дающее пользователю возможность выбирать тег из списка имеющихся тегов, отображаемого после ввода нескольких символов. Для реализации функции автодополнения придется внести изменения как в представление, так и в программный код, потому что раскрывающийся список будет заполняться информацией из внутреннего источника данных.

Сначала внесем изменения в клиентскую часть. Добавим компонент с функцией автодополнения в определения ресурсов в заголовке страницы /views/user/post.gsp:

```
<gui:resources components="['tooltip','datePicker','autoComplete']"/>
```

После этого мы можем использовать в представлении тег `autoComplete`. Функция автодополнения имеет множество параметров настройки. Рассмотрим сначала сам тег:

```
Tags: <gui:autoComplete
                id="tags"
                resultName="tags"
                labelField="name"
                idField="id"
                controller="post"
                action="autoTags"
                delimChar=" "
      />
```

Это самый настоящий вызов функции. Как можно было бы предположить, атрибуты `controller` и `action` определяют действие контроллера, возвращающее данные для заполнения раскрывающегося списка. Атрибут `delimChar` определяет символ, разделяющий элементы списка. В данном случае нам требуется возможность вводить такие теги, как

«groovy grails news», и всякий раз, когда нажимается клавиша пробела, получать другой раскрывающийся список с именами тегов.

Остальные атрибуты, `resultName`, `labelField` и `idField`, имеют отношение к данным в формате JSON, возвращаемым контроллером. Чуть ниже мы увидим, что действие контроллера возвращает данные в формате JSON, которые выглядят, как показано ниже:

```
{"tags":[{"id":1,"name":"grails"},{"id":2,"name":"groovy"}]}
```

Поле `resultName` определяет имя корневого элемента JSON. Поля `idField` и `labelField` определяют имена элементов внутри данных JSON, содержащие значения для поля с функцией автодополнения (`id` и `name`). В этом примере мы используем только поле `name`, но поле `idField` является обязательным, поскольку элемент управления создает скрытые поля (`tags` и `id` в предыдущем примере), содержащие подпись и идентификатор соответственно.

В заключение реализуем логику работы, выполняющую запрос на основе символов, введенных пользователем. Рассмотрим программный код в листинге 8.15.

Листинг 8.15. Реализация функции автодополнения имен тегов

```
import grails.converters.JSON

class PostController {

    ...
    def autoTags = {
        def queryTerm = params.query ← ❶ Извлекает искомые слова для запроса из параметров
        User user = session.user.attach()

        def matchingTags = Tag.findAllByUser(user).
            ↪ findAll { tag -> ← ❷ Выбирает все теги, соответствующие искомому слову
            tag.name.toLowerCase().startsWith(queryTerm)
        }
        def tagList = matchingTags.collect { tag -> ← ❸ Создается список из значений полей id и name тегов, соответствующих критерию поиска
            [id: tag.id, name: tag.name]
        }
        def jsonResult = [
            tags: tagList
        ]
        render jsonResult as JSON ← ❹ Возвращает результаты в формате JSON
    }
}
```

Строка, введенная пользователем, находится в свойстве `params.query` ❶, и нам остается лишь отыскать любые совпадения с тегами пользователя. Поиск должен выполняться без учета регистра символов, поэтому

сначала мы извлекаем все теги пользователя, а затем выполняем итерации по ним, отыскивая те, что начинаются с введенной строки ❷.

Получив набор совпадений, нам необходимо преобразовать их в формат, который мог бы использоваться элементом управления с функцией автодополнения. Сначала мы создаем список пар значений свойств `id`/`name` тегов ❸, а затем возвращаем полученный результат в виде объекта JSON ❹. Обратите внимание, что имя корневого элемента в объекте `jsonResult` (`tags`) соответствует значению атрибута `resultName` тега в представлении.

Закончив реализацию действия, нам не остается ничего иного, как откинуться на спинку кресла и насладиться работой поля с функцией автодополнения. Наш результат изображен на рис. 8.17.

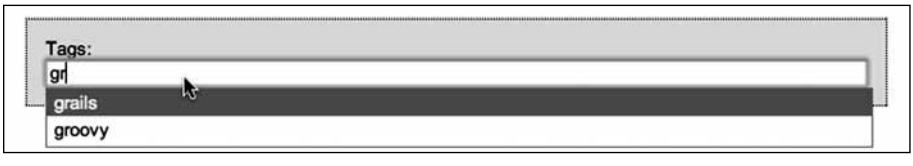


Рис. 8.17. *Поле ввода с функцией автодополнения в действии*

Реализацией возможности автодополнения мы завершаем наш тур по популярным расширениям для платформы Grails и заканчиваем улучшение внешнего вида приложения Hubbub.

8.6. В заключение

В этой главе мы обсудили большое число расширений и посмотрели, где искать популярные особенности, которые вы, вероятно, захотите добавить в свое следующее приложение.

Мы показали вам, как устанавливаются и связываются расширения; рассказали, как создавать диаграммы и графики и как добавить в приложение поддержку операций с электронной почтой. Кроме того, мы исследовали множество вариантов реализации функции поиска для добавления в приложение возможности полнотекстового поиска.

В заключение с помощью расширения GrailsUI мы придали пользовательскому интерфейсу приложения Hubbub законченный внешний вид, добавив всплывающие подсказки, редактор форматированного текста, элемент выбора даты и поле с функцией автодополнения. Нам пришлось немало потрудиться, зато удобство использования нашего приложения повысилось чрезвычайно.

Ниже приводятся несколько важных положений, которые вы должны вынести из этой главы:

- *Не усложняйте.* Отдавайте предпочтение типу данных `simple`, определяя значение атрибута `dataType` при создании диаграмм. Адреса

URL в этом случае получаются короче, что позволяет определять большие объемы данных.

- *Используйте файл для настройки расширения Mail*. Установите значение атрибута `mail.grails.default.from` в файле Config.groovy, чтобы хранить адрес отправителя в одном месте. Благодаря этому вам больше не придется использовать поле `From` при обращении к службе поддержки электронной почты.
- *Сохраняйте настройки механизма поиска в файле*. Всегда устанавливайте собственные файлы с настройками расширения Searchable, чтобы иметь возможность помещать файлы с индексами в наиболее подходящее для этого место.
- *Индексируйте данные избирательно*. Будьте внимательны при включении полей в индексы. Избегайте индексирования конфиденциальных данных и данных, изменяющихся одновременно из разных мест (из индексирования поля `clickCounter` не выйдет ничего хорошего).
- *Исследуйте возможности расширения GrailsUI*. В состав расширения GrailsUI входит масса дополнительных компонентов, которые вы можете использовать. Теперь вы знакомы с самыми основами, поэтому вам будет легко добавлять новые компоненты в свой арсенал. Найдите время, чтобы ознакомиться с документацией к расширению GrailsUI и изучить доступные особенности.

В этой главе мы много времени уделили пользовательскому интерфейсу, поэтому теперь хорошо бы уравновесить работу над интерфейсной частью и работу над внутренней реализацией. У вас будет возможность продолжить развитие навыков применения платформы Grails для разработки как интерфейсной части, так и внутренней реализации в следующей главе, где вы займетесь добавлением последовательностей действий в стиле мастера на примере создания простого интернет-магазина для приложения Hubbub.

В этой главе:

- Веб-потоки и где они используются
- Реализация простых последовательностей действий в стиле мастера
- Реализация ветвления сложных условий
- Подпотоки и упрощение логики работы
- Использование служб с сохранением информации о состоянии в области видимости потока
- Тестирование веб-потоков

9

Реализация мастеров и последовательностей с помощью веб-потоков

К настоящему моменту мы рассмотрели много вопросов. Мы подробно изучили основы контроллеров, представлений и моделей – как они взаимодействуют и как их тестировать; существенно улучшили пользовательский интерфейс нашего приложения и при этом многое узнали о разработке приложений на платформе Grails. Однако мы еще не исследовали основы реализации последовательностей действий.

Как быть, если у вас имеется процедура регистрации, занимающая несколько страниц? Или если при расчете за покупку необходимо предоставить пользователю выбор способа доставки, исходя из его места проживания? Вы сможете решить эти проблемы при текущем уровне знаний контроллеров в связке с механизмом перенаправления и скрытых переменных форм, но при этом вы создали бы свой механизм определения текущего состояния, а в зависимости от задачи сложность ее реализации может возрастать очень быстро.

Механизм веб-потоков в платформе Grails спроектирован именно для таких случаев и делает создание многостраничных последовательностей операций с необязательными шагами более понятным и управляемым. Веб-потоки предлагают простой в использовании и настраиваемый до тонкостей механизм сохранения состояний, прекрасно подходящий для реализации последовательностей действий в веб-приложениях без применения тяжеловесных механизмов управления.

Посмотрим, как с помощью веб-потоков можно за полдня создать интернет-магазин Hubbub.

9.1. Что такое веб–поток?

В терминологии платформы Grails веб-поток – это последовательность логически связанных страниц, через которые должен пройти пользователь, чтобы достичь конечной точки назначения. Пользователи могут пропускать отдельные шаги в зависимости от данных, которые ими вводятся (например, когда пользователю не требуется указывать какой-то нестандартный способ доставки). В результате выбора различных параметров они могут оказываться на совершенно разных заключительных страницах (например, на странице, сообщающей об окончании заполнения заказа, или на странице, извещающей об отсутствии товара на складе).

Рассмотрим идеальный поток, состоящий из нескольких шагов, в виде блок-схемы. В нашем примере мы будем использовать приложение Hubbub и дадим пользователям возможность выбирать некоторые товары, имеющиеся в магазине, и вводить информацию о способе доставки. Затем мы проверим номер кредитной карты и переадресуем пользователя на страницу подтверждения заказа. На рис. 9.1 изображена первая версия алгоритма работы потока для интернет-магазина Hubbub. Сначала мы реализуем основной веб-поток, а затем на протяжении главы будем добавлять некоторые дополнительные более сложные шаги и подпотоки.

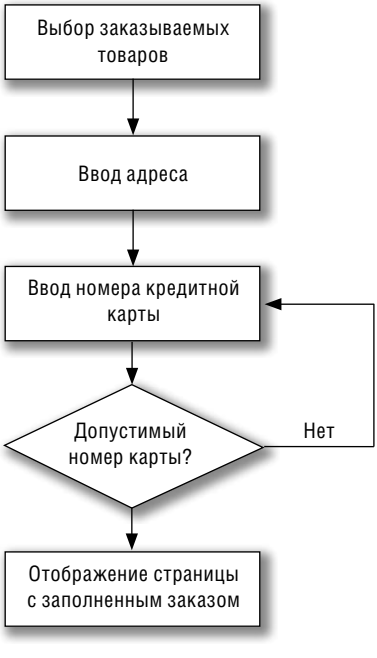


Рис. 9.1. *Блок-схема алгоритма работы основного веб-потока*

Наш первый поток достаточно прост, но даже на этом этапе видны несколько основных преимуществ использования механизма веб-потоков. Что такого дает нам веб-поток, что не может дать решение, основанное на стандартных контроллерах платформы Grails? Веб-поток обеспечивает следующие возможности:

- Запоминает, на какой стадии процедуры оформления заказа находится пользователь (вследствие чего действия кнопок `Next` (Далее) и `Previous` (Назад) зависят от контекста).
- Предоставляет простой предметно-ориентированный язык (Domain-Specific Language, DSL) для определения происходящего на каждом этапе процесса. Это позволяет легко увидеть, какой этап выполняется.
- Упрощает добавление новых шагов в процесс, освобождая от необходимости проводить реорганизацию большого количества методов действий контроллера.
- Обеспечивает широкие возможности в реализации принятия решений (например, на этапе проверки номера кредитной карты) и в выборе маршрута, на основе принятого решения.
- Вводит две новые области видимости: *потока* (*flow*) и *диалога* (*conversation*) – что позволяет создавать переменные, которые будут доступны только внутри потока и будут автоматически удаляться по завершении работы потока.

Веб-потоки в состоянии обеспечить весьма широкие возможности для любой части приложения, где необходимо реализовать взаимодействие, состоящее из нескольких шагов.

Но прежде чем приступать к использованию веб-потоков, вы должны понять, что они подходят не для всех типов приложений; основное их предназначение – реализация многоэтапных операций в стиле мастеров. Они абсолютно применимы для разбиения формы, содержащей 100 полей, на последовательность из нескольких страниц, или для реализации замысловатых алгоритмов, когда, например, требуется проверить номер кредитной карты, получить от пользователя адрес и способ доставки, а затем предложить ему страницу подтверждения заказа. Но не пытайтесь ограничить свой новый сайт в стиле Flickr взаимодействиями в стиле веб-потоков – вы потратите больше времени на борьбу с платформой, чем на решение насущных проблем.

9.1.1. Наш первый поток: мастер приема заказа

Теперь, когда у вас есть некоторое представление о ситуациях, в которых имеет и не имеет смысла применять веб-потоки, можно переходить к созданию нашего простого потока оформления заказа и изучению того, как взаимодействуют его составляющие.

Вы будете приятно удивлены, узнав, что все определения веб-потоков производятся в контексте стандартного контроллера. Начнем с созда-

ния в приложении Hubbub контроллера интернет-магазина, где пользователи смогут приобретать товары, имеющие отношение к сообществу Hubbub:

```
grails create-controller com.grailsinaction.shop
```

Создав контроллер `shop`, можно приступать к определению нашего первого потока. Для определения всех действий, изображенных на рис. 9.1, мы будем использовать язык Webflow DSL. Назовем наш поток `orderFlow`; в соответствии с соглашениями имена действий должны оканчиваться на «Flow» (поток). Это позволит платформе Grails автоматически определять действия, содержащие определения на языке Webflow DSL. Кроме того, потоки получают свой собственный адрес, то есть наш поток `orderFlow` будет иметь адрес URL /hubbub/shop/order.

Теперь можно определить поток. В листинге 9.1 определяется родительский элемент потока.

Листинг 9.1. Определение пустого потока, куда будут добавляться шаги процедуры

```
package com.grailsinaction

class ShopController {

    def index = {
        redirect(action: "order")
    }

    def orderFlow = {
    }

}
```

Это всего лишь заготовка. Обратите внимание, что мы включили в контроллер действие `index`. Как вы помните, если в контроллере имеется только одно действие, оно становится действием по умолчанию, но из-за специальной природы адресов URL данное правило к веб-потокам не применяется. Если вы хотите получить очевидный адрес URL (перенаправить адрес /shop на адрес /shop/order), вы должны взять дело в свои руки.

Обратите также внимание на то, что хотя мы присвоили потоку имя `orderFlow`, внутри контроллера мы должны обращаться к действию по имени без окончания «Flow»; именно поэтому перенаправление в действии `index()` выполняется по имени `"order"`, а не `"orderFlow"`.

В листинге 9.1 приводится определение потока, но он пока не выполняет никаких операций. Начинать определение потоков проще всего с создания скелета всего потока; затем можно наполнять его логикой. Без лишней суеты мы набросаем скелет определения потока, после чего на каждом шаге будем понемногу наращивать на нем мясо. В листин-

ге 9.2 приводится скелет определения потока, готовый к наполнению его некоторой логикой.

Листинг 9.2. Скелет определения потока

```
def orderFlow = {

    displayProducts {
        on("next") { ← Обработка кнопки Next (Далее) в форме
            // сохранить информацию о выбранных товарах
        }.to("enterAddress") ← Переход в состояние enterAddress
        on("cancel").to("finish")
    }

    enterAddress {
        on("next") {
            // сохранить адрес
        }.to("enterPayment")
        on("previous").to("displayProducts")
    }

    enterPayment {
        on("next") {
            // сохранить информацию о способе оплаты
        }.to("validateCard")
        on("previous").to("enterAddress")
    }

    validateCard {
        action { ← Действует, как точка ветвления
            // выполнить некоторые проверки
            if (params.validCard) {
                log.debug "Valid Card!!"
                valid()
            } else {
                log.debug "Invalid Card!!"
                invalid()
            }
        }
        on("valid").to("orderComplete")   ← Ветви алгоритма в зависимости от
        on("invalid").to("enterPayment")    значения, возвращаемого action
    }

    orderComplete {
        // display order
        on("finished").to("finish")
    }
    finish { ← Конечная точка потока
        redirect(controller:"homePage", action: "index")
    }

}
```

Хотя это всего лишь скелет процесса, здесь присутствует логика выполнения в стиле мастеров. Вы можете видеть, как каждое состояние веб-потока соответствует определенному этапу на блок-схеме (рис. 9.1). После каждого шага `on()` веб-поток выполняет переход к новому этапу `to()`. Чуть ниже мы займемся подробностями реализации, а пока посмотрите, можете ли вы уловить суть работы потока в целом, и насколько он соответствует нашей блок-схеме.

Теперь, когда вы имеете некоторое представление о том, как выполняется реализация потоков, можно поближе познакомиться с тем, как осуществляется переход от одного этапа к другому. Начнем с исследования определений `on().to()`.

Параметр execution: кнопка «Назад» и проблемы с постоянными ссылками

Начав работать с веб-потоками, вы заметите, как постоянно изменяется добавляемый в конец адреса URL параметр, определяющий состояние веб-потока (`execution=e1s1` со значением, которое изменяется при каждом изменении состояния).

Предположим, что пользователь создал закладку на страницу где-то в середине потока, или создал ссылку на нее где-то на внешнем ресурсе, или щелкнул на кнопке Назад броузера после того, как поток завершился, и попытался повторно отправить форму из середины потока. В таких случаях платформа Grails перехватит запрос к завершившемуся потоку или к потоку, время действия которого истекло, и перенаправит пользователя на первый этап потока. Если вы заглянете в журналируемые сообщения веб-потока (врезка «Отладка необычных ошибок веб-потоков»), вы увидите, как платформа перехватывает их:

```
[6431198] servlet.FlowHandlerAdapter Restarting a new execution of
          previously expired/ended flow 'order'
```

Как правило, возврат в начало потока – это самый лучший способ обработки таких ошибок. Если вам необходимо реализовать собственную логику обработки ошибок обращения к завершившимся потокам, проще всего будет это сделать с помощью фильтра, перехватывающего ошибку.

9.1.2. Анатомия состояния потока

Каждый шаг потока называется *состоянием*. Первое состояние, в котором оказывается веб-поток, называется *начальным состоянием* (*start state*), каковым в нашем случае является `displayProducts`.

При открытии страницы /shop/order платформа вызовет `displayProducts` для продолжения работы. Рассмотрим определение этого состояния:

```
displayProducts {
    on("next") {
        // сохранить информацию о выбранных товарах
    }.to("enterAddress")
    on("cancel").to("finish")
}
```

Как состояние `displayProducts` вызывает правила `"next"` и `"cancel"`? Ответ на этот вопрос лежит в плоскости соглашений о работе представления. В случае с состоянием `displayProducts` платформа Grails сначала отобразит страницу представления в файле /views/shop/order/displayProduct.gsp, а затем будет ждать, пока пользователь отправит форму. Поток продолжит свою работу в зависимости от того, на какой кнопке будет выполнен щелчок.

В листинге 9.3 приводится определение простой страницы `displayProduct`, с которой мы начнем. (Не забывайте, что она должна храниться в подкаталоге «order» в каталоге /views/shop, то есть мы находимся на один уровень ниже, чем обычно.)

Листинг 9.3. Выполнение шагов потока из представления

```
<html>
    <head>
        <title>Shop for Official Merchandise</title>
        <meta content="main" name="layout"/>
    </head>
    <body>

        Display Products Details Here
        <g:form action="order"> ← ❶ Имя формы соответствует имени потока
            <g:submitButton name="next" value="Next"/> ← ❷ Имя кнопки
                                                соответствует имени перехода
            <g:submitButton name="cancel" value="Finished Shopping"/>
        </g:form>

    </body>
</html>
```

Выглядит, как обычная форма GSP платформы Grails, в которой присутствует всего несколько элементов, имеющих отношение к веб-потоку. Во-первых, обратите внимание, что в качестве имени формы выбрано имя `"order"`, а не `orderFlow` ❶. Представления в потоке всегда указывают на одно и то же действие, а механизм веб-потока определяет, в какое состояние следует перейти, опираясь на действия пользователя (с помощью параметра `execution`, который добавляется веб-потоком к каждому адресу URL, о чем говорилось во врезке выше, где обсуждался параметр `execution`).

Во-вторых, обратите внимание на особую важность имен кнопок ❷. Имя кнопки определяет соответствующее действие в реализации, поэтому будьте внимательны при их написании.

Заключительная часть действия – предложение .to(), которое сообщает веб-потоку, к какому состоянию следует перейти далее. Когда пользователь щелкает на кнопке Cancel (Отменить), весь механизм приходит в движение. Форма отправляется на сервер, веб-поток следует за предложением on() и оказывается в конечной точке, где производится переход на домашнюю страницу. В листинге 9.4 демонстрируется реализация перехода в потоке.

Листинг 9.4. Предложение on() отображает кнопку Cancel (Отменить) на следующее логическое действие

```
displayProducts {
    on("next") {
        // сохранить информацию о выбранных товарах
    }.to("enterAddress")
    on("cancel").to("finish")
}

finish { ← Завершает работу потока, так как здесь опущено предложение to()
    redirect(controller:"homePage", action: "index")
}
```

Обратите внимание на отсутствие предложений .to() у состояния finish. Любое состояние, в котором отсутствует предложение .to(), называется *конечным состоянием* (*end state*). Такое состояние является специальным случаем состояний, поскольку после перехода в конечное состояние уничтожаются все данные о состоянии потока (например, информация об этапе, на котором находился пользователь, и все данные в области видимости потока, о которой мы поговорим ниже).

В нарушение соглашений: отображение собственных представлений

Имена представлений веб-потоков следуют стандартным соглашениям (имя файла представления соответствует названию состояния), но вы легко можете преодолеть это ограничение в случае необходимости, воспользовавшись методом render(). Например, состояние displayProducts в листинге 9.4 могло бы вызывать представление с другим именем, как показано ниже:

```
displayProducts {
    render(view: "standardProducts")
    on("next") {
        // capture products
    }.to("enterAddress")
    on("cancel").to("finish")
}
```

9.2. Работа с веб–потоками

Теперь, когда вы немного представляете, как веб-потоки обрабатывают состояния и выбирают маршруты, можно попробовать применить эти знания для реализации нашего потока оформления заказа. В данном разделе вы изучите основы реализации потоков:

- Как сохранять объекты в области видимости потока
- Как выполнять привязку данных и обработку ошибок в потоках
- Как программно реализовывать состояния с принятием решения и как программно выбирать маршрут дальнейшего движения к различным состояниям

Начнем с исследования области видимости потока – новой области видимости, предоставляемой платформой Grails для хранения объектов, которые должны быть доступны только на протяжении времени существования веб-потока.

9.2.1. Область видимости потока: лучше, чем область видимости кадра, и дешевле, чем область видимости сеанса

Веб-потоки платформы Grails привносят с собой новую область видимости: *область видимости потока* (*flow scope*). Данные, находящиеся в этой области видимости, продолжают существовать, пока существует сам поток (то есть пока пользователь не попадет в конечное состояние, подобное состоянию `finish` в предыдущем примере). Область видимости потока обеспечивает эффективный способ хранения и использования данных на разных этапах потока, и она отлично подходит для хранения объектов, создаваемых внутри потока. В примере с приложением Hubbub мы могли бы сохранять товары, отобранные пользователем на первом этапе потока, и использовать их на этапе подтверждения заказа и выставления счета.

Находясь в одном из состояний потока, вы можете обращаться к области видимости потока явно, как и к другим областям видимости. Например:

```
displayProducts {
    on("next") {
        flow.order = new Order(params)
    }.to("enterAddress")
    on("cancel").to("finish")
}
```

← Явно сохраняет объект в области видимости потока

Это явный способ обращения к области видимости потока, но существует еще более удобная возможность. Когда объекты возвращаются состоянием потока (в виде отображения, как это обычно принято в контрол-

лерах), они автоматически сохраняются в области видимости потока. Например, можно написать такую реализацию:

```
displayProducts {
    on("next") {
        def orderDetails = new Order(params)
        [ order: orderDetails, orderStartDate: new Date() ]
    }.to("enterAddress")
    on("cancel").to("finish")
}
```

Сохраняет объект в области видимости потока

Объекты `orderDetails` и `orderStartDate` автоматически будут сохранены в области видимости потока. Но эта область видимости имеет собственные ограничения, и они весьма существенны. Самое серьезное ограничение состоит в том, что все объекты, сохраняемые в области видимости потока, должны поддерживать интерфейс `Serializable`.

Как быть с интерфейсом Serializable?

Требование поддержки интерфейса `Serializable` всеми объектами, сохраняемыми в области видимости потока, означает, что объекты простых типов, таких как `Strings` и `Date`, без исключений могут сохраняться в области видимости потока, но если потребуется сохранять объекты предметной области, они должны реализовать интерфейс `Serializable`. Хуже того: чтобы удовлетворить требования, предъявляемые интерфейсом `Serializable`, все классы предметной области, связанные отношениями с данным классом, также должны реализовать интерфейс `Serializable`. Это может повлечь необходимость внесения массы изменений!

Если прежде вам не приходилось сталкиваться с интерфейсом `Serializable` в языке Java, заметим, что вам следует начать с изучения документации Javadoc для `java.io.Serializable`. В большинстве случаев использования веб-потоков вам достаточно будет просто добавить `implements Serializable` к определениям своих классов предметной области. Однако с интерфейсом `Serializable` связаны некоторые сложные, трудно обнаруживаемые ситуации, которые могут приводить в замешательство начинающих разработчиков приложений на платформе Grails, – особенно когда появляется необходимость сохранять объекты, поддерживающие интерфейс `Serializable`, в коллекциях.

Многие элементы, которые вам придется сохранять в области видимости потока, являются объектами, созданными ранее, и вам едва ли захочется возвращаться обратно и отмечать все классы предметной области как поддерживающие интерфейс `Serializable` только для того,

чтобы получить возможность хранить их в области видимости потока. К счастью, управляющие объекты, которые можно объявить с поддержкой интерфейса Serializable, предлагают неплохое решение этой проблемы.

Продолжим наше исследование веб-потоков, чуть больше углубившись в практические приемы проверки данных в веб-потоках.

9.2.2. Стратегии привязки и проверки данных

Мы обсудили некоторые сложности сохранения объектов данных в области видимости потока, но веб-потоки предоставляют нам ряд других возможностей обработки данных форм. Одной из самых замечательных является поддержка потоками управляющих объектов. Мы рассматривали управляющие объекты в главе 5, когда изучали способы обработки форм, – они представляют собой отличное решение текущей проблемы, позволяющее избежать необходимости реализовать интерфейс Serializable в классах предметной области.

Допустим, что наше представление displayProducts дает возможность указать количество приобретаемых футболок и бейсболок. Реализуется это просто, как показано ниже:

```
<g:form action="order">
    Shirts: <g:textField name="numShirts" value="0"/>
    Hats: <g:textField name="numHats" value="0"/>
    <g:submitButton name="next" value="Next"/>
    <g:submitButton name="cancel" value="Finished Shopping"/>
</g:form>
```

Нам требуется гарантировать, что пользователь не сможет заказать более 10 футболок или 10 бейсболок. Напоминает классическую проблему проверки ограничений, правда? В листинге 9.5 приводится пример поддерживающего интерфейс Serializable объекта, который определяет диапазон количества заказываемых предметов от 0 до 10; сюда же мы можем добавить любые другие проверки, соответствующие решаемой задаче.

Листинг 9.5. Управляющие объекты, используемые в потоке, должны поддерживать интерфейс Serializable

```
class OrderDetailsCommand implements Serializable {

    int numShirts
    int numHats

    boolean isOrderBlank() {
        numShirts == 0 && numHats == 0
    }

    static constraints = {
```

```
            numShirts(range: 0..10)
            numHats(range: 0..10)
        }
    }
```

В листинге 9.6 показано, как можно привязать управляющий объект к состоянию `displayProducts`.

Листинг 9.6. Отображение управляющего объекта в этап потока

```
displayProducts {
    on("next") { OrderDetailsCommand odc ->          ❶ Привязка данных формы к объекту
        if (odc.hasErrors() || odc.isOrderBlank()) {  ❷ Проверка правильности
            flow.orderDetails = odc                   ❸ Сохранение недопустимого
                                                        заказа в области видимости потока
            return error()                            ❹ Останавливает продвижение пользователя по потоку
        }
        [ orderDetails: odc, orderStartDate: new Date() ]
    }.to("enterAddress")
    on("cancel").to("finish")
}
```

Как и обычные действия (вне веб-потока), состояния веб-потока, выполняющие привязку к управляющему объекту, должны определить этот объект в первом аргументе замыкания ❶. Механизм привязки данных в платформе Grails передает параметры запроса в управляющий объект и устанавливает свойства объекта `errors`, если были обнаружены несоответствия установленным ограничениям ❷.

Единственное, с чем вы еще ранее не встречались, – это вызов метода `error()` ❹. Возврат значения методом `error()` предписывает веб-потоку вернуться к предыдущему представлению, чтобы пользователь мог исправить обнаруженные ошибки, но тогда пользователь должен иметь доступ к информации об ошибках, чтобы исправить их. Для этого мы сохраняем управляющий объект в области видимости потока ❸, в итоге мы получаем возможность использовать коллекцию ошибок для отображения на странице представления.

Все это приобретет для вас больше смысла, когда вы увидите эту реализацию в действии. В листинге 9.7 приводится измененная версия представления `displayProducts`, где на этот раз предусмотрена обработка ошибок.

Листинг 9.7. Измененная версия представления displayProducts с обработкой ошибок

```
<g:hasErrors bean="${orderDetails}">
    <div class="errors">
        <g:renderErrors bean="${orderDetails}"/>
    </div>
</g:hasErrors>
```

```
<g:form action="order">
    Shirts: <g:textField name="numShirts"
                  value="${orderDetails?.numShirts}"/>
    Hats: <g:textField name="numHats" value="${orderDetails?.numHats}"/>
    <g:submitButton name="next" value="Next"/>
    <g:submitButton name="cancel" value="Finished Shopping"/>
</g:form>
```

Обратите внимание, что теперь мы отображаем все ошибки, которые присутствуют в объекте `orderDetails`, сохраненном в области видимости потока. Кроме того, мы предусмотрели заполнение полей значениями свойств объекта, чтобы показать пользователю ошибочные значения. На рис. 9.2 показаны сообщения об ошибках в действии.

Это именно то, что нам требовалось. Вы могли заметить, что кроме всего прочего была выполнена настройка сообщений об ошибках. Для этого была использована поддержка механизмом интернационализации (i18n) пакетов сообщений (представленная в разделе 4.1.2), которая доступна и в веб-потоках. Для создания сообщений, представленных на рис. 9.2, мы добавили в файл /grails-app/i18n/messages.properties следующие строки:

```
OrderDetailsCommand.numShirts.range.toosmall=
➥ You may not order less than {3} shirts.
OrderDetailsCommand.numShirts.range.toobig=
➥ You may not order more than {4} shirts.
OrderDetailsCommand.numHats.range.toosmall=
➥ You may not order less than {3} hats.
OrderDetailsCommand.numHats.range.toobig=
➥ You may not order more than {4} hats.
```

Примечание

Чтобы освежить в памяти знания о поддержке механизмом интернационализации (i18n) пакетов сообщений, вернитесь к главе 4.

Теперь, когда мы реализовали в нашем потоке проверку накладываемых ограничений и сохранение управляющих объектов в области видимости потока, можно поближе познакомиться с состояниями потока, которые вообще не имеют пользовательского интерфейса: с *состояниями действий*.

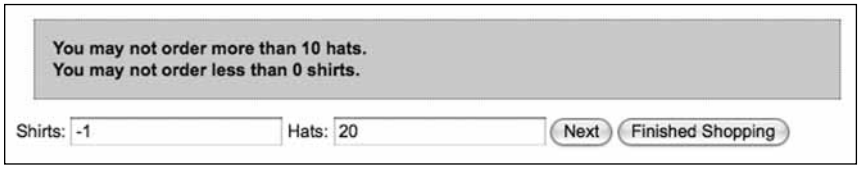


***Рис. 9.2.** Наша форма отображает сообщения об ошибках и заполняет поля ошибочными данными*

9.2.3. Принятие решений программным способом в состояниях действий

До сих пор все наши решения управлялись пользовательским интерфейсом: пользователь щелкал на кнопке Next (Далее) или Previous (Назад), или форма содержала недопустимые данные. Но мы пока не рассматривали возможность программного управления направлением движения потока. Нам может потребоваться проверить номер кредитной карты или предложить специальные способы доставки товаров для пользователей, проживающих на большом удалении.

У подобных этапов потока нет собственного пользовательского интерфейса, зато есть возможность изменять направление движения потока на основе результатов внутренней обработки данных. В контексте веб-потоков такие этапы называют *состояниями действий* (*action states*), чтобы отличать их от более типичных *состояний с переходами* (*transition states*), с которыми мы имели дело до сих пор.

Возьмем в качестве примера реализацию проверки кредитной карты. После того как пользователь укажет информацию о своей кредитной карте, нам необходимо проверить ее с помощью одной из служб электронных платежей. Если пользователь указал правильный номер кредитной карты и баланс позволяет выполнить покупку, поток может продолжать работу, но если кредитная карта не проходит проверку, нам следует вернуть пользователю форму ввода номера кредитной карты. Реализовать подобную логику работы потока можно с помощью состояния действия.

Реализуем этап простой проверки кредитной карты, который будет терпеть неудачу, если пользователь вводит информацию утром. Нам требуется получить результат, подобный показанному на рис. 9.3.

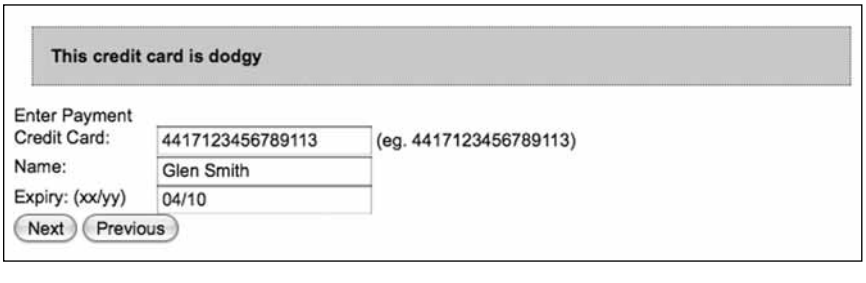


Рис. 9.3. *Потоки поддерживают возможность переопределения сообщений об ошибках с помощью пакетов сообщений*

Форма, как приведенная ниже, проходит обычную последовательность представление-состояние в процессе проверки значений полей, что может быть реализовано так:

```
enterPayment {
    on("next") { PaymentCommand pc ->
        flow.pc = pc
        if (pc.hasErrors()) {
            return error()
        }
    }.to("validateCard")
    on("previous").to("enterAddress")
}
```

Здесь выполняются основные проверки, и если все в порядке, совершается переход к новому состоянию действия: `validateCard`.

Определение состояния действия начинается с ключевого слова `action`, благодаря чему платформа Grails может определить, что для данного состояния не требуется отображать представление. В остальном тело состояния потока похоже на любое другое состояние. Обычно состояния действий определяют и возвращают собственные имена переходов, что делает тело потока более удобочитаемым. В листинге 9.8 мы возвращаем значение `invalid()` или `valid()`, в зависимости от результатов проверки кредитной карты.

Листинг 9.8. Реализация состояния действия, выполняющего проверку кредитной карты

```
validateCard {
    action {
        def validCard = new Date().hours > 11 // Требуется время после полудня
        if (validCard) {
            valid()
        } else {
            flow.pc.errors.rejectValue("cardNumber",
                "card.failed.validation",
                "This credit card is dodgy")
            invalid()
        }
    }
    on("valid").to("orderComplete")
    on("invalid").to("enterPayment")

}
```

← Отвергает кредитную карту с собственным кодом ошибки

В листинге 9.8, когда проверка терпит неудачу, мы добавляем собственный объект ошибки в управляющий объект `PaymentCommand` (`pc`), находящийся в области видимости потока. Это позволит представлениям использовать тег `hasErrors`, предлагаемый платформой Grails, для отображения текста нашего сообщения.

Обсуждение выше достаточно полно описывает, что делают состояния действий и как их использовать. Состояния действий прекрасно подходят для представления сложной логики принятия решения без вмеше-

ния в работу действия с переходом, а также для организации ветвления и запуска подпотоков, с которыми мы познакомимся вскоре.

Теперь вы знакомы со всеми основными особенностями веб-потоков, которые вы, вероятно, захотите применить при создании своего следующего приложения на платформе Grails. В следующем разделе мы познакомим вас с некоторыми дополнительными концепциями веб-потоков – на практике они используются не так часто, но их удобно использовать в некоторых ситуациях.

9.3. Дополнительные особенности веб-потоков

Теперь у вас должно иметься четкое представление об основах реализации прикладных потоков, однако веб-потоки обладают еще целым рядом дополнительных особенностей, которые могут пригодиться в определенных ситуациях. В этом разделе мы исследуем некоторые из них, включая подпотоки и диалоги. Начнем со знакомства со службами в области видимости потока.

9.3.1. Службы области видимости потока

К настоящему моменту мы изучили, как сохранять в области видимости потока объекты, обладающие поддержкой интерфейса `Serializable`, для того чтобы их можно было передавать из одного состояния потока в другое. Но мы пока не знакомы с возможностью использования служб области видимости потока, которые можно рассматривать как легковесный эквивалент компонента сеанса JavaBean с сохранением состояния.

Службы области видимости потока удобно использовать, когда необходимо получить объект службы, который будет использоваться только в пределах потока, и управлять данными, имеющими отношение только к текущему потоку. Хорошим примером такой службы может служить функция проверки кредитной карты. В начале потока мы выполняем проверку кредитной карты, а затем мы могли бы получить доступ к коду ошибки или к проверенному номеру карты, сохраненному в пределах службы.

Приведем в порядок наш предыдущий пример и поэкспериментируем с созданием службы области видимости потока, выполняющей проверку кредитной карты. Службы области видимости потока похожи на обычные службы и создаются стандартным способом:

```
grails create-service com.grailsinaction.CreditCard
```

Все точно так же, как и при создании любой другой службы. Но в отличие от случая со стандартной службой, необходимо явно указать область видимости службы с помощью атрибута `scope`:

```
class CreditCardService implements Serializable {
```

```
    static scope = "flow"
    static transactional = false
}
```

Как и все объекты, сохраняемые в области видимости потока, данные службы должны поддерживать интерфейс `Serializable`. Когда служба определяется таким способом, к контроллерам, в которых используется служба, применяются стандартные правила внедрения служб (вернитесь к главе 4, если вам требуется освежить в памяти информацию о службах).

Следует отметить, что хотя объявление службы находится в контроллере (а не в определении потока на языке DSL), каждый новый экземпляр потока будет получать новый экземпляр внедренной службы – новый объект службы создается для каждого потока, а не для контроллера.

Ниже приводится измененная версия контроллера `ShopController` с внедренной службой:

```
class ShopController {

    def creditCardService

    // ... остальные действия контроллера
}
```

После того как служба будет объявлена в контроллере, ее использование внутри любого состояния потока будет гарантировать, что обращения будут выполняться к уникальной службе, созданной специально для этого потока. Следовательно, вы можете применять службы в состояниях действий. В листинге 9.9 приводится переделанная реализация действия `validateCard`, в которой теперь используется служба области видимости потока.

Листинг 9.9. Вызов службы с сохранением состояния для проверки кредитной карты

```
validateCard {
    action {
        def validCard = creditCardService.checkCard(
            flow.pc.cardNumber,                          ← Вызов службы области
            flow.pc.name,                                  видимости потока
            flow.pc.expiry)
        if (validCard) {
            valid()
        } else {
            flow.pc.errors.rejectValue("cardNumber",
                "card.failed.validation",
                "This credit card is dodgy")
            invalid()
        }
    }
```

```
        on("valid").to("orderComplete")
        on("invalid").to("enterPayment")
    }
```

Обратите внимание, что мы просто вызываем службу `creditCardService`, а о сохранении состояния службы в области видимости потока заботится сама платформа Grails. Не требуется никакого волшебства с `flow.creditCardService`.

Самое большое преимущество служб области видимости потока состоит в том, что в последующих состояниях потока вы можете благополучно обращаться к данным, сохраненным службой. Такая свобода означает, что мы можем реализовать службу `CreditCardService` как компонент с сохранением состояния и пребывать в уверенности, что у нас никогда не возникнут конфликты между состояниями веб-потока, использующими эту службу. В листинге 9.10 приводится базовая реализация службы с сохранением состояния.

Листинг 9.10. Абстрактная проверка кредитной карты в службе с сохранением состояния

```
package com.grailsinaction

class CreditCardService implements Serializable {  <-- Гарантирует реализацию службой интерфейса Serializable

    static scope = "flow"  <-- Отмечает как службу области видимости потока
    static transactional = false

    boolean cardChecked = false
    String cardNumber
    String name
    String expiry
    Date checkedAt

    boolean checkCard(String aNumber, String aName, String anExpiry) {
        (cardNumber, name, expiry) = [ aNumber, aName, anExpiry ]
        log.debug "Validating ${cardNumber} for ${name} with expiry ${expiry}"

        // вызвать удаленную службу здесь
        checkedAt = new Date()
        cardChecked = true
        println "Card is ${cardChecked}"
        return cardChecked

    }
}
```

Как объяснялось в главе 4, службы без сохранения информации о состоянии создаются обычно потому, что по умолчанию они используются единственными экземплярами объектов. Но в листинге 9.10 выполняется сохранение информации о кредитной карте в самой службе. Позд-

нее в потоке мы сможем безопасно обращаться к этим данным или вызывать другие методы службы, использующие эти же данные, будучи уверенными, что никакой другой объект не изменит их к тому времени.

Поскольку службы области видимости потока сохраняются в потоке, они обязаны поддерживать интерфейс `Serializable`, как и любые другие объекты, сохраняемые в потоке. Этим они отличаются от стандартных служб; вы знаете, что делать при появлении ошибки, гласящей: «object in flow state could not be serialized» («объект не может быть сохранен в потоке»).

> **Как быть, если мне потребуется служба, которая могла бы использоваться и в области видимости потока, и в области видимости контроллера?**
>
> Мы знаем, о чем вы размышляете, – вы хотите использовать службу в области видимости потока, но при этом иметь возможность вызывать некоторые ее методы из стандартного контроллера. Очень жаль, но это практически всегда говорит о том, что вы что-то недостаточно продумали.
>
> Основное назначение служб области видимости потока заключается в сохранении информации о состоянии. Если у вас возникает желание смешать методы с сохранением и без сохранения состояния, то вы, скорее всего, подразумеваете методы, которые должны принадлежать двум разным классам служб.
>
> Одно из решений данной проблемы состоит в том, чтобы выделить в отдельную службу программный код, предполагающий сохранение информации о состоянии в области видимости потока, а потом внедрить в нее службу без сохранения состояния. Это означает, что вы сможете многократно использовать логику службы без сохранения состояния, использовать компонент с сохранением состояния и одновременно остаться верным принципу DRY (Don't Repeat Yourself – избегай повторений).

9.3.2. Подпотоки и диалоги

До сих пор мы рассматривали возможности перехода от состояния к состоянию и ветвления в пределах единственного потока. Но по мере разработки потока все усложняется. Вам может потребоваться выполнять переходы между различными частями потока, ваши процессы могут принимать весьма запутанный вид, и тогда область видимости потока может начать напоминать свалку. Для таких ситуаций платформа Grails предоставляет возможность создавать подпотоки, которые позволяют реализовать некоторую логику работы в виде отдельного, самостоятельного потока.

Подпотоки обладают следующими преимуществами:

- Подпотоки можно многократно использовать в нескольких родительских потоках.
- Область видимости потока становится более упорядоченной, потому что она доступна только подпотоку и очищается по завершении его работы.
- Группировка логики работы в подпотоки делает основные потоки более короткими и более самодостаточными, а реализацию – более удобной в сопровождении.
- Вы можете обеспечить совместное использование данных родительскими и дочерними потоками, поэтому вам не придется дублировать данные потока.

Наша процедура вычисления стоимости посылки и ее доставки – отличный кандидат на выделение ее в отдельный подпоток. Мы дадим пользователю возможность выбирать наиболее подходящий для него способ доставки, и если такой выбор будет сделан, мы будем отправлять пользователя в подпоток, обрабатывающий различные параметры доставки. На рис. 9.4 изображен измененный фрагмент нашей блок-схемы (из рис. 9.1), где после этапа ввода адреса появился новый подпоток.

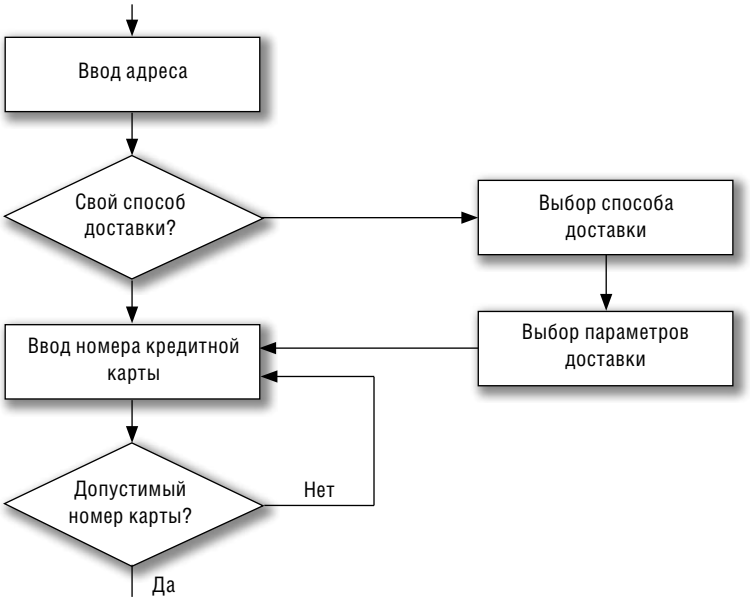


Рис. 9.4*. Новый подпоток, обрабатывающий параметры доставки, после этапа ввода адреса*

На странице ввода адреса пользователю предоставляется возможность установить флажок Custom Shipping (Свой способ доставки); мы будем ис-

пользовать его, чтобы запустить подпоток. Для сохранения информации о способе доставки нам потребуется управляющий объект: определение этого класса `ShippingCommand` приводится в листинге 9.11. Ограничения для данных мы добавим позднее, а пока будем использовать его для простой передачи данных.

Листинг 9.11. Объект ShippingCommand, используемый для сохранения информации о способе доставки

```
class ShippingCommand implements Serializable {

    String address
    String state
    String postcode
    String country
    boolean customShipping
    String shippingType
    String shippingOptions

}
```

Теперь реализуем запуск подпотока из основного потока `orderFlow`. Нам потребуется обеспечить совместное использование информации о способе доставки родительским и дочерним потоками. В настоящий момент данные в виде объекта `ShippingCommand` сохраняются в области видимости потока, но область видимости основного потока недоступна подпотокам. Поэтому нам необходимо создать *область видимости диалога* (*conversation scope*), где будут храниться данные, совместно используемые текущим потоком и всеми подпотоками. Внесем изменения в реализацию и сохраним информацию о способе доставки в области видимости диалога, как показано в листинге 9.12.

Листинг 9.12. Сохранение данных в области видимости диалога обеспечивает возможность передачи данных подпотокам

```
enterAddress {
    on("next") { ShippingCommand sc ->
        conversation.sc = sc ← Помещает управляющий объект в область
        if (sc.hasErrors()) {   видимости диалога
            return error()
        }
    }.to("checkShipping")
    on("previous").to("displayProducts")
}
```

Сохранив информацию о способе доставки в области видимости диалога, можно приступать к реализации ветвления на тот случай, если у пользователя появятся свои требования к способу доставки. Это отличный шанс применить на практике наши знания о действиях потока. В листинге 9.13 приводится реализация действия `checkShipping`, в котором сосредоточена логика выбора способа доставки.

Листинг 9.13. Использование состояния действия для проверки наличия у пользователя своих требований к способу доставки

```
checkShipping {
    action {
        if (conversation.sc.customShipping) { ❶ Обращение к переменной в области видимости диалога
            custom() ❷ Вызывает переход к выбору способа доставки
        } else {
            standard() ❸ Вызывает переход, соответствующий случаю стандартного способа доставки
        }
    }
    on("custom").to("customShipping") ❹ Переход к состоянию customShipping
    on("standard").to("enterPayment") ❺ Переход к состоянию enterPayment
}
```

В листинге 9.13 мы сначала проверяем управляющий объект, чтобы определить, выбрал ли пользователь какой-то иной способ доставки ❶. Если выбрал, мы вызываем переход `custom()` ❷, который осуществляет переход пользователя к состоянию `customShipping` ❹. Если пользователь предпочел стандартный способ доставки, мы вызываем переход `standard()` ❸, и поток переходит к состоянию `enterPayment`, как обычно ❺.

Теперь пришло время заняться реализацией состояния `customShipping` потока. Это новое состояние потока реализует переход к новому подпотоку, и здесь вам потребуются знание некоторых особенностей. Сначала взгляните на листинг 9.14, чтобы познакомиться с новой конструкцией `subflow()`.

Листинг 9.14. Запуск подпотока из родительского потока

```
customShipping { ❶ Определение состояния, с которого начинается новый подпоток

    subflow(customShippingFlow) ❷ Имя подпотока должно соответствовать имени текущего состояния с окончанием Flow
    on("goBack").to("enterAddress")
    on("standardShipping") {
        conversation.sc.customShipping = false
    }.to("enterPayment")
    on("customShipping").to("enterPayment")
}
```

Первое, что вам следует знать о подпотоках, – это то, что имя имеет большое значение. Имя подпотока ❶ должно соответствовать имени состояния ❷ без окончания «Flow» – переход в подпоток `customShippingFlow` должен производиться в состоянии с именем `customShipping`. Это немножко неудобное соглашение, от которого, вероятно, откажутся в будущих версиях платформы Grails, но пока ему необходимо следовать.

Но откуда взялись все эти результаты состояния (`goBack`, `standardShipping`, `customShipping`)? Это значения конечного состояния, возвращаемые подпотоком. Они приобретут для вас больше смысла, когда вы увидите определение подпотока, описанием которого мы и займемся дальше.

В листинге 9.15 мы добавили в контроллер `OrderController` новое определение подпотока `customShipping`.

Листинг 9.15. Реализация подпотока обработки нестандартного способа доставки

```
def customShippingFlow = {

    selectShippingType {                          ❶ Дает пользователю возможность выбрать
        on("next") {                                свой способ доставки
            conversation.sc.shippingType = params.shippingType
        }.to("selectShippingOptions")
        on("standardShipping").to("standardShipping")
        on("previous").to("goBack")
    }

    selectShippingOptions {                       ❷ Объявление дополнительного этапа
        on("previous").to("selectShippingType")     определения параметров способа доставки
        on("next") {
            conversation.sc.shippingOptions =     ❸ Сохраняет параметры доставки
                params.shippingOptions              в области видимости диалога
        }.to("customShipping")
    }

    customShipping()                              ❹ Определение значений, возвращаемых
    standardShipping()                              родительскому потоку
    goBack()
}
```

Как видно на блок-схеме (рис. 9.4), выбор собственного способа доставки реализуется достаточно просто. Это двухступенчатый процесс, при котором пользователь сначала выбирает способ доставки ❶, а затем определяет ее параметры ❷. На этом операция выбора заканчивается.

Подпотоки выглядят так же, как любые другие потоки. Они имеют всего два отличия:

- Для хранения данных, общих для родительского и дочернего потоков, используется область видимости диалога (а не область видимости потока).
- Конечные состояния подпотока используются для выбора состояния в родительском потоке.

В листинге 9.15, когда пользователь щелкает на кнопке Next (Далее), находясь в состоянии `selectShippingOptions` ❸, вызывается конечное состояние `customShipping()`. Оно является конечным, потому что за ним не следует других предложений `.on()` ❹, что приводит к передаче этого значения действию `customShipping` в родительском потоке `OrderFlow`, определение которого выглядит, как показано ниже:

```
customShipping {
    subflow(customShippingFlow)
```

```
        on("goBack").to("enterAddress")
        on("standardShipping") {
            conversation.sc.customShipping = false
        }.to("enterPayment")
        on("customShipping").to("enterPayment")
    }
```

Состояние `customShipping` вызывает переход к следующему состоянию `enterPayment`, и поток продолжает работу, как обычно. За счет перемещения логики обработки способа доставки в отдельный поток нам удалось убрать значительную долю логики из потока `orderFlow`, и мы обеспечили возможность многократного использования `shippingFlow`.

Но от нашего внимания ускользнули некоторые детали общей мозаики. Мы рассмотрели логику работы подпотока, но пока ничего не сказали о том, где находятся файлы GSP для этого нового подпотока. Они следуют тем же соглашениям, что применяются к родительскому потоку. Если учесть, что подпоток `customShippingFlow` определен в контроллере `OrderController` (как в данном случае), файл selectShippingType.gsp должен находиться в каталоге /views/order/customShipping/.

Мы рассмотрели массу дополнительных сведений о веб-потоках и подпотоках, но у нас осталась неисследованной еще одна область: тестирование. Как правильно тестировать все эти потоки и подпотоки, чтобы убедиться, что все действует так, как мы ожидаем? Пришло время заняться изучением поддержки тестирования веб-потоков.

9.4. Тестирование веб-потоков

Как и в случае с любыми другими компонентами платформы Grails, вы должны иметь представление о том, как тестировать веб-потоки. Способность веб-потоков сохранять информацию о своем состоянии делает их тестирование несколько более сложным по сравнению с обычными контроллерами, однако платформа Grails предлагает некоторые встроенные возможности имитации необходимых деталей мозаики. Рассмотрим простой испытательный тест, чтобы получить представление с чего начинать.

До сих пор мы трудились над созданием контроллера обработки заказа, поэтому начнем тестирование с него. Тестирование любых веб-потоков должно выполняться в виде интеграционных тестов, так что давайте создадим заготовку интеграционного теста:

```
grails create-integration-test com.grailsinaction.OrderFlowIntegration
```

Теперь нам нужно дополнить этот тест, добавив наследование класса `WebFlowTestCase`, базового для всех тестов, предназначенных для тестирования веб-потоков. Реализация простого испытательного теста приводится в листинге 9.16.

Отладка необычных ошибок веб-потоков

Когда у вас появятся потоки и подпотоки со сложной организацией, вы неизбежно столкнетесь с некоторыми необычными ошибками, такими как ошибка невозможности сохранения объекта в потоке, ошибка с кодом 404 и другими. В подобных ситуациях у вас будут два варианта действий.

Во-первых, вы можете запустить команду `grails clean`. Проблема может возникать всего лишь на этапе компиляции, которая может быть решена очисткой классов и повторной компиляцией.

Если этот вариант не дал положительных результатов, можно попробовать второй вариант – и изменить уровень журналирования в файле /grails-app/conf/Config.groovy, чтобы можно было увидеть все отладочные сообщения веб-потоков. Для этого добавьте следующую строку:

```
log4j = {
    debug 'org.springframework.webflow'
}
```

А затем перезапустите сервер. Вы получите массу различных отладочных сообщений, но они помогут вам определить, какие состояния вызываются и какие объекты сохраняются. Это потребует определенного времени, зато у вас, по крайней мере, появятся предположения об источнике проблем.

Если ваша проблема имеет отношение к представлению (например, вы получаете ошибку с кодом 404), попробуйте установить отладочный уровень журналирования для `org.springframework`. Но еще лучше – установите расширение runtimeLogging, это позволит вам изменять уровни журналирования прямо во время выполнения приложения, на странице /hubbub/runtimeLogging. Вы будете получать множество отладочных сообщений, но они дадут вам возможность докопаться до сути проблемы (и, возможно, запустить систему отслеживания ошибок JIRA).

Листинг 9.16. Простой интеграционный тест для нашего потока

```
package com.grailsinaction

import grails.test.*

class OrderFlowIntegrationTests extends WebFlowTestCase {

    def shopController = new ShopController()
```

❶ Для тестирования потока необходимо наследовать класс WebFlowTestCase

❷ Создается контроллер тестируемого потока

```
    def getFlow() { shopController.orderFlow } ← ❸ Определяется тестируемый поток

    void testBasicOrder() {

        startFlow()                                    ← ❹ Запуск потока и переход
        assertCurrentStateEquals "displayProducts"       к состоянию по умолчанию

    }
}
```

Как уже говорилось выше, все испытательные тесты веб-потоков являются интеграционными тестами и должны расширять класс `WebFlowTestCase` ❶. Наследование класса `WebFlowTestCase` необходимо, чтобы создать метод `getFlow()` ❸, возвращающий замыкание, представляющее веб-поток. В данном случае мы работаем с потоком `orderFlow`. Затем создается экземпляр контроллера `ShopController`, который может потребоваться нам позже ❷, чтобы сымитировать передачу действиям отображения `params`. В заключение вызывается реализация нашего метода `testBasicOrder()`, который в свою очередь вызывает метод `startFlow()` ❹.

Как следует из имени, метод `startFlow()` – это первое действие нашего потока и это один из нескольких добавляемых классом `WebFlowTestCase` методов, которые используются для запуска действий в потоке. Наиболее важные из этих методов перечислены в табл. 9.1.

Таблица 9.1. Методы класса WebFlowTestCase, имеющие отношение к потокам

Метод	Описание
`startFlow()`	Выполняет переход к первому состоянию текущего потока
`signalEvent(String)`	Имитирует событие потока, вызываемое пользователем
`setCurrentState(String)`	Запускает новый поток и сразу же переходит к заданному состоянию
`getFlowScope()`	Получает область видимости текущего потока
`getConversationScope()`	Получает текущую область видимости диалога

После запуска потока мы можем с помощью метода `assertCurrentStateEquals()` ❹ проверить, действительно ли был выполнен переход к состоянию `displayProducts()`. Этот и другие методы проверки, которые также добавляются классом `WebFlowTestCase`, позволяют убедиться, что поток находится в требуемом состоянии. В табл. 9.2 перечислены некоторые из наиболее важных методов проверки, доступных при тестировании веб-потоков.

Таблица 9.2. Методы проверки класса WebFlowTestCase, имеющие отношение к потокам

Метод	Описание
`assertCurrentStateEquals(String)`	Проверяет, является ли текущее состояние потока активным и совпадает ли имя состояния с указанной строкой
`assertFlowExecutionActive()`	Проверяет, что поток не достиг конечного состояния
`assertFlowExecutionEnded()`	Проверяет, что поток достиг конечного состояния и завершил работу
`assertFlowExecutionOutcomeEquals(String)`	Позволяет проверить совпадение имени возвращаемого значения потока с указанной строкой после его завершения

Теперь, когда вы познакомились с основами тестирования веб-потоков, можно перейти к созданию более полного испытательного теста для нашего потока обработки заказа.

9.4.1. Обработка входных параметров

В главе 7 вы многое узнали о тестировании контроллеров, поэтому процедура передачи входных параметров в контроллеры не должна быть вам в диковинку. Но при тестировании веб-потоков вам придется столкнуться с несколькими новыми острыми ситуациями.

Мы попробуем передавать состоянию `displayProducts` некоторые недопустимые параметры и посмотрим, как можно расширить наш испытательный тест, чтобы обеспечить наиболее полный охват веб-потока. В листинге 9.17 приводится испытательный тест, в котором имитируется ситуация, когда количество заказываемых товаров превышает допустимое значение.

Листинг 9.17. Передача значений и имитация ошибочных ситуаций

```
void testBasicOrder() {

    startFlow()
    assertCurrentStateEquals “displayProducts”

    shopController.params.numShirts = 1
    shopController.params.numHats = 1000 ← ❶ Имитация ошибочного значения в поле формы с помощью params
    signalEvent('next')

    assertTrue getFlowScope().orderDetails.hasErrors() ← ❷ Подтверждение ошибки
    assertCurrentStateEquals “displayProducts”
```

```
        shopController.params.numHats = 1   ← ❸ Исправление ошибки в поле
        signalEvent('next')                    и повторная передача значений
        assertCurrentStateEquals "enterAddress" ← ❹ Подтверждение перехода потока
                                                   к следующему состоянию
}
```

После запуска потока мы имитируем передачу формы заказа с недопустимым количеством бейсболок ❶. Управляющий объект содержит ограничение, требующее, чтобы количество заказываемых предметов находилось в диапазоне от 0 до 10.

Имитация события `next` вызывает отправку параметров действию `displayProduction` (показанному выше в листинге 9.6), что приводит к появлению ошибки проверки ограничений и возврату к состоянию `displayProducts`. Напомним, что все эти действия направлены на тестирование:

```
displayProducts {
    on("next") { OrderDetailsCommand odc ->
        if (odc.hasErrors() || odc.isOrderBlank()) {
            flow.orderDetails = odc  ← Помещает ошибочный заказ обратно
            return error()              в область видимости потока
        }
        [ orderDetails: odc, orderStartDate: new Date() ]
    }.to("enterAddress")
    on("cancel").to("finish")
}
```

Обратите внимание, что когда проверка ограничений терпит неудачу, мы помещаем управляющий объект обратно в область видимости потока, благодаря чему мы можем восстановить значения полей формы при повторном ее отображении. Поскольку ошибочный заказ оказывается в области видимости потока, наш испытательный тест может убедиться, что объект `orderDetails` содержит соответствующий атрибут `errors`, и пользователь был возвращен к состоянию `displayProducts` ❷.

Убедившись, что обработка ошибок выполняется корректно, мы исправляем недопустимое значение и повторно имитируем передачу формы ❸. На этот раз проверка ограничений должна закончиться успехом, и поток должен оказаться в состоянии `enterAddress` ❹.

Это достаточно полный тест, и он поможет вам получить представление о том, как писать свои собственные тесты. Мы узнали, как имитировать события, как обрабатывать параметры и даже как обращаться с ошибками. Но у нас остались еще не исследованными подпотоки и завершившиеся потоки. Теперь перейдем к ним.

9.4.2. Тестирование переходов в подпотоки

Вспомните, в разделе 9.3.2 мы вынесли логику обработки нестандартного способа доставки в отдельный подпоток. Сейчас мы напишем тест для этого подпотока. Но прежде мы должны убедиться, что при вклю-

чении флажка Custom Shipping (Свой способ доставки) пользователь отправляется в подпоток, а при выключении выполняется переход к состоянию enterPayment.

В листинге 9.18 приводится дополненная реализация теста, которая сначала проверяет работу для случая стандартного способа доставки, а затем возвращается и отмечает флажок, что должно привести к переходу в подпоток.

Листинг 9.18. Переход в подпоток

```
void testSubflowTrigger() {

    setCurrentState('enterAddress') <- ❶ Переход к состоянию enterAddress

    shopController.params.customShipping = false  <- ❷ Имитация стандартного
    signalEvent('next')                               способа доставки
    assertEquals false, getConversationScope().sc.customShipping
    assertCurrentStateEquals "enterPayment" <- ❸ Убедиться, что выбран
                                                  стандартный способ доставки
    signalEvent('previous')                   <- ❹ Имитировать щелчок на
    assertCurrentStateEquals "enterAddress"       кнопке Previous (Назад)

    shopController.params.customShipping = true
    signalEvent('next')
    assertEquals true, getConversationScope().sc.customShipping
    assertCurrentStateEquals "selectShippingType" <- ❺ Убедиться, что отображена
                                                      страница выбора способа доставки
}
```

Тест начинается с непосредственного перехода в состояние enterAddress ❶. Затем мы имитируем снятие флажка Custom Shipping (Свой способ доставки) и щелчок на кнопке Next (Далее) ❷. Когда выбран стандартный способ доставки, пользователь должен оказаться в состоянии enterPayment, значит, мы можем явно убедиться, что переход в подпоток был пропущен ❸.

Затем мы имитируем щелчок на кнопке Previous (Назад) и проверяем, привело ли это к возврату в состояние enterAddress ❹. Мы имитируем установку флажка Custom Shipping (Свой способ доставки), что приводит к запуску подпотока и переходу к первому его состоянию – на страницу selectShippingType ❺.

Мы могли бы протестировать передачу параметров в подпоток и процедуру оформления заказа целиком, но мы оставим это вам в качестве самостоятельного упражнения.

Теперь, когда у нас имеется интеграционный тест для подпотока, можно написать испытательный тест, полностью тестирующий сам подпоток. Это даст нам возможность узнать, как тестируется завершение работы потока.

9.4.3. Тестирование завершения потока

Последнее, что нам осталось протестировать, – это корректное завершение потока. Наш поток, обрабатывающий выбор нестандартного способа доставки, имеет весьма интересные характеристики завершения, что делает его превосходным кандидатом для экспериментов. Листинг 9.19 воспроизводит поток целиком.

Листинг 9.19. Поток обработки нестандартного способа доставки имеет несколько вариантов завершения

```
def customShippingFlow = {

    selectShippingType {
        on("next") {
            conversation.sc.shippingType = params.shippingType
        }.to("selectShippingOptions")
        on("standardShipping").to("standardShipping")
        on("previous").to("goBack")
    }

    selectShippingOptions {
        on("previous").to("selectShippingType")
        on("next") {
            conversation.sc.shippingOptions = params.shippingOptions
        }.to("customShipping")
    }

    customShipping()
    standardShipping()
    goBack()
}
```

Прежде всего создадим интеграционный тест:

```
grails create-integration-test
↪       com.grailsinaction.CustomShippingFlowIntegration
```

Создав заготовку испытательного теста, можно наполнить его некоторой логикой тестирования, чтобы проверить наш новый подпоток. В листинге 9.20 демонстрируется логика, имитирующая действия пользователя по выбору способа доставки и вводу дополнительной информации и завершение потока в конечном состоянии `customShipping`.

Листинг 9.20. Испытательный тест проверки завершения потока, обрабатывающего нестандартный способ доставки

```
package com.grailsinaction

import grails.test.*

class CustomShippingFlowIntegrationTests extends WebFlowTestCase {
```

```
    def shopController = new ShopController()

    def getFlow() { shopController.customShippingFlow }

    void testFlowTermination() {

        startFlow()
        getConversationScope().sc = new ShippingCommand()
        shopController.params.shippingType = "express"
        signalEvent("next")
        shopController.params.shippingOptions = "same-day"
        signalEvent("next")
        assertFlowExecutionEnded()                    <- ❶ Проверяется, что поток
        assertFlowExecutionOutcomeEquals(                завершился в корректном
                              "customShipping")          состоянии
    }
}
```

Испытательный тест запускает подпоток, после чего он выбирает способ доставки и определяет дополнительные ее параметры. Как только эти события будут запущены, можно проверить, что поток завершился и что конечным состоянием потока является состояние `customShipping` ❶. Вызов метода `assertFlowExecutionEnded()` является избыточным, поскольку заключительная проверка `assertFlowExecutionOutcomeEquals()` гарантирует, что поток находится в конечном состоянии, но мы все равно добавили его для полноты картины.

На этом мы завершаем обзор приемов тестирования веб-потоков. Теперь коротко повторим, что мы узнали в этой главе.

9.5. В заключение

В этой главе вы получили полное представление о веб-потоках платформы Grails – настраиваемом до тонкостей механизме сохранения состояний, предназначенном для работы с последовательностями логически связанных страниц в приложениях на платформе Grails. Вы познакомились с основами работы состояний и переходов и реализовали большое количество особенностей, типичных для многоэтапных операций в стиле мастеров.

Мы познакомили вас с уникальным понятием состояний потоков, с областями видимости потока и диалога и применили их для сохранения управляющих объектов, поддерживающих проверку ограничений, при передаче данных в потоке. Мы также исследовали некоторые дополнительные особенности веб-потоков, такие как подпотоки и службы области видимости потока.

Мы завершили главу подробным введением в тестирование веб-потоков и рассмотрением основ запуска потоков, тестирования ошибок и конечных состояний.

Это был исчерпывающий тур, и по пути мы познакомились с множеством интересных приемов:

- *Не злоупотребляйте потоками*. Вы должны знать, для решения каких проблем веб-потоки подходят, и использовать их, только когда в этом есть смысл. Типичная область применения веб-потоков – последовательности операций, где требуется сохранять информацию о состоянии между этапами, поэтому старайтесь не привлекать их для решения не характерных для них задач.
- *Покорите сложные формы*. Используйте веб-потоки для обработки сложных форм – разбейте форму на множество страниц и организуйте пропуск этапов, которые не должны навязываться всем пользователям.
- *Отдавайте предпочтение области видимости потока (flow scope)*. Отдавайте предпочтение области видимости потока, когда вы стоите перед выбором между областью видимости потока и областью видимости сеанса. Веб-потоки автоматически очищают свою область видимости, что позволит вам более эффективно использовать память сервера.
- *Разберитесь с параметром* `execution`. Изучите принцип действия параметра `execution` и то, как он влияет на кнопку Back (Назад) броузера и постоянные ссылки. Проектируйте свои потоки так, чтобы максимально уменьшить это влияние.
- *Используйте управляющие объекты*. Используйте управляющие объекты для проверки ограничений, накладываемых на значения полей форм, в каждом состоянии своего потока. Используйте самоописательные имена для валидаторов управляющих объектов и собственных процедур, чтобы они однозначно сообщали об их предназначении (например, `isOrderBlank()`).
- *Упрощайте реализацию применением состояний действия*. Упрощайте принятие решений в состояниях с помощью состояний действия. Используйте осмысленные имена переходов, чтобы логика работы была более очевидной.
- *Используйте службы области видимости потока*. Используйте службы области видимости потока для организации в потоках компонентов с запоминанием состояния. Модель параллельной обработки в этом случае значительно упрощается; кроме того, службы могут удаляться по завершении потока.
- *Пишите тесты*. Пишите испытательные тесты для веб-потоков, как и для любых других компонентов платформы Grails. Это позволит вам убедиться, что потоки действуют именно так, как вы и предполагали. Написать тест оказывается быстрее, чем непрерывно производить тестирование интерфейса в броузере с привлечением человека; кроме того, тесты помогут проверить работоспособность после перехода на новую версию платформы Grails.

На примере создания интернет-магазина Hubbub мы дали вам исчерпывающее введение в веб-потоки платформы Grails и добавили ряд новых особенностей в приложение Hubbub. В следующей главе мы обратим свое внимание на реализацию в приложении Hubbub ограничений, связанных с безопасностью, – мы гарантируем доступ пользователей только к тем особенностям приложения Hubbub, на использование которых у них имеется разрешение.

В этой главе:

- Почему вы должны уделять внимание обеспечению безопасности
- Как защитить свое приложение от нападений
- Реализация управления доступом

10

Не впускайте незнакомцев, или к вопросу о безопасности

В последних двух главах мы существенно улучшили пользовательский интерфейс приложения Hubbub, поэтому теперь мы займемся вопросом выпуска приложения Hubbub в дикие джунгли Интернета. Это означает, что вы должны прийти к пониманию опасностей, подстерегающих приложение, и обеспечить ему защиту от самых страшных злоключений. Добро пожаловать в мир безопасности.

10.1. Почему безопасность имеет такое значение

Мы уверены, что вы знакомы с некоторыми проблемами безопасности веб-приложений и, возможно, даже с некоторыми приемами, позволяющими уменьшить риск. Мы рассмотрим большую их часть применительно к платформе Grails, благодаря чему вы будете знать, какие меры можно предпринять, чтобы защитить свое приложение от нападений и от несанкционированного доступа. К концу главы вы с полной уверенностью в безопасности сможете представить свое веб-приложение реальному миру с реальными пользователями.

Однако прежде чем перейти к обсуждению подробностей, мы расскажем небольшую историю. Компания А была благополучным банком, и ее руководством было принято решение реализовать предоставление услуг клиентам банка через Интернет. Специалисты понимали, что такая система будет находиться под постоянной угрозой нападения различных злоумышленников, стремящихся получить доступ к информации о клиентах и к их деньгам, поэтому они создали систему, которая превратила их портал в виртуальный эквивалент Форт Нокса[1]. Ника-

[1] Fort Knox – хранилище золотого запаса США. – *Прим. ред.*

кие нападения с участием человека не могли преодолеть защиту системы, клиенты никогда не отправляли секретную информацию на сервер целиком, и все операции выполнялись через SSL. Система была неприступной.

Но однажды один из служащих банка скопировал информацию о тысячах клиентах на ноутбук, чтобы продолжить работу дома. На следующее утро по дороге на работу он случайно оставил ноутбук в поезде, и тот пропал. Информация, хранившаяся в этом ноутбуке в незашифрованном виде, стоила намного дороже, чем сам ноутбук. Это стало огромной проблемой для компании, ей нужно было известить об инциденте всех клиентов, которых он коснулся. Многие из них закрыли свои счета и перешли в другие банки. Репутации банка был нанесен непоправимый ущерб.

Это вымышленная история, и мы надеемся, что никакой банк ни при каких условиях не позволит своим служащим копировать информацию о клиентах (особенно в незашифрованном виде) и выносить ее за пределы учреждения. И все же имели место несколько громких случаев, когда ноутбуки оставлялись в общественных местах, что приводило к потере секретной информации!

Суть истории заключается в том, что безопасность – это не продукт и не технология. Это не предмет, который можно взять с полки. Даже если доступ к вашему приложению возможен только через SSL, оно еще не в безопасности. Безопасность – это и процесс, и образ мышления. Мы не ожидаем, что вы тут же начнете настаивать на изменении организационного мышления вашей компании, но мы считаем важным помнить мораль рассказанной истории при разработке приложений, даже если выработаете над своим персональным проектом в свободное время.

Рассмотрим некоторые из угроз, с которыми может столкнуться ваше приложение, и способы защиты от них.

10.2. Защита от злоумышленников

Вероятно, вам приходилось слышать о сетевых червях, которые за минуты распространяются по всему миру и заражают веб-сайт за веб-сайтом, о фактах порчи страниц веб-сайтов и о приложениях, из которых обманным путем извлекалась секретная информация теми, кто не должен был иметь доступ к ней. Определенно нет недостатка в желающих потратить некоторое время на подобные неприглядные действия. Ваша задача состоит в том, чтобы помешать им сделать нечто подобное с вашими приложениями.

Самое главное помнить о том, что вы разрабатываете, и всегда рассматривать свои разработки с позиции обеспечения безопасности. Если вы сможете думать как злоумышленник, будет еще лучше! Только не превращайтесь в него и не обвиняйте нас в этом. Мы стараемся избегать юридических хитросплетений.

В данном разделе мы проведем вас через несколько простых шагов и процедур, которые помогут существенно снизить риск вторжения и защититься от большинства потенциально возможных видов нападений. Несмотря на то что представленные здесь примеры рассматриваются в контексте платформы Grails, демонстрируемые в них концепции можно применять к разработке приложений на любом языке программирования.

Начнем наше обсуждение с одной из самых уязвимых сторон программного обеспечения: обработки данных, введенных извне.

10.2.1. Проверяйте все вводимые данные

Представьте, что вам требуется отыскать в приложении Hubbub все сообщения, отмеченные тегом «grails». Таких сообщений могут быть миллионы, потому что Grails – чрезвычайно крутая тема для обсуждения, и если приложение Hubbub попытается вернуть *всех* их, это может повредить как серверу, так и клиенту. Чтобы предотвратить такое поведение, приложение Hubbub по умолчанию возвращает 50 сообщений. Но допустим, что вам требуется получить только 10 самых последних сообщений.

В приложении Hubbub данную возможность можно обеспечить, принимая параметр `max` (который мог бы содержать значение 10) и передавая его реализации запроса:

```
def search = {
    def max = params.max?.toInteger() ?: 50
    def results = Post.findAllByTag("grails", [ max: max ])
    ...
}
```

Следует отметить, что данный фрагмент не соответствует реализации, используемой в приложении Hubbub, но он служит иллюстрацией к важному замечанию. Вы можете посмотреть на этот код и подумать, что в нем все делается правильно, однако в нем заключен один фундаментальный недостаток: он доверяет пользователю. Представьте, что произойдет, если пользователь попытается ввести такой адрес URL: *http://localhost:8080/hubbub/post/search?query=grails&max=1000000000*.

Значение параметра `max` равно одному миллиарду! В этом случае действие попытается вернуть и отобразить все сообщения с тегом «grails» (которых, конечно же, намного меньше миллиарда). Несколько подобных запросов могут привести к остановке сервера. Это классическая атака «отказ в обслуживании» (Denial of Service, DoS).

Мораль этого примера незатейлива: никогда не доверяйте внешним источникам данных – проверяйте все, что используете. Простейшим решением проблемы в данном конкретном примере было бы ограничение верхнего предела значения `max`:

```
def max = params.max?.toInteger() ?: 50
max = Math.min(max, 500)
```

Такое маленькое изменение даст пользователям возможность запросить до 500 сообщений и не больше. Это простой случай, но как быть, если имеется несколько параметров, которые требуется проверять во многих методах?

Один из подходов заключается в использовании управляющих объектов, которые позволяют объявлять ограничения и выполнять проверку наличия нарушений этих ограничений. Благодаря тому что управляющие объекты могут многократно использоваться в различных методах, вы получаете систему, сопровождение которой намного проще, чем проверка всех параметров вручную. Кроме того, вы получаете в свое распоряжение все остальные преимущества управляющих объектов, которые вы видели выше в этой книге.

Если вы не хотите возиться с созданием управляющих объектов, попробуйте использовать удивительный метод `bindData()`. В следующем примере выполняется привязка всех параметров, за исключением параметра с именем «created»:

```
def update = {
    def post = new Post()
    bindData(post, params, ["created"])
    ...
}
```

В главе 5 была представлена эта версия метода с «черным списком», когда в третьем аргументе методу `bindData()` передается список полей класса предметной области, которые не должны участвовать в привязке данных. Мы исключили это поле потому, что пользователь не должен иметь возможность изменять время создания сообщения. Кроме того, метод `bindData()` исключает некоторые свойства по умолчанию: `id`, `version`, `metaClass` и `properties` (последние два имеют отношение к языку программирования Groovy, поэтому можете пока забыть о них). Следует заметить, что этот метод динамически внедряется в контроллеры, поэтому его нельзя использовать в службах или в классах предметной области.

«Ага!» – мы уже слышим ваши возгласы: «Если я выполняю проверку на стороне клиента, значит, мне не нужно беспокоиться о ней на стороне сервера!» Проверка на стороне клиента, безусловно, полезна для самого пользователя, но было бы ошибкой отказываться от проверки на стороне сервера.

Двойная проверка

Всякий раз, занимаясь разработкой веб-приложения, помните, что интерфейсом к нему является не броузер, а протокол HTTP. Любой желающий может отправить запрос вашему приложению из любого клиентского приложения, а также может даже использовать утилиту

Опасность привязки данных в классах предметной области

Как вы уже знаете, существует возможность выполнять прямую привязку параметров к объектам предметной области. Это достаточно опасно, потому что привязка выполняется для любых значений параметров, имена которых совпадают с именами свойств класса. Атакующий может попытаться изменить свойство, доступ к которому он не должен иметь.

Вместо такой прямой привязки можно воспользоваться управляющим объектом и выполнить привязку его свойств к свойствам объекта предметной области, например так:

```
def update = { PostCommand cmd ->
    if (!cmd.hasErrors()) {
        def post = new Post(cmd.properties)
        ...
    }
    ...
}
```

telnet и вручную формировать запросы HTTP. В конце концов, протокол HTTP – это достаточно простой протокол, предназначенный для передачи текстовой информации. В такой ситуации реализованная вами проверка на стороне клиента просто не будет задействована. Кроме того, пользователь может отключить поддержку JavaScript в своем броузере.

Для вашего приложения это означает, что оно может получить от клиента все, что угодно, и, скорее всего, так оно и будет. Вот почему необходимо выполнять проверку не только на стороне клиента, но и на стороне сервера. Мы понимаем, что это нарушает принцип DRY (Don't Repeat Yourself – избегай повторений), но, к сожалению, такова цена, которую приходится платить общедоступному серверу за защиту богатых клиентов. Это правило незыблемо независимо от типа используемого клиента, будь то JavaScript в броузере, Flash-приложение или апплет Java.

Вышесказанное относится не только к объекту `params`. Что если при обработке запроса, содержащего данные, у вас возникает потребность читать их непосредственно из запроса?

Другие типы запросов

Как вы увидите в следующей главе, запросы могут содержать данные в формате XML, JSON или даже в каком-то вашем собственном формате. В таких ситуациях информация обычно не попадает в объект `params`, поэтому приемы привязки данных, которые известны нам к настоящему моменту, просто не будут действовать. Что же тогда можно сделать?

Если это возможно, то полученные данные следует преобразовать в отображение параметров и значений, которое затем можно было бы использовать в методе `bindData()`. Такой подход применим к данным в формате JSON. Если же речь идет о данных в формате XML, вам следует подумать об использовании языка XML Schema. Он позволяет добавлять ограничения для значений атрибутов и содержимого элементов, что обеспечивает проверку входных данных на декларативном уровне.

Если ни один из этих методов не подходит, вам придется реализовать собственную процедуру проверки, даже если это будет означать выполнение проверки вручную, как для параметра `max` в примере выше. Вариант реализации с отсутствием проверки данных вообще не должен рассматриваться (по крайней мере, как безопасный).

Теперь вы должны понимать, насколько важна проверка входных данных для безопасности приложения. К счастью, простая проверка данных, осторожное отношение к привязке данных и наличие ограничений позволят вам легко защититься от злонамеренных запросов HTTP. Как вы увидите в примерах исходных текстов приложения Hubbub этой главы, мы внесли все необходимые изменения в контроллеры.

Такой программный код даст вам уверенность, что ваше веб-приложение остается защищенным в опасном мире, но не думайте, что на этом история заканчивается. В будущем вам, скорее всего, придется писать приложения, которые будут получать данные из различных источников или в различных форматах, для которых нельзя будет задействовать механизмы привязки данных. В таких ситуациях важно помнить, что входные данные должны проверяться во что бы то ни стало.

Обычно самой по себе проверки входных данных бывает недостаточно. Опасность входных данных не всегда бывает очевидной, и порой бывает трудно ее обнаружить. Это означает, что вы должны внимательно следить за всем, что вы *делаете* с входными данными.

10.2.2. Экранируйте все выходные данные

Представьте, что в приложении Hubbub имеется действие контроллера, отображающее все сообщения указанного пользователя. Например, ниже приводится запрос, который реализует эту возможность:

```
def userId = “peter”
Post.findAll(“from Post as post WHERE post.user.userId = ‘${userId}’”)
```

Для выбора сообщений данный запрос использует локальное свойство `userId`. Выглядит достаточно невинно, не правда ли?

Двинемся дальше и попробуем выполнить его в консоли Grails – скопируйте эти две строки и выполните их. В зависимости от хранящихся у вас данных вы должны увидеть нечто похожее на то, что изображено на первом снимке с экрана на рис. 10.1.

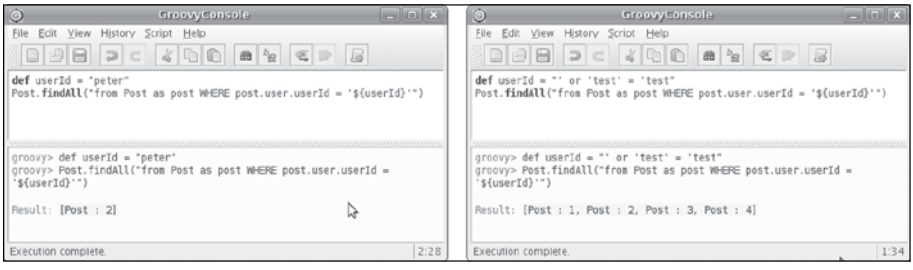


Рис. 10.1. *Сначала был выполнен запрос HQL с достаточно безопасным входным значением (слева), а во второй раз была предпринята попытка выполнить нападение типа «инъекция SQL» (справа). В последнем случае по запросу были возвращены все сообщения, хранящиеся в базе данных.*

Данные и консоль

Если ваша база данных настроена так, что она заново создается при каждом запуске приложения или в ней вообще отсутствуют данные, тогда примеры в этом разделе будут не особенно наглядны. Вы можете быстро наполнить базу данных какими-нибудь данными, выполнив следующие две строки в консоли:

```
def bs = new BootStrap()
bs.init()
```

Теперь представьте, что злоумышленник изменил строку запроса URL таким образом, что параметр `userId` получил значение, как в примере ниже:

```
def userId = "' or 'test' = 'test"
Post.findAll("from Post as post WHERE post.user.userId = '${userId}'")
```

Запрос остался прежним, но значение параметра `userId` выглядит уже не как идентификатор. Взгляните, что получится, когда это значение попадет в запрос:

```
... WHERE post.user.userId = '' or 'test' = 'test'
```

О, подлый! Злоумышленник изменил запрос так, что теперь он возвращает все сообщения, а это легко может привести к отказу сервера! Данный пример нападения называется «инъекцией SQL». Результат такого нападения изображен на втором снимке с экрана на рис. 10.1.

Проверка входных данных может остановить выполнение запроса клиента еще до того, как данные попадут в запрос к базе данных, но во многих случаях проверка не может предотвратить использование опасных значений, не блокируя при этом допустимые значения. К счастью, су-

ществует простое решение: экранируя входные значения перед вставкой их в текст запроса к базе данных, вы препятствуете нападению.

Примечание

Вам редко придется страдать от этой проблемы в приложениях на платформе Grails, потому что и динамические методы поиска, и запросы с критериями автоматически экранируют свои аргументы.

Мы завершим этот небольшой пример модифицированной версией запроса HQL, которая защищена от нападения такого вида, потому что в ней предусмотрено экранирование значения параметра `userId`:

```
def userId = "' or 'test' = 'test"
Post.findAll("from Post as post WHERE post.user.userId = ?", [ userId ])
```

Это эквивалент параметризованного запроса JDBC.

Теперь рассмотрим еще один похожий пример. Запустите сервер Hubbub, зарегистрируйтесь в приложении и попробуйте отправить такое сообщение (дословно!):

```
<script type="text/javascript">alert("Got ya!")</script>
```

Что произошло? Появился диалог с сообщением «Got ya!» Теперь всякий раз, когда вы будете обновлять страницу со списком сообщений, будет появляться данный диалог. Мда, не очень-то приятно. Вы только что продемонстрировали нападение типа «межсайтовый скриптинг» (XSS). Это одна из разновидностей нападения инъекции, но в данном случае инъекции разметки HTML и программного кода JavaScript, а не SQL.

Не удивительно, что данная проблема решается так же, как и в случае с инъекцией SQL: экранировать данные, введенные пользователем, еще до того, как они будут отображены в веб-странице. Это можно сделать вручную, вызвав метод `encodeAsHTML()` и передав ему текст, который требуется отобразить, или автоматически, добавив (или изменив) следующую строку в файле grails-app/conf/Config.groovy:

```
grails.views.default.codec = "html"
```

Благодаря этому параметру настройки платформа Grails будет экранировать все выражения в файлах GSP, создающие разметку HTML, и вам не придется даже пальцем шевельнуть!

Внимание

Хотя параметр `grails.views.default.codec` является действенным средством защиты во многих случаях, он не экранирует значения в атрибутах тегов GSP и не экранирует вывод, генерируемый самими тегами. Теги сами должны выполнять необходимое экранирование или кодирование.

Чтобы увидеть действие этого параметра настройки на практике, взгляните на следующий фрагмент файла GSP:

```
<div>${post.content}</div>
<g:textField name="test" value="${post.content}"/>
```

В первой строке содержимое сообщения экранируется автоматически, потому что обращение к нему производится из самостоятельного выражения. Но платформа Grails не делает то же самое со значением атрибута во второй строке. Именно поэтому реализация тега `textField` выполняет аналогичную операцию:

```
attrs["value"].encodeAsHTML()
```

В общем случае, когда в файлах GSP используются кодек HTML и стандартные теги платформы Grails, вам не о чем беспокоиться. Но помните, что при создании собственных тегов GSP вы должны учитывать необходимость экранирования, если они напрямую выводят какие-либо входные данные.

Мы почти закончили обсуждение рассматриваемой темы, но мы считаем необходимым упомянуть, что экранирование или кодирование ввода пользователя не будет действовать, если вы не укажете правильный кодек для выполнения этой работы. Например, если входное значение используется для создания гиперссылки, используйте метод `encodeAsURL()`; если оно используется где-то в разметке HTML, используйте кодек HTML.

Мы привели лишь два примера атак, связанных с инъекцией, но существуют и другие разновидности такого рода нападений, например инъекция программного кода (когда у пользователя появляется возможность ввести программный код на языке Groovy, который затем выполняется приложением), инъекция команд операционной системы (все, что передается методу `Runtime.exec()` или его эквивалентам) и инъекция путей к файлам. Во всех этих случаях вы должны убедиться, что источник входных данных заслуживает доверия; если не заслуживает, экранируйте данные или обрабатывайте их каким-либо другим способом.

Вместе с проверкой входных данных экранирование выходных данных защитит вас от целого класса потенциальных атак. Потенциальные последствия нападения могут быть весьма серьезными, но мы надеемся, что вы понимаете, насколько просто защитить свое приложение от подобных атак. Будьте бдительны, всегда рассуждайте так: «Я использую входные данные, которым не могу доверять, что я могу сделать с ними?»

До сих пор мы занимались методами работы с входными данными, получаемыми из источников, не пользующихся доверием, но если бы ваше приложение хранило конфиденциальную информацию о пользователях, вы столкнулись бы с другой проблемой. Как передавать эту информацию, чтобы никто не смог ее прочитать?

10.2.3. SSL, шифрование и свертка сообщений

Представьте, что интернет-портал вашего банка позволяет отправлять идентификатор и пароль пользователя (или другую удостоверяющую информацию) в незашифрованном виде. Безопасно ли это? Мы так не думаем, и такой банк очень быстро потеряет всех своих клиентов. Проблема состоит в том, что хотя, на ваш взгляд, вы отправляете информацию непосредственно на веб-сайт, ее путь зачастую не такой прямой, что дает злоумышленникам возможность подсмотреть ее.

Стандартное решение заключается в использовании протокола HTTPS, который шифрует все данные, курсирующие между броузером и сервером, за счет использования протокола защищенных соединений (Secure Sockets Layer, SSL). Как дополнительное преимущество протокол SSL требует, чтобы сервер опубликовал свой сертификат SSL; это дает пользователям уверенность, что они пользуются настоящим веб-сайтом, а не его подделкой.

Чтобы запустить приложение на платформе Grails в режиме использования протокола HTTPS, достаточно передать команде `run-app` или `run-war` ключ `--https`. После этого вы сможете получить доступ к приложению по адресу URL `https://localhost:8443/...`, а также по более привычному адресу `http://localhost:8080/...`.

Если вы запустите приложение Hubbub или свое собственное приложение таким способом, вы заметите, что ваш броузер будет не очень счастлив при обращении к нему. На рис. 10.2 показано, что думает броузер Firefox о приложении Hubbub, работающем по протоколу HTTPS. Проблема заключается в том, что платформа Grails по требованию создает *самоподписанный* сертификат SSL. В сущности, ваше приложение говорит: «верьте мне, я действительно то, за что я себя выдаю». В данном случае вы знаете, что можете доверять приложению, потому что вы сами запустили сервер, но как в этом смогут убедиться настоящие пользователи?

Чтобы избежать появления подобных сообщений в броузере, вам необходимо получить сертификат SSL, подписанный компанией, выдающей сертификаты (Certificate Authority, CA), такой как VeriSign, Thawte, Comodo или Go Daddy. Большинство (если не все) броузеров распознают и доверяют этим компаниям по умолчанию, поэтому когда пользователи попадают на веб-сайт, имеющий сертификат, подписанный одной из этих компаний, они видят изображение запертого замка (или другой признак, говорящий о безопасном соединении с сайтом) и не получают предупреждающих сообщений или сообщений об ошибках.

Как получить сертификат, подписанный одной из этих компаний? На сайтах компаний даются четкие инструкции, но в общем виде процедура выглядит так:

1. С помощью команды `keytool`, входящей в состав JDK, создать частный ключ для своего сервера.

2. Сгенерировать запрос на получение подписанного сертификата (Certificate Signing Request, CSR).
3. Передать запрос компании, выдающей сертификаты.
4. Дождаться, когда компания пришлет вам подписанный сертификат, и импортировать его в свое хранилище ключей.

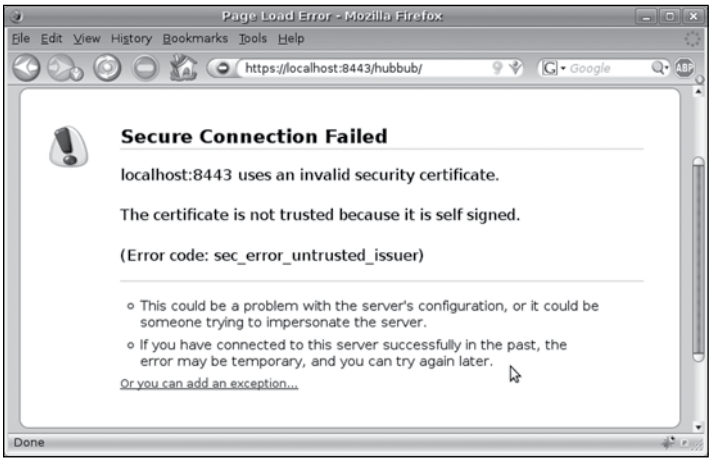


Рис. 10.2. *Попытка доступа к приложению Hubbub по протоколу HTTPS с помощью броузера Firefox 3. Броузер не доверяет самоподписанным сертификатам, но вы можете принять его и добавить в исключения (ссылка внизу окна).*

Использование протокола SSL является, пожалуй, самым знакомым для разработчиков веб-приложений способом обеспечить шифрование данных, но это не означает, что протокол SSL применим только для веб-приложений. Если вам когда-либо придется задуматься о реализации шифрования или электронной подписи (являющейся одной из разновидностей шифрования) вручную, ознакомьтесь со следующими рекомендациями:

- Используйте стандартные алгоритмы шифрования и хеширования – не пытайтесь реализовать свой собственный!
- Не используйте алгоритмы с известными уязвимостями, такие как DES, SHA-1 и MD5.
- Не передавайте частные ключи!

Как вы увидите ниже, упомянутые алгоритмы SHA-1 и MD5 не являются алгоритмами шифрования – это хеш-функции, создающие представление фиксированной длины для указанных данных. Такое представление называется «хешем», или «дайджестом». Суть применения хешей состоит в том, что в случае изменения данных должен измениться и их хеш. Это позволяет определить, не были ли данные подделаны,

в том случае, если известен оригинальный хеш. Подумайте об использовании алгоритма SHA-256 или SHA-512 вместо SHA-1 и MD5. На данный момент для этих алгоритмов не обнаружено уязвимостей, и они могут использоваться так же, как и более старые хеш-функции.

Примечание

Если вам интересно узнать, почему алгоритмы SHA-1 и MD5 считаются уязвимыми, начните с Википедии, где имеются статьи для этих двух функций.

Мы уже использовали хеш-функцию в приложении Hubbub для организации хранения паролей пользователей, и это достаточно распространенный прием. Если веб-сайт сбрасывает ваш пароль вместо того, чтобы прислать его вам, значит, скорее всего, он хранит пароли именно таким способом – из хеша невозможно получить оригинальные данные (пароль в данном случае). Это алгоритм одностороннего шифрования. Основная причина такого использования функции хеширования состоит в том, что даже если кто-то получит доступ к базе данных, он не сможет использовать пароль, чтобы выдать себя за другого пользователя.

Это был достаточно краткий обзор криптографии, и мы лишь слегка коснулись поверхности. Для большинства людей использование SSL позволяет решать любые проблемы, связанные с обеспечением конфиденциальности. Достаточно лишь убедиться, что пользователи не смогут получить доступ к конфиденциальной информации по протоколу HTTP (не обладающему возможностью шифрования). Прежде чем двинуться дальше, мы хотели бы упомянуть сайт The Legion of the Bouncy Castle (*http://www.bouncycastle.org/*), где можно найти открытую библиотеку Java Cryptography Extension (JCE), поддерживающую несколько алгоритмов шифрования, таких как Blowfish. В этой библиотеке имеются даже классы для работы с OpenPGP.

Мы охватили три области, которым требуется уделять особое внимание, но осталась еще одна. Она не является уязвимой, но не менее важна.

10.2.4. Не отдавайте информацию

Представьте на мгновение, что вы один из тех злоумышленников, которые желают взломать приложение. Что бы вы сделали? Вы могли бы попробовать провести некоторые разновидности атак типа «инъекция», с которыми мы уже познакомились, но разработчик закрыл эти бреши. Значит, вам придется попробовать множество различных атак, чтобы отыскать уязвимое место.

Другой подход – попробовать определить, на какой платформе базируется веб-приложение. Если это известно, можно сузить круг поисков вариантов взлома, ограничившись известными уязвимостями этой платформы. Определить платформу достаточно просто. Взгляните на следующий адрес URL: *http://www.somecompany.com/orders/buy.php*.

Если только это не какой-то хитрый ход, можно смело предположить, что данный веб-сайт основан на технологии PHP.

Приложения на платформе Grails не позволяют делать такие очевидные выводы, но и у них есть определенные недостатки. Попробуйте при запущенном приложении Hubbub ввести в адресной строке броузера такой адрес URL: *http://localhost:8080/hubbub/path/unknown*.

Вы увидите результат, изображенный на рис. 10.3: теперь любой атакующий знает, что приложение Hubbub является веб-приложением на языке Java, работающим под управлением веб-сервера Jetty. Хуже того, если ваше приложение возбудит исключение, платформа Grails отобразит стандартную страницу с сообщением об ошибке. Благодаря этому атакующий будет знать, что приложение основано на платформе Grails.

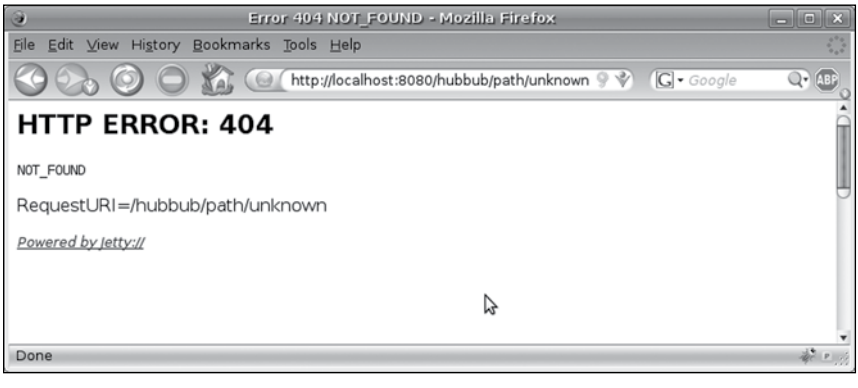


Рис. 10.3. *По умолчанию при обращении к неизвестной странице отображается информация о платформе. Попробуйте предотвратить получение такой полезной информации атакующими.*

Вернемся к главе 5, где была представлена особенность, которая может помочь в этой ситуации: отображение кодов ответов в контроллеры. В идеале было бы желательно, чтобы все ответы с кодами ошибок, которые обычно генерируются платформой Grails, отображались на свои собственные представления:

```
class UrlMappings {
    ...
    “403”(controller: “errors”, action: “forbidden”)
    “404”(controller: “errors”, action: “notFound”)
    “500”(controller: “errors”, action: “internalError”)
}
```

В следующей главе вы увидите, что существует масса других кодов состояния HTTP, которые можно было бы отобразить, но вообще отображать имеет смысл только те коды, которые используются вашим приложением. Если какой-либо частью приложения используются другие

коды состояния, отличные от указанных здесь, добавьте соответствующие записи в отображение.

К сожалению, на этом история не заканчивается. Некоторые части инфраструктуры GSP в настоящее время жестко запрограммированы на отображение представления grails-app/views/error.gsp в случае появления исключения в странице GSP. При этом механизм, который мы только что рассматривали, оказывается незадействованным, и приложение на весь мир объявляет, что оно реализовано на платформе Grails. Чтобы ликвидировать эту проблему, мы рекомендуем вам модифицировать страницу GSP так, чтобы она отправляла ошибку с кодом «500» в режиме эксплуатации:

```
<%@ page import="grails.util.Environment" %>
<g:if test="${Environment.current == Environment.PRODUCTION}">
    ${response.sendError(500)}
</g:if>
<g:else>
    ...
</g:else>
```

Это изменение означает, что действующее приложение не будет сообщать всем и вся сведения об используемой платформе, однако вы попрежнему сможете наблюдать полезные сообщения об ошибках с трассировочной информацией при разработке или тестировании.

Теперь можете вздохнуть с облегчением – мы закончили обзор типичных уязвимостей веб-приложений и приемов их устранения. Теперь вы знаете достаточно, чтобы суметь ликвидировать типичные утечки; к тому же у вас теперь имеются сложившийся образ мышления и комплект инструментальных средств, которые помогут вам укрепить безопасность.

Все, что мы обсуждали в этом разделе, и некоторые дополнительные сведения можно найти в статье «CWE/SANS Top 25 Most Dangerous Programming Errors» (*http://cwe.mitre.org/top25/*) и в статье «Top 10» на сайте проекта информационной безопасности OWASP (*http://www.owasp.org/index.php/Top_10_2007*). Вам следует обязательно посетить эти веб-сайты – здесь вы найдете массу информации об опасностях, поджидающих разработчиков приложений. Кроме того, на сайте проекта OWASP вы найдете несколько полезных инструментов, таких как Scarab, которые позволят автоматизировать процесс тестирования вашего приложения на наличие типичных уязвимостей.

Прежде чем мы перейдем к другому существенному аспекту обеспечения безопасности приложений, мы должны упомянуть о том, что регулярные обзоры программного кода – отличный способ борьбы с уязвимостями. Пока целый коллектив просматривает программный код с позиции безопасности, ваше приложение остается в безопасности.

Защита от злоумышленников – самая важная составляющая системы безопасности приложения, и если в процессе разработки программного кода вы постоянно будете помнить о возможных уязвимостях, вы сможете существенно уменьшить риск их появления. Если какая-то часть вашего приложения должна быть доступна только какому-то определенному пользователю или группе пользователей, вам необходимо гарантировать, что никто не сможет получить доступ к той части приложения, на работу с которой у него нет прав.

10.3. Управление доступом

Во многих приложениях возникает необходимость идентификации пользователя, либо потому что некоторая информация должна быть доступна только определенному (известному) человеку, либо потому что необходимо ограничить доступ к некоторой их функциональности. Возможно, в приложении потребуется следить за тем, кто и что делает. В приложении Hubbub, например, было бы нежелательно, чтобы анонимные пользователи могли оставлять сообщения, поэтому приложение Hubbub должно знать, кто их создает. Кроме того, мы должны гарантировать, что пользователь сможет изменять только свой собственный профиль. Никто другой не должен иметь такой возможности, за исключением, может быть, системного администратора.

Именно в этом заключается суть управления доступом. Область применения управления доступом широка: от простого гарантирования, что приложением смогут пользоваться только настоящие люди (в противоположность «ботам»), до использования сложных правил, основанных на многочисленных разрешениях, проектах и группах. Приложение Hubbub попадает в первую категорию, но мы постараемся предоставить достаточно информации, чтобы вы могли двигаться дальше, опираясь на основы.

> **Терминология**
>
> Термины «аутентификация» и «авторизация» получили широкое распространение и обозначают практически то же, что и термин «управление доступом», но последний пользуется большим предпочтением в кругах, связанных с безопасностью. В различных платформах безопасности также используются свои собственные термины, что может существенно осложнять их сравнение. Если вы испытываете сомнение относительно термина, выбирайте тот, который кажется вам более понятным.

Проблема управления доступом включает в себя выяснение ответов на два основных вопроса:

- Действительно ли человек, участвующий во взаимодействии, является тем, за кого себя выдает (аутентификация)?
- Имеет ли он право выполнять данное действие (авторизация)?

Простое управление доступом может быть реализовано посредством стандартных фильтров и сеансов платформы Grails, однако практически всегда предпочтительнее использовать одно из доступных расширений системы безопасности. Зачем повторно изобретать колесо? Ниже перечислены основные кандидаты на использование:

- *Расширение Authentication*. Расширение Authentication – это легковесная реализация аутентификации, не зависящая от сторонних библиотек. Оно отлично подходит для веб-сайтов, где необходимо обеспечить возможность регистрации пользователей, но без сложных требований, связанных с авторизацией.
- *Расширение JSecurity*. Расширение JSecurity обеспечивает богатые возможности управления доступом и шифрования посредством библиотеки Apache JSecurity (в настоящее время проект носит название Incubator). Несмотря на то что расширение JSecurity пока не получило широкой известности, вот уже на протяжении нескольких лет оно используется в действующих системах.
- *Расширение Spring Security (Acegi)*. Расширение Spring Security, ранее называвшееся Acegi Security, – хорошо известная и широко распространенная платформа обеспечения безопасности для веб-приложений на языке Java. Оно обеспечивает наиболее полный комплект инструментов аутентификации среди трех расширений, перечисленных здесь. Начинающие разработчики часто находят это расширение чересчур трудным для понимания, однако оно значительно преуспело в сокрытии наиболее сложных моментов.

Мы не включили в этот список пару стандартных возможностей: компонент Java Authentication and Authorization Service (JAAS), входящий в состав JDK, и реализацию управления доступом J2EE, предоставляемую контейнерами J2EE. Несмотря на то что они являются стандартными, они не получили широкого распространения в сообществе Grails; кроме того, JAAS является довольно сложным в изучении и использовании. Однако если вам потребуется использовать их, документация для обоих компонентов вполне применима к приложениям на платформе Grails.

В веб-приложениях расширения действуют подобно привратникам, как показано на рис. 10.4. Они решают, можно ли передавать тот или иной запрос приложению, опираясь на определенные правила. Если запрос считается допустимым, расширение отображает ответ приложения. В противном случае он может потребовать от клиента зарегистрироваться или может отобразить страницу «доступ запрещен».

Мы не в состоянии охватить все эти расширения в данной главе, поэтому мы сконцентрируем наше внимание на расширении Spring Secu-

rity. Большинство идей, которые мы будем обсуждать, в равной степени применимы и к другим расширениям, поэтому у вас не должно возникать никаких неприятностей при их использовании и изучении их документации.

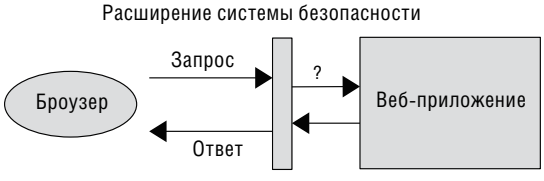


Рис. 10.4. *Порядок участия типичного расширения системы безопасности платформы Grails в цикле запрос/ответ. Под знаком вопроса «?» подразумевается, что запрос может передаваться или не передаваться приложению по усмотрению расширения*

10.3.1. Введение в расширение Spring Security

Управление доступом невозможно без определения текущего пользователя. В настоящее время приложение Hubbub обеспечивает это определение, требуя от пользователя введения своего идентификатора и пароля на странице регистрации и сохраняя соответствующий экземпляр объекта `User` в рамках сеанса HTTP (при условии, что процедура аутентификации завершилась успехом). Нам необходимо заменить данный простой механизм более функциональным расширением Spring Security.

Сначала требуется установить расширение:

```
grails install-plugin acegi
```

Имя расширения может показаться немного странным, но оно приобретет некоторый смысл, когда вы узнаете, что расширение было создано, когда платформа носила название Acegi Security.

Теперь, после того, как расширение было установлено, у вас есть доступ к небольшой коллекции сценариев, один из которых занимается созданием классов предметной области, необходимых для расширения Spring Security. Без них вы не сможете осуществлять управление доступом.

Если бы вы создавали приложение, ориентированное на использование расширения Spring Security с самого начала, этот процесс выглядел бы более просто:

```
grails create-auth-domains org.example.User org.example.Role
➥ org.example.Requestmap
```

Эта команда создает три класса в каталоге grails-app/domain/org/example: `User`, `Role` и `Requestmap`. Подробнее мы расскажем о них ниже; сначала нам необходимо учесть тот факт, что класс `User` у нас уже существу-

ет, и мы хотели бы сохранить его. Поэтому вместо предыдущей команды мы воспользуемся следующей:

```
grails create-auth-domains DummyUser
➥ com.grailsinaction.Role Requestmap
```

В этом случае существующий класс `User` не будет перезаписан, однако теперь расширение настроено на получение информации о пользователе из класса `DummyUser`.

Не беспокойтесь, эту проблему легко исправить. Расширение может использовать любой класс предметной области для пользователя, пока он содержит поля с именем пользователя и паролем, с признаком, свидетельствующим, что учетная запись может или не может использоваться, и с коллекцией разрешений или ролей. В нашем классе `User` уже имеются первые два поля, поэтому нам нужно лишь добавить два недостающих поля:

```
class User {
    String userId
    String password
    boolean enabled

    static hasMany = [ authorities: com.grailsinaction.Role,
                ... ]

    static belongsTo = com.grailsinaction.Role
    ...
}
```

Нам необходимо также исправить класс `Role`, поскольку сейчас он связан с классом `DummyUser`; мы изменяем поле отношения, как показано ниже:

```
class Role {
    static hasMany = [people: User]
    ...
}
```

Поскольку у нас имеется свой собственный класс `User`, нам больше не требуется класс `DummyUser`. Его удаление избавит нас от лишнего беспорядка в приложении. Поскольку автоматически сгенерированные страницы не будут действовать без класса предметной области, мы должны удалить также контроллер `DummyUserController` и соответствующие ему представления (grails-app/views/dummyUser).

Классы `User` и `Role` формируют основу системы управления доступом, определяющей, кому и какие действия доступны. Это достаточно простая модель, как показано на рис. 10.5. Последнее, что осталось сделать, – изменить реализацию класса `BootStrap`, чтобы присвоить всем предопределенным пользователям права или роли. Пользователи, не

наделенные необходимыми правами, не смогут зарегистрироваться. Как выглядит соответствующий код, нам пока не важно, но вы сможете увидеть изменения ниже в примерах программного кода приложения Hubbub для этой главы. Кроме того, в примерах вы увидите, что мы используем объект `authenticateService` расширения для шифрования пароля перед сохранением его в базе данных:

```
authenticateService.encodePassword("password")
```

Сейчас мы двинемся дальше, но не надо беспокоиться. Мы еще вернемся к этой службе ниже.

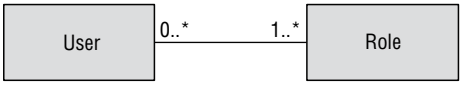


***Рис. 10.5.** Отношение между основными классами предметной области, используемыми системой управления доступом. Модель данных имеет очень простой вид*

Далее нам необходимо настроить расширение Spring Security на работу с нашим собственным классом `User`. Когда вы запустили команду `create-auth-domains`, она тайком создала дополнительный файл в каталоге grails-app/conf: SecurityConfig.groovy. Этот файл позволяет настраивать многие аспекты расширения Spring Security; для этого файла даже имеется отдельная страница со справочной информацией: *http://www.grails.org/AcegiSecurity+Plugin+-+Customizing+with+SecurityConfig*.

В настоящее время содержимое файла SecurityConfig.groovy в приложении Hubbub имеет следующий вид:

```
security {
    active = true
    loginUserDomainClass = "DummyUser"
    authorityDomainClass = "com.grailsinaction.Role"
    requestMapClass = "Requestmap"
}
```

Последние три значения должны быть знакомы вам, потому что именно они передавались команде `create-auth-domains`. Вы, вероятно, уже догадались, что мы сделаем теперь: мы заменим значение `loginUserDomainClass` на `com.grailsinaction.User`. Вуаля! Теперь расширение Spring Security будет использовать наш собственный класс `User`.

Внимание

Будьте внимательны при обращении с параметром настройки `active`: если присвоить ему значение `false`, работа Spring Security будет запрещена, и вы потеряете возможность управлять доступом! Что еще более важно, то же произойдет, если удалить этот параметр из файла или сам файл.

Но это еще не все. По умолчанию расширение ожидает, что класс пользователя будет содержать поля с именами `username` и `passwd`, но у нас эти поля имеют имена `userId` и `password`. Чтобы исправить эту проблему, добавим следующие два параметра в файл SecurityConfig.groovy:

```
userName = "userId"
password = "password"
```

Пока все замечательно. Расширение Spring Security пока еще ничего не делает, но оно настроено и готово к работе. Все, что нам осталось сделать, – выбрать цель и выстрелить, или, точнее говоря, настроить адреса URL, которые требуется защитить.

10.3.2. Защита адресов URL

Начнем с самой простой модели защиты приложения Hubbub: правом доступа ко всем страницам обладают только аутентифицированные пользователи, *исключение* составляет домашняя страница, которая должна быть доступна любому желающему. Реализовать это с помощью расширения можно тремя способами:

- *Статически* – определив правила в файле SecurityConfig.groovy.
- *Динамически* – сохранив правила в базе данных в виде объекта `Requestmaps`.
- *В виде аннотаций* – объявив правила в контроллерах и службах с помощью аннотаций.

Теперь вы знаете, зачем нужен класс предметной области `Requestmap`! Он отображает адреса URL на разрешения, или «права доступа» в терминологии Spring Security. Этот прием удобно использовать, когда требуется изменить настройки безопасности после развертывания приложения. Аннотации позволяют обезопасить контроллеры независимо от адресов URL, используемых для доступа к ним. Их удобно использовать также для управления доступом к методам служб.

Однако в приложении Hubbub мы будем использовать старомодный способ статической настройки отчасти потому, что в этом случае все настройки безопасности хранятся в одном месте, а отчасти потому, что нам нужно будет поговорить о порядке определения правил. Поскольку мы не предполагаем применять динамические правила защиты адресов URL, мы можем удалить класс `Requestmap`, а также соответствующие ему контроллер и представления.

Посмотрим теперь на настройки, которые мы будем использовать для реализации требований, предъявляемых к управлению доступом. В листинге 10.1 приводится содержимое файла SecurityConfig.groovy после добавления правил. Эти правила гарантируют, что к домашней странице и к странице регистрации смогут обращаться все пользователи, но все остальные страницы приложения смогут просматривать только

аутентифицированные пользователи. Кроме того, мы гарантируем свободный доступ ко всем файлам JavaScript, CSS и изображениям.

Листинг 10.1. Настройки расширения Spring Security для простой модели управления доступом в приложении Hubbub

```
security {
    active = true
    cacheUsers = false ← Запрещает использовать кэш Hibernate

    loginUserDomainClass = "com.grailsinaction.User"
    authorityDomainClass = "com.grailsinaction.Role"
    userName = "userId"
    password = "password"

    useRequestMapDomainClass = false ← ❶ Запрещает динамические правила
    requestMapString = """\
CONVERT_URL_TO_LOWERCASE_BEFORE_COMPARISON
PATTERN_TYPE_APACHE_ANT
/=IS_AUTHENTICATED_ANONYMOUSLY
/login/auth=IS_AUTHENTICATED_ANONYMOUSLY ← ❷ Правила доступа к адресам URL
/js/**=IS_AUTHENTICATED_ANONYMOUSLY
/css/**=IS_AUTHENTICATED_ANONYMOUSLY
/images/**=IS_AUTHENTICATED_ANONYMOUSLY
/plugins/**=IS_AUTHENTICATED_ANONYMOUSLY
/**=IS_AUTHENTICATED_FULLY
"""
}
```

Первая половина файла содержит определения, которые вы уже видели выше, а вторая половина – это то, что интересует нас прямо сейчас. Чтобы использовать статические настройки, нам сначала необходимо отключить обработку динамических настроек в объекте `Requestmap` ❶. Далее следуют сами правила защиты адресов URL в виде многострочного текста, присваиваемого параметру `requestMapString` ❷. Обратите внимание, что каждое правило располагается в отдельной строке.

Первые две строки в правилах – это общие настройки, которые обычно включаются во все конфигурации расширения Spring Security. Первая из них вынуждает Spring Security приводить все адреса URL к нижнему регистру символов перед сравнением их с правилами. Например, такой адрес URL

```
http://localhost:8080/hubbub/starPost/showAll
```

перед сравнением превратится в

```
http://localhost:8080/hubbub/starpost/showall
```

Следовательно, правила в `requestMapString` должны содержать только символы нижнего регистра. Вторая общая настройка указывает, что расширение Spring Security будет использовать стандартный для Ant

способ сопоставления адресов URL с правилами. Это означает, что для определения правила для всех файлов .js следует использовать шаблон `/**/*.js`, который указывает, что он соответствует любому файлу с расширением .js, находящемуся в любом каталоге.

За общими настройками следуют необходимые нам правила. Поскольку нам требуется, чтобы домашняя страница была доступна всем желающим, первое правило связывает определение `IS_AUTHENTICATED_ANONYMOUSLY` с корневым адресом URL (/). Нам необходимо также обеспечить свободный доступ к странице регистрации (/login/auth), в противном случае никто не сможет зарегистрироваться! Статические ресурсы, такие как файлы JavaScript, CSS и изображения (предоставляемые приложением и расширениями), также находятся в свободном доступе. В заключение мы разрешаем доступ ко всем остальным адресам URL только аутентифицированным пользователям, то есть тем, кто благополучно зарегистрировался.

Особое внимание следует уделять порядку следования правил, потому что расширение Spring Security применяет первое правило, соответствующее запрошенному адресу URL. Например, если последнее правило сделать первым (/**), доступ ко всем страницам смогут получить только аутентифицированные пользователи независимо от того, какие правила следуют ниже. Чем более точный шаблон адреса URL используется в правиле, тем выше в списке оно должно находиться.

Теперь приложение Hubbub защищено от анонимного доступа. Если пользователь не имеет учетной записи, он не сможет оставлять сообщения. Попробуйте сделать это, запустив сервер и введя в броузере адрес *http://localhost:8080/hubbub/post/timeline*. Вы автоматически будете перенаправлены на страницу регистрации расширения, как показано на рис. 10.6.

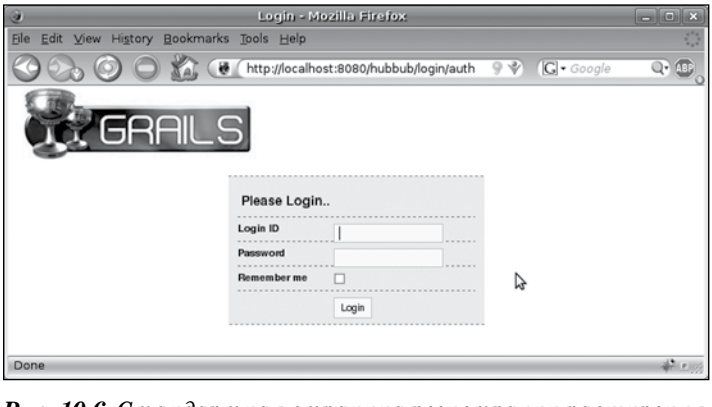


Рис. 10.6. *Стандартная страница регистрации расширения Spring Security с флажком «Remember me» (Запомнить)*

Вы можете попробовать зарегистрироваться, воспользовавшись одной из существующих учетных записей, таких как `glen` с паролем `password`, но это вызовет ошибку на стороне сервера. Нам все еще необходимо интегрировать процедуру регистрации с остальной частью приложения, которое в настоящее время ожидает найти экземпляр объекта `User` в области видимости сеанса.

Прежде чем мы сделаем это, давайте посмотрим на флажок Remember me (Запомнить) на странице регистрации расширения. Мы уверены, что эта особенность знакома вам по различным веб-сайтам, и она прекрасно подходит для использования в приложении Hubbub. Приложение не нуждается в обеспечении высокого уровня безопасности, поэтому мы можем позволить своим пользователям получать доступ к приложению без выполнения всякий раз процедуры регистрации. Проблема состоит в том, что, вернувшись к приложению Hubbub спустя некоторое время, вы обнаружите, что не можете получить доступ к нему. Броузер приходит в замешательство, потому что его снова и снова перенаправляют на одну и ту же страницу. Расширение Spring Security обнаруживает, что информация о пользователе сохранена и обходит страницу регистрации. К сожалению, для доступа к большинству страниц приложения требуется полная аутентификация пользователя, поэтому происходит перенаправление на страницу /login/denied. Но эта страница тоже требует, чтобы пользователь был аутентифицирован; мы попали в замкнутый круг!

Внимание

Никогда не позволяйте пользователям выбирать флажок Remember me (Запомнить), если приложение не поддерживает такую возможность. Если вы используете страницу регистрации, предоставляемую расширением, удалите флажок Remember me (Запомнить).

Нам требуется обеспечить поддержку флажка Remember me (Запомнить) в приложении Hubbub, и для этого мы должны изменить правило

```
/**=IS_AUTHENTICATED_FULLY
```

как показано ниже:

```
/**=IS_AUTHENTICATED_REMEMBERED
```

Вот и все! Теперь поддержка флажка Remember me (Запомнить) будет действовать, и броузер не попадет в замкнутый круг.

Теперь, когда мы разобрались в нашей проблеме, можно попробовать улучшить управление доступом к приложению Hubbub.

10.3.3. Сохранение информации о текущем пользователе

Управление доступом выглядит многообещающим, и при этом нам не приходится много трудиться. Пока управление доступом не достаточно работоспособно, но этот недостаток можно быстро исправить. Первое, что нам необходимо сделать, – исправить страницу со списком сообщений пользователя, которая в настоящий момент генерирует ошибку с кодом «500», потому что зарегистрировавшийся пользователь пока не имеет права доступа к ней.

Решить эту проблему можно двумя способами. Во-первых, в момент регистрации пользователя мы можем сохранить экземпляр класса `User` в области видимости сеанса, для чего потребуется изменить реализацию автоматически созданного контроллера, выполняющего регистрацию (grails-app/controllers/LoginController.groovy). Но это излишне, потому что все необходимое можно получить непосредственно от расширения Spring Security. Зачем расходовать ценное пространство области видимости сеанса, если прямой необходимости в этом нет? Так что мы воспользуемся вторым способом: изменим программный код приложения таким образом, чтобы получать информацию о пользователе от расширения Spring Security.

Один из способов извлечь эту информацию состоит в том, чтобы получить текущий контекст Spring Security из класса `SecurityContextHolder` и оттуда извлечь информацию о пользователе. Этот контекст содержит всю информацию системы безопасности, имеющую отношение к текущему пользователю, поэтому в нем находятся практически все сведения, которые только могут потребоваться. Но работать с контекстом не особенно удобно, поэтому расширение предоставляет удобную службу аутентификации. Эта служба содержит массу полезных методов, большая часть которых перечислена в табл. 10.1.

Таблица 10.1. Перечень полезных методов, предоставляемых службой аутентификации расширения

Метод	Описание
`isLoggedIn()`	Возвращает `true`, если текущий пользователь зарегистрировался; в противном случае возвращается `false`
`ifAllGranted(roles)`	Возвращает `true`, если текущий пользователь обладает всеми указанными ролями; в противном случае возвращается `false`. Аргумент `roles` – это список имен ролей, разделенных запятыми, в виде строки
`ifNotGranted(roles)`	Действует подобно методу `ifAllGranted()`, но возвращает `true`, если пользователь не обладает ни одной из указанных ролей

`ifAnyGranted(roles)`	Действует подобно методу `ifAllGranted()`, но возвращает `true`, если пользователь обладает хотя бы одной из указанных ролей
`userDomain()`	Возвращает объект предметной области, представляющий текущего пользователя
`encodePassword(pwd)`	Шифрует данный пароль с помощью указанного в настройках алгоритма и возвращает полученный хеш; это хеш может сохраняться в поле `password` класса `User`
`isAjax(request)`	Возвращает `true`, если данный запрос класса `HttpServletRequest` выглядит, как запрос AJAX

Как вы уже, наверно, поняли, для получения экземпляра класса `User` мы будем использовать метод `userDomain()`. Единственное, на что следует обратить внимание, – метод возвращает *обособленный* экземпляр, то есть он не привязан к сеансу Hibernate. Это означает, что мы не сможем получить доступ к каким-либо его коллекциям или сохранить изменения в нем, пока не присоединим его к сеансу. В данном случае, так как нам требуется получить самую свежую информацию о пользователе, мы будем повторно извлекать ее из базы данных.

Ниже приводятся изменения, которые нужно внести в контроллер сообщений:

```
class PostController {
    def authenticateService
    ...
    def list = {
        def user = authenticateService.userDomain()
        user = User.get(user.id)
        ...
    }
    ...
    def add = {
        def content = params.postContent
        if (content) {
            def user = authenticateService.userDomain()
            user = User.get(user.id)

            if (user) {
                ...
            }
        }
        ...
    }
    ...
}
```

Служба аутентификации расширения будет внедрена платформой Grails в свойство `authenticateService`, после чего мы сможем использовать ее для получения информации о текущем пользователе.

Этим мы исправили проблему в контроллере, но если вы теперь попытаетесь обратиться к странице /post/timeline, вы по-прежнему получите сообщение об ошибке. Дело в том, что шаблон GSP боковой колонки все еще пытается получить информацию о пользователе непосредственно из сеанса HTTP. Первое, что приходит в голову, – реализовать обращение к службе аутентификации в шаблоне (весьма неуклюжее и не очень хорошее решение) или добавить информацию о пользователе в модель, возвращаемую действием (более хорошее решение). К сожалению, шаблон боковой колонки используется в макете, поэтому все представления, использующие этот макет, будут требовать, чтобы модель содержала объект с информацией о пользователе.

К счастью, мы можем не волноваться об этой проблеме, потому что расширение предоставляет библиотеку тегов, которая реализует все, что нам требуется. Вместо такой реализации

```
<dd>${session.user.userId}</dd>
```

наш шаблон grails-app/views/post/_sidebar.gsp может использовать специальный тег, позволяющий выводить любые поля объекта с информацией о пользователе:

```
<dl>
    <dt>User</dt>
    <dd><g:loggedInUserInfo field="userId">Guest</g:loggedInUserInfo>
</dl>
```

Этот тег можно также использовать как метод, как показано в примере ниже:

```
<g:createLink action="following" controller="friend"
          id="${loggedInUserInfo(field: 'userId')}"/>
```

После внесения этих изменений приложение Hubbub с ревом вернется к жизни.

Пока мы не ушли далеко от темы тегов GSP, отметим, что мы можем добавить следующий блок в любой наш макет, чтобы дать пользователям регистрироваться в любой момент:

```
<g:isLoggedIn>
    <g:link controller="logout" action="index">sign out</g:link>
</g:isLoggedIn>
```

Тег `isLoggedIn` также предоставляется расширением. Если пользователь уже зарегистрировался с помощью Spring Security, этот тег просто выводит свое содержимое. В данном случае его содержимое – это ссылка на контроллер `logout`, созданный расширением ранее (в каталоге приложения grails-app/controllers).

Теперь управление доступом действует, как ожидалось. Только аутентифицированные (или сохраненные) пользователи имеют доступ к странице /post/timeline, а новые сообщения корректно ассоциируются с текущим зарегистрированным пользователем. Мы можем даже вывести ссылку, позволяющую пользователю выходить из приложения. Что нам еще осталось?

Одна из проблем состоит в том, что теперь у нас имеются две страницы регистрации, и было неплохо избавиться от одной из них. Сохраним нашу прежнюю форму регистрации, которая находится на домашней странице.

10.3.4. Использование собственной страницы регистрации

Настройка расширения на использование нашей домашней страницы в качестве страницы регистрации выполняется просто – достаточно добавить следующие строки в файл SecurityConfig.groovy:

```
loginFormUrl = "/"
defaultTargetUrl = "/post/timeline"
```

Первый параметр обеспечивает перенаправление на домашнюю страницу, когда потребуется выполнить процедуру аутентификации, а второй предписывает расширению Spring Security загружать страницу с сообщениями пользователя в случае успешной регистрации. После того как пользователь пройдет аутентификацию на домашней странице, расширение Spring Security перенаправит его обратно на оригинальную страницу.

К сожалению, это еще не все, что мы должны сделать. Существующая форма регистрации не будет действовать, как нам требуется, потому что для правильной ее работы поля формы должны иметь специальные имена, которые будут распознаваться расширением Spring Security. Установка этих имен предоставит нам как дополнительное преимущество возможность добавить флажок Remember me (Запомнить). Ниже приводится новая реализация формы в файле grails-app/views/_sidebar.gsp:

```
<form action="${resource(file: 'j_spring_security_check')}" method="POST">
    <table>
        <tr><td>User Id:</td><g:textField name="j_username"/></td></tr>
        <tr>
            <td>Password:</td>
            <td><input name="j_password" type="password"/></td>
        </tr>
        <tr>
            <td>Remember:</td>
            <td><input type="checkbox"
                       name="_spring_security_remember_me"/></td>
        </tr>
```

```
        <tr><td colspan="2"><g:submitButton name="login" value="Login"/></
td>
    </table>
</form>
```

Критически важными элементами в этом фрагменте являются адрес URL в атрибуте `action` и имена полей для ввода идентификатора пользователя, пароля и флажка Remember me (Запомнить).

Теперь, когда все действует, как надо, и мы реализовали простое управление доступом, можно посмотреть, как его можно протестировать. Нам действительно необходимо убедиться, что все действует, как требуется, и что в ходе дальнейшей разработки мы ничего не нарушим!

10.3.5. Тестирование управления доступом

Расширение Spring Security, как и другие платформы обеспечения безопасности, для управления доступом к приложению использует комбинации фильтров сервлетов и других способов перехвата запросов. Это осложняет тестирование, поскольку не существует классов, на основе которых можно было бы реализовать модульный тест, а фильтры не участвуют в интеграционных тестах. Оба типа тестов не пригодны для тестирования системы управления доступом. Единственным средством являются функциональные тесты.

На рис. 10.7 изображена логика, которую нам требуется протестировать. Схема не очень подробная, но она в достаточной степени иллюстрирует, чего мы хотим добиться с помощью тестов. Мы хотим убедиться, что при любых условиях будут отображаться именно те страницы, которые должны.

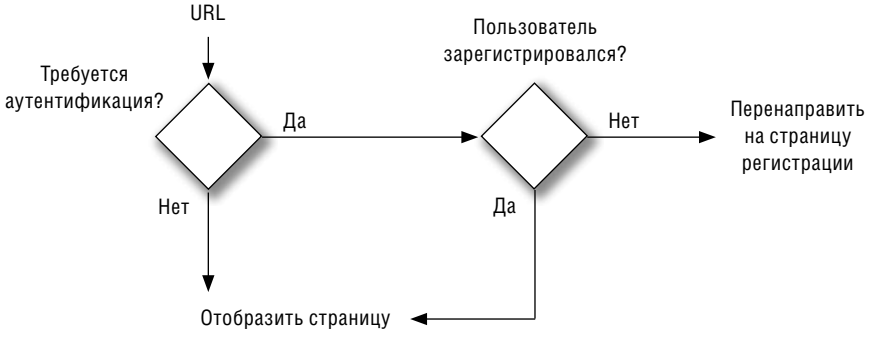


Рис. 10.7. *Простая логическая последовательность управления доступом, которую требуется проверить с помощью функционального теста. Ожидаемое поведение зависит от адреса URL запроса и от того, зарегистрирован ли пользователь*

Написать тест, подтверждающий поведение, изображенное на диаграмме, достаточно просто, и его реализацию вы можете видеть в листин-

ге 10.2. Как и в главе 7, для организации доступа к определенным адресам URL мы используем расширение Functional Test, только сейчас некоторые из этих адресов защищены системой управления доступом.

Листинг 10.2. Функциональный тест для проверки системы управления доступом

```
package com.grailsinaction

class AccessControlTests extends functionaltestplugin.FunctionalTestCase {
    void testAnonymousAccess() {
        get("/")                    <- ❶ Разрешается анонимный доступ
        checkHomePage()
    }

    void testTimeline() {
        get("/post/timeline")       <- ❷ Анонимный доступ отклоняется
        checkHomePage()

        login("peter", "test")      <- ❸ Проверка неудачной попытки регистрации
        checkHomePage()

        login()
        checkTimelinePage()

        post("/logout")
        get("/post/timeline")       <- ❹ Проверка работы процедуры входа
        checkHomePage()
    }

    void checkHomePage() {
        assertTitle "Hubbub » Howdy"
    }

    void checkTimelinePage() {
        assertTitle "Hubbub » New Post"
        assertContentContains "What are you hacking on right now?"
    }

    void login(userId = "peter", password = "password") {
        post("/j_spring_security_check") {
            j_username = userId
            j_password = password
        }
    }
}
```

Сначала мы проверяем наличие у пользователя доступа к домашней странице или к странице регистрации ❶. Таким же способом следует проверять все страницы, допускающие возможность анонимного доступа. Затем выполняется попытка обратиться к защищенной странице,

которой в данном случае является персональная страница пользователя со списком его сообщений. Неавторизованный пользователь должен быть перенаправлен на страницу регистрации ❷, к такому же результату должна приводить и неудачная попытка регистрации ❸.

Тестирование заканчивается проверкой случая успешной регистрации, а также проверкой, что переход по адресу URL для выхода из приложения действительно приводит к выходу ❹. Этим мы преследуем две цели: во-первых, мы проверяем, что реализация выхода из приложения действует, как ожидалось; а во-вторых, гарантируем, что последующие тесты будут производиться с неаутентифицированным пользователем. Мы попытались оставить приложение в том же состоянии, в каком оно находилось в начале теста.

Примечание

Программный код в листинге 10.2 демонстрирует пользу выделения программного кода во вспомогательные методы службы, такие как `login()`. Эти методы могут быть собраны в базовый класс, чтобы все ваши функциональные тесты могли пользоваться их преимуществами.

Поздравляем! Теперь все действует, как надо, и мы получили систему, готовую к дальнейшему совершенствованию. Вспомните, как много нам пришлось потрудиться из-за того, что у нас уже имелись классы предметной области и представления. Все было бы намного проще и быстрее, если бы мы использовали возможности, предоставляемые расширением по умолчанию. С другой стороны, проделав нелегкий путь, мы исследовали несколько аспектов использования платформы Spring Security и расширения, что, несомненно, пригодится вам в будущем.

В оставшейся части главы мы рассмотрим более сложные приемы, которые помогут вам удовлетворить различные требования к безопасности и позволят внести некоторые полезные усовершенствования в приложение Hubbub.

10.4. Дальнейшее исследование Spring Security

Spring Security – это мощная платформа, обеспечивающая различные возможности аутентификации и авторизации, многие из которых поддерживаются расширением. В одной главе просто невозможно обсудить все, что касается Spring Security, – для этого потребовалось бы написать целую книгу. Поэтому мы не будем пытаться сделать невозможное, и в этом разделе мы представим вам наиболее типичные ситуации, требующие особого обращения. Сначала мы восполним некоторые пробелы, существующие в системе управления доступом приложения Hubbub в ее текущем состоянии.

10.4.1. Реализация регистрации пользователя

Возможно, вы уже заметили проблему в управлении пользователями в приложении Hubbub: у нас нет пользователей! В настоящее время класс `BootStrap` создает несколько стандартных учетных записей, однако от приложения Hubbub будет не много пользы, если у нас не будет возможности добавлять новых пользователей.

Мы могли бы создать пользовательский интерфейс управления пользователями на основе классов `User` и `Role`. Расширение может даже помочь нам в этом, предоставляя команду `generate-manager`. Однако не в отношении приложения Hubbub. Пользователей едва ли привлечет перспектива отсылать системному администратору сообщение с просьбой создать учетную запись. Типичное решение заключается в том, чтобы предоставить пользователям возможность самим регистрировать учетные записи; этим мы сейчас и займемся.

И снова в этом нам поможет расширение, предоставив команду, с которой мы начнем реализацию возможности самостоятельной регистрации новых пользователей:

```
grails generate-registration
```

Эта команда создаст все необходимые контроллеры и представления и даже загрузит библиотеки JavaMail и Activation, с помощью которых расширение сможет отправлять по электронной почте подтверждения о регистрации. Более того, созданная форма регистрации будет использовать механизм CAPTCHA (Completely Automatic Public Turing Test to Tell Computers and Humans Apart – полностью автоматический тест Тьюринга для различения компьютеров и людей) для предотвращения возможности регистрации спам-ботов.

Это практически все, что нам необходимо сделать, но у реализации возможности регистрации новых пользователей, предоставляемой расширением, имеется ряд дополнительных требований, которые мы должны удовлетворить. Во-первых, нам требуется добавить в класс `User` следующие поля:

```
class User {
    ...
    String email
    String userRealName
    boolean emailShow = false
    ...
    static constraints = {
        ...
        email(nullable: true, blank: true)
        userRealName(nullable: true, blank: true)
    }
    ...
}
```

Мы добавили дополнительные ограничения, потому что сейчас класс `BootStrap`, создавая учетные записи, не заполняет новые поля, и, следовательно, нам необходимо сделать их необязательными. Это пример целесообразности. Следует отметить, что класс предметной области с информацией о пользователе, создаваемый расширением, по умолчанию включает эти дополнительные поля.

Мы не должны также забывать, что реализация регистрации новых пользователей добавляет несколько дополнительных адресов URL, которые должны быть доступны для анонимных пользователей. По этой причине мы должны добавить следующие два правила в настройки системы безопасности:

```
/captcha/**=IS_AUTHENTICATED_ANONYMOUSLY
/register/**=IS_AUTHENTICATED_ANONYMOUSLY
```

Какой смысл иметь форму регистрации новых пользователей, если доступ к ней будет для них закрыт? Не забывайте, что данные правила должны предшествовать правилам для адресов `/**`; в противном случае доступ к captcha и к страницам регистрации смогут получить только аутентифицированные пользователи.

После внесения этого последнего изменения можно опять запустить приложение Hubbub и обратиться к странице регистрации нового пользователя по адресу *http://localhost:8080/hubbub/register/*. Вы увидите очень простую форму, но с ее помощью вы уже сможете зарегистрировать новую учетную запись и тут же воспользоваться ею. Кроме того, благодаря тому что расширение создает контроллеры и представления в каталоге проекта, вы сможете настроить внешний вид и поведение страниц в соответствии с остальной частью приложения.

Дополнительной особенностью реализации регистрации новых пользователей, о которой мы еще не упомянули, является возможность отправки подтверждений по электронной почте. Вы можете настроить расширение так, чтобы оно отправляло сообщения пользователям, успешно зарегистрировавшим свои учетные записи. По умолчанию эта возможность выключена, но вы можете включить ее в файле с настройками системы безопасности, добавив следующий параметр:

```
useMail = true
```

Затем вам потребуется настроить свой сервер SMTP, чтобы позволить расширению отправлять сообщения. Полный перечень параметров настройки, имеющих отношение к электронной почте, вы найдете на странице с описанием настроек SecurityConfig (*http://www.grails.org/AcegiSecurity+Plugin+-+Customizing+with+SecurityConfig*).

Обратите внимание, что возможности для работы с электронной почтой весьма ограничены и включают в себя только возможность отправки уведомлений. Вы не сможете отправлять сообщения, содержащие код или ссылку для активации соответствующей учетной записи, поэтому

если вам требуется именно это, вам придется реализовать такую особенность самостоятельно. Имеющиеся возможности тем не менее позволили нам обеспечить регистрацию новых пользователей с минимальными усилиями и получить начальную реализацию, которую можно развивать дальше.

На этом мы практически закончили реализацию системы безопасности для приложения Hubbub, но у нас осталась еще пара проблем, которые мы могли бы решить. Кроме того, сейчас самое время узнать, для чего служит класс `Role`.

10.4.2. Усиление ограничений доступности

Для назначения и определения прав доступа платформа Spring Security использует понятие *полномочий*: кто и какие действия может выполнять. Реализация этого понятия обладает весьма широкими возможностями, но поскольку концепция полномочий является довольно абстрактной, многим сложно в ней разобраться.

Расширение обходит эту проблему, оперируя лишь именованными полномочиями, которые называются *ролями*. Суть ролей очень проста – это всего лишь имена. Пользователю может присваиваться произвольное число ролей, которые дают ему или ей право доступа к любому адресу URL, ограниченному любой из этих ролей. В табл. 10.2 демонстрируется, как комбинируются правила и роли для определения права доступа пользователя к определенному адресу URL. Если пользователь обладает любой из требуемых ролей, ему предоставляется доступ к странице. Обратите внимание, что в этих примерах `ROLE_USER` и `ROLE_ADMIN` – это строки. Кроме того, если пользователю не присвоено ни одной роли, он или она не сможет даже зарегистрироваться.

Как выполняется присваивание ролей? За счет связывания экземпляра класса `Role` с экземпляром класса `User` либо программно, как это делается в классе `BootStrap` в примерах исходного программного кода приложения Hubbub, либо с помощью интерфейса управления пользователями. Последний вариант идеально подходит для случаев, когда необходимо иметь возможность выдавать или отзывать привилегии у пользователей во время выполнения приложения.

Таблица 10.2. Как правила и роли соотносятся друг с другом

Правило требует	Пользователь имеет	Доступ разрешен?
`ROLE_USER`	`ROLE_USER`	Да
`ROLE_ADMIN`	`ROLE_USER`	Нет
`ROLE_USER,ROLE_ADMIN`	`ROLE_USER`	Да

Типичный экземпляр класса `Role` может содержать имя `ROLE_USER` с описанием «A known, registered user of the system» (Известный системе

зарегистрированный пользователь). Такую роль можно сделать необходимой для доступа к любому адресу URL, для чего достаточно добавить соответствующее правило:

```
/profile/**=ROLE_USER
```

Это сразу же порождает вопрос: в чем разница между ролями IS_AUTHENTICATED_REMEMBERED,

IS_AUTHENTICATED_FULLY и ROLE_USER?

- IS_AUTHENTICATED_REMEMBERED – это встроенная в платформу Spring Security роль. Обеспечивает доступность страниц для пользователей, которые прошли аутентификацию или были запомнены.
- IS_AUTHENTICATED_FULLY – это встроенная в платформу Spring Security роль. Обеспечивает доступность страниц для пользователей, прошедших аутентификацию. Не разрешает доступ пользователям, которые были *запомнены*.
- ROLE_USER – это определяемая пользователем роль, которая обеспечивает доступ к страницам только тем пользователям, которым она была присвоена. Применяется независимо от того, был ли пользователь аутентифицирован или запомнен.

Рассмотрим более конкретный пример. Предположим, что у нас имеется интерфейс управления пользователями, сгенерированный командой расширения generate-manager. Необходимо, чтобы никто не имел доступ к этому интерфейсу, потому что изменение информации о пользователе, а также добавление, удаление и отключение учетных записей – очень серьезные операции. Что требуется сделать для этого?

Сначала нужно реализовать создание новой роли на этапе загрузки в классе BootStrap, например с именем ROLE_ADMIN, и присвоить ее пользователю:

```
def role = new Role(authority: "ROLE_ADMIN", description: "A super user.")
def admin = new User(userId: "dilbert", ...).save()
role.addToPeople(admin)
role.save()
```

Затем необходимо ограничить с помощью этой роли доступ к адресам URL интерфейса управления пользователями:

```
/user/**=ROLE_ADMIN
/role/**=ROLE_ADMIN
/requestmap/**=ROLE_ADMIN
```

Вот и все! Теперь инерфейс управления пользователями будет доступен только администраторам.

Все достаточно просто, и мы думаем, что вы согласитесь с нами. Но здесь есть одна проблема. Представьте, что пользователи могут редактировать свои учетные данные. Мы можем закрыть доступ к странице Edit Profile (Изменить учетные данные) ролью, но тогда любой, обладаю-

щий требуемой ролью, получит доступ к любому профилю – это совсем не то, что нам требуется! Данный случай является примером ситуации, когда мы выходим за рамки возможностей Spring Security и вынуждены реализовывать решение самостоятельно.

Самый простой способ решения проблемы заключается в использовании фильтров платформы Grails. Все, что нужно сделать, – настроить перехватчик `before` контроллера профиля и выполнить проверку принадлежности запрошенного профиля текущему пользователю. Реализация такого фильтра приводится в листинге 10.3. Она должна храниться в файле grails-app/conf/SecurityFilters.groovy. Если идентификатор текущего пользователя не совпадает с идентификатором пользователя в URL запроса, доступ блокируется.

Листинг 10.3. Ограничение доступа с помощью фильтра

```
package com.grailsinaction

class SecurityFilters {
    def authenticateService ← ❶ Внедряется необходимая служба

    def filters = {
        profileChanges(controller: "profile", action: "edit,update") {
            before = {
                def currUserId = authenticateService.userDomain().userId
                if (currUserId != params.userId) { ← ❷ Сравнение идентификаторов
                    redirect(controller: "login", action: "denied")
                    return false
                }
                return true
            }
        }
    }
}
```

Решение на основе фильтра выглядит на удивление простым. Первое, на что следует обратить внимание: для получения информации о текущем пользователе мы используем службу аутентификации ❶. Внутри фильтра мы сравниваем идентификатор текущего пользователя с параметром `userId` запроса ❷. Если они не совпадают, выполняется перенаправление на стандартную страницу «доступ закрыт» расширения, и возвращается значение `false` для предотвращения дальнейшей обработки запроса платформой Grails. В противном случае мы пропускаем запрос. Вы не сможете создать более простую реализацию, чем эта!

На этом мы заканчиваем обсуждение вопросов, связанных с авторизацией. Вы узнали достаточно много, чтобы суметь решить большинство проблем, с которыми вам придется столкнуться. Однако для разрешения ситуаций, где потребуется более тонкое разграничение прав доступа, вам могут потребоваться списки управления доступом платформы Spring Security (Access Control Lists, ACL). Хотя на момент написания этих строк

расширение не поддерживало их непосредственно, эта поддержка может быть реализована в самом ближайшем будущем. Как вариант вы могли бы рассмотреть возможность использования мощного и гибкого расширения JSecurity. За дополнительной информацией о расширениях обращайтесь к их домашним страницам на веб-сайте Grails.

Итак, мы закончили с авторизацией, что еще нам осталось? Существует ряд весьма занятных ситуаций, имеющих отношение к аутентификации, которые, на наш взгляд, могут заинтересовать вас.

10.4.3. Другие средства аутентификации

Процедура аутентификации, использовавшаяся нами до сих пор, опиралась на имена пользователей и их пароли, хранящиеся в базе данных. Несмотря на то что это один из самых распространенных подходов, он не единственный. Платформа Spring Security (и расширение) поддерживает множество схем аутентификации через так называемые провайдеры аутентификации (authentication providers). Возможно, вы будете удивлены, узнав, что приложение Hubbub уже использует три из них:

- `AnonymousAuthenticationProvider`
- `RememberMeAuthenticationProvider`
- `GrailsDaoAuthenticationProvider`

Последний из них использует для аутентификации наш класс `User`.

В большинстве случаев включение в работу того или иного провайдера выполняется довольно просто, потому что для этого достаточно лишь добавить несколько строк в настройки системы безопасности. Теперь мы коротко познакомимся с некоторыми из наиболее полезных провайдеров. Дополнительные сведения о них, включая учебные руководства и справочную информацию, вы можете найти на веб-сайте расширения.

LDAP

Наиболее часто используемый в корпоративных решениях, облегченный протокол доступа к каталогу (Lightweight Directory Access Protocol, LDAP) представляет собой прикладной программный интерфейс для управления доступом к любым иерархическим структурам данных. Одним из наиболее популярных применений протокола является хранение учетных записей пользователей и предоставление собственной схемы аутентификации. Если в вашей компании все данные о пользователях хранятся на сервере LDAP, вам, вероятно, потребуется обеспечить аутентификацию пользователей с помощью этого протокола. Служба сертификатов Microsoft Active Directory реализует протокол LDAP, поэтому вы сможете реализовать доступ и с ее помощью.

OpenID

Предел мечтаний реализации аутентификации в веб-приложениях заключается в том, чтобы дать пользователю возможность единожды зарегистрироваться и сразу же получить доступ к широкому кругу различных веб-приложений. OpenID – это одна из технологий, нацеленных на воплощение мечты в жизнь. В настоящее время идентификаторы пользователей и адреса URL часто бывает сложно запомнить, но они приобретают все большую сферу влияния благодаря поддержке таких компаний, как Yahoo!, Microsoft, Google и MySpace.

Когда регистрация в приложении производится с помощью OpenID, аутентификация выполняется посредством удаленной службы OpenID, которая затем перенаправляет пользователя обратно, в оригинальное приложение. В настоящее время существует уже несколько таких служб на выбор.

Facebook connect

Если вы никогда не слышали о Facebook, где вы тогда были? Эта технология использует подход, аналогичный OpenID, за исключением того, что в ее случае существует единственный поставщик услуг: Facebook. Это означает, что вы можете регистрироваться в приложении, используя свою учетную запись Facebook и предоставляя тем самым приложению некоторую информацию о себе из социальной сети.

Простая аутентификация

Это самая древняя технология аутентификации в Веб. Она основана на использовании заголовков и кодов состояний HTTP; и платформа Spring Security поддерживает ее с помощью фильтров, а не с помощью провайдера аутентификации.

В наши дни этот способ редко используется веб-сайтами, в частности из-за того, что информация об аутентификации передается на сервер в открытом виде. Кроме того, этот способ дает дополнительную нагрузку, поскольку данные аутентификации передаются с каждым запросом. Во многих случаях в этом нет никакой необходимости, однако такое поведение имеет определенный смысл для программных клиентов, не желающих задействовать механизм cookies. В действительности при выполнении обмена данными по протоколу HTTPS данный способ аутентификации идеально подходит для клиентов службы REST, когда вашему приложению необходимо обеспечить управление доступом. Подробнее о службах REST будет рассказываться в следующей главе.

По умолчанию расширение и его контроллер регистрации поддерживают только один тип провайдера, но это можно легко исправить. В настройках системы безопасности нужно добавить параметр с именем `providerNames` и в качестве его значения указать список желаемых провайдеров аутентификации:

```
providerNames = ['facebookAuthProvider',
                 'daoAuthenticationProvider',
                 'anonymousAuthenticationProvider',
                 'rememberMeAuthenticationProvider']
```

После этого останется лишь изменить контроллер и форму регистрации, чтобы включить поддержку нескольких способов аутентификации.

Я уверен, что нам удалось показать, насколько широкими возможностями обладает Spring Security; в этом разделе мы затронули лишь те из них, что лежат на самой поверхности. Эта платформа безопасности поддерживает большое разнообразие провайдеров аутентификации, из которых вы можете выбирать; она обеспечивает возможность управления доступом к методам служб; и, кроме того, она имеет свой механизм событий. За дополнительной информацией обо всех этих особенностях обращайтесь к электронной документации как для расширения, так и для самой платформы Spring Security.

Теперь можно вздохнуть с облегчением – вы сделали это! В данной главе мы узнали много нового, но вам предстоит узнать еще больше. Однако не стоит беспокоиться – мы дали вам надежный фундамент, на основе которого вы сможете построить свое здание знаний о рассмотренной предметной области.

10.5. В заключение

Безопасность – очень сложная тема, которая зачастую требует, чтобы разработчик обдумывал решение задачи с разных сторон. Вы видели примеры нескольких типов нападений, с которыми может столкнуться ваше приложение на платформе Grails. Но основное внимание мы уделяли приемам, которые вы должны применять, а не нападениям, которым они противостоят. В ходе разработки проще помнить правило «проверяй все входные данные», чем «защищайся против атак типа инъекция SQL/кода/всего, что угодно».

Следует также задумываться об устанавливаемых правилах управления доступом. Использование платформы безопасности, такой как Spring Security, может облегчить реализацию управления доступом в приложении, но при этом уровень безопасности полностью зависит от правил, которые вы определяете. Тщательно изучите специфические требования приложения, прежде чем приступать к реализации управления доступом.

Еще один важный момент, который следует обдумать, – насколько высокий уровень безопасности должен быть реализован в приложении. Насколько важная информация хранится в нем? Каковы могут быть последствия, если злоумышленнику удастся замаскироваться под истинного пользователя? Вы можете реализовать массу проверок, но на-

сколько оправданы будут затраты на разработку и, возможно, неудобства для пользователя? Например, приложение Hubbub не содержит серьезной конфиденциальной информации, но оно должно препятствовать возможности оставлять сообщения под чужими именами. С другой стороны, банковское приложение требует реализации значительно большего числа проверок и должно гарантировать, что никто не сможет не то, чтобы изменить, но даже прочитать информацию о пользователе.

Ниже приводятся несколько советов и рекомендаций, которые помогут вам защитить ваше приложение:

- *Запомните девиз: «сила в глубине защиты»*. Если вокруг вашего приложения имеется единственный слой защиты, пройдя его, нападающий получит неограниченный доступ ко всему приложению. Вы должны также выполнять проверку входных данных в своих контроллерах и использовать возможности автоматического экранирования параметров запроса.
- *Знайте границы доверия*. Граница доверия – это линия, по одну сторону которой имеется доверие, а по другую – нет. Именно на этих границах вы должны быть особенно бдительными при вычищении входных данных и проверке безопасности исходящих данных.
- *Тесты, тесты и еще раз тесты!* Обязательно проверяйте невосприимчивость приложения к распространенным типам нападений с помощью функциональных тестов и инструментов автоматизированной проверки, таких как Scarab проекта OWASP.
- *Обфускация не может использоваться взамен системы безопасности*. Сокрытие или искажение информации может оказаться полезным – в конце концов, не стоит рекламировать то, что может помочь нападающему. Но если вы полагаетесь на обфускацию, вы сильно рискуете скомпрометировать свое приложение. Этот прием может использоваться в дополнение к другим методикам, но никак не взамен их.
- *Помните о безопасности в процессе разработки*. Не забывайте, что обеспечение безопасности – это процесс и образ мышления. Чтобы обезопасить приложение, вы должны постоянно помнить о побочных эффектах программного кода, который вы пишите.
- *Выполняйте обзоры программного кода*. Регулярные обзоры программного кода помогут не только выявить ошибки программирования и слабые места проектирования, они также помогут обнаружить проблемы, связанные с безопасностью. Нет ничего лучше, чем привлечение еще одной пары глаз для выявления потенциальных опасностей.
- *Используйте существующие платформы и инструменты*. Независимо от того, насколько бдительны вы будете, ошибки будут появляться. Инструменты и платформы безопасности – проверенное

и широко используемое оружие. Они позволят быстро обнаруживать и ликвидировать уязвимости. Зачем рисковать, используя собственные решения?

Напоследок нам хотелось бы еще раз напомнить, что для платформы Grails существует несколько платформ безопасности, поэтому если вам потребуется использовать какую-нибудь из них, на наш взгляд, стоит потратить некоторое время на выяснение того, насколько она соответствует вашим нуждам.

Обладая знаниями об обеспечении безопасности приложения, можно теперь посмотреть, как немного приоткрыть свое приложение и разрешить доступ к нему не только людям, использующим броузеры, но и программным клиентам.

В этой главе:

- Обеспечение доступа к приложению из программных клиентов
- Реализация прикладного интерфейса REST по протоколу HTTP
- Экспортирование услуг для удаленных клиентов

11

Удаленный доступ

Теперь, когда у вас имеется полное представление об основных способах защиты приложения от нападений и неавторизованного доступа, пришло время посмотреть, как можно организовать взаимодействие приложений на платформе Grails с другими программами (не являющимися веб-броузерами).

Особое положение контроллеров и представлений GSP демонстрирует, как много делает платформа Grails, обеспечивая возможность создания веб-приложений, действующих в веб-броузерах, однако зачастую нет причин ограничивать себя только пользовательским интерфейсом – любой, кто пользовался сайтом Twitter, знает, что для просмотра и создания сообщений можно использовать множество различных клиентов. Интерфейс HTML не очень удобен для таких программных клиентов, так как обычно они сами управляют представлением информации. Нам необходимо обеспечить для программных клиентов возможность оперировать данными, находящимися в системе, то есть реализовать общедоступный программный интерфейс.

Такой прикладной интерфейс может быть реализован разными способами, и в этой главе мы представим вам несколько наиболее часто используемых из них. Основное внимание мы сосредоточим на технологии, ставшей популярной в последние несколько лет, и прекрасно подходящей для веб-приложений: REST. Платформа Grails обладает базовой поддержкой данной технологии, и это еще одна причина познакомиться с ней поближе. Мы также взглянем на пару дополнительных возможностей, которые могут лучше соответствовать нашим требованиям: расширение Remoting (основанное на применении механизма RMI[1]) и веб-службы (основанные на использовании протокола SOAP[2]).

---

[1] RMI – Remote Method Invocation (удаленный вызов метода). – *Прим. перев.*

[2] SOAP – Simple Object Access Protocol (простой протокол доступа к объектам). – *Прим. перев.*

К концу этой главы вы будете понимать, как легко и быстро спроектировать и реализовать общедоступный прикладной интерфейс в приложении на платформе Grails, избежав при этом некоторых неприятностей. Кроме того, вы будете обладать необходимым объемом знаний для принятия решения о том, на какой основе следует выполнять свою реализацию, чтобы она лучше соответствовала требованиям приложения.

Начнем мы с добавления в приложение Hubbub реализации простого прикладного интерфейса, который позволит просматривать и отправлять сообщения с помощью программных клиентов.

11.1. Решение на базе REST

На первом этапе создания прикладного интерфейса для приложения необходимо решить, какие действия должны быть доступны клиентам. В конце концов, если какая-то особенность недоступна в прикладном интерфейсе, клиент не может использовать ее. Выбор операций, которые должны поддерживаться прикладным интерфейсом, не всегда является простой задачей, но в процессе разработки у вас есть возможность поэкспериментировать. Как только интерфейс будет опубликован, вам следует избегать изменений в нем, чтобы не нарушить работоспособность использующих его клиентов.

Совет

Добавление новых операций в прикладной интерфейс обычно вызывает меньше проблем, чем изменение или удаление существующих. Лучше всего сначала реализовать минимально необходимый набор операций, а затем развивать интерфейс по мере необходимости.

Приложение Hubbub могло бы иметь весьма обширный прикладной интерфейс, но мы пока реализуем самые простые операции. Клиенты должны иметь следующие возможности:

- Получать список существующих сообщений
- Создавать новые сообщения
- Извлекать содержимое из существующих сообщений

Обратите внимание, что в этом списке не содержится информации о реализации, и это лучшее начало. Как только вы опишете прикладной интерфейс в такой форме, можно переходить к его реализации.

В языке Java прикладной интерфейс обычно реализуется в виде общедоступных интерфейсов с четко определенным поведением, но такой подход непригоден для веб-приложений, поскольку клиентские приложения могут быть написаны не только на языке Java. Какие еще варианты у нас имеются? Посмотрим, как взаимодействует веб-броузер с нашим веб-приложением. Они обмениваются информацией с помо-

щью платформонезависимого протокола HTTP, так почему бы нам не использовать его?

11.1.1. Первые шаги в REST

Протокол HTTP можно представить себе просто как язык общения в Веб, но на его основе реализован архитектурный стиль REST (Representational State Transfer – передача представления состояния), который можно использовать не только для организации взаимодействий с веб-броузерами, но и в любых других взаимодействиях типа клиент/сервер. Основная идея заключается в том, что приложение должно рассматриваться как коллекция ресурсов, над которыми можно выполнять некоторые операции с помощью нескольких методов. Возьмем в качестве примера Википедию: каждая страница – это отдельный ресурс, который можно извлечь, изменить или удалить. Более того, можно даже создавать новые страницы.

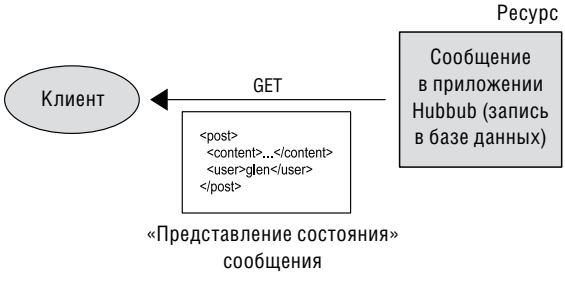


Рис. 11.1*. Ключевые понятия технологии REST. Сообщение в приложении Hubbub (экземпляр класса предметной области) – это наш ресурс, а фрагмент разметки XML, возвращаемой клиенту, – это представление состояния данного ресурса*

Откуда взялось название *REST*? При выполнении каждой операции (за исключением операции удаления) производится передача содержимого определенной страницы между клиентом (броузером) и Википедией. Содержимое страницы – это представление состояния (данных) страницы. На рис. 11.1 изображен другой пример, где в качестве ресурса рассматривается сообщение в приложении Hubbub.

Стоит только представить себе весь масштаб Веб и все его разнообразие, и вам станет понятно, почему технология REST приобрела такую популярность. Одно из преимуществ этой концепции заключается также в ее простоте и в том, что протокол HTTP предоставляет практически готовую реализацию (сама по себе технология REST сродни шаблону проектирования).

Четырьмя основными операциями (или методами), поддерживаемыми протоколом HTTP, являются `GET`, `POST`, `PUT` и `DELETE`. Первые две очень ча-

сто реализуются обычными веб-приложениями и используются броузерами. Каждый раз, когда вы открываете веб-страницу, броузер отправляет запрос GET, а когда вы отправляете данные на веб-сайт, они обычно передаются в виде запроса POST.

Чтобы реализовать прикладной интерфейс REST в приложении Hubbub, необходимо решить, какие ресурсы должны быть доступны, и обеспечить обработку приложением всех четырех методов HTTP для каждого из ресурсов. В следующих примерах мы будем рассматривать только сообщения в приложении Hubbub, но применяемый к ним подход можно использовать для представления других операций в приложении, таких как добавление и удаление друзей.

Как вы уже знаете из опыта работы в Интернете, чтобы получить доступ к ресурсу, необходимо указать его адрес URL. В идеале адреса URL, которые вы используете, должны быть:

- *Уникальными для каждого ресурса* – каждому адресу URL должен соответствовать единственный ресурс. Например, у вас не должно быть общего адреса URL, идентифицирующего различные сообщения. Вы можете определять несколько адресов URL для одного и того же ресурса, но во избежание путаницы лучше этого не делать.
- *Долгоживущими* – адрес URL всегда должен указывать на один и тот же ресурс, предпочтительно до скончания дней. Это может выглядеть, как слишком жесткое требование, и спецификация HTTP признает это, предусматривая временное или постоянное перенаправление. Вы все же должны стремиться к обеспечению как можно более длительного существования ваших адресов URL, в противном случае вы рискуете испортить свою репутацию из-за появления недействующих ссылок – одного из проклятий Веб.

На что должны быть похожи адреса URL сообщений в приложении Hubbub в соответствии с этими пунктами списка? Чем проще, тем лучше, тогда почему бы не использовать адрес *http://.../posts* для представ-

Выбор удачных адресов URL

Мы отметили только два требования к адресам URL, но они не единственные, определяющие удачный подбор адресов URL. Некоторые считают, что адреса URL должны быть к тому же короткими, простыми для восприятия и запоминания. Они не связаны непосредственно с технологией REST, но могут помочь разработчикам и пользователям. Вы можете отыскать дополнительные рекомендации, выполнив поиск в Веб по фразе «good URL design».[1]

[1] Для русскоговорящих читателей можно порекомендовать выполнить поиск по фразе «удобные адреса URL». – *Прим. перев.*

ления всех адресов в системе и адреса вида *http://.../posts/$id* для представления отдельных сообщений, где `$id` – это внутренний идентификатор сообщения?

Примечание

Наши адреса URL включают имя сервера (и, возможно, контекст сервлета). Пока идентификаторы сообщений будут уникальны в пределах приложения, каждый адрес URL гарантированно будет уникальным в глобальном смысле.

Теперь возникает вопрос выбора идентификаторов сообщений. Очевидным выбором является идентификатор экземпляра класса предметной области, но хотя таково поведение по умолчанию механизма скаффолдинга платформы Grails, существует несколько моментов, осложняющих дело, о которых вам следует знать.

Выбор идентификаторов ресурсов

Для прикладного интерфейса REST нам необходимы долгоживущие адреса URL. В стандартном приложении на платформе Grails идентификаторы генерируются базой данных (это первичный ключ). Если на протяжении всего своего существования приложение работает с одной и той же базой данных, никаких проблем не возникает. Но как быть, если с ростом популярности приложения появляется потребность перенести данные в более крупную базу данных?

Для этого вы, вероятно, экспортировали бы данные в инструкции SQL (включая текущие идентификаторы каждой записи), а затем импортировали бы их в новую базу данных. Этот прием сработает для существующих записей, но что произойдет, когда приложение создаст новый экземпляр класса предметной области? Если вам повезет, функция автоинкремента начнет автоматически создавать новые идентификаторы, продолжив существующие значения идентификаторов в соответствующей таблице, но вы не можете полагаться на это.

Еще один недостаток использования первичных ключей в адресах URL заключается в том, что злоумышленник сможет легко предсказывать допустимые значения. Кроме того, для злоумышленника это может служить подсказкой о реализации приложения. Возможно, для вас это не проблема, но если вашему приложению предстоит хранить конфиденциальную информацию, вам следует рассмотреть возможность использования альтернативного источника идентификаторов.

Например, в качестве первичного ключа в классах предметной области можно использовать значения GUID (Globally Unique Identifier – глобально уникальный идентификатор) или UUID (Universally Unique Identifier – универсальный уникальный идентификатор), или можно выделить другое (но уникальное) поле, которое будет хранить значение идентификатора для адреса URL (например, имя пользователя или

адрес электронной почты). Приложение Hubbub не хранит конфиденциальную информацию, поэтому для простоты реализации мы будем использовать стандартный первичный ключ.

Итак, мы определились с форматом адресов URL для сообщений, теперь можно перейти к реализации каждого из методов HTTP.

Предоставление доступа к ресурсам

До сих пор в данной книге мы видели только прямое «один к одному» отображение адресов URL на действия контроллеров. Мы могли бы продолжить использовать этот подход и создать действие, которое выполняло бы различные операции в зависимости от метода HTTP, как показано ниже:

```
def handlePosts = {
    if (request.method == "GET") {
        ...
    }
    else if (request.method == "POST") {
        ...
    }
}
```

Такая реализация будет решать возложенные на нее задачи, но эти условные конструкции неудобны в сопровождении, и, кроме того, они осложнят рефакторинг программного кода в будущем.

Попробуем использовать иной подход, воспользуемся механизмом отображения адресов URL:

```
class UrlMappings {
    def mappings = {
        ...
        "/posts"(controller: "postRest") {
            action = [ GET: "list", POST: "save" ]
        }
        "/posts/$id"(controller: "postRest") {
            action = [ GET: "show", PUT: "update", DELETE: "delete" ]
        }
    }
}
```

Как видите, имеется возможность с помощью параметра отображения URL `action` определить отображение, которое свяжет методы HTTP с именами действий. В данном случае запросы `GET` к адресу /posts будут обрабатываться действием `list`, тогда как запросы `PUT` к адресу /posts/$id будут обрабатываться действием `update`. Обратите внимание, что для упрощения реализации мы отобразили адреса URL не на существующий контроллер `post`, а на отсутствующий пока контроллер `postRest`. Чуть ниже вы поймете, почему необходимо иметь отдельный контроллер для обработки запросов REST.

POST и PUT

Нередко при реализации технологии REST возникает недопонимание, когда следует применять метод POST, а когда PUT. Согласно спецификации (*http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html*) метод PUT используется для создания и обновления ресурсов *с заданными адресами URL*.

В большинстве случаев при использовании платформы Grails идентификатор нового ресурса неизвестен до его создания, следовательно определить конкретный адрес URL для метода PUT невозможно. По этой причине для родительского адреса URL, например /posts, выполняется метод POST, который создаст подчиненный ресурс, такой как /posts/10.

В случае существующего ресурса его идентификатор известен, поэтому для его изменения можно использовать метод PUT.

От отображения без контроллера, стоящего за ним, мало проку, поэтому сейчас мы его создадим. Мы должны принять одно важное решение – определить, в какой форме должны передаваться сообщения. В терминологии REST этот элемент называется «представлением состояния». Теоретически можно выбрать любой формат, даже двоичный, но на сегодняшний день, пожалуй, самым часто используемым форматом является XML. Отчасти по этой причине мы будем использовать формат XML; кроме того, платформа Grails предоставляет специализированные синтаксические конструкции, которые упрощают работу с этим форматом.

Наши обработчики запросов GET (действия list и show) реализуются очень просто:

```
package com.grailsinaction

import grails.converters.XML

class PostRestController {
    def list = { render Post.list() as XML }
    def show = {
        Post post = Post.get(params.id)
        render post as XML
    }
}
```

Сложно придумать более простую реализацию, чем эта. Оба действия генерируют данные в формате XML, используя некоторую магию, с которой мы познакомимся в следующем подразделе. А пока опробуем наше действие show, запустив приложение и введя в броузере адрес *http://localhost:8080/hubbub/posts/1*. Вы должны увидеть страницу, содержа-

щую разметку XML; она будет немного напоминать фрагмент, изображенный на рис. 11.1.

Как у нас получился ответ в формате XML? Чтобы понять это, необходимо чуть больше узнать о конвертерах платформы Grails.

11.1.2. Сериализация и десериализация объектов предметной области

Класс `XML` является примером конвертера платформы Grails. Единственное назначение конвертеров заключается в преобразовании, или сериализации, объектов предметной области в определенный формат, такой как XML. Grails по умолчанию предоставляет всего два конвертера (`XML` и `JSON`), однако вы можете добавлять свои собственные, наследуя класс `AbstractConverter` и реализуя необходимые методы (за дополнительной информацией обращайтесь к документации с описанием API).

За кулисами платформа Grails переопределяет метод `asType()` языка Groovy для классов предметной области (с помощью расширения converters, входящего в состав ядра), благодаря чему у вас имеется возможность использовать конструкции `as XML` и `as JSON`. Можно также использовать ключевое слово `as` со своими собственными конвертерами. Но это лишь часть истории: конструкция `as XML` создает не строку с разметкой XML, а экземпляр класса `XML`. Чтобы создать соответствующий фрагмент XML в текстовом виде, необходимо вызвать либо метод `render(Writer)`, либо `render(HttpServletResponse)` экземпляра конвертера, как показано ниже:

```
def xml = obj as XML
xml.render(response)
```

Конструкция `render post as XML`, используемая в контроллере, вызывает последний из этих двух методов.

Расширение converters не только переопределяет метод `asType()`, но и добавляет в контроллеры дополнительный метод `render(Converter)`, который делегирует конвертеру операцию записи ответа. Кроме записи ответа, конвертер устанавливает также соответствующий тип содержимого.

Вот так практически без усилий мы реализовали для работы с сообщениями метод `GET`, который возвращает их в формате XML. Следующий шаг – реализовать создание новых сообщений методом HTTP `POST`. Для этого можно воспользоваться динамическим свойством, добавленным к запросу расширением converters, как показано в реализации действия `save` в листинге 11.1.

Листинг 11.1. Обработка запроса POST, содержащего разметку XML

```
package com.grailsinaction

class PostRestController {
```

```
    ...
    def save = {
        def xml = request.XML ← ❶ Извлекает фрагмент XML из запроса
        def post = new Post()
        post.content = xml.content.text()          ← ❷ Инициализирует сообщение
        post.user = User.get(xml.user.@id.text())      из данных в формате XML

        def markup
        if (post.save()) {
            markup = {
                status("OK")
            }
        }
        else {
            markup = {
                status("FAIL")
            }
        }
        render contentType: "text/xml; charset=utf-8",   ← ❸ Отправляет ответ
              markup                                         в формате XML
    }
    ...
}
```

В самом начале мы извлекаем фрагмент с разметкой XML из запроса ❶. Объект, возвращаемый при обращении к свойству `XML`, является экземпляром класса `GPathResult`, который является типичным результатом парсинга разметки XML с помощью класса `XmlSlurper` языка Groovy. Затем мы используем этот результат для заполнения нового экземпляра класса `Post` ❷ данными, полученными в запросе. Мы предполагаем, что корневой элемент фрагмента XML содержит дочерние элементы `content` и `user`, подобно фрагменту XML, созданному конструкцией `post as XML`:

```
<post>
    <content>...</content>
    <user id="..."/>
</post>
```

В завершение мы сохраняем новое сообщение и отправляем обратно ответ в формате XML ❸, свидетельствующий, что сообщение было благополучно добавлено.

Будучи добропорядочными гражданами, мы обычно включаем в ответ дополнительную информацию о причинах отказа, но мы не сделали этого здесь, чтобы сохранить программный код простым. В идеале мы установили бы также код состояния HTTP в ответе, такой как 400. Поближе с кодами состояний мы познакомимся в конце раздела 11.3.

Теперь в нашем приложении Hubbub для работы с сообщениями имеется действующий прикладной интерфейс REST, но не следует принимать его на веру. Почему бы вам его не опробовать? Проверить обработку метода `GET` очень просто, для этого достаточно ввести в адресной стро-

Типы содержимого XML

При отправке содержимого в формате XML по протоколу HTTP имеется возможность выбора из нескольких типов MIME: `text/xml`, `application/xml` или `application/...+xml`. Последний – это шаблон для конкретных диалектов языка XML, таких как `application/xhtml+xml` для XHTML.

Если в вашем случае содержимое ответа соответствует какому-то определенному диалекту языка XML, для которого определен свой тип MIME, лучше будет указывать этот тип. В противном случае для содержимого XML документ RFC-3023 рекомендует указывать тип `text/xml`, если любой пользователь сможет прочитать *и понять* его, и `application/xml` – если нет. Вот почему инструменты, не понимающие разметку XML, будут отображать содержимое типа `text/xml` как простой текст.

И последнее замечание. Если вы предпочтете указывать тип содержимого как `text/xml`, добавьте параметр `charset`, как это сделано в нашем примере. Иначе добропорядочные программные клиенты будут по умолчанию использовать кодировку US-ASCII.

ке броузера соответствующий URL и посмотреть, какой будет получен ответ. Обработку других методов HTTP проверить с помощью броузера значительно сложнее, потому что для отправки форм веб-броузеры используют только метод `POST` (причем без использования формата XML!) и в большинстве своем не поддерживают методы `PUT` и `DELETE`.

Для опробования этих методов вам потребуется специальный клиент REST, который позволит отправлять запросы HTTP любых типов с любым содержимым. Если вы пользуетесь веб-броузером Firefox, у вас есть прекрасная возможность установить расширение Poster. Другой вариант – использовать клиента, написанного на языке Java и имеющего пользовательский интерфейс, реализованный на базе библиотеки Swing (*http://code.google.com/p/rest-client/*). А если вам нравится командная строка, вы не найдете ничего лучше, чем гибкая утилита curl (*http://curl.haxx.se/*).

Чтобы проверить реализацию прикладного интерфейса REST, вы можете попробовать отправить приложению Hubbub по адресу URL /posts запрос `POST` со следующим содержимым в формате XML:

```
<post>
    <content>I'm posting this via REST!</content>
    <user id="1"/>
</post>
```

Убедитесь, что вы используете действительный идентификатор пользователя – при использовании базы данных HSQLDB или MySQL зна-

чение "1" должно быть вполне подходящим, но при работе с базой данных PostgreSQL, имеющей настройки по умолчанию, это может быть не так. На рис. 11.2 изображен результат выполнения данного запроса с помощью расширения Poster для броузера Firefox.

Тестирование вручную с помощью такого клиента, как Poster, отлично подходит для экспериментов, но для постоянного использования, как вы уже знаете, лучше использовать автоматизированные тесты. Это может показаться сложным, но на самом деле реализация автоматизированного теста не сильно сложнее, чем создание тестов для стандартных контроллеров.

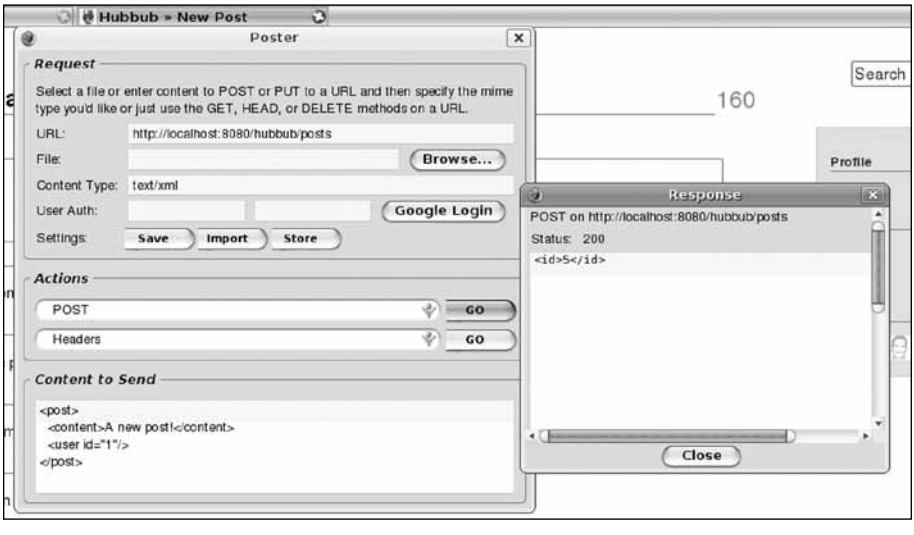


***Рис. 11.2.** Отправка тестового запроса POST приложению Hubbub с помощью расширения Poster для броузера Firefox. Запрос должен создать новое сообщение для указанного пользователя, а в ответе должен быть возвращен идентификатор нового сообщения*

11.1.3. Тестирование прикладного интерфейса

Для реализации технологии REST мы используем контроллеры, поэтому мы можем создавать для них такие же модульные тесты, как и для обычных контроллеров, что описывалось в главе 7. Наследуя класс `ControllerUnitTestCase`, наши тесты получают все необходимые для тестирования средства, включая дополнительные особенности, которые мы еще не обсуждали.

В листинге 11.2 приводится реализация модульного теста для контроллера `postRest`, который мы создали выше в предыдущем подразделе. Этот тест позволяет убедиться в корректной работе обоих действий `show` и `save`, хотя он и не настолько полон, как хотелось бы.

Листинг 11.2. Модульный тест для контроллера, реализующего интерфейс REST

```
package com.grailsinaction

import grails.test.ControllerUnitTestCase

class PostRestControllerUnitTests extends ControllerUnitTestCase {
    protected void setUp() {
        super.setUp()
        mockDomain(Post, [
            new Post(content: "First"),
            new Post(content: "Second"),
            new Post(content: "Third") ])
    }

    void testShow() {
        this.controller.params.id = 2
        this.controller.show()

        def responseContent = this.controller.response.contentAsString
        def xml = new XmlSlurper().parseText(
            responseContent) <-- ❶ Парсинг ответа в формате XML

        assertEquals "Second", xml.content.text()
    }

    void testSave() {
        mockDomain(User, [ new User(userId: "peter") ]

        setXmlRequestContent {
           post {                          <-- ❷ Устанавливает содержимое запроса
               content("A new message")        в формате XML
               user(id: 1)
           }
        }
        this.controller.save()

        assertEquals "<status>OK</status>", <-- ❸ Проверка содержимого ответа
            this.controller.response.contentAsString          в формате XML
    }
}
```

Вы уже знакомы с основными принципами реализации теста, поэтому мы остановимся лишь на том, что является новым для вас. Чтобы протестировать действие `show`, мы определяем значение параметра `id`, производим непосредственный вызов действия и затем анализируем содержимое ответа с помощью класса `XmlSlurper` ❶. Зачем мы анализируем? Анализ формата XML – достаточно сложное дело, потому что при использовании этого формата ни пробелы, ни порядок следования атрибу-

тов элементов не имеют значения. Вы можете получить документ XML, располагающийся в нескольких строках, с оформленными отступами, или семантически эквивалентный документ в более компактной форме, занимающий единственную строку! Как только вы начинаете использовать простую операцию сравнения строк при анализе XML, вы открываете двери перед возможными проблемами. Использование процедуры парсинга строки в формате XML в первую очередь делает ваш тест более устойчивым в случае изменения форматирования документа XML.

Чтобы протестировать действие `save`, необходимо предоставить ему для обработки некоторое содержимое в формате XML. Это легко можно сделать с помощью метода `setXmlRequestContent()` ❷. В данном примере мы создаем фрагмент XML с использованием синтаксиса построителя разметки. Вы можете также передать содержимое в виде строки с помощью метода `setXmlRequestContent(String)`. После вызова любой из указанных методов действие сможет получить доступ к содержимому в формате XML, обратившись к входному потоку запроса и к его динамическому свойству `XML`.

В заключение выполняется проверка успешности выполнения операции посредством анализа содержимого ответа ❸. Нет смысла заниматься парсингом фрагмента XML, содержащего единственный элемент, потому что весьма маловероятно, что его форматирование изменится. Поэтому здесь мы использовали простую операцию сравнения строк.

Вот и все! Модульное тестирование контроллера REST реализуется очень просто. В следующем разделе мы продемонстрируем несколько дополнительных особенностей платформы модульного тестирования, но вы и так уже изучили многое из того, что вам необходимо знать. Мы рекомендуем выполнять также функциональное тестирование прикладного интерфейса REST, чтобы окончательно убедиться, что все действует, как надо, но реализация такого теста зависит от того, какую платформу тестирования вы используете. На момент написания этих строк расширение Functional Test не поддерживало возможность отправки запросов в формате XML, но она была включена в планы на будущее и, возможно, будет реализована к тому моменту, когда вы будете читать эти строки. Проверьте домашнюю страницу расширения.

После того как мы получили подтверждение, что наша реализация прикладного интерфейса REST действует, как ожидалось, что еще нам осталось сделать? Один из недостатков выбранного нами подхода состоит в том, что мы ограничили себя только форматом XML. А если попробовать добавить поддержку формата JSON? Вы, возможно, сразу подумали, что для этого придется создать новый комплект действий контроллера, но тогда вам пришлось бы отделить адреса URL, использующие формат XML, от адресов, использующих формат JSON, – это не так! Существует другой путь: согласование формата содержимого.

11.2. Согласование формата представления

Представьте себе путешественника, блуждающего по рынку где-нибудь в самом глухом уголке мира. Он видит нечто, что ему захотелось купить, но чтобы сделать покупку, ему необходимо вступить в общение с продавцом. Первый шаг состоит в том, чтобы договориться о языке, который был бы понятен обеим сторонам – даже если это импровизированный язык жестов. Аналогично клиент и сервер должны договориться о формате данных, участвующих в обмене, при использовании технологии REST.

Платформа Grails предоставляет возможность, называемую «согласованием формата содержимого» (content negotiation), которая осуществляет управление различными форматами и обслуживанием договоренностей с клиентом. Клиент может указать предпочтительный формат содержимого различными способами, а платформа Grails на стороне сервера упрощает возможность отображения содержимого на основе этого формата. На рис. 11.3 в общих чертах показано, как действует данный механизм.

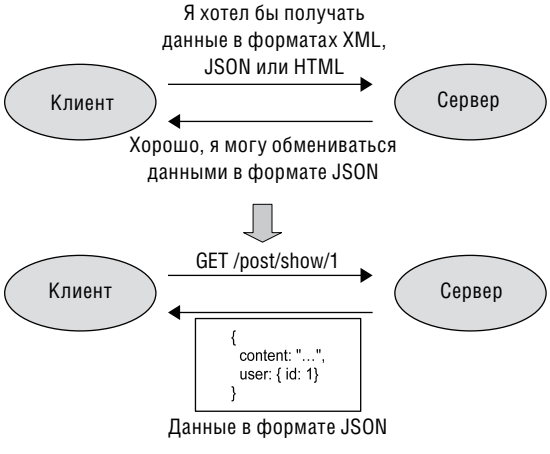


Рис. 11.3. *Процедура согласования формата содержимого в действии. Клиент сообщает серверу о форматах содержимого, которые он поддерживает, а сервер возвращает содержимое в одном из этих форматов*

Какое влияние на реализацию прикладного интерфейса REST окажет согласование формата содержимого? С точки зрения интересов приложения, необходимо рассмотреть всего два аспекта: обработку содержимого входящих запросов и отображение соответствующего содержимого в ответ. Для обработки запросов необходимо написать программный код, который будет обслуживать все необходимые типы содержимого, поэтому вам приятно будет узнать, что платформа Grails унифицирует доступ к данным запроса с помощью стандартного объекта `params`.

Примечание

Согласование формата содержимого предполагает симметрию. Это означает, что платформа Grails ожидает, что тип содержимого запроса будет совпадать с типом содержимого ответа, за исключением случаев с методами GET и DELETE.

11.2.1. REST и свойство params

Вы уже знаете, что объект params содержит адрес URL и данные POST (из формы HTML). Единственное, что вы пока не знаете, – это то, что при активной функции согласования формата содержимого платформа Grails может добавлять данные запроса в форматах XML и JSON. Безусловно, это поможет понять, как структурированные данные запроса отображаются в объект params, поэтому мы коротко рассмотрим, как произвольный документ XML превращается в параметр с данными.

Предположим, что запрос содержит следующий фрагмент XML:

```
<post>
    <content>A new post.</content>
    <tags>
        <tag>Music</tag>
        <tag>US</tag>
    </tags>
    <user id="10">
        <userId>dilbert</userId>
        <password>hjdj23r0fn</password>
    </user>
</post>
```

Заметьте, что этот фрагмент XML взят исключительно в демонстрационных целях – он не будет восприниматься приложением Hubbub.

Предположим, что клиент и сервер договорились об использовании формата XML (как это происходит, вы увидите в подразделе 11.2.3), тогда платформа Grails добавляет в объект params элемент с ключом post (в соответствии с именем корневого элемента) и со следующим отображением в качестве его значения:

```
[ content: "A new post.",
  tags: "Music US",
  "user.id": "10",
  user: [ userId: "dilbert", password: "hjdj23r0fn" ],
  "user.userId": "dilbert",
  "user.password": "hjdj23r0fn"
]
```

Каждому элементу в документе XML, имеющему атрибут id, соответствует вложенное отображение. Все остальные элементы отображаются в строки (даже содержимое элемента tags). Это отображение может

быть привязано к любому управляющему объекту или к любому объекту предметной области, как это происходит с обычными параметрами данных:

```
Post post = new Post(params["post"])
...
```

Предыдущая строка должна быть вам знакома по примерам программного кода предшествующих контроллеров, и этот прием будет действовать независимо от того, в каком формате передаются данные – XML, JSON или HTML.

Платформа Grails не выполняет автоматический парсинг содержимого запросов, поэтому вам придется явно включить данную возможность. Сделать это можно двумя способами: либо с помощью параметра `resource`, либо с помощью параметра `parseRequest` в настройках отображения адресов URL.

Параметр `resource` должен заместить параметры `controller` и `action` в отображении адресов URL для интерфейса REST:

```
class UrlMappings {
    "/posts/$id?"(resource: "postRest")
    ...
}
```

Это отображение применит поведение по умолчанию интерфейса REST к контроллеру `postRest`, в результате чего метод `GET` окажется отображен на действие `show`, метод `POST` – на действие `save`, метод `PUT` – на действие `update`, и метод `DELETE` – на действие `delete`. Кроме того, оно заставит платформу Grails автоматически выполнять парсинг содержимого соответствующих запросов.

Параметр `parseRequest`, в свою очередь, воздействует *только* на парсинг запросов. Подобно параметру `resource`, он используется в настройках отображения адресов:

```
class UrlMappings {
    "/items/$id?"(controller: "item", parseRequest: true) {
        action = [ GET: "list", POST: "save" ]
    }
    ...
}
```

Благодаря одному простому параметру ваше действие получает возможность обратиться к содержимому в формате XML либо через объект `params`, либо через свойство `request.XML`.

Совет

Параметр `resource` сложный в использовании. Как вы помните, при обработке метода `PUT` ожидается, что адрес URL будет содержать значение `id`, а при обработке `POST` – нет. Корректная реализация такого поведения с применением параметра `resource` может потребовать больше усилий, чем вообще без его

использования! Если вам требуется обеспечить корректное поведение прикладного интерфейса REST, мы рекомендуем избегать использования параметра `resource`.

Прежде чем двинуться дальше, следует сделать еще несколько важных замечаний, касающихся отображения разметки XML в элементы объекта `params`. Во-первых, отношения между объектами будут учитываться, только если указан атрибут `id`. Это означает, например, что вы не сможете создать одновременно сообщение *и* ассоциированного с ним пользователя, но вы сможете создать новое сообщение от лица существующего пользователя. Во-вторых, отношения «один-ко-многим» вообще не поддерживаются.

Наконец, представления по умолчанию, сгенерированные динамически механизмом скаффолдинга, не добавляют префиксы к именам своих полей ввода, поэтому они не будут работать с предыдущим фрагментом программного кода контроллера (`post = new Post(params["post"])`). Динамически созданная страница редактирования могла бы содержать, например, такую разметку HTML:

```
<input type="text" name="content" id="content" value="..." />
```

Но для корректной работы она должна иметь вид:

```
<input type="text" name="post.content" id="post.content" value="..." />
```

Заметьте, что имя поля ввода содержит префикс `post.`, который обеспечивает доступность значения этого поля через `params["post"].content`.

Если вы можете жить с этими ограничениями, тогда объект `params` вполне пригоден для обработки запросов в стиле REST. Другие возможности мы рассмотрим в разделе 11.3, а пока отвлечемся от обработки содержимого запросов и посмотрим на ответы.

11.2.2. Поддержка нескольких форматов содержимого ответов

Рассмотрим действие `show`, созданное нами для контроллера `postRest`. В его текущей реализации оно всегда возвращает ответ в формате XML; но как быть, если клиент требует, чтобы данные о сообщении возвращались в формате JSON?

Вероятно, первым вашим желанием будет проверить формат запроса, что возможно благодаря тому, что платформа Grails добавляет в объект запроса динамическое свойство `format`:

```
def show = {
    Post post = Post.get(params.id)
    if (request.format == "xml") {
        render post as XML
    }
```

```
        else if (request.format == "json") {
            render post as JSON
        }
        else { ... }
    }
```

Этот прием действует, но мы ожидаем от платформы Grails большего. В листинге 11.3 демонстрируется более простой и понятный способ, основанный на использовании специального блока `withFormat`. Вы можете опробовать его с помощью броузера, добавив в адрес URL параметр `format` и указав в качестве значения один из поддерживаемых форматов, такой как `xml` или `json` (например, *http://.../post/1?format=json*). Почему действует этот параметр в адресе URL, вы узнаете в следующем подразделе, когда мы познакомимся с тем, как платформа Grails определяет формат запроса.

Листинг 11.3. Использование конструкции withFormat для создания ответов в требуемом формате

```
package com.grailsinaction

class PostRestController {
    def show = {
        Post post = Post.get(params.id)

        withFormat {  <-- ❶ Обслуживает разные форматы
            html {
                [ post: post ]  <-- Возвращает модель для представления HTML
            }
            xml {
                [ post: post ]  <-- ❷ Возвращает модель для представления XML
            }
            json {
                render post as JSON
            }
        }
    }
}
```

Блок `withFormat` ❶ имеет достаточно простой синтаксис. Для каждого поддерживаемого формата добавляется дополнительный блок с названием формата. В этом примере мы реализовали возможность ответа на запросы в форматах HTML, XML и JSON. Если действие не имеет соответствующего элемента для формата запроса, платформа Grails выбирает первый из них (в данном случае `html`).

Каждый блок формата следует интерпретировать как отдельное действие контроллера. Значит, если требуется отобразить соответствующее представление (в данном случае `show.gsp`), блок должен возвращать отображение, как это сделано для форматов HTML и XML ❷. Допускается также напрямую вызывать метод `render()`, если это необходимо (как это

сделано для формата JSON). Главное требование, которое вам необходимо знать, заключается в том, что блок `withFormat` должен быть последним фрагментом программного кода в действии контроллера.

Модульное тестирование конструкции withFormat

Если вам потребуется провести модульное тестирование действия, использующего конструкцию `withFormat`, для этого вам потребуется определить формат запроса, как показано ниже:

```
this.controller.request.format = "xml"
```

Вам даже не придется имитировать метод `withFormat`, потому что его имитация уже реализована в классе `ControllerUnitTestCase`.

В последнем абзаце кое-что может показаться вам странным. Фактически мы заявили, что для отображения форматов HTML и XML используется представление show.gsp, но как такое может быть? Для нас вряд ли будет желательно возвращать одно и то же содержимое в обоих форматах, и нам требуется убедиться, что для ответа выбран соответствующий формат. Не волнуйтесь: эта проблема легко решается еще одним замечательным соглашением, принятым в платформе Grails. Вы можете создавать файлы GSP, следуя соглашению об именовании <действие>.<формат>.gsp, то есть вы можете создавать отдельные представления для каждого формата.

Так, для нашего примера можно было бы создать представление views/postRest/show.xml.gsp, которое выглядит примерно так:

```
<%@ page contentType="text/xml; charset=utf-8" %>
<post>
    <content>${post.content}</content>
    <user id="5"/>
</post>
```

Это стандартное представление GSP, отображающее разметку XML, а не HTML. Самой важной в предыдущем фрагменте является директива `@page`, которая гарантирует, что ответ будет в формате XML. Без этой строки тип ответа был бы `text/html`, что было бы не только неправильно, но и в принципе неприменимо для клиентов, которые обрабатывают разметку HTML и XML по-разному.

Несмотря на то что для совместной работы с блоком `withFormat` мы использовали представление, для специфического формата такие представления будут прекрасно работать и без этого блока. В действительности, если в листинге 11.3 мы откажемся от обработчика формата JSON, то сможем избавиться и от блока `withFormat`, потому что для представления формата XML используется та же модель, что и для представления формата HTML:

> **Порядок следования форматов**
>
> Порядок объявления обработчиков в блоке `withFormat` важен, поскольку если параметр `format` имеет значение `all` или вообще не имеет значения, платформа Grails будет использовать первый обработчик. Некоторые старые версии веб-броузеров отправляют заголовок `Accept`, который платформа Grails интерпретирует как запрос с форматом `all`, поэтому мы рекомендуем всегда в первую очередь объявлять обработчик `html` (если, конечно, ваше действие поддерживает формат HTML). Поближе с заголовком `Accept` мы познакомимся в следующем разделе.

```
def show = {
    Post post = Post.get(params.id)
    [ post: post ]
}
```

Это означает, что блок `withFormat` можно использовать, только если каждый формат обрабатывается по-разному.

К настоящему моменту мы уже многое рассказали о согласовании форматов содержимого, но мы упустили один важный момент: как платформа Grails определяет формат запроса?

11.2.3. Как определяется формат запроса

Определение формата запроса выполняется платформой Grails на основе нескольких факторов. Они перечислены ниже в порядке убывания значимости:

1. Расширение файла в адресе URL.
2. Параметр `format` в строке URL.
3. Тип содержимого запроса (в соответствии с заголовком `Content-Type` протокола HTTP).
4. Заголовок HTTP `Accept`.

Рассмотрим их по очереди.

Расширение файла в адресе URL

Первый из этих факторов достаточно очевиден. Для адреса URL, такого как *http://localhost: 8080/hubbub/post/1.xml*, платформа Grails установит формат `xml` в соответствии с фрагментом адреса, следующим за последней точкой (.). Это означает, что оба файла, 1.xml и 1.tmp.xml, соответствуют формату `xml`. Однако, если использовать такой подход для предоставления одного и того же ресурса в разных форматах, вы получите несколько адресов URL, представляющих единственный ресурс, а это не самое лучшее решение, о чем уже говорилось выше.

Следует также заметить, что расширение интерпретируется как формат, только если оно известно платформе Grails. Однако это не повод для паники: если заглянуть в файл Config.groovy для вашего проекта, можно увидеть свойство `grails.mime.types` с отображением в качестве значения. Платформа Grails распознает любые форматы, названия которых определены в отображении в виде ключей.

Ниже приводится пример настройки по умолчанию:

```
grails.mime.types = [ html: ['text/html','application/xhtml+xml'],
                      xml: ['text/xml', 'application/xml'],
                      text: 'text/plain',
                      js: 'text/javascript',
                      rss: 'application/rss+xml',
                      atom: 'application/atom+xml',
                      css: 'text/css',
                      csv: 'text/csv',
                      all: '*/*',
                      json: ['application/json','text/json'],
                      form: 'application/x-www-form-urlencoded',
                      multipartForm: 'multipart/form-data' ]
```

По умолчанию платформа Grails распознает большинство наиболее важных типов содержимого, используемых в Веб. Вы легко можете добавить свои форматы, вставив в отображение дополнительные элементы.

Внимание

По умолчанию платформа Grails удаляет расширение файла из адреса URL, если обнаружено соответствие известному формату. Это означает, что вы не сможете использовать шаблон /post/$filename.xml в настройках отображения адресов URL. Вы можете либо использовать шаблон без расширения, такой как /post/$filename, и определять формат содержимого по значению свойства `format` запроса, либо запретить такое поведение платформы, установив параметр `grails.mime.file.extensions` в файле с настройками в значение `false`.

Параметр format в строке URL

Действие параметра `format` в строке URL подобно действию расширения файла, но параметр `format` не оказывает влияния на имя ресурса, потому что он является параметром запроса: *http://localhost:8080/hubbub/post/1?format=xml*. Это, пожалуй, самый простой и самый надежный способ для клиента запросить данные в определенном формате.

Тип содержимого запроса

Последние два элемента в списке отличаются от предыдущих тем, что они имеют отношение не к простым именам форматов, а к типам MIME. Именно по этой причине параметр настройки `grails.mime.types` имеет вид отображения, а не списка или множества: платформа Grails долж-

на знать, как исходя из типа MIME получить соответствующее имя формата.

Заголовок `Content-Type` достаточно прост, но область его применения ограничена, потому что большинство запросов `GET` не имеют или не должны иметь типа содержимого, ассоциированного с ним. В действительности этот заголовок имеет смысл только для запросов `POST` и `PUT`. Но даже тогда следует быть осторожными: броузеры отправляют данные с типом MIME `application/x-www-form-urlencoded`, и при этом они обычно ожидают получить ответ в формате HTML. Это один из примеров, когда симметрия, предполагаемая механизмом согласования формата содержимого в платформе Grails, не соблюдается.

Заголовок HTTP Accept

Заголовок `Accept` является еще одной затруднительной особенностью: он содержит список предпочтительных типов содержимого, которые может обрабатывать клиент, вместе с рейтингами (значения `q`), которые соответствуют степени их предпочтения.

Рассмотрим, например, следующее значение заголовка `Accept`:

```
application/xhtml+xml,application/xml,text/html;q=0.9;text/xml;q=0.8
```

Согласно этому заголовку клиент отдает предпочтение форматам XHTML и XML (указывая тип MIME `application/xml`) перед форматом HTML, который в свою очередь более предпочтителен, чем тип MIME `text/xml`. Рейтинг `q` может иметь значение от 0 до 1, большему значению соответствует более высокое предпочтение. Если некоторый тип MIME не имеет значение `q` рейтинга (как в случае с форматом XHTML в данном примере), платформа Grails присвоит ему рейтинг `1`.

Мы могли бы рассмотреть заголовок `Accept` и порядок его интерпретации более подробно, но мы не думаем, что это поможет вам на данном этапе. Дополнительную информацию, если она вам понадобится, вы сможете найти в справочном руководстве к платформе Grails. Проблема состоит в том, что для определения формата запроса из заголовка `Accept` платформа Grails использует сложный (отчасти даже эвристический) алгоритм. Он реализован так, что в результате вы будете получать формат `html` запроса для большинства популярных веб-броузеров, даже хотя все они используют совершенно различные заголовки `Accept`. К сожалению, такая сложность означает, что заголовки `Accept` других типов клиентов могут вообще не действовать.

Приведенное описание возможно слишком абстрактно, чтобы понять его с первого прочтения, но, говоря простыми словами, оно означает, что не стоит опираться на заголовок `Accept` для согласования формата содержимого, так как это может быть чревато трудностями. Мы рекомендуем использовать параметр `format` запроса везде, где это возможно: он прост и непротиворечив в своем поведении.

Пока мы обсуждаем тему заголовка `Accept`, вспомните о специальном формате `all`, упоминавшемся выше. Он существует по той простой причине, что некоторые старые версии броузеров отправляют заголовок `Accept` со значением `*/*`, которое означает: «Я принимаю содержимое любого типа». Этот тип MIME соответствует формату `all`, в чем можно убедиться, взглянув на отображение `grails.mime.types` в файле с настройками.

Это были основы (и не только основы) поддержки технологи REST в платформе Grails. Теперь вы владеете всеми необходимыми инструментами и сможете быстро реализовать интерфейс REST для любого приложения на платформе Grails. Вопрос состоит только в том, как эффективнее использовать эти инструменты?

11.3. Практика реализации интерфейса REST

Реализация простого интерфейса REST для приложения на платформе Grails сводится к простому использованию особенностей, которые мы продемонстрировали к настоящему моменту. Неприятность заключается в наличии некоторых проблем, связанных со встроенной поддержкой. В этом разделе мы представим вам некоторые наиболее практичные приемы и объясним причины, стоящие за ними.

11.3.1. Поддерживайте стабильность API

Начнем с одной из самых интересных особенностей: конвертеров. Конвертеры позволяют одной строкой программного кода сериализовать целую группу объектов предметной области в формат XML, JSON или ваш собственный. Представьте, что произойдет, если вы измените класс предметной области, например удалив некоторое поле? Сериализованная версия класса также изменится – поле, которые вы удалили, исчезнет из представления в формате XML (или в любом другом, в какой выполняется преобразование).

Это может стать источником проблем, потому что интерфейсы REST обычно являются общеизвестными прикладными программными интерфейсами (API), а это означает, что они должны быть стабильными и обеспечивать обратную совместимость. Если вы реализовали создание ответов в формате XML (например) с использованием стандартного конвертера, ваш общеизвестный API начинает зависеть от внутренних особенностей реализации, что действительно не очень хорошо! В идеале вы должны отделить свой интерфейс REST от реализации.

С другой стороны монеты под названием «конвертеры», у нас имеются проблемы с извлечением данных из запросов в формате XML или JSON. Как было показано выше, платформа Grails автоматически извлекает из запросов данные форм HTML, фрагменты XML и JSON и заполняет этими данными отображение `params`. Это здорово, потому что в конеч-

ном итоге данные оказываются в отображении, но не в случае формата XML. Парсер игнорирует все атрибуты, за исключением `id`, и создает вложенные отображения только для элементов с атрибутом `id`. Возьмем в качестве примера произвольный фрагмент XML:

```
<item category="general">
    <name>New message</name>
    <owner>
        <id>glen</id>
        <firstName>Glen</firstName>
        <lastName>Smith</lastName>
    </owner>
</item>
```

Атрибут `category` в корневом элементе не появится в объекте `params`, а также будет потеряна структурированная информация, находящаяся внутри элемента `owner`, в котором отсутствует атрибут `id`. Эта информация слишком важна, чтобы терять ее.

Имеется ли альтернативное решение? Для по-настоящему стабильного API вы должны определить, какие запросы и ответы поддерживаются для каждого адреса URL, и строить свою реализацию вокруг них. Это позволит отказаться от конвертеров, которые для XML гораздо хуже, чем парсинг запросов с помощью класса `XmlSlurper` и отображение ответов с помощью метода `render()` (с использованием синтаксиса построителя разметки) или чем GSP для генерирования ответа.

Как быть с запросами и ответами в формате JSON?

Обычно при реализации интерфейса REST в качестве основного формата представления ресурсов используется XML, поэтому базовые рекомендации и приемы реализации интерфейса REST по большей части касаются именно этого формата. К формату JSON вполне применимы те же рекомендации, а вот приемы применимы в меньшей степени. Например, структура данных в формате JSON хорошо вписывается в структуру вложенных отображений в объекте `params`, поэтому лучше будет позволить платформе Grails автоматически выполнять парсинг запросов JSON и извлекать всю необходимую информацию из отображения `params`.

Что касается ответов, формат JSON плохо соответствует синтаксису GSP, поэтому использование метода `render()` для отображения разметки JSON выглядит предпочтительнее, чем попытка создать представление, такое как show.json.gsp.

Разработчики, создающие клиентов для вашего приложения, вне всяких сомнений по достоинству оценят детальное определение API, которое должно включать адреса URL и структуру сообщений. Вы мог-

ли бы предоставить это описание в виде электронной документации, но в настоящее время существует два стандарта, позволяющие формализовать определение интерфейса REST: язык описания веб-приложения (Web Application Description Language, WADL) и язык определения веб-служб (Web Service Definition Language, WSDL) версии 2.0.

У нас нет большого опыта работы ни с одним из них, но использование WADL или WSDL 2.0 обеспечит дополнительную поддержку. Кроме того, при помощи специализированного языка гораздо проще будет точно описать поведение прикладного интерфейса. Однако определения ни на одном из этих языков нельзя назвать удобочитаемыми, поэтому совсем нелишним будет иметь документацию, описывающую интерфейс понятным языком.

Примечание

Учебник по языку WSDL 2.0 можно найти по адресу *http://www.w3.org/TR/2007/REC-wsdl20-primer-20070626/*, а спецификацию языка WADL – по адресу *https://wadl.dev.java.net/wadl20061109.pdf*.

Теперь, когда вы познакомились с основными приемами поддержания стабильности своего прикладного интерфейса REST, можно посмотреть на них в действии, воспользовавшись в качестве примера приложением Hubbub.

11.3.2. Применяем теорию на практике

Вы познакомились с теорией, теперь мы применим некоторые рассуждения к приложению Hubbub. Начнем с определения отображения адресов URL, которое вы уже видели выше:

```
class UrlMappings {
    def mappings = {
        "/posts"(controller: "postRest") {
            action = [ GET: "list", POST: "save" ]
        }
        "/posts/$id"(controller: "postRest") {
            action = [ GET: "show", PUT: "update", DELETE: "delete" ]
        }
    }
}
```

Теперь нам требуется рассмотреть, как формируются сообщения в формате XML для каждого из этих отображений. Для простоты мы будем рассматривать только методы `GET` и `POST` для URL /posts и `GET` для /posts/$id, однако примеры исходного программного кода приложения Hubbub включают полную реализацию.

Запрос `GET` для URL /posts/$id должен возвращать ответ в формате XML, который выглядит, как показано ниже:

```
<post>
    <content>Message content goes here</content>
    <created>2008-01-05</created>
    <user>
        <username>glen</username>
        <fullName>Glen Smith</fullName>
    </user>
</post>
```

Этот фрагмент немного напоминает сообщение, сгенерированное конструкцией `render post as XML`, но в нем отсутствует идентификатор объекта предметной области. Вместо него мы возвращаем имя пользователя (соответствующее значению поля `userId` в классе `User`), который тоже однозначно идентифицирует пользователя в системе.

Проблема с идентификаторами объектов предметной области состоит в их неочевидности, она может осложнить отладку прикладного интерфейса. Представьте, что вы пытаетесь определить источник ошибки в приложении Hubbub, просматривая содержимое сообщений в формате XML, которые передаются между клиентом и приложением. Что бы вы предпочли видеть: целочисленный идентификатор или имя пользователя? Кроме того, использование идентификаторов объектов предметной области ведет к утечке информации об особенностях реализации вашего прикладного интерфейса.

В ответ на запрос `GET` к URL /posts будет возвращаться корневой элемент `<posts>`, содержащий множество дочерних элементов `<post>`. Запрос `POST` будет выглядеть еще проще:

```
<post>
    <content>New content goes here</content>
</post>
```

Этот запрос создаст новое сообщение в приложении Hubbub от имени зарегистрированного пользователя. Вопрос регистрации пользователя с помощью интерфейса REST мы обсудим в конце данного раздела.

Теперь, когда мы определили формат сообщений, можно перейти к реализации на стороне сервера. В листинге 11.4 приводится последняя версия нашего контроллера `postRest`, реализующая прикладной интерфейс, который мы только что определили.

Листинг 11.4. Разбор запросов в формате XML вручную в контроллере, реализующем интерфейс REST

```
package com.grailsinaction

class PostRestController {
    def authenticateService
    def postService

    def list = {
        [ posts: Post.list() ] ← ❶ Используется представление для формата XML
```

```
    }

    def show = {
        def post = Post.get(params.id)
        if (post) {
            return [ post: post ]
        }
        else {
            response.sendError(404) ← ❷ Если значение id не найдено,
        }                                генерируется ошибка с кодом 404
    }

    def save = {
        def user = authenticateService.userDomain()
        def newContent = request.XML.content.text() ← ❸ Извлекает содержимое
                                                       XML из запроса
        def post = postService.createPost(user.id, newContent)

        if (post && !post.hasErrors()) {
            response.status = 201          ← ❹ При добавлении нового сообщения
            render ""                         генерируется ответ с кодом 201
        }
        else {
            response.status = 403 ← ❺ Генерируется сообщение об ошибке
            render contentType: "text/xml", encoding: "utf-8", {
                errors {
                    post?.errors?.fieldErrors?.each { err ->
                        field(err.field)
                        message(g.message(error: err))
                    }
                }
            }
        }
    }
}
```

В листинге 11.4 мы ограничились только поддержкой формата XML, чтобы сократить объем программного кода. Реализация действия `list` получилась очень простой, потому что мы перенесли создание ответа XML в представление ❶.

То же относится и к реализации действия `show`; ниже приводится его представление GSP (grails-app/views/postRest/show.xml.gsp):

```
<%@ page contentType="text/xml; charset=utf-8" %>
<post>
    <content>${post.content}</content>
    <created>
        <g:formatDate date="${post.created}" format="yyyy-MM-dd"/>
    </created>
    <user>
        <userId>${post.user.userId}</userId>
```

```
            <fullName>${post.user.profile.fullName</fullName>
        </user>
    </post>
```

Действие `show` имеет более интересную реализацию, чем действие `list`, потому что в нем дополнительно предусмотрена отправка сообщения об ошибке с кодом 404 ❷ в случае, если не будет найдено ни одного сообщения с запрошенным идентификатором. Подобное использование кодов состояний протокола HTTP – распространенная практика, и подробнее об этом будет рассказываться в конце данного подраздела. Обычно нет необходимости возиться с кодами состояний, потому что код 200, используемый по умолчанию, прекрасно подходит для случаев успешного завершения работы действий.

О методе sendError()

Использование метода `sendError(int code)` для создания ответа является стандартным способом отправки сообщения об ошибке с кодом состояния HTTP в веб-приложениях на языке Java. Однако этот метод имеет некоторые ограничения. Во-первых, вы не сможете с его помощью отправить содержимое ответа. Во-вторых, ответ будет иметь тип содержимого `text/html`.

В нашем распоряжении имеются два способа преодолеть эти ограничения: вручную установить код состояния ответа и добавить содержимое ответа обычным способом или добавить отображение кода состояния в настройки отображения адресов URL:

```
"500"(controller: "errors", action: "serverError")
```

Действие `save` получилось чуть более сложным, хотя его реализация должна выглядеть для вас знакомой. Сначала мы получаем объект с информацией о текущем пользователе, чтобы новое сообщение можно было ассоциировать с ним, и затем извлекаем содержимое нового сообщения из запроса в формате XML ❸. После этого мы делегируем создание сообщения соответствующей службе и проверяем, действительно ли оно было создано.

Согласно спецификации протокола HTTP в результате успешного создания нового ресурса должен возвращаться код состояния 201, что мы и делаем ❹. Кроме того, мы должны определить значение HTTP-заголовка `Location`, но проблема с обратным отображением адресов URL не позволяет сделать это без жесткого определения адреса URL в программном коде. С другой стороны, если новое сообщение не было создано по какой-либо причине, мы создаем ответ в формате XML, который содержит информацию о причинах отказа ❺. Помимо этого, чтобы показать, что запрос потерпел неудачу и повторная попытка не приведет к успеху, мы используем код состояния 403.

Использование кодов состояния протокола HTTP

Выбрать код состояния для использования в том или ином ответе бывает совсем непросто, поэтому мы перечислили некоторые из наиболее часто применяемых кодов в табл. 11.1, сопроводив каждый из них рекомендациями по использованию. Полный перечень с описаниями вы найдете по адресу *http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html*.

Таблица 11.1. Наиболее часто используемые коды состояния HTTP и рекомендации по их использованию

Код	Краткое наименование	Описание
200	OK	Запрос выполнен успешно.
201	Created (Создано)	Новый ресурс создан. Заголовок `Location` должен содержать «как можно более точный» URI нового ресурса, а содержимое ответа должно включать список возможных URI.
301	Redirected Permanently (Перемещено окончательно)	Ресурс был окончательно перемещен по другому URI
302	Found (Найдено)	Указывает, что запрошенный ресурс временно доступен по другому URI. Этот код состояния возвращается методом `redirect()`.
400	Bad Request (Плохой запрос)	Содержимое запроса сформировано неправильно или содержит ошибку. Например, этот код состояния можно использовать при получении сообщения XML, неправильного для конкретного адреса URL, или при содержании недопустимой разметки XML в сообщении.
401	Unauthorized (Не авторизован)	Для доступа к ресурсу требуется авторизация клиента. Этот код допускается, только когда используется один из стандартных методов аутентификации по протоколу HTTP (например, базовый).
403	Forbidden (Запрещено)	Запрос не может быть выполнен из-за имеющихся ограничений. Это многоцелевой код состояния, который может использоваться при обнаружении ошибок в ходе проверки ограничений и любых других возникших на стороне клиента ошибок, которые не соответствуют другим кодам состояния.
404	Not Found (Не найдено)	Ресурс не найден – это самый известный код состояния!
500	Internal Server Error (Внутренняя ошибка сервера)	Обычно этот код указывает на ошибку, возникшую в реализации сервера. Контейнеры сервлетов возвращают код 500 в случае появления необработанного исключения.

При разумном использовании коды состояния могут служить неплохим дополнением к прикладному интерфейсу REST и помочь предоставить клиентам ценную и легкодоступную обратную связь. Объединив их с соответствующим содержимым ответов, вы сможете создать систему, способную обрабатывать практически любые ситуации.

Коды состояния также имеют отношение к способу аутентификации программных клиентов через интерфейс REST. Если вы посмотрите на табл. 11.1, то увидите подсказку: код 401.

Аутентификация программных клиентов

В нашем приложении Hubbub реализована строгая система управления доступом, поэтому для выполнения операций клиенты должны пройти процедуру аутентификации. Люди могут сделать это на главной странице, где присутствует форма регистрации, но программные клиенты не могут работать со страницами HTML. Они могли бы напрямую обратиться по адресу URL, где выполняется регистрация, но для этого они должны обеспечить поддержку соответствующих cookies.

Самое простое решение, предоставляемое протоколом, – базовая аутентификация. Клиенты могут отправлять идентификационную информацию с помощью заголовков HTTP в каждом запросе. Если указана верная идентификационная информация, им предоставляется доступ к ресурсу. В противном случае сервер возвращает код состояния 401.

Этот подход имеет две основные проблемы. Во-первых, вам придется передавать идентификационную информацию при каждом запросе. Причем приложение будет вынуждено каждый раз проверять достоверность этой информации. Данная проблема обычно не является значительной. Во-вторых, идентификационная информация передается в открытом виде, поэтому любой сможет узнать ее, перехватив трафик между клиентом и сервером.

Для приложения Hubbub ни одна из этих проблем не является достаточно существенной, потому что мы не храним конфиденциальную информацию. Для других систем предпочтительнее использовать базовую аутентификацию в сочетании с протоколом HTTPS.

На этом мы завершаем обсуждение прикладного интерфейса REST. Основные принципы его построения чрезвычайно просты, и платформа Grails предлагает вам в помощь неплохой набор особенностей, от реализации до тестирования. Добавьте сюда подходящее расширение системы безопасности, и все готово.

Как вы уже видели, REST – мощный и гибкий архитектурный шаблон, который может использоваться во многих приложениях. Однако это не означает, что он пригоден для всех ситуаций. Рассмотрим простой пример: приложение, которое просто выводит все, что было ему отправлено. В таком приложении нет никаких ресурсов, есть только операция вывода, поэтому выбор шаблона REST для его реализации будет ошибкой.

Более подходящим будет решение, ориентированное на выполнение операций; в следующем разделе мы рассмотрим несколько таких реализаций, которые основаны на применении служб, а не контроллеров.

11.4. Командно-ориентированные удаленные взаимодействия

Поддержка технологии REST в платформе Grails основана на использовании контроллеров, потому что при этом выполняются операции над ресурсами и создается представление их содержимого. Существует еще одна широко используемая модель удаленных взаимодействий, основанная на службах и методах. Эта модель будет более понятна программистам, работающим с объектно-ориентированными языками, потому что им каждый день приходится вызывать методы объектов. Такой подход знаком им очень хорошо.

В языке Java реализован свой протокол удаленных взаимодействий, который, возможно, уже знаком вам: вызов удаленных методов (Remote Method Invocation, RMI). Он позволяет вызывать методы Java-объектов, находящихся в другой виртуальной машине Java или даже на другом компьютере.

Мы начнем с того, что покажем вам, как обеспечить доступность служб приложения на платформе Grails с помощью RMI. Затем мы познакомим вас с другими протоколами и обсудим достоинства и недостатки каждого из них (включая RMI). В заключение мы рассмотрим одну из крупнейших технологий удаленных взаимодействий: веб-службы. Хотя мы не являемся поклонниками этой технологии, она получила настолько широкое распространение, что заслуживает обсуждения. Но сначала реализуем возможность вызова методов удаленного объекта.

11.4.1. Расширение Remoting

Один из самых простых способов обеспечить доступ программных клиентов к приложению заключается в реализации возможности удаленного вызова процедур (Remote Procedure Calls, RPC) в службе Grails. В приложениях на языке Java для этой цели традиционно используется протокол RMI, и мы продемонстрируем, насколько просто реализовать поддержку этого протокола, на примере службы сообщений в приложении Hubbub.

Начнем с установки расширения Remoting:

```
grails install-plugin remoting
```

Затем нам придется немного потрудиться. Службы платформы Grails обычно не реализуют интерфейсы, но без реализации хотя бы одного интерфейса служба не может быть экспортирована ни по протоколу RMI, ни по любому другому протоколу. Интерфейс – это определение

поддерживаемого прикладного интерфейса API, а служба Grails – реализация. Значит, нам необходимо создать интерфейс для нашей службы сообщений.

Где должно находиться определение интерфейса? Лучше всего для этих целей подходит каталог src, где находится весь программный код, не имеющий прямого отношения к платформе Grails. Какой язык программирования использовать, Java или Groovy? В интерфейсах нет ничего сложного, поэтому нет большого смысла писать их на языке Groovy. Пишите интерфейсы на языке Java, и вы сможете копировать их в проекты и на языке Java, и на языке Groovy.

Мы обеспечим доступность для удаленных клиентов только метода `createPost()`, поэтому интерфейс получится очень простым:

```
package com.grailsinaction;

public interface RemotePostService extends java.rmi.Remote {
    long createPost(String userId, String content)
                    throws java.rmi.RemoteException;
}
```

Сохраните это определение в файле src/java/com/grailsinaction/RemotePostService.java. Мы дали интерфейсу имя `RemotePostService`, а не `PostService`, чтобы избежать конфликта имен с реализацией службы. К сожалению, протокол RMI вынуждает наследовать интерфейс `Remote` и объявлять все методы как возбуждающие исключение `RemoteException`; если бы не это требование, программный код получился бы проще и понятнее.

Теперь, когда мы определили интерфейс, необходимо изменить реализацию службы так, чтобы она использовала его. В настоящий момент сигнатура метода службы выглядит так:

```
Post createPost(long userId, String content)
```

Она не соответствует сигнатуре, описываемой интерфейсом, поэтому нам нужно либо изменить текущую реализацию, либо добавить дополнительный метод. Так как у нас уже имеется программный код, использующий существующий метод, мы добавим новый метод. Получившийся результат приводится в листинге 11.5, где также показано, как активировать поддержку RMI в службе.

Листинг 11.5. Реализация поддержки протокола RMI в службе сообщений приложения Hubbub

```
import com.manning.graina.hubbub.RemotePostService

class PostService implements RemotePostService {
    static expose = [ "rmi" ] ← ❶ Экспортирование службы по протоколу RMI

    long createPost(String username, String content) { ← Реализация метода интерфейса
        def user = User.findByUserId(username)
```

```
            if (!user) throw new RuntimeException("'$username' not found.")

            def post = createPost(user.id, content)
            if (!post) throw new RuntimeException("Creation failed.")
            return post.id
        }

        Post createPost(long userId, String content) {
            ...
        }
    }
```

Включение поддержки протокола RMI в службе выполняется просто: достаточно добавить статическое свойство `expose` и записать в него список, содержащий значение `"rmi"` ❶. В этот список можно добавить и другие протоколы, как вы увидите немного ниже.

Мы преднамеренно постарались избежать некоторых сложностей, воспользовавшись в сигнатуре метода только элементарными типами данных; вот почему метод возвращает значение типа `long`, а не экземпляр класса `Post`. Это не означает, что вы не можете передавать между клиентом и сервером такие типы, как `Post` (вы, конечно же, сможете это сделать), но для этого придется приложить дополнительные усилия.

Во-первых, вам потребуется обеспечить доступность типа данных на стороне клиента. Лучший способ реализовать это – упаковать все интерфейсы и определения типов данных в отдельный файл JAR и использовать его на стороне клиента и на стороне сервера, включив его как зависимость в оба проекта. В случае приложения Hubbub мы могли бы назвать его hubbub-client.jar. Во-вторых, определения типов должны находиться в пакетах, поскольку только классы в пакете по умолчанию могут обратиться к другим классам, находящимся в нем. В-третьих, объекты классов предметной области не всегда бывает просто передать по сети, а клиенты определенно не должны видеть определения ограничений, отображений и другую внутреннюю информацию. По этой и другим причинам разработчики обычно стремятся копировать все необходимые данные из объектов предметной области в так называемые объекты переноса данных (Data Transfer Objects, DTO) – простые классы, содержащие только поля данных. Это все, что мы хотели сказать по данному поводу.

Клиентское приложение удобно использовать для тестирования службы, поэтому мы включили простое приложение клиента в состав примеров исходных текстов для данной книги. Вы найдете его в подкаталоге rmi-client в каталоге с примерами для этой главы. Он включает простой файл сборки для утилиты Ant, который позволит запустить клиента, как показано ниже:

```
ant -Dmessage='This is a new Hubbub post via RMI.' run
```

Если вы чувствуете в себе заинтересованность, подумайте о создании второго приложения на платформе Grails, которое использовало бы расширение Remoting для доступа к службе сообщений приложения Hubbub по протоколу RMI. Подробнее узнать о том, как превратить приложение на платформе Grails в удаленного клиента, можно в документации расширения: *http://grails.org/Remoting+Plugin*.

Теперь, когда вы знаете, как экспортировать службу Grails по протоколу удаленных взаимодействий, можно познакомиться с другими протоколами и сравнить их друг с другом.

11.4.2. Сравнение протоколов удаленных взаимодействий

Помимо протокола RMI расширение Remoting обеспечивает поддержку еще трех протоколов, которые можно включить, добавив соответствующие значения в свойство `expose`: `httpinvoker`, `hessian` и `burlap`. Все эти протоколы прекрасно уживаются друг с другом, поэтому вы можете добавлять их в список `expose` в любых комбинациях и даже все четыре сразу. Какой из них следует использовать и почему?

RMI

RMI – основной протокол удаленных взаимодействий, но это не означает, что он лучший. Поддержка протокола RMI появилась в JDK версии 1.1, поэтому он существует достаточно давно. Если ваши клиенты будут создаваться только на языке Java, имеет смысл подумать об использовании протокола RMI, так как он не имеет других зависимостей, кроме самой библиотеки JDK.

Главным недостатком протокола RMI помимо того, что он поддерживается только языком Java, является его насильственность. Любой интерфейс, экспортируемый по протоколу RMI, должен наследовать класс `java.rmi.Remote`, а все его методы должны объявляться как способные возбуждать исключение `java.rmi.RemoteException`. В наше время разработчики справедливо ожидают, что такие особенности реализации не должны просачиваться в общедоступный API.

Использованию протокола RMI также не способствует то, что он требует отдельной регистрации на сервере и не работает поверх протокола HTTP. Это уменьшает его шансы на использование в сетях с брандмауэрами. К счастью, существует альтернатива, которая лишена этих проблем и даже проще в применении.

HTTP invoker

В состав платформы Spring входит свой собственный протокол, который называется «HTTP invoker». По своей сути это протокол RMI, действующий поверх HTTP благодаря использованию стандартного механизма сериализации языка Java. Следовательно, он прекрасно рабо-

тает в сетях с брандмауэрами, и вам не потребуется регистрировать его на сервере, потому что местоположение удаленной службы определяется адресом URL.

В этом заключается самое большое его преимущество перед протоколом RMI, но больше всего нам нравится его ненасильственность. Ваши классы и интерфейсы не должны наследовать или реализовать какие-либо специфические классы и интерфейсы. По этим причинам мы настоятельно рекомендуем вам использовать этот протокол вместо RMI.

Для нас было бы непростительно, если бы мы не упомянули отрицательные стороны протокола. Как и протокол RMI, он реализован только в языке Java, но существует дополнительное ограничение: ваш клиент должен будет использовать платформу Spring. Является ли это недостатком, зависит от конкретных требований, но мы предпочитаем видеть в этом хороший повод использовать платформу Spring во всех своих приложениях!

Hessian и Burlap

Оставшиеся два протокола были разработаны одной и той же компанией, Caucho. В действительности Burlap – это XML-версия протокола Hessian (двоичный протокол). Главное преимущество этих протоколов перед другими двумя состоит в том, что они могут использоваться не только в клиентах на языке Java. Кроме того, в качестве транспортного протокола они используют протокол HTTP, поэтому у вас не должно возникать проблем с брандмауэрами.

Из недостатков, с которыми вы можете столкнуться, можно назвать слишком сложную иерархию объектов. Это обусловлено тем, что протокол не использует стандартный механизм сериализации языка Java. Кроме того, вы можете обнаружить, что протокол Hessian действует немного медленнее, чем HTTP invoker, хотя, возможно, не настолько, чтобы беспокоиться об этом.

Мы рекомендуем использовать протокол Hessian, только если потребуется обеспечить доступ к приложению для клиентов, написанных не на языке Java. Что касается протокола Burlap, мы не видим в нем особого смысла. Единственное его преимущество перед протоколом Hessian заключается в том, что вы сможете читать сообщения, передаваемые между клиентом и сервером, потому что они имеют формат XML, и, значит, этот протокол удобно использовать для отладки.

Как вы могли убедиться, и технология REST, и расширение Remoting обеспечивают простые способы организации взаимодействий между приложениями, но вследствие их простоты они испытывают недостаток некоторых мощных особенностей, таких как безопасная и надежная передача сообщений, управление маршрутизацией сообщений, и многих других. Если простые решения не отвечают вашим требованиям, вероятно, вам следует окунуться в мир веб-служб.

11.4.3. Веб–службы и протокол SOAP

Если вам приходилось работать с языком Java в течение некоторого времени, вы, скорее всего, слышали о веб-службах. Многие знаменитости, известные в мире программного обеспечения, вложили значительные усилия в развитие и продвижение этой технологии. Безусловно, обещаемые преимущества веб-служб выглядят весьма привлекательно: безопасная и надежная передача сообщений через Интернет, полная функциональная совместимость различных систем. В прошлом сложность, присущая веб-службам, вызывала проблемы с функциональной совместимостью. Однако усилия таких организаций, как организация по обеспечению совместимости веб-служб (Web Services Interoperability Organization, WS-I), несколько улучшили ситуацию.

Что такое веб-службы? По сути это обычные службы, взаимодействующие с клиентами по протоколу SOAP, обычно поверх HTTP. Некоторые используют термин «веб-служба», подразумевая также интерфейс REST, однако все технологии, находящиеся под зонтиком веб-службы, основаны на использовании протокола SOAP.

Первоначально протокол SOAP сам по себе являлся реализацией механизма удаленного вызова процедур (Remote Procedure Call, RPC), использующей язык XML для описания методов, их аргументов и возвращаемых данных. В настоящее время его лучше рассматривать как систему передачи сообщений, где вы отправляете веб-службе сообщение в формате XML и получаете в ответ другое сообщение. На рис. 11.4 изображена простота природы большинства взаимодействий по протоколу, основанных на передаче запросов и ответов, а также структура запроса SOAP. Как видите, модель запроса SOAP вообще не отличается от запроса HTTP.

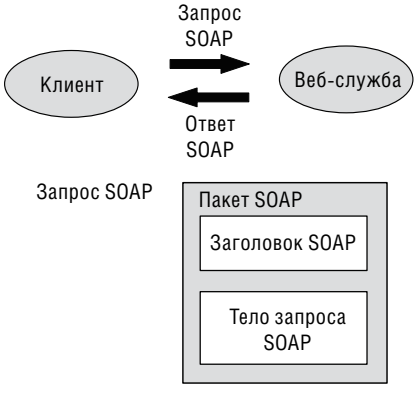


Рис. 11.4. *Порядок взаимодействия клиента и веб-службы с использованием модели запрос-ответ. Ниже схематически изображена структура запроса SOAP – обратите внимание на сходство с протоколом HTTP*

Мир веб-служб огромен и заслуживает отдельной книги. Мы не сможем охватить все аспекты веб-служб в этом коротком разделе, но мы покажем вам небольшой пример использования одного из расширений для платформы Grails и затем поговорим о том, когда следует использовать веб-службы (если вообще их следует использовать).

Расширения для работы с веб-службами

На момент написания этих строк существовало два основных расширения для работы с веб-службами. Оба расширения, xfire и axis2, используют различные библиотеки Java. Расширение xfire основано на библиотеке XFire, которую в настоящее время заменила библиотека CXF. Скорее всего, в ближайшее время расширение также перейдет на использование библиотеки CXF. В следующем примере мы будем использовать расширение axis2, основанное на использовании библиотеки Apache Axis 2.

Сначала необходимо установить расширение axis2:

```
grails install-plugin axis2
```

Затем мы экспортируем службу сообщений как веб-службу, добавив строку `axis2` в список `expose`:

```
class PostService {
    static expose = [ "rmi", "axis2" ]
    ...
}
```

Вот и все! Не верите? Запустите приложение и откройте в броузере адрес URL: *http://localhost:8080/hubbub/services/post?wsdl*. Если все действует, как надо, вы должны увидеть довольно жутко выглядящий фрагмент XML.

Диалект языка XML носит название язык определения веб-служб (Web Service Definition Language, WSDL), и он используется для описания таких особенностей интерфейса веб-службы, как допустимые операции и содержимое сообщений SOAP. Вы должны будете передать это описание на языке WSDL, чтобы предпочитаемая вами библиотека для работы с веб-службами или набор инструментальных средств смогли сгенерировать классы, необходимые для создания клиентского приложения.

Данный пример лишь слегка касается проблемы использования протокола SOAP и веб-служб, поэтому по нему трудно судить о сложности этой технологии или о ее преимуществах. Следует ли вам идти этим путем?

Использовать или не использовать SOAP

Несмотря на кажущуюся простоту создания веб-служб на базе протокола SOAP, если вы хотите всерьез заняться веб-службами, вам придется начать с определения сообщений запросов и ответов, а не с реализации службы, как и в случае использования технологии REST. Это озна-

чает, что вам придется иметь дело с WSDL и с пространствами имен XML – возможно, вам даже придется вспомнить свои навыки владения XML Schema. Как вы уже видели, WSDL – не самый лаконичный диалект языка XML и не самый простой в постижении для простых смертных. С другой стороны, существуют инструменты, которые помогут вам создавать и использовать описания на языке WSDL.

Такая сложность делает веб-службы мощными и гибкими инструментами, обладающими многими особенностями, основанными на SOAP и WSDL, такими как: вложения, для включения в сообщения двоичных данных; WS-Addressing, для определения маршрута движения сообщения; WSS, для управления доступом к веб-службам и шифрования сообщений; и другими. В действительности диапазон расширений базовой платформы поистине впечатляющий.

На наш взгляд, за все это приходится платить снижением производительности разработчика, потому что веб-службы сложны для понимания и в отладке. Конечно, существуют инструменты, способные облегчить вашу жизнь, но если мы что-то и поняли за те годы, что мы занимались разработкой программного обеспечения, так это то, что, в конечном счете, все равно приходится разбираться с тем, что происходит за кулисами. Это совершенно не означает, что вам никогда не следует заниматься веб-службами и стоит держаться подальше от них. Мы рекомендуем вам изучить все имеющиеся возможности, чтобы вы могли принимать обоснованное решение в каждой конкретной ситуации. Излишняя сложность приводит к снижению производительности труда, но решение, не соответствующее вашим требованиям, – это еще хуже.

Несмотря на рост популярности технологии REST в последние годы, мы надеемся что, нам удалось продемонстрировать, что она – не единственная возможность реализации взаимодействий между приложениями. Когда вы контролируете создание не только сервера, но и клиента, реализация удаленных взаимодействий может быть выполнена быстрее и проще. Однако веб-службы часто формируют основу решений для системы обслуживания запросов (Service-Oriented Architecture, SOA). Главное, как всегда, правильно выбрать инструмент для решения поставленной задачи.

11.5. В заключение

Предоставление программным клиентам доступа к вашему приложению требует подхода, отличающегося от подхода к реализации обычных пользовательских интерфейсов на базе разметки HTML. Настоящие пользователи должны получать данные в таком виде, чтобы их легко было прочитать и понять, в то время как для программных клиентов важны только сами данные. Можно было бы положиться на использование клиентов, выполняющих анализ обычной разметки HTML, получаемой от вашего приложения, но это неудобно и ненадежно. Как вы

уже видели, нет ничего сложного в том, чтобы снабдить свое приложение интерфейсом, который облегчит жизнь программным клиентам, так почему бы не использовать такую возможность?

В этой главе мы занимались в основном реализацией архитектурного шаблона REST, который идеально подходит для использования в веб-приложениях и поддерживается платформой Grails. Вы легко можете создать для своего приложения безопасный и надежный прикладной интерфейс, опираясь на использование протокола HTTP.

Несмотря на то что значительную долю внимания мы уделили технологии REST, вы должны знать и о других возможностях – механизмы, ориентированные на выполнение операций, могут лучше соответствовать целям вашего прикладного интерфейса. Кроме того, разработчики на языке Java обычно стремятся использовать технологии удаленных взаимодействий, основанные на службах, простых для понимания и в работе, потому что они базируются на знакомых концепциях интерфейсов и методов.

Ниже приводятся несколько рекомендаций, которые помогут повысить привлекательность вашего прикладного интерфейса для разработчиков:

- *Используйте простейшее из решений, удовлетворяющих вашим потребностям.* Разработчики часто попадаются на шумиху, развернутую вокруг некоторых технологий, и чувствуют себя обязанными использовать их, потому что они приобрели или приобретают большую популярность. Не забывайте, что от вас требуется как можно быстрее создать полезный прикладной интерфейс, а не продвигать технологии.

  Отыщите самое простое решение, удовлетворяющее вашим потребностям. Если в ваших руках находится разработка не только сервера, но и клиента, таким решением может быть расширение Remoting, даже хотя его использование не считается чем-то крутым. Если у вас сложные требования, единственным выбором могут оказаться веб-службы. Однако во многих ситуациях технология REST оказывается достаточно мощной и простой, чтобы стать оптимальным решением.

- *Тщательнее обдумывайте структуру своих адресов URL.* Хорошими считаются долгоживущие, непротиворечивые и простые для понимания адреса URL. И хотя неудачные адреса URL, скорее всего, не окажут существенного влияния на реализацию богатого особенностями прикладного интерфейса, они могут осложнить жизнь тем, кто будет ими пользоваться. Это только оттолкнет пользователей. В Интернете можно найти массу статей о проектировании удобной структуры адресов URL.

- *Используйте принцип обмена сообщениями при работе с технологией REST.* Если вы для своего приложения разрабатываете интерфейс REST, который, как предполагается, будет использоваться на

протяжении нескольких лет, очень важно отделить прикладной интерфейс от реализации. Это означает, что сначала вы должны спроектировать структуру адресов URL и содержимое сообщений, которые будут составлять соответствующие запросы и ответы.

Определившись с сообщениями, опишите их в документации, чтобы дать пользователям возможность узнать, как вы используете свой прикладной интерфейс. Кроме того, подумайте о возможности использования WADL или WSDL 2.0 для создания формального описания.

- *Используйте наиболее подходящие по смыслу методы HTTP.* Отобразите различные методы HTTP в действия, которые соответствуют ожиданиям и требованиям методов. Например, не отображайте метод `GET` в действие, которое добавляет, удаляет или изменяет данные на сервере, потому что в этом случае будет нарушена семантика, определяемая протоколом HTTP.
- *Используйте коды состояния HTTP.* Возвращать код состояния 200 в ответ на каждый запрос и об успешности или об ошибке запроса сообщать в содержимом ответа – просто. Просто, но это плохая практика. Вы могли бы задействовать гибкий и однозначный механизм сообщений об ошибках, который будет понятен любым клиентам HTTP (таким как броузеры), а не только тем, что разрабатывались специально для работы с вашим прикладным интерфейсом.
- *Используйте механизмы управления доступом.* Обычно бывает нежелательно предоставлять пользователям неограниченный доступ ко всем данным в системе. То же относится и к программным клиентам. Помните, что к вашему приложению по протоколу HTTP может обратиться любой желающий, и это не всегда будет один из добропорядочных клиентов. Обязательно предусмотрите соответствующее управление доступом в своем интерфейсе REST.

От взаимодействий между приложениями мы переходим к реализации системы внутренних взаимодействий, которая позволит вам координировать работу различных частей приложения, или нескольких внутренних приложений, или подсистем.

В этой главе:

- Как действует обмен сообщениями
- Как улучшить масштабируемость архитектуры
- Как реализовать очереди и темы с помощью популярного сервера JMS
- Реализация планирования заданий и управление ими
- Возможности планирования постоянных и групповых заданий

12

Обмен сообщениями и планирование заданий

В предыдущей главе мы потратили некоторое время на исследование способов организации удаленных взаимодействий в приложениях на платформе Grails и узнали, как создаются и используются сообщения, передаваемые между приложениями. В этой главе мы продолжим изучение этой темы, но теперь будем рассматривать способы передачи сообщений внутри приложения. В частности, мы исследуем, как различные компоненты приложения могут взаимодействовать *друг с другом* по мере происхождения различных событий в приложении. Одним из наиболее популярных способов организации таких взаимодействий является реализация очередей сообщений, архитектуры, иногда называемой *промежуточным программным обеспечением для рассылки сообщений* (Message-Oriented Middleware, MOM).

Если вам приходилось вращаться в корпоративных кругах, вы, вероятно, уже использовали или слышали об архитектурах MOM. Вы могли бы подумать, что это какая-то тяжеловесная, устаревшая технология, которая никак не вымрет. Ничего подобного. В действительности с появлением архитектуры предоставления услуг (Service-Oriented Architecture, SOA), такой как сервисная шина предприятия (Enterprise Service Bus, ESB), и с ростом сайтов социальных сетей в стиле Web 2.0 мы наблюдаем взрыв интереса к архитектурам обмена сообщениями.

Вас может удивить, почему такого рода архитектуры получили широкое распространение в последние годы. Если вы посмотрите на любое крупное веб-приложение, от Twitter до Digg и LinkedIn, вы обнаружите, что они поддерживаются обширной инфраструктурой обмена сообщениями. Архитектуры обмена сообщениями получили такое большое распространение на этих сайтах по трем основным причинам:

- Они обеспечивают возможность создания слабо связанных архитектур, что позволяет замещать элементы инфраструктуры, не вынуждая клиента простаивать в ожидании.
- Они обеспечивают высокую масштабируемость – вы можете добавлять дополнительные компоненты, которые будут заниматься обработкой вашей очереди.
- Они обеспечивают высокую надежность и гарантируют, что ваши сообщения и транзакции не будут потеряны в системе.

В этой главе мы добавим в приложение Hubbub связь с популярной системой обмена мгновенными сообщениями Jabber. Когда мы закончим реализацию, вы сможете отправлять сообщения в свою учетную запись в приложении Hubbub с помощью клиента обмена мгновенными сообщениями. Мы также реализуем связь в обратную сторону, что позволит вам получать в своем клиенте обмена мгновенными сообщениями уведомления от своих друзей по приложению Hubbub. Попутно вы познакомитесь с особенностями распространенных сценариев обмена сообщениями и получите некоторое представление о том, как можно их использовать в своих текущих проектах.

Но обмен сообщениями – не единственная асинхронная особенность. Во многих ситуациях бывает необходимо иметь легковесное решение, позволяющее планировать выполнение заданий в определенное время. Необходимо реализовать ежедневное резервное копирование? Отправлять по электронной почте ежедневные обзоры? Заново генерировать индекс для полнотекстового поиска? Каждому разработчику время от времени приходится иметь дело с подобными запланированными событиями, и для этих целей платформа Grails предоставляет надежный и легко настраиваемый планировщик на базе популярной платформы Quartz. Мы рассмотрим различные способы планирования заданий – как писать задания, создающие ежедневные обзоры, как включать и выключать их в процессе работы приложения и как действует система планирования в кластеризованном окружении.

С особенностями планирования мы познакомимся немного ниже, а пока займемся обменом сообщениями.

12.1. Автостопом по работе с сообщениями

Механизмы передачи сообщений существуют уже целую вечность, но чаще всего они использовались для организации обмена сообщениями на уровне предприятия, поэтому вы можете ничего не знать о том, как действует подобная архитектура. В этом разделе мы обсудим основы обмена сообщениями и научим вас отправлять и получать сообщения с помощью сервера Java Message Service (JMS). Готовьтесь!

12.1.1. Учимся мыслить асинхронно: когда можно использовать системы обмена сообщениями?

Часто компоненты приложения требуют для своей работы существенных затрат времени и ресурсов, и от них не требуется, чтобы они выполняли свою работу мгновенно. Одним из примеров является создание документа PDF для отправки пользователю по электронной почте. Когда пользователь заказывает такой документ, вы сообщаете ему, что файл PDF будет отправлен по электронной почте, но сама работа не может быть выполнена немедленно. Создание документа PDF может потребовать значительных затрат вычислительных мощностей, и было бы нежелательно задерживать работу остальных пользователей, пока сервер занимается созданием файла PDF для единственного пользователя. Практичнее может оказаться создать и отправить документ PDF где-нибудь в течение следующей минуты или около того. Но это должно быть сделано в самое ближайшее время, и выполнение этого задания должно быть гарантировано.

Это классический пример, когда может пригодиться система обмена сообщениями, а такой подход «сделать как можно скорее» известен как *асинхронная обработка*. Вот как все могло бы быть реализовано за кулисами. Когда пользователь запрашивает документ, указывая адрес пересылки, сообщение с необходимой информацией помещается во внутреннюю очередь. Эта операция может быть выполнена немедленно, и вы тут же можете сообщить пользователю, что документ PDF будет отправлен по электронной почте. Другой процесс, возможно, даже на другом сервере (в том же сегменте сети, с доступом к почтовому серверу), извлекает запрос из очереди, генерирует документ PDF и отправляет его пользователю по электронной почте. На рис. 12.1 изображено, как запрос на получение документа PDF попадает к целевому процессу через очередь сообщений.

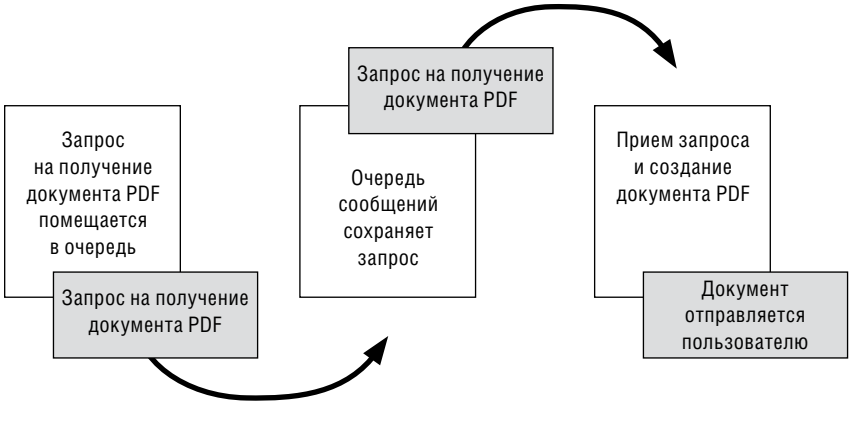


Рис. 12.1. *Передача запроса на получение документа PDF целевому процессу через очередь сообщений*

Одна из интересных особенностей такого асинхронного подхода состоит в том, что сервер сообщений будет хранить запрос клиента, пока не появится возможность выполнить его. Если процедура создания документов PDF является самым узким местом, у вас может появиться несколько клиентов, ожидающих, пока не подойдет очередь создавать и отправлять их документы PDF; сервер сообщений обеспечит семантику транзакций, гарантируя удаление запросов из очереди, как только они будут обслужены.

Теперь, когда вы понимаете, для каких операций может использоваться асинхронный подход, можно познакомиться с терминологией реализации, которую вы должны знать. Попробуем реализовать наш первый механизм обмена сообщениями для приложения Hubbub.

12.1.2. Терминология механизмов обмена сообщениями: поставщики, потребители, темы и очереди

Прежде чем переходить к реализации механизма обмена сообщениями, необходимо познакомиться с некоторыми основными терминами JMS. Везде, и в документации к расширениям, и в статьях, предполагается, что вы знаете, что такое темы, очереди и поставщики, поэтому мы сначала опишем их и дадим вам некоторое представление о том, в каких ситуациях какие приемы обмена сообщениями применяются.

Во-первых, на сцене JMS играют два основных актера:

- *Поставщики* создают и помещают сообщения в очередь.
- *Потребители* извлекают сообщения из очереди.

В примере с документом PDF веб-приложение (поставщик) помещает в очередь новый запрос на отправку документа PDF, а приложение, отправляющее документ по электронной почте (потребитель), извлекает запрос из очереди.

Как взаимодействуют потребители и поставщики? Сервер JMS предлагает две основные модели взаимодействий:

- *Очереди*. Очереди действуют по принципу «первым пришел, первым ушел», при этом каждое сообщение, помещаемое поставщиком в очередь, обрабатывается одним (и только одним) потребителем. Иногда это называют *двухточечным* (point-to-point) способом обмена сообщениями, как показано на рис. 12.2.
- *Темы*. При использовании тем применяется *широковещательная* модель передачи сообщений, когда все потребители, подписавшиеся на получение сообщений с определенной темой, получают свою копию сообщения. Поставщик помещает в очередь одно сообщение, которое затем дублируется и передается множеству потребителей одновременно. Эта схема изображена на рис. 12.3.

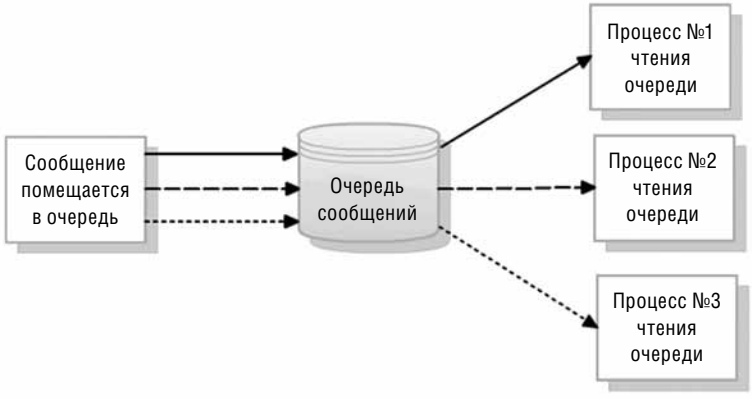


Рис. 12.2. *Очереди сообщений действуют как механизм доставки из одной точки в другую, при этом каждое сообщение передается единственному процессу*

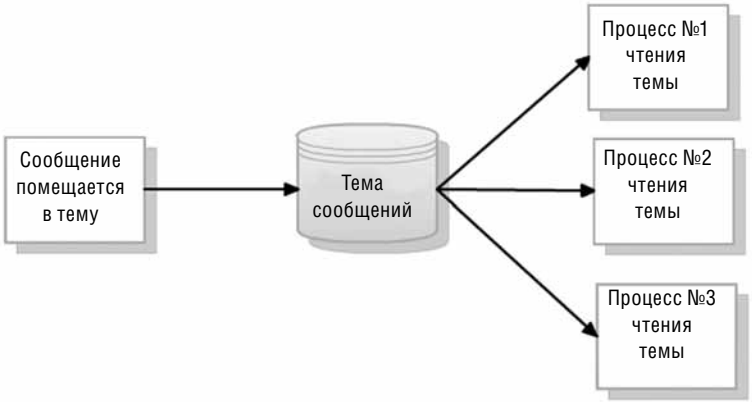


Рис. 12.3. *Сообщения из темы рассылаются всем клиентам*

Одним из примеров, где может использоваться архитектура, основанная на темах сообщений, является приложение мониторинга сети. Например, когда обнаруживается выход из строя некоторого устройства системы, приложение мониторинга может послать широковещательное сообщение с определенной темой, чтобы известить остальные компоненты системы о необходимости перейти на использование альтернативного устройства. В такой ситуации все подписчики темы обработают поступившее сообщение. В нашем примере с документом PDF нам необходимо, чтобы каждый запрос на создание файла PDF обрабатывался только один раз, поэтому нам следует использовать очередь.

12.1.3. Установка и настройка расширения JMS

Покончив с теорией, рассмотрим, как реализовать простую систему обмена сообщениями для приложения Hubbub. Сначала установим расширение JMS:

```
grails install-plugin jms
```

Затем нам необходимо принять решение о выборе механизма обмена сообщениями. В настоящее время доминирующие позиции занимают несколько механизмов, среди которых имеются как открытые реализации (ActiveMQ, Open MQ, JBoss Messaging), так и коммерческие (IBM WebSphere MQ). Одно из требований спецификации Java EE заключается в том, что вместе с сервером приложений должен распространяться контейнер JMS, поэтому, возможно, решение уже было принято за вас (например, вместе с Glassfish распространяется Open MQ – замечательный сервер JMS).

Но если вы планируете развертывать приложение на основе контейнера сервлета (такого как Tomcat или Jetty), тогда вы сами сможете выбрать реализацию механизма.

Между различными реализациями существуют лишь незначительные отличия в настройке, поэтому в данной главе мы будем использовать ActiveMQ, популярный открытый сервер обмена сообщениями.

Примечание

Если вы получите сообщение об ошибке компиляции «Compilation error: java.lang.NoClassDefFoundError: javax/jms/MessageListener» при установке расширения JMS или при первом запуске приложения после установки, значит, вам необходимо скопировать JAR-файл J2EE в каталог lib своего приложения на платформе Grails. В этом JAR-файле находится определение интерфейсов и классов поддержки JMS. Если вы используете Open MQ, можно воспользоваться файлом jms.jar. Если вы используете ActiveMQ, скопируйте файл activemq-all-5.1.0.jar.

Установка ActiveMQ

ActiveMQ – это открытая, свободно распространяемая и популярная реализация системы обмена сообщениями; она используется, например сайтом LinkedIn (*http://hurvitz.org/blog/2008/06/linkedin-architecture*). В данной главе мы будем использовать эту реализацию.

Чтобы установить ее, загрузите копию ActiveMQ по адресу *http://activemq.apache.org/* и распакуйте полученный файл в желаемое место для установки. Никакой дополнительной настройки не требуется, поэтому вы можете сразу же запустить сервер сообщений командой `/activemq/bin/activemq`. По завершении процесса запуска вы сможете получить доступ к консоли ActiveMQ путем ввода в броузере адреса

http://localhost:8161/admin/. На рис. 12.4 изображен интерфейс консоли в действии.

Консоль ActiveMQ позволяет просматривать существующие очереди и темы сообщений и проверить, как они передаются, – исследованием консоли мы займемся чуть позднее. А сейчас, когда у нас имеется действующий сервер сообщений, можно настроить приложение Hubbub на его использование.

Рис. 12.4. *Доступ к консоли ActiveMQ можно получить с помощью броузера*

Настройка механизма обмена сообщениями

После установки расширения JMS и запуска сервера сообщений необходимо выполнить следующие действия, чтобы настроить приложение на работу с ним:

- Настроить службу передачи сообщений в файле /grails-app/conf/spring/resources.groovy.
- Скопировать JAR-файл(ы) с реализацией механизма JMS в каталог /lib проекта.

Займемся сначала настройкой службы передачи сообщений. Каждый механизм JMS предоставляет свой фабричный класс подключения, который отвечает за создание соединения с вашим механизмом JMS. В случае использования ActiveMQ фабричному классу требуется передать имя хоста и номер порта сервера сообщений, поэтому дополним файл resources.groovy, чтобы сообщить расширению всю нужную ему информацию. Необходимые изменения приводятся в листинге 12.1.

Листинг 12.1. Изменения в файле resources.groovy, обеспечивающие возможность соединения с ActiveMQ

```
import org.apache.activemq.ActiveMQConnectionFactory ← Импорт фабричного класса-брокера

beans = {

    connectionFactory(ActiveMQConnectionFactory)  { ← Определение брокера соединения
        brokerURL = "tcp://localhost:61616" ← Настройка брокера соединения
    }

}
```

Далее необходимо скопировать JAR-файл ActiveMQ в каталог lib приложения. Для текущей версии ActiveMQ требуется скопировать единственный файл activemq-all-5.1.0.jar. Он содержит определение класса `ActiveMQConnectionFactory`.

Теперь, когда все настроено и готово к работе, можно задействовать мощь расширения JMS для платформы Grails и передать с его помощью несколько сообщений.

12.2. Использование расширения JMS для платформы Grails

Расширение JMS для платформы Grails обеспечивает простой способ передачи и получения сообщений из тем и очередей JMS. Подобно большинству расширений для платформы Grails оно использует принцип преимущества соглашений перед настройками, гарантируя тем самым, что вы будете тратить свое время на отправку сообщений, а не на настройку очередей (хотя вы можете переопределить любые соглашения, если захотите указать свои собственные имена для очередей).

В этом разделе мы рассмотрим основы отправки сообщений в очередь и извлечения их из очереди. Мы также снабдим приложение Hubbub шлюзом к системе обмена мгновенными сообщениями.

12.2.1. Наша потрясающая возможность в приложении Hubbub: интеграция с Jabber

Рассмотрим, какие возможности передачи сообщений нам требуется реализовать в приложении Hubbub. Было бы очень здорово интегрировать наше приложение с системой обмена мгновенными сообщениями, поэтому для приложения Hubbub мы напишем простой шлюз обмена мгновенными сообщениями, который свяжет его с одной из популярных систем обмена мгновенными сообщениями. В качестве примера мы напишем шлюз обмена сообщениями для доступа к Jabber, популярной открытой системе обмена мгновенными сообщениями, которая может обеспечить доступ к другим клиентам (AIM, Yahoo! IM и так далее).

Если пользователи приложения Hubbub зарегистрируют свои учетные записи в системе обмена мгновенными сообщениями, мы сможем позволить им отправлять свои сообщения с помощью их любимого клиента, а в случае, когда они подключены к сети, мы сможем даже передавать им сообщения других пользователей, чтобы они могли оставаться в курсе событий. При добавлении пользователем нового сообщения мы будем помещать его в очередь сообщений (чтобы затем передать его всем, кто следит за дискуссией и находится в данный момент в сети). Аналогично, когда пользователь будет отправлять сообщение боту приложения Hubbub с помощью клиента обмена мгновенными сообщениями, мы будем помещать его в очередь входящих сообщений для последующей передачи на веб-сайт Hubbub. Для чтения и отправки мы будем использовать транспорт обмена мгновенными сообщениями.

На рис. 12.5 схематически изображен порядок следования сообщений через очередь между приложением Hubbub и шлюзом.

Но прежде чем приступить к реализации шлюза, нам необходимо узнать, как помещаются исходящие сообщения в очередь JMS.

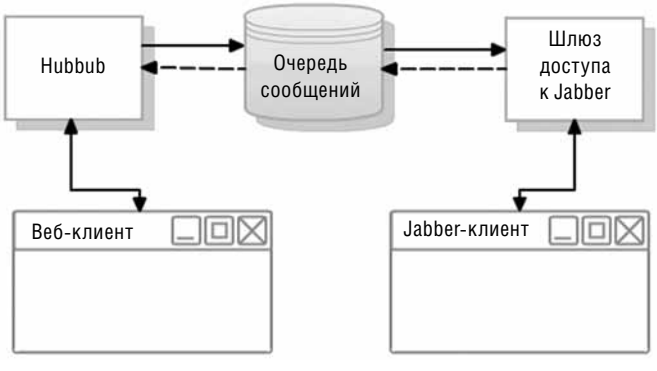


***Рис. 12.5.** Архитектура шлюза к системе Jabber*

12.2.2. Отправка сообщений JMS

Действие расширения JMS выражается в добавлении новых методов ко всем службам и контроллерам. Какой из методов следует вызывать, зависит от того, куда отправляется сообщение – в очередь или в тему. В табл. 12.1 перечислены методы для каждого типа места назначения.

Для очередей и тем существует по два метода, которые являются аналогами друг друга, поэтому вы можете использовать тот метод, который вам больше понравится. В наших примерах мы будем использовать метод `sendQueueJMSMessage()` для работы с очередями и метод `sendTopicJMSMessage()` для работы с темами, потому что их имена более явно говорят о том, что они делают (что будет хорошим примером для других разработчиков).

Таблица 12.1. Имена методов для каждого типа места назначения

Место назначения	Метод
Очередь	sendJMSMessage()
	sendQueueJMSMessage()
Тема	sendPubSubJMSMessage()
	sendTopicJMSMessage()

Независимо от того, работаете ли вы с темами или очередями, все методы принимают одни и те же параметры. Первый параметр – это имя объекта назначения (то есть имя очереди или темы на сервере сообщений), а второй параметр – само сообщение. ActiveMQ не требует, чтобы вы предварительно создавали очереди, но требования других механизмов могут отличаться.

В листинге 12.2 приводится реализация класса `JabberService`, который мы будем использовать для передачи сообщений. Мы будем помещать в очередь отображение, хранящее все данные сообщения.

Листинг 12.2. Реализация службы Jabber

```
class JabberService {

    void sendMessage(post, jabberIds) {

        log.debug "Sending jabber message for ${post.user.userId}..."
        sendQueueJMSMessage("jabberOutQ",
            [ userId: post.user.userId
              content: post.content,
              to: jabberIds.join(",") ] )
    }

}
```

← Помещает отображение в очередь

Теперь в нашем распоряжении имеется вся инфраструктура, правда, у нас пока нет инструмента, с помощью которого мы могли бы извлекать сообщения из очереди. Напишем испытательный тест, который отправит несколько сообщений. В листинге 12.3 приводится простой тест для испытания службы.

Листинг 12.3. Испытание нашей службы Jabber с помощью интеграционного теста

```
class JabberServiceTests extends GroovyTestCase {

    def jabberService

    void testWriteToQueue() {

        def post = [user: [userId: 'chuck_norris'],
            content: 'is backstroking across the atlantic']
```

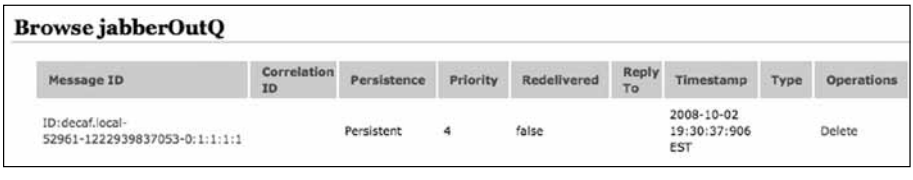
```
            def jabberIds = ["glen@grailsinaction.com",
                "peter@grailsinaction.com" ]
            jabberService.sendMessage(post, jabberIds)

        }

    }
```

Убедитесь, что сервер ActiveMQ запущен, и затем запустите тест командой `grails test-app JabberService`. Этот тест продемонстрирует возможность добавления элементов в очередь.

По окончании выполнения теста откройте в броузере страницу с адресом консоли ActiveMQ: *http://localhost:8161/admin*. На рис. 12.6 показано, как выглядит наша очередь `jabberOutQ`, после того как мы выбрали пункт меню Queues (Очереди).



Browse jabberOutQ

Message ID	Correlation ID	Persistence	Priority	Redelivered	Reply To	Timestamp	Type	Operations
ID:decaf.local-52961-1222939837053-0:1:1:1:1		Persistent	4	false		2008-10-02 19:30:37:906 EST		Delete

***Рис. 12.6.** Обзор очереди Jabber с помощью веб-интерфейса консоли ActiveMQ*

Вы можете щелкнуть на любом сообщении и убедиться, что все данные надежно сохранены. На рис. 12.7 показано, как выглядит содержимое отдельного сообщения.

Мы просмотрели все наши сообщения по порядку и убедились, что отображения с данными сохранены в очереди в ожидании, пока потребители извлекут их. Прежде чем мы перейдем к изучению способов извлечения сообщений из очереди, рассмотрим, какие типы данных могут помещаться в очередь.

Вы только что убедились, что сообщения были сохранены в очереди JMS, и вас может заинтересовать вопрос, какие типы данных могут сохраняться в очереди. Вы можете передавать системе JMS несколько простых типов данных, которые перечислены в табл. 12.2.

Таблица 12.2. Простые типы данных, которые могут включаться в сообщения JMS

Тип	Пример
`String`	`"Message from ${user.name}"`
`Map`	`[ name: "glen", age: "36", job: "stunt programmer" ]`
`byte[]`	`image.getBytes()`
`Object`	Любые объекты, реализующие интерфейс `Serializable`

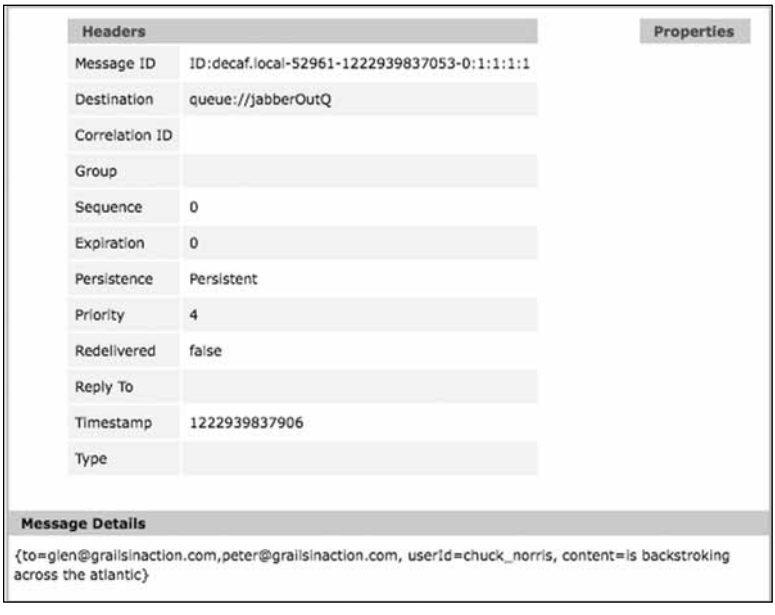


Рис. 12.7. *Исследование сообщения в очереди*

Хотя некоторые предпочитают в виде сообщений передавать строки, содержащие текст в формате XML, мы считаем, что сообщения с данными типа `Map` обладают большей гибкостью. Использование типа данных `Map` позволяет легко добавлять новые свойства к объектам, включаемым в сообщения, и при этом не приходится беспокоиться о появлении проблем в программном коде, выполняющем разбор данных где-нибудь в системе.

Как выполняется преобразование типов?

Если прежде вам приходилось сталкиваться с механизмами JMS, вы уже знаете, что JMS поддерживает свою собственную систему типов данных (`TextMessage`, `MapMessage`, `BytesMessage` и так далее). Расширение JMS автоматически выполняет приведение входных данных к соответствующему типу, освобождая вас для работы над своим приложением.

За кулисами расширение JMS использует стандартный класс Spring JMS `Template`. По умолчанию этот класс выполняет все преобразования с помощью класса `SimpleMessageConverter` платформы Spring, который обслуживает приведение простых типов данных, перечисленных в табл. 12.2.

Мы закончили тяжелую работу по передаче данных в нашу очередь, теперь нам предстоит заняться реализацией чтения сообщений из очереди, чтобы полностью завершить работу с очередями.

12.2.3. Чтение из очереди

Используя преимущество модели, основанной на соглашениях, расширение JMS существенно упрощает процесс чтения из очереди. Для простой реализации нам потребуется следующее:

- Добавить свойство в класс службы, чтобы превратить его в потребителя очереди или темы.
- Реализовать метод `onMessage()` для обработки входящих сообщений.
- Переопределить значения параметров по соглашению (где это необходимо), чтобы привести в соответствие имена очередей и количество потоков выполнения потребителя.

Рассмотрим каждый из этих шагов в отдельности, чтобы вы могли реализовать их и запустить.

Реализация первого метода onMessage()

В первую очередь следует заметить, что только службы могут играть роль конечных точек JMS. Чтобы уведомить расширение, что служба является конечной точкой JMS, необходимо включить в определение класса следующую строку:

```
static expose = [‘jms’]
```

Свойство `expose` используется многими расширениями удаленных взаимодействий (XFire, Remoting, Jabber), и вы можете использовать его для обозначения службы как конечной точки SOAP и JMS в одном и том же классе службы.

Далее необходимо добавить реализацию метода `onMessage()`, который будет вызываться расширением при поступлении нового сообщения. В результате мы получаем законченный класс обработки сообщений. В листинге 12.4 приводится реализация новой особенности.

Листинг 12.4. Обработка входящих сообщений в службе

```
class JabberService {

    static expose = [‘jms’]

    static destination = “jabberInQ” ← ❶ Имя прослушиваемой очереди

    static listenerCount = 5 ← ❷ Количество принимающих потоков

    void onMessage(msg) {
        log.debug “Got Incoming Jabber Response from: ${msg.jabberId}”
        try {
```

```
            def profile = Profile.findByJabberAddress(msg.jabberId)
            if (profile) {
                profile.user.addToPosts(new Post(content: msg.content))
            }
        } catch (t) { ← ❸ Перехватывает возникающие ошибки
            log.error "Error adding post for ${msg.jabberId}", t
        }
    }

    void sendMessage(post, jabberIds) {
        log.debug "Sending jabber message for ${post.user.userId}..."
        def msg = [userId: post.user.userId,
            content: post.content, to: jabberIds.join(",")]
        sendQueueJMSMessage("jabberOutQ", msg)
    }

}
```

Обратите внимание, как мы определили имя очереди в свойстве `destination` ❶. Следуя соглашениям, расширение JMS конструирует имя очереди из имени класса службы, поэтому для службы `JabberService` по умолчанию очередь получит имя «Jabber». В данном примере нам требуются отличные друг от друга очереди входящих и исходящих сообщений, поэтому мы переопределили имя очереди с помощью свойства `destination`, сообщив тем самым расширению, какое имя должна иметь очередь.

Мы изменили также количество потоков выполнения, которые будут принимать сообщения из очереди ❷. По умолчанию используется один поток, но мы увеличили их число до 5, потому что мы ожидаем большое число сообщений в очереди.

Наконец, мы предусмотрели обработку исключительных ситуаций ❸. В некоторых серверах сообщений происходит сбой при чтении из открытых соединений, если клиенты не соблюдают правила обмена данными, поэтому мы предусматриваем корректный выход при получении случайных или неправильно сформированных сообщений.

Выход на остановке: приложение, реализующее шлюз к системе Jabber

Теперь, когда в нашем приложении Hubbub имеется действующий интерфейс обмена сообщениями, можно написать приложение для связи с протоколом Jabber. Чтобы упростить реализацию, мы создадим приложение шлюза как обычное приложение на платформе Grails и воспользуемся расширениями JMS и Jabber для обеспечения связи с Интернетом.

Установить расширение Jabber можно с помощью обычного механизма установки расширений платформы Grails:

```
grails install-plugin jabber
```

Своим принципом действия расширение Jabber очень напоминает расширение JMS, с которым вы уже познакомились. Расширение Jabber определяет классы служб, обладающие свойством `expose = ['jabber']`, и автоматически добавляет к ним метод `sendJabberMessage()`. Если служба определяет замыкание `onJabberMessage()`, расширение будет вызывать его всякий раз, когда в настроенную очередь будет попадать любое сообщение из системы Jabber.

После установки расширений JMS и Jabber вся остальная реализация приложения заключена в одном классе службы, как показано в листинге 12.5.

Листинг 12.5. Служба шлюза, выполняющая передачу сообщений между приложением Hubbub и системой Jabber

```
class GatewayService {

    static expose = ['jabber', 'jms'] <-- ❶ Активирует поддержку JMS и Jabber

    static destination = "jabberOutQ" <-- ❷ Имя очереди JMS

    void onMessage(msg) { <-- ❸ Принимает входящие сообщения JMS

        log.debug "Incoming Queue Request from:
          ↪ ${msg.userId} to: ${msg.to} content: ${msg.content}"
        def addrs = msg.content.split(",")
        addrs.each {addr ->

log.debug "Sending to: ${addr}"
            sendJabberMessage(addr, msg.content) <-- ❹ Отправляет сообщения
        }                                              в очередь Jabber
    }

    void onJabberMessage(jabber) { <-- ❺ Принимает входящие сообщения Jabber
        log.debug "Incoming Jabber Message Received
          ↪ from ${jabber.from()} with body ${jabber.body}"
        def msg = [jabberId: jabber.from, content: jabber.body]
        sendQueueJMSMessage ("jabberInQ", msg) <-- ❻ Отправляет сообщения
                                                   в очередь JMS
    }
}
```

Примечание

Полные исходные тексты приложения включены в примеры исходного программного кода для этой главы и находятся в каталоге jabber-gateway.

Реализация службы `GatewayService` начинается с настройки на прием сообщений обоих типов, JMS и Jabber ❶, и с определения имени очере-

ди JMS (`destination`), которая будет прослушиваться службой ❷. Далее следует реализация метода `onMessage()` ❸, выполняющего прием входящих сообщений JMS и их передачу по адресу Jabber извлеченному из сообщения ❹.

Служба реализует также метод `onJabberMessage()` ❺, который принимает сообщения Jabber и помещает их в очередь сообщений приложения Hubbub, которое затем обрабатывает их и добавляет в списки сообщений пользователей ❻.

Написав всего 20 строк кода, мы реализовали двунаправленный шлюз между Jabber и JMS! Как видите, использование расширений может существенно уменьшить объем программного кода, который вам придется сопровождать.

> **Когда использовать темы, а когда очереди?**
>
> Теперь мы прекрасно представляем, как с помощью очереди выполняются передача и прием сообщений JMS. Для данного случая лучше всего подходит именно очередь, потому что нам требуется, чтобы каждое сообщение обрабатывалось всего один раз.
>
> Темы же идеально подходят для передачи широковещательных сообщений – когда о произошедшем событии необходимо известить всех потребителей. Представьте себе систему, выполняющую мониторинг сети и отслеживающую случаи запуска и остановки различных служб. Узел сети, выполняющий мониторинг других серверов, мог бы сообщать об остановке того или иного сервера всем остальным компонентам системы. Темы идеально подходят для случаев, когда требуется широковещательная рассылка сообщений.

На этом мы заканчиваем изучение основ обмена сообщениями. Теперь займемся исследованием другой, более легковесной альтернативы для асинхронных взаимодействий: механизма планирования заданий платформы Grails.

12.3. Механизм планирования заданий платформы Grails

Мы рассмотрели архитектуры обмена сообщениями и познакомились с некоторыми простыми способами использования асинхронных взаимодействий для упрощения систем, увеличения их гибкости и масштабируемости. Однако получение такой надежной и производительной архитектуры обеспечивается весьма обширной инфраструктурой.

Иногда бывает необходимо иметь простое асинхронное решение, позволяющее запускать некоторые функции в запланированное время (например, создание ежедневных отчетов, обновление индексов или создание ежедневных резервных копий). Для удовлетворения таких потребностей платформа Grails предлагает фантастический, простой в использовании механизм планирования, детальным исследованием которого мы сейчас и займемся.

12.3.1. Разработка задания, создающего ежедневные обзоры

Поддержка возможности планирования заданий в платформе Grails реализована в виде расширения Quartz. Quartz – это популярная библиотека Java с широкими возможностями планирования, а расширение Quartz обеспечивает простой доступ ко всем этим возможностям. Попробуем задействовать его, чтобы ежедневно отправлять по электронной почте всем пользователям приложения Hubbub обзоры, содержащие краткое изложение сообщений, оставленных их сторонниками за прошедший день.

Начнем с установки расширения:

```
grails install-plugin quartz
```

После установки расширения становятся доступными две новые команды:

```
grails create-job
grails install-quartz-config
```

Первая используется для создания шаблонов новых заданий (по своему действию близко напоминает команду `grails create-service`), а вторая устанавливает нестандартный файл с настройками расширения Quartz (который необходим только для реализации сложных случаев, таких как кластеризация, о котором мы поговорим немного ниже).

Чтобы обеспечить отправку ежедневных обзоров по электронной почте, нам необходимо создать новое задание, которое будет запускаться каждую ночь:

```
grails create-job DailyDigest
```

Этот вновь созданный класс задания будет помещен в файл grails-app/jobs/DailyDigestJob.groovy.

Самый простой способ использовать задание заключается в том, чтобы определить значение `timeout` в миллисекундах. Каждый раз по истечении указанного интервала расширение будет запускать задание. В листинге 12.6 приводится каркас задания, выполняющего отправку ежедневных обзоров по электронной почте.

Листинг 12.6. Простое задание отправки ежедневных обзоров (с использованием значения timeout)

```
class DailyDigestJob {

    def timeout = 24 * 60 * 60 * 1000 ← Запускать задание один раз в сутки
    def startDelay = 60 * 1000 ← Задержать первый запуск задания на 1 минуту

    def execute() {
        log.debug "Starting the Daily Digest job."
        // ... создание ежедневного обзора
        log.debug "Finished the Daily Digest job."
    }
}
```

Обратите внимание, что мы добавили поле `startDelay`, которое определяет начальную задержку перед тем, как расширение запустит задание. Это может быть удобно при небольших значениях в поле `timeout` (порядка нескольких секунд), когда необходимо убедиться, что до того как задание будет запущено первый раз, приложение на платформе Grails полностью пройдет этап начальной загрузки.

На данном этапе у вас вполне может возникнуть желание реализовать всю логику работы в самом классе задания. Такая возможность поддерживается платформой, но это практически всегда ошибочное решение. Задания поддерживают те же возможности внедрения, что и другие компоненты платформы, поэтому гораздо лучше реализовать вызов внедренной службы, чем выполнять реализацию задания непосредственно в классе. Прием, основанный на внедрении, позволяет создавать архитектуру, более удобную для тестирования, а также способствует многократному использованию программного кода. В листинге 12.7 представлена переработанная версия нашего нового задания.

Листинг 12.7. Простое задание отправки ежедневных обзоров

```
class DailyDigestJob {

    def timeout = 24 * 60 * 60 * 1000
    def startDelay = 60 * 1000

    def dailyDigestService

    def execute() {
        log.debug "Starting the Daily Digest job."
        dailyDigestService.sendDailyDigests()      ← Логика работы заключена
        log.debug "Finished the Daily Digest job."    в классе службы
    }
}
```

Благодаря наличию поля `dailyDigestService` расширение Quartz сначала убедится, что все связанные компоненты находятся на месте, и только потом запустит задание.

Теперь, когда задание по подготовке ежедневных обзоров готово к работе, можно заново продумать наш механизм планирования. До сих пор мы использовали простейший метод планирования, предоставляемый расширением Quartz. Он отлично подходит для заданий, которые должны выполняться каждые несколько секунд. Но нам предпочтительнее было бы иметь возможность производить рассылку обзоров в одно и то же время суток: например в 1 час ночи, когда серверы не так нагружены. Чтобы получить возможность планирования на календарное время, нам необходимо познакомиться с планировщиком cron.

12.3.2. Точное планирование с помощью планировщика cron

Если у вас имеется некоторый опыт работы с операционной системой UNIX, вы, вероятно, знаете о службе cron. Служба cron – это функциональная особенность UNIX, позволяющая планировать выполнение заданий в определенные дни в определенное время или в определенные дни недели или месяца со всей возможной гибкостью. Однако вместе с гибкостью вы получаете весьма замысловатый синтаксис, который может понравиться разве что фанатам командной строки. На рис. 12.8 изображены основные элементы выражения cron.

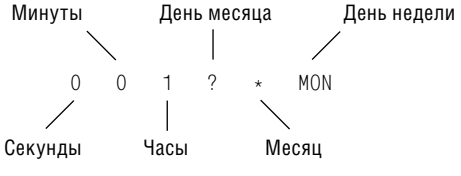


Рис. 12.8. *Основные элементы выражения cron*

Как видно на рис. 12.8, каждому полю в выражении cron соответствует свой интервал времени. Выражение в данном примере предписывает планировщику cron запускать задание в 1 час ночи каждый понедельник (MON).

Выражения планировщика cron обеспечивают непревзойденную гибкость планирования заданий, вот только синтаксис может вызывать затруднения. Во всех полях выражения cron допускается указывать числа, шаблонные символы (*), диапазоны (5-15), множества (5,10,15) или приращения (10/15). Имеются особые случаи для полей месяца и дня недели, где можно использовать специальные литералы. В поле месяца можно использовать такие выражения, как JAN-MAR, а в поле дня недели – такие выражения, как MON-FRI.

Понять синтаксис выражений cron будет намного легче, увидев его в действии. В табл. 12.3 перечислены некоторые наиболее часто используемые выражения.

Таблица 12.3. Несколько часто используемых выражений планировщика cron

Выражение	Описание
`0 0,45 1 ? * MON-FRI`	Каждый рабочий день недели в 1 час ночи и в 1:45 ночи
`0 0/15 1 ? * MON`	Каждые 15 минут, с 1 часа ночи до 1:45 ночи по понедельникам
`0 0 10-12 1 * ?`	В 10 часов утра, в 11 часов утра и в 12 часов дня первого числа каждого месяца
`0 0 0 1 1 ?`	В полночь в новогоднюю ночь

Совет

На веб-сайте проекта Quartz в разделе Documentation (Документация) имеется исчерпывающее справочное руководство по выражениям cron со множеством примеров по адресу: *http://www.opensymphony.com/quartz/wikidocs/CronTriggers%20Tutorial.html*.

Получив немного опасных знаний, попробуем переделать задание отправки ежедневных обзоров так, чтобы оно запускалось в 1 час ночи каждый рабочий день недели. Новая версия нашего задания приводится в листинге 12.8.

Листинг 12.8. Простое задание отправки ежедневных обзоров с нашими настройками планировщика cron

```
class DailyDigestJob {

    def cronExpression = "0 0 1 ? * MON-FRI" <-- Определяет выражение в стиле cron для задания Quartz

    def dailyDigestService

    def execute() {
        log.debug "Starting the Daily Digest job."
        dailyDigestService.sendDailyDigests()
        log.debug "Finished the Daily Digest job."
    }

}
```

На этом мы заканчиваем изучение простых способов планирования выполнения операций в приложениях на платформе Grails и переходим к исследованию некоторых более сложных приемов планирования.

12.4. Дополнительные возможности планирования

Мы рассмотрели приемы планирования заданий, способные удовлетворить большую часть потребностей применения планировщика в при-

ложениях на платформе Grails, и у вас уже, наверно, появились свои идеи по добавлению подобных заданий в свои будущие приложения. Но у нас впереди еще много неисследованного. В этом разделе мы будем создавать, запускать и управлять заданиями программно, а также создадим административный пользовательский интерфейс, с помощью которого мы сможем управлять всем этим непосредственно из приложения. Помимо этого мы рассмотрим возможность совместного использования данных заданиями и возможность распределения заданий в кластере. К концу раздела вы узнаете обо всех достоинствах (и недостатках) всех этих возможностей.

Сначала познакомимся с тем, как расширение планирования обрабатывает задания с сохранением информации о состоянии и задания, позволяющие запускать несколько копий.

12.4.1. Запуск нескольких копий задания и задания с сохранением состояния

По умолчанию расширение Quartz создает новый экземпляр класса задания и вызывает его всякий раз, когда требуется запустить задание. Но иногда возникают ситуации, когда нежелательно иметь два экземпляра задания, работающих одновременно.

Представьте, что в приложении Hubbub реализована возможность отправки извещений по СМС. Задание запускается каждые 10 секунд и проверяет наличие неотправленных СМС-сообщений. Если обнаруживаются такие сообщения, задание передает их веб-службе СМС-сообщений, которая их отправляет. Но что произойдет, если на обработку одного СМС-сообщения уходит 60 секунд? Ваше задание будет запускаться снова, и снова, и снова в течение одной и той же минуты, что в результате приведет к отправке нескольких (раздражающих) сообщений. Эту проблему можно было бы обойти с помощью поля `processed` в самом сообщении или с использованием для отправки очереди JMS; но предположим, что установленные правила исключают такие возможности, тогда вам необходим способ не допустить одновременного выполнения нескольких копий задания.

Расширение Quartz позволяет защититься от проблемы одновременно выполняющихся копий одного и того же задания с помощью свойства `concurrent`. Эта возможность демонстрируется в листинге 12.9.

Листинг 12.9. Использование свойства concurrent для предотвращения запуска нескольких копий задания

```
class SmsSenderJob {

    def timeout = 10000 // запускать задание каждые 10 секунд
    def concurrent = false

    def execute() {
```

```
        log.error "Sending SMS Job at ${new Date()}"
        Thread.sleep(20000) ← Имитация задержки, вызванной веб-службой
        log.error "Finished SMS Job at ${new Date()}"
    }
}
```

Если запустить это приложение, можно увидеть, что хотя запуск задания планируется производить каждые 10 секунд, имитация задержки позволяет запускать задание, только если оно уже не было запущено. Ниже приводится вывод этого примера реализации задания:

```
[22642] task.SmsSenderJob Sending SMS Job at Tue Oct 14 13:40:38 EST 2008
[42643] task.SmsSenderJob Finished SMS Job at Tue Oct 14 13:40:58 EST 2008
[42646] task.SmsSenderJob Sending SMS Job at Tue Oct 14 13:40:58 EST 2008
[62649] task.SmsSenderJob Finished SMS Job at Tue Oct 14 13:41:18 EST 2008
[62653] task.SmsSenderJob Sending SMS Job at Tue Oct 14 13:41:18 EST 2008
[82654] task.SmsSenderJob Finished SMS Job at Tue Oct 14 13:41:38 EST 2008
```

Важно понять, что если задание запланировано на запуск, но уже была запущена его копия, новое задание просто не запускается, а не приостанавливается на время, пока не завершится предыдущая его копия.

Другое следствие определения задания как не допускающего параллельного выполнения нескольких копий состоит в том, что расширение создаст задание как класс, реализующий интерфейс `StatefulJob`. Это означает, что заданию будет доступна область для сохранения информации между вызовами задания, которая называется `jobDataMap`. В нашем примере шлюза для сообщений СМС мы могли бы завести счетчик неудачных попыток отправки сообщений и выдавать предупреждающее сообщение в случае появления большого числа неудачных попыток. Как реализовать такую возможность, показано в листинге 12.10.

Листинг 12.10. Задание с сохранением состояния получает в свое распоряжение постоянный контекст

```
class SmsSenderWithTimeoutJob {
    def timeout = 10000 // запускать задание каждые 10 секунд
    def concurrent = false

    def execute(context) { ← ❶
        log.debug "Sending SMS Job at ${new Date()}"

        def failCounter = context.jobDetail.jobDataMap['failCounter'] ?: 0
        log.debug "Failed Counter is ${failCounter}"
        try {
            // здесь следует добавить вызов службы отправки СМС-сообщений
            failCounter = 0
        } catch (te) {
            log.error "Failed invoking SMS Service"
            failCounter++
            if (failCounter == 5) {
                log.fatal "SMS has not left the building."
```

```
            }
        }
        context.jobDetail.jobDataMap['failCounter'] = failCounter ← ❷
        log.debug «Finished SMS Job at ${new Date()}»
    }
}
```

Обратите внимание, что в этом примере мы изменили метод `execute()` – теперь он принимает аргумент типа `jobContext` ❶.

Второе, на что следует обратить внимание, – теперь имеется возможность сохранять в контексте любые объекты, реализующие интерфейс `Serializable` ❷: числа, даты, строки, коллекции и так далее. Опробуйте этот пример, чтобы убедиться, что значение счетчика сохраняется между вызовами задания.

12.4.2. Приостановка и возобновление работы задания с сохранением состояния программным способом

Итак, мы исследовали задания с сохранением состояния и рассмотрели, как мы можем избежать одновременного выполнения нескольких его копий. Но что если нам понадобится реализовать программное управление планированием?

Планировщик Quartz позволяет приостанавливать и возобновлять отдельные задания, группы заданий и даже сам планировщик. Чтобы упростить возможность управления заданием, его необходимо поместить в группу. В листинге 12.11 приводится наша первая попытка создать задание, допускающее возможность приостановки.

Листинг 12.11. Свойство group упрощает возможность управления заданием программным способом

```
class ControllableJob {

    def timeout = 5000 // запускать задание каждые 5 секунд
    def concurrent = false

    def group = "myServices" ← Определяет свойство group для управления
                               заданием программным способом
    def execute() {
        println "Controllable Job running..."
    }
}
```

Обратите внимание, что мы определили в задании атрибут `group`. Позднее мы будем использовать планировщик для получения доступа к этому заданию по имени группы, указанному в свойстве `group`.

Однако сейчас нам необходимо научиться управлять самим планировщиком. Неудивительно, что это может быть реализовано с помощью стандартного механизма внедрения платформы Grails. В листин-

ге 12.12 мы создали контроллер, в котором реализовано управление нашими заданиями.

Листинг 12.12. Контроллер, реализующий возможности приостановки и возобновления заданий

```
class JobAdminController {

    def quartzScheduler ← ❶ Обретается ссылка на объект планировщика Quartz

    def index = { redirect(action:'show') }

    def show = {

        def status = ""

                                                    Определяется, какая операция
        switch(params.operation) { ← ❷ выбрана пользователем
            case 'pause':
                quartzScheduler.pauseJob("ControllableJob", "myServices")
                status = "Paused Single Job"
                break
            case 'resume':
                quartzScheduler.resumeJob("ControllableJob", "myServices")
                status = "Resumed Single Job"
                break
            case 'pauseGroup':
                quartzScheduler.pauseJobGroup("myServices")
                status = "Paused Job Group"
                break
            case 'resumeGroup':
                quartzScheduler.resumeJobGroup("myServices")
                status = "Resumed Job Group"
                break
            case 'pauseAll':
                quartzScheduler.pauseAll()
                status = "Paused All Jobs"
                break
            case 'resumeAll':
                quartzScheduler.resumeAll()
                status = "Resumed All Jobs"
                break
        }

        return [ status: status ]

    }
}
```

В контроллере `JobAdminController` мы представили вашему вниманию несколько важных аспектов управления заданиями. Во-первых, мы определили свойство `quartzScheduler`, которое внедряет планировщик ❶. Ин-

струкция `switch` демонстрирует различные способы приостановки и возобновления заданий – по имени, по названию группы или всех заданий сразу ❷.

В листинге 12.13 приводится реализация простого пользовательского интерфейса, с помощью которого можно управлять планировщиком.

Листинг 12.13. Простой веб-интерфейс управления заданиями

```
<html>
    <head>
        <title>Job Admin</title>
        <style>
            div#status {
                margin: 1em;
                padding: 1em;
                border: 1px solid blue;
                background: lightblue;
            }
            body {
                font-family: “Trebuchet MS”,Helvetica;
            }
        </style>
    </head>

    <body>
        <h1>Job Admin</h1>

        <g:if test="${status}">
            <div id="status">
                ${status}
            </div>
        </g:if>

        <g:form action="show">
            <fieldset>
                <legend>Job Admin Operations</legend>
                <label for="operation">Select an operation:</label>
                <g:select id="operation" name="operation"
                    from="${ [
                    ‘pause’, ‘resume’,
                    ‘pauseGroup’, ‘resumeGroup’,
                    ‘pauseAll’, ‘resumeAll’
                    ] }" />
                <g:submitButton name="go" value="Go"/>
            </fieldset>
        </g:form>

    </body>
</html>
```

Откройте страницу *http://localhost:8080/quartz/jobAdmin/show*, для того чтобы увидеть административный интерфейс в действии, как показано на рис. 12.9.

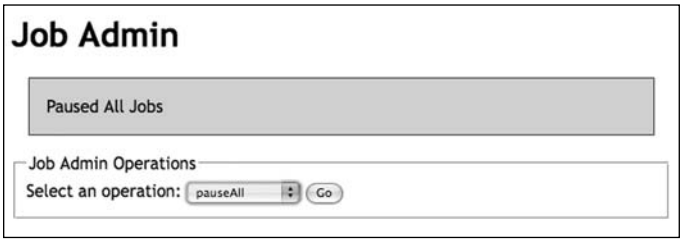


Рис. 12.9. *Приостановка задания с помощью нового административного интерфейса*

Наблюдение за тем, что выводит платформа Grails в командной строке, показывает, что задание `ControllableJob` (и другие задания) может запускаться и останавливаться с помощью административного интерфейса. Такой способ управления может пригодиться при реализации административного раздела в ваших приложениях, где вам может потребоваться возможность приостанавливать запланированные задания для ликвидации последствий чрезвычайных ситуаций (например, в случае остановки сервера поставщика услуг передачи СМС-сообщений).

Теперь мы знаем, как реализовать задания, сохраняющие информацию о своем состоянии между вызовами и не допускающими возможность одновременного выполнения нескольких копий. Но что произойдет с запланированными заданиями и их контекстами `jobDataMap` в случае перезапуска сервера приложения? И что будет происходить, когда мы попытаемся запустить наши задания в кластере? Чтобы ответить на эти вопросы, вам необходимо чуть больше узнать о том, как библиотека Quartz реализует сохранение информации.

12.4.3. Хранение данных и работа в кластере

Механизмом по умолчанию хранения данных в расширении Quartz является класс `RAMJobStore`, и, как вы уже, наверно, догадались, исходя из имени, этот класс обеспечивает высокую скорость работы, но не обеспечивает долговременное хранение данных. Если перезапустить сервер, все имеющиеся задания будут завершены, а любые данные, хранящиеся в `jobDataMaps`, будут утеряны. Если вам требуется обеспечить долговременное хранение заданий, обладающих возможностью сохранения состояния, замените класс `RAMJobStore` каким-нибудь другим классом, реализующим возможность долговременного хранения, таким как `JDBCJobStore`.

Для этого необходимо создать файл с настройками для расширения Quartz:

```
grails install-quartz-config
```

Предыдущая команда создаст новый файл /grails-app/conf/QuartzConfig.groovy, в котором вы можете активировать механизм хранения JDBC. Содержимое файла выглядит, как показано ниже:

```
quartz {
    autoStartup = true
    jdbcStore = false
}
```

Если параметр `jdbcStore` установить в значение `true`, состояние заданий будет сохраняться в базе данных. Но перед этим нужно создать необходимые таблицы.

Запрос SQL, создающий таблицы, можно найти в каталоге установки расширения Quartz /plugins/quartz-{version}/src/templates/sql/tables/. Здесь присутствуют сценарии SQL для всех распространенных баз данных, поэтому вы можете импортировать сценарии и создавать таблицы с помощью своего любимого инструмента администрирования.

Как только в базе данных будут созданы необходимые таблицы, можно изменить настройки в файле QuartzConfig.groovy, включив сохранение информации в базе данных:

```
quartz {
    autoStartup = true
    jdbcStore = true
}
```

Теперь обеспечивается долговременное сохранение состояния заданий; но нам требуется внести еще одно изменение, прежде чем мы перезапустим измененную реализацию в листинге 12.10 и проверим, действительно ли состояние задания сохраняется между перезапусками приложения.

По умолчанию все задания Quartz помечены как недолговременные, то есть их состояние не должно сохраняться в базе данных. Поэтому прямо сейчас сделаем одно из заданий долговременным, как показано в листинге 12.14.

Листинг 12.14. Определение долговременности задания

```
class SmsSenderWithTimeoutJob {
    def timeout = 10000 // запускать задание каждые 10 секунд
    def concurrent = false
    def volatility = false
    def execute(context) { ... }
}
```

Теперь все готово. Перезапустим приложение и посмотрим, действительно ли задание сохранило значение счетчика.

Специальное примечание для пользователей HSQLDB

Возможность долговременного сохранения данных при использовании базы данных HSQLDB была нарушена в версии Quartz 1.6. Возможно, к тому времени, когда вы будете читать эти строки, данная проблема будет исправлена. Если же этого не произойдет, вам потребуется создать файл /src/java/quartz.properties, чтобы определить свои строки запросов для расширения Quartz. Поместите следующую строку в файл, она исправит имеющуюся ошибку:

```
org.quartz.jobStore.selectWithLockSQL=
↪ SELECT * FROM {0}LOCKS WHERE LOCK_NAME = ?
```

Если вам потребуется проверить долговременность хранения данных в HSQLDB, измените свойство `url` своего источника данных на версию, обеспечивающую долговременное сохранение, например:

```
jdbc:hsqldb:file:devDB;shutdown=true
```

Этим исследованием возможности долговременного хранения информации о состоянии заданий мы завершаем наш тур по механизмам планирования заданий и обмена сообщениями в платформе Grails. Теперь коротко вспомним все, что мы узнали по пути.

12.5. В заключение

В этой главе мы покрыли значительную территорию асинхронных взаимодействий. Сначала мы рассказали вам о природе асинхронных взаимодействий и типах приложений, где они могут использоваться. Затем мы познакомили вас с основной терминологией, связанной с передачей сообщений, и перешли к подробному изучению расширения JMS.

После того как мы применили наши навыки владения расширением JMS для реализации в приложении Hubbub шлюза доступа к системе Jabber, мы провели исследование легковесных механизмов планирования заданий, доступных в платформе Grails. Мы рассмотрели использование выражений в стиле cron для реализации рассылки ежедневных обзоров по электронной почте и затем обсудили возможность программного управления механизмом планирования.

Ниже приводятся некоторые рекомендации:

- *Ознакомьтесь с конфигурацией своего сервера приложения*. Ваш сервер приложения, вероятно, уже обладает поддержкой механизма JMS, и вы можете опробовать его. Если ваше приложение выполняется в контейнере сервлета, лучшим выбором для вас будут Open MQ и ActiveMQ.

- *Правильно выбирайте между очередями и темами*. Очереди должны использоваться, когда необходимо, чтобы каждое сообщение обрабатывалось единственным потребителем, а темы – когда необходима широковещательная рассылка сообщений.
- *Отдавайте предпочтение отображениям*. При выборе типа данных для сообщений, помещаемых в очередь или в тему, отдавайте предпочтение отображениям – они обеспечат большую гибкость в ходе разработки вашего API.
- *Переопределяйте соглашения по мере необходимости*. Не бойтесь переопределять значения по умолчанию для параметров настройки (такие как имена очередей и тем), если это упростит дальнейшее сопровождение вашей системы. Совсем не лишней будет возможность догадаться по имени очереди, для каких целей она используется (например, `smsIncomingQueue` и `smsOutgoingQueue`).
- *Определяйте пропускную способность*. Количество потоков, прослушивающих очередь, должно соответствовать ожидаемой пропускной способности. Не стройте предположений на пустом месте – простое профилирование поможет вам определить возможности системы.
- *Используйте консоль*. Консоль администратора расширения ActiveMQ позволит вам видеть, что происходит в очередях – используйте возможности этого инструмента. При работе с другими серверами сообщений можно использовать отличный инструмент Hermes – открытую, позволяющую просматривать очереди альтернативу, которая способна взаимодействовать с любым контейнером JMS.
- *Отделяйте логику*. Не помещайте логику реализации в классы заданий расширения Quartz. Используйте механизм внедрения служб и помещайте логику работы в классы служб, это обеспечит простоту тестирования и большее удобство использования.
- *Отдавайте предпочтение cron*. Выражения в стиле cron кратки и выразительны, и они обеспечивают большую гибкость по сравнению с использованием свойства `timeout`.
- *Обеспечьте возможность программного управления заданиями*. Всегда определяйте в своих заданиях атрибут `group`, чтобы позднее можно было реализовать программное управление ими.
- *Обеспечьте возможность использования заданий в кластерах*. Если приложение выполняется в кластере, вы можете предусмотреть сохранение информации о состоянии в базе данных, чтобы иметь возможность совместного использования информации о состоянии и распределять работу между узлами кластера.

В следующей главе мы перейдем к изучению некоторых более сложных случаев использования механизма GORM. Мы исследуем возможности кэширования данных, оптимизации производительности и интеграции с устаревшими базами данных.

IV

Расширенные возможности Grails

В четвертой части книги мы представим вам некоторые расширенные возможности, предлагаемые платформой Grails. Вы узнаете о настройке производительности, интеграции с уже имеющимися базами данных, о транзакциях в базах данных, о настройке процесса сборки и даже о том, как создавать и передавать в общее пользование собственные расширения.

В главе 13 мы подробнее рассмотрим GORM – реализацию объектно-реляционной модели в платформе Grails. Вы многое узнаете о редко используемых возможностях моделирования, которые способны сэкономить ваше время. Вы также исследуете способы кэширования, профилирования производительности и некоторые примеры интеграции с уже имеющимися базами данных.

В главе 14 вы изучите основы интеграции платформы Grails с платформой Spring. Вы узнаете различные способы определения компонентов платформы Spring и взаимодействия с ними в своем приложении. Мы также коснемся транзакций – как они действуют и где их можно использовать.

Настройка инфраструктуры сборки – важная часть процесса разработки программного обеспечения, и в главе 15 мы углубимся в устройство системы сборки платформы Grails. Сначала вы узнаете, как добавлять собственные команды Grails. Затем мы покажем вам, как интегрировать систему сборки платформы Grails с системами сборки Ant и Maven, двумя наиболее часто используемыми инструментами сборки проектов на языке Java. Затем мы расскажем о стратегиях переноса данных по мере роста вашего приложения и познакомим со сторонними расширениями, которые помогут вам в этом.

Закончим мы подробным обзором принципов разработки расширений для Grails. Вы узнаете о жизненном цикле расширений и о том, как до-

бавлять новые особенности в платформу Grails. Вы также узнаете, как передавать свои новые расширения в репозиторий расширений Grails, чтобы ими могли пользоваться другие разработчики.

К концу этой части ваши навыки владения платформой Grails поднимутся на совершенно новый уровень. Вы будете готовы создавать высоко масштабируемые приложения социальных сетей в стиле Web 2.0. Вспомните о нас, когда добьетесь больших успехов.

В этой главе:

- Специализированные возможности классов предметной области
- Советы по организации кэширования, профилированию и оптимизации
- Работа с несколькими объектами dataSource в одном приложении
- Стратегии интеграции с имеющимися базами данных

13

Тонкое искусство владения механизмом GORM

В главе 3 мы впервые представили вам классы предметной области. Все было просто: мы сохраняли данные, проверяли их и использовали динамические методы поиска. Вы далеко можете уйти с этими знаниями; возможно даже пройти весь путь к созданию своих первых приложений на платформе Grails. Но затем вы упретесь в стену. По мере притока пользователей в ваши новые приложения социальных сетей вас начнет беспокоить вопрос увеличения эффективности ваших запросов. Вы начнете задумываться о возможности кэширования запросов. Вам потребуется реорганизовать свои классы предметной области и проверить все свои циклы. Короче говоря, вам потребуется снять привязные ремни и заняться серьезным искусством владения механизмом GORM. Когда вы столкнетесь с проблемой производительности и масштабируемости, эта глава поможет вам.

Если сейчас вы занимаетесь разработкой приложений уровня предприятия на языке Java, вероятно, у вас уже имеются некоторые базы данных (и, может быть, имеются существующие отображения Hibernate). Мы исследуем все особенности интеграции приложений на платформе Grails с существующими таблицами, источниками данных JNDI и другими компонентами, которые могут использоваться на вашем предприятии. Но вы не сможете разобраться с высокоуровневыми понятиями, если у вас не будет некоторого представления о том, как действуют основные механизмы, поэтому мы начнем наши исследования с самого низа и посмотрим, как платформа Grails взаимодействует с библиотекой Hibernate и какие настройки можно выполнить, чтобы добиться более высокой производительности.

13.1. Искусство моделирования предметной области

В главе 3 мы исследовали типичные виды отношений в модели предметной области: 1:1, 1:m и m:n. Но при разработке приложений на платформе Grails вам, вероятно, придется столкнуться с ситуациями, когда ни один из трех стандартных типов отношений не будет удовлетворять вашим потребностям. В этом разделе мы займемся исследованием некоторых редко используемых возможностей моделирования предметной области и начнем с наследования.

13.1.1. Исследование возможностей наследования

Реляционные базы данных не предназначены для хранения объектно-ориентированных данных, и отношения наследования не так просто отобразить на реляционную модель представления данных. Признавая это, давайте тем не менее посмотрим, какие возможности *может* предложить механизм GORM для работы с классами предметной области, использующими механизм наследования. Для обслуживания наследования механизм GORM предлагает две основные возможности: организацию отдельных таблиц для каждой иерархии и организацию таблиц для каждого подкласса.

По умолчанию GORM создает одну таблицу на иерархию, это означает, что базовый класс и его подклассы сохраняются в одной и той же таблице.

Прояснить ситуацию нам поможет пример. Модифицируем приложение Hubbub, добавив в него поддержку возможности отправлять специальные, отмеченные звездочками сообщения, которые всегда будут отображаться со звездочкой и с описанием причин, почему они сопровождаются звездочкой. Все сторонники пользователя будут обращать внимание на такие сообщения, помеченные специальным ярлыком.

Мы могли бы реализовать класс `StarPost`, добавив следующие определения в существующий класс `Post`:

```
boolean starred
String reasonStarred
```

Но в таком случае практически во всех сообщениях свойство `starred` будет иметь значение `false`, а свойство `reasonStarred` – значение `null`. Это означает, что нам придется ослабить ограничения.

Давайте реализуем нашу возможность в стиле объектно-ориентированного программирования и создадим соответствующий подкласс от класса предметной области `Post`, как показано в листинге 13.1.

Листинг 13.1. Класс StarPost, наследующий класс Post

```
package com.grailsinaction
```

```
class StarPost extends Post {  ← Наследует свойства класса Post

    String reason

    static constraints = {  ← Дополнительное ограничение,
        reason(maxSize: 50)     накладываемое на сообщение
    }
}
```

Теперь у нас имеется реализация класса `StarPost`, который наследует все свойства базового класса `Post`. Это означает, что теперь мы должны изменить реализацию класса `Tag`, чтобы явно обработать новое отношение:

```
static belongsTo = [ User, Post, StarPost ]
```

После внесения изменений в класс `Tag` и при наличии готового к работе класса `StarPost` можно создать интеграционный тест, который позволит убедиться, что отношения действуют именно так, как ожидалось. В листинге 13.2 приводится реализация интеграционного теста для проверки нового класса `StarPost`.

Листинг 13.2. Интеграционный тест для проверки класса StarPost

```
package com.grailsinaction

import grails.test.*

class StarPostIntegrationTests extends GrailsUnitTestCase {

    void testInheritQuery() {

        def u = new User(userId: 'glen', password: 'password').save()
        u.addToPosts(new Post(content: "First Post"))
        u.addToPosts(new Post(content: "Second Post"))
        u.addToPosts(new StarPost(content: "Third Post",
                                  reason: "I shine on 3"))
        assertEquals 3, Post.countByUser(u)  ← ❶ Подсчитывает число сообщений
        assertEquals 1, StarPost.countByUser(u)  ← ❷ Подсчитывает число
    }                                               сообщений со звездочкой
}
```

Когда у нас имеется несколько классов предметной области, связанных отношением наследования, как в данном случае, мы можем использовать преимущества полиморфных запросов. Мы можем извлекать все сообщения пользователя ❶ или только сообщения, отмеченные звездочками ❷, используя механизм полиморфных запросов GORM, который поддерживается динамическими методами поиска.

Мы говорили, что по умолчанию механизм GORM использует по одной таблице на иерархию. Один из недостатков такого подхода состоит в том, что вы не можете использовать ограничение `nullable` в подклассах (потому что экземпляры базового класса вставляются в ту же са-

мую таблицу, что и экземпляры его подклассов). Но это весьма незначительное ограничение в сравнении с широтой возможностей полиморфных запросов.

Как быть, если вам потребуется использовать отдельную таблицу для каждого подкласса? Это возможно, но при этом придется немного поколдовать с языком GORM DSL. Добавьте следующее отображение в корневой класс (в данном случае класс `Post`):

```
static mapping = {
    tablePerHierarchy false
}
```

При создании отдельной таблицы для каждого подкласса для загрузки экземпляров механизм GORM будет использовать внешнее соединение таблиц, что может приводить к существенному снижению производительности при работе с глубокими иерархиями. Но при этом у вас появляется возможность использовать ограничение `nullable` для проверки полей классов предметной области; вот такая сделка.

13.1.2. Встраивание классов предметной области

Иногда бывает желательно воспользоваться удобствами, которые предоставляются объектами предметной области, но при этом избежать создания отдельной таблицы. Такая потребность обычно может возникнуть, когда уже имеется некоторая реализация, в которой определено логическое отношение 1:1, но все свойства объектов реализованы в виде отдельных полей одной большой таблицы. В подобных ситуациях механизм GORM предлагает использовать возможность *встраивания*. Встраивание позволяет определить таблицу как два или более связанных объектов.

Рассмотрим в качестве примера наш объект `Profile`. Для каждого пользователя имеется один объект `Profile`, содержащий адрес электронной почты и другие личные данные. А теперь представьте, что мы решили устранить нагрузку управления отдельной таблицей `Profile` и просто встроить информацию об учетной записи непосредственно в таблицу `User`. В листинге 13.3 показано, как следует изменить классы, чтобы обеспечить такую возможность.

Листинг 13.3. Внедрение связанных классов в единую таблицу

```
class User {

    String userId
    String password
    Date signupDate = new Date()
    Profile profile
    static embedded = ['profile' ] ← Объект Profile будет сохраняться
    // ...                             в той же записи, что и объект User
}
```

```
class Profile {  ← Встраиваемый класс Profile объявляется в том же файле

    byte[] photo

    String fullName
    // ...
}
```

Как видите, встраивание выполняется с помощью конструкции `embedded`. Кроме того, чтобы механизм встраивания заработал, класс `Profile` должен быть объявлен в том же файле Groovy, что и класс `User` (User.groovy); в противном случае в базе данных будет создана пустая таблица `Profile`.

Встраивание позволяет сохранить объектно-ориентированную семантику без напрасного расходования табличного пространства в базе данных.

13.1.3. Использование отображений для создания таблиц на скорую руку

Использование классов предметной области для управления данными – вполне логичный выбор, однако в некоторых ситуациях бывает желательно избежать сложностей, связанных с созданием класса.

Представьте, что приложение Hubbub поддерживает набор некоторых расширений. Пользователи могут добавлять эти виджеты в свои домашние страницы и настраивать их свойства. Мы могли бы создать класс с именем `PluginProperty` со строковыми полями для хранения имени и значения свойства, но это немного неудобно. Все, что нам требуется, – это отобразить пары ключ/значение в хранилище данных.

В таких случаях механизм GORM позволяет объявлять подобные элементы как отображения. В листинге 13.4 приводится пример сохранения свойств в виде отображения.

Листинг 13.4. Использование отображений для организации хранения данных на скорую руку

```
class User {

    String userId
    String password
    Map pluginProperties  ← Сохраняет ключи и значения отображения как поля типа varchar(255)
    // ... остальные элементы класса
}
```

За кулисами платформа Grails сохраняет `pluginProperties` как отображение пар ключ/значение в виде полей типа `varchar(255)`, поэтому вы можете сохранять только строковые значения.

Ниже показано, как протестировать эту возможность:

```
user = new User(userId: 'glen', password: 'notlong',
            pluginProperties: ['colour':'green']).save()

user.pluginProperties = [ one: "1", two: "2" ]
```

Сохранение строк в свободной форме может быстро превратить обслуживание приложения в ночной кошмар, поэтому используйте эту возможность с большой осторожностью. Ее удобно тем не менее использовать для создания на скорую руку таких таблиц, как `pluginProperties`.

13.1.4. События в модели предметной области

Механизм GORM выполняет массу операций при извлечении информации из базы данных. Но иногда бывает необходимо интегрировать в GORM свои собственные особенности. Например, представьте реализацию возможности контроля, которая журналирует изменения в модели предметной области всякий раз, когда выполняется модификация объекта. Реализация GORM предлагает механизм событий как раз для таких целей, обеспечивая возможность производить собственные операции непосредственно перед тем и после того, как GORM сохранил или загрузил объект.

В механизме GORM доступно семь основных событий:

- `beforeInsert`
- `beforeUpdate`
- `beforeDelete`
- `afterInsert`
- `afterUpdate`
- `afterDelete`
- `onLoad`

Если вам потребуется перехватить любое из них, определите замыкание с соответствующим именем, как показано ниже:

```
def beforeDelete = {
    log.warn "${id} is toast by ${jsec.principal}"
}
```

Ни одно из замыканий не принимает никаких аргументов и не возвращает никаких значений, поэтому если внутри замыкания потребуется получить какую-либо информацию, она должна храниться в самом классе предметной области или извлекаться посредством текущего потока выполнения. (В предыдущем примере мы извлекаем пользователя JSecurity из текущего потока выполнения.)

Механизм GORM обеспечивает еще одну удобную возможность, которую можно использовать при работе с событиями, – получение текущего времени. При обработке событий часто возникает необходимость запоминать время создания и изменения класса предметной области.

Например, можно перехватить событие `beforeUpdate` и установить время изменения (или перехватить событие `beforeInsert` и установить время создания). Для таких ситуаций механизм GORM предлагает удобное соглашение: если двум полям типа `Date` объекта присвоить имена `dateCreated` и `lastUpdated`, GORM автоматически будет записывать в эти поля значение текущего времени:

```
class User {
    Date dateCreated
    Date lastUpdated
    String userId
    // ...
}
```

Это особый случай, но он является одним из самых типичных случаев использования событий (наряду с операцией журналирования), поэтому весьма удобно иметь такую встроенную особенность. Мы использовали ее при создании объекта `Post` в главе 3, и она действительно позволила сэкономить наше время.

Теперь, когда мы рассмотрели некоторые крайние случаи моделирования предметной области, можно заняться рассмотрением некоторых рекомендаций по увеличению производительности запросов.

13.2. Искусство кэширования: переходим от 2 пользователей к 2[10]

В предыдущих главах мы производили сложные запросы, нимало не беспокоясь о том, насколько тяжело дается их выполнение базе данных. Такой подход годится для сайтов, которые посещают несколько сотен человек в день, но если вы стремитесь к увеличению числа посетителей, вам необходимо исследовать возможность настройки кэширования и производительности.

В этом разделе мы рассмотрим, как механизм GORM выполняет кэширование, и покажем, как вы можете определить, какие части приложения могут извлечь выгоду из кэширования и какие настройки можно подкрутить. Прежде всего нужно осмыслить тот факт, что GORM использует механизм кэширования второго уровня библиотеки Hibernate.

13.2.1. Настройки Hibernate: должны ли вы использовать кэш второго уровня?

При взаимодействиях с объектной моделью Hibernate в игре все время участвует кэш первого уровня – сеанс. Когда в ходе запроса выполняется изменение нескольких свойств того или иного объекта (например, изменение времени последней регистрации пользователя и IP-адреса), библиотека Hibernate может объединить эти операции в едином SQL-запросе `UPDATE`.

Но иногда кэша первого уровня бывает недостаточно. Рассмотрим ситуацию с отношением 1:m (например, когда каждый пользователь может иметь множество сообщений). Каждый запрос, содержащий вызов `user.posts.each { }`, будет вызывать повторное обращение к базе данных. Если сообщения изменяются нечасто (или вообще не изменяются), вся эта масса запросов будет приводить к пустой трате времени.

Примечание

Повторный вызов `user.posts.each` в одном и том же запросе не будет вызывать повторное обращение к базе данных, потому что коллекция кэшируется в сеансе.

В подобных ситуациях можно задействовать *кэш второго уровня*. Когда поддержка кэша второго уровня включена для класса предметной области, библиотека Hibernate сначала выполнит поиск в кэше и только потом обратится к базе данных. Библиотека Hibernate реализует также удаление объектов из кэша при добавлении, редактировании и удалении сообщений.

Единственный случай, когда следует проявлять осторожность в использовании кэша второго уровня, – при работе в кластере серверов приложений. В такой ситуации вам необходимо гарантировать, что ни один из серверов не сможет получить доступ к устаревшим данным, настроив механизм кэширования на работу в кластере.

Как правило, ожидается, что кэш второго уровня обеспечит прирост производительности. Но как его правильно настроить и отрегулировать? Прежде чем перейти к регулировкам, настроим основные параметры кэширования.

13.2.2. Настройка кэша

Теперь, когда вы знаете, что делает кэш второго уровня, можно заняться изучением его настройки для использования в приложении на платформе Grails. В первую очередь вам следует познакомиться с файлом DataSource.groovy. Вы уже видели этот файл ранее, когда настраивали параметры соединения с базой данных, однако в этом файле имеется раздел `hibernate`, который мы еще не исследовали. Рассмотрим его:

```
hibernate {
    cache.use_second_level_cache=true
    cache.use_query_cache=true
    cache.provider_class='com.opensymphony.oscache.
    ➥ hibernate.OSCacheProvider'
}
```

В данном случае кэш второго уровня активирован, как и кэш запросов (с которым мы познакомимся ниже). Кроме того, указан класс, реализующий механизм кэширования, – этот параметр сообщает библиотеке

Hibernate, какую библиотеку кэширования использовать (их существует несколько). Платформа Grails по умолчанию использует библиотеку OSCache, но мы будем использовать другую библиотеку, Ehcache, потому что она обеспечивает более высокую производительность:

```
cache.provider_class='org.hibernate.cache.EhCacheProvider'
```

Примечание

Большинство пользователей библиотеки Hibernate знакомы с Ehcache как с наиболее часто используемым механизмом кэширования, тогда почему платформа Grails по умолчанию использует OSCache? Это обусловлено тем, что OSCache более производительна на этапе перезагрузки Grails в среде разработки. Если вы привыкли применять библиотеку Ehcache, вы свободно можете переключиться на ее использование, когда приложение будет полностью готово к эксплуатации.

Каждая реализация кэширования имеет собственный механизм настройки. Библиотека Ehcache использует единственный файл XML (ehcache.xml), который должен находиться в корне пути к классам (classpath). Скопируйте его либо в каталог /src/java, либо в каталог /grails-app/conf. В листинге 13.5 демонстрируется пример содержимого файла ehcache.xml, с которого мы и начнем.

Листинг 13.5. Пример файла с настройками ehcache.xml

```
<ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="ehcache.xsd">

    <defaultCache
        maxElementsInMemory="1000"
        eternal="false"
        timeToIdleSeconds="3600"
        timeToLiveSeconds="3600"  <-- Предельный срок хранения объектов в кэше
        overflowToDisk="false"        составляет 1 час (3600 секунд)
        diskPersistent="false"
        diskExpiryThreadIntervalSeconds="120"
        memoryStoreEvictionPolicy="LRU"
    />

</ehcache>
```

Полное описание библиотеки Ehcache далеко выходит за рамки этой книги, но мы отметим, что с данными настройками по умолчанию объекты будут храниться в кэше 1 час (3600 секунд), после чего они будут обновлены в результате выполнения следующего запроса к базе данных.

Библиотека Ehcache поддерживает множество параметров настройки, включая возможность создания распределенного кэша, возможность сохранения кэша на диске при перезапуске приложения и различные политики удаления неиспользуемых и устаревших элементов.

Библиотека имеет отличную документацию (*http://ehcache.sourceforge.net/*), поэтому обращайтесь к ней, чтобы узнать больше.

13.2.3. Кэширование отдельных классов предметной области

Чтобы включить кэширование для отдельных классов предметной области, требуется сделать чуть больше. В каждый класс предметной области, который требуется кэшировать, нужно добавить блок `mapping`, как показано ниже:

```
static mapping = {
    cache true
}
```

Или, если требуется более полный контроль над параметрами кэширования, их можно определить в виде отображения:

```
static mapping = {
    cache: "read-write"
}
```

С помощью параметров настройки кэширования можно определить, как библиотека Hibernate будет обслуживать одновременные обращения к кэшу (табл. 13.1). Помните, что Hibernate записывает данные в кэш всякий раз, когда вы создаете, изменяете или удаляете экземпляры классов предметной области. Если имеются данные, которые никогда не изменяются (например, список часовых поясов), вы безопасно можете установить параметр `cache` в значение `read-only`, что даст некоторый прирост производительности.

Таблица 13.1. Платформа Grails поддерживает все стандартные стратегии кэширования Hibernate; здесь перечислены наиболее часто используемые

Значение параметра cache	Описание
`true`	То же, что и значение `read-write`
`read-only`	Может использоваться только для кэширования статических данных, в противном случае неизбежно возникнут проблемы с параллельным доступом
`read-write`	Значение по умолчанию; реализует возможность параллельного обращения к кэшу

В приложении Hubbub мы настраиваем кэширование сообщений пользователей. В подобных случаях нам потребуется использовать стратегии кэширования, чтобы обеспечить возможность перехода в дереве объектов от объекта `User` к объекту `Post`. Для этого нам нужно немного изменить класс `User`:

```
static mapping = {
    cache: true
    posts cache:true
}
```

Совет

Запомните, что механизм кэширования необходимо включать в двух местах. Сначала необходимо включить кэширование глобально в файле DataSource.groovy (по умолчанию включено). А затем необходимо включить кэширование для отдельных классов предметной области.

Но это только полдела. Теперь, когда объекты `Post` сохраняются в кэше, нам может потребоваться отрегулировать параметры синхронизации данного кэша. Вернемся к файлу ehcache.xml и добавим новый элемент с именем `name`, значение которого соответствует имени класса предметной области:

```
<cache
    name="com.grailsinaction.Post" ← Настройки кэширования объектов Post
    maxElementsInMemory="10000"
    eternal="false"
    timeToIdleSeconds="300"
    timeToLiveSeconds="600"
    overflowToDisk="false"
/>

<cache
    name="com.grailsinaction.User" ← Настройки кэширования объектов User
    maxElementsInMemory="10000"
    eternal="false"
    timeToIdleSeconds="300"
    timeToLiveSeconds="600"
    overflowToDisk="false"
/>
```

Примечание

Если вы помещаете классы предметной области в пакеты, вы должны будете также указать имя пакета в поле `name`, как это сделано в предыдущем примере (`com.grailsinaction.Post`).

В настройках выше мы определили период простоя равным 300 секундам (то есть если к экземпляру объекта не будет обращений в течение 300 секунд, он будет удален из кэша) и максимальный период хранения в кэше равным 600 секундам (независимо от того были ли обращения к объекту, он в любом случае будет удален из кэша через 600 секунд). Это достаточно агрессивные настройки, но они отлично подходят для нужд профилирования.

Совет

Не существует универсальной стратегии кэширования, подходящей для всех случаев. Выбор стратегии зависит от того, как объекты используются системой. Если в системе операция записи выполняется редко, а операция чтения – часто (как в случае с сообщениями в приложении Hubbub), есть смысл распространить кэширование на определенный уровень. Когда в следующем разделе мы будем рассматривать проблему профилирования производительности, вы познакомитесь с некоторыми инструментами, которые можно использовать для оценки значений параметров кэширования, наилучшим образом удовлетворяющих вашим потребностям.

Теперь, когда мы отрегулировали все настройки кэша, займемся профилированием и посмотрим, как можно обнаруживать в приложении узкие места, влияющие на его производительность.

13.2.4. Достаточно разговоров, займемся профилированием

На первом этапе профилирование можно начать с проверки некоторых запросов SQL, которые создаются приложением. Вы можете внести следующие изменения в файл DataSource.groovy, чтобы заставить механизм GORM выводить запросы в консоль:

```
development {
    dataSource {
        logSql = "true"
    }
}
```

Это позволит вам получить некоторое представление о том, какие запросы генерируются, но вы не узнаете, как долго база данных выполняет эти запросы. Для этого необходимо выполнить профилирование запросов, и самый простой способ сделать это в платформе Grails – воспользоваться P6Spy.

P6Spy – это небольшой драйвер JDBC, который обертывает ваш действующий драйвер и предоставляет простой секундомер. Он позволяет узнать, сколько миллисекунд потребовалось базе данных, чтобы удовлетворить ваш запрос. Существует даже расширение для платформы Grails, упрощающее работу с драйвером P6Spy, поэтому установим его:

```
grails install-plugin p6spy
```

В процессе установки расширение изменит файл DataSource.groovy и добавит в него закомментированный раздел P6Spy с настройками драйвера базы данных. Изменим наш класс `dataSource` в разделе `development`, чтобы задействовать новую версию P6Spy:

```
development {
    dataSource {
```

```
            dbCreate = "update"
            driverClassName =
                "com.p6spy.engine.spy.P6SpyDriver" ← Изменить только класс драйвера
            url = "jdbc:hsqldb:mem:devDB"
        }
    }
```

Обратите внимание, что мы не изменяем адрес URL – драйвер P6Spy будет использовать имя пользователя, пароль и URL нашей базы данных. Но мы должны сообщить ему действительный драйвер базы данных, а для этого необходимо изменить файл с настройками P6Spy: spy.properties. Расширение создает шаблон файла в каталоге grails-app/conf. Рассмотрим теперь его содержимое. Обратите внимание на параметр `realdriver`, который сообщает драйверу P6Spy, с какой базой данных он будет работать:

```
realdriver=org.hsqldb.jdbcDriver
```

По умолчанию используется база данных HSQLDB, поэтому нам не придется ничего менять.

Теперь, когда профилирование настроено, можно запустить приложение Hubbub и посмотреть на собранные данные. Вы заметите, что теперь в корневом каталоге проекта появился файл spy.log. В нем вы найдете продолжительность выполнения каждого запроса, переданного базе данных. Ниже приводится одна строка, извлеченная из файла spy.log:

```
13:40:23|0|1|statement||create table account_account
↪ (user_followers_id bigint, user_id bigint, user_following_id bigint)
```

Это конечно хорошо, но нам хотелось бы видеть более удобочитаемые результаты. Представляем вам SQL Profiler: небольшое приложение с интерфейсом на базе библиотеки Swing, которое обрабатывает файлы, полученные от расширения P6Spy, и отображает запросы, занявшие самое продолжительное время. Вы можете загрузить это приложение по адресу: *http://sourceforge.net/projects/sqlprofiler*.

Приложение SQL Profiler прослушивает сокет, куда расширение P6Spy выводит журналируемые данные, так что вам необходимо изменить файл с настройками p6spy.properties, чтобы сообщить расширению, куда следует выводить данные. Настроенный файл p6spy.properties вы можете найти в исходных текстах приложения Hubbub.

Закончив настройку параметров журналирования, запустите приложение SQL Profiler до того, как вы запустите команду `grails run-app`, и понаблюдайте за потоком сообщений, появляющихся в процессе взаимодействия с приложением. На рис. 13.1 показан поток запросов SQL, обрабатываемых профилировщиком. Щелкните на кнопке Pause (Пауза) в верхней части окна, чтобы ознакомиться с результатами анализа.

В верхней панели приложения SQL Profiler отображаются запросы и время, затраченное на их выполнение. Использование базы данных

HSQLDB, размещаемой в оперативной памяти, может замаскировать проблемы производительности, но при использовании настоящей базы данных (такой как PostgreSQL или MySQL) вы получите значения, более близкие к эксплуатационным. Щелчок на элементе в верхней панели приведет к выводу в нижней панели полного текста запроса SQL, благодаря чему вы сможете увидеть, какие соединения таблиц используются.

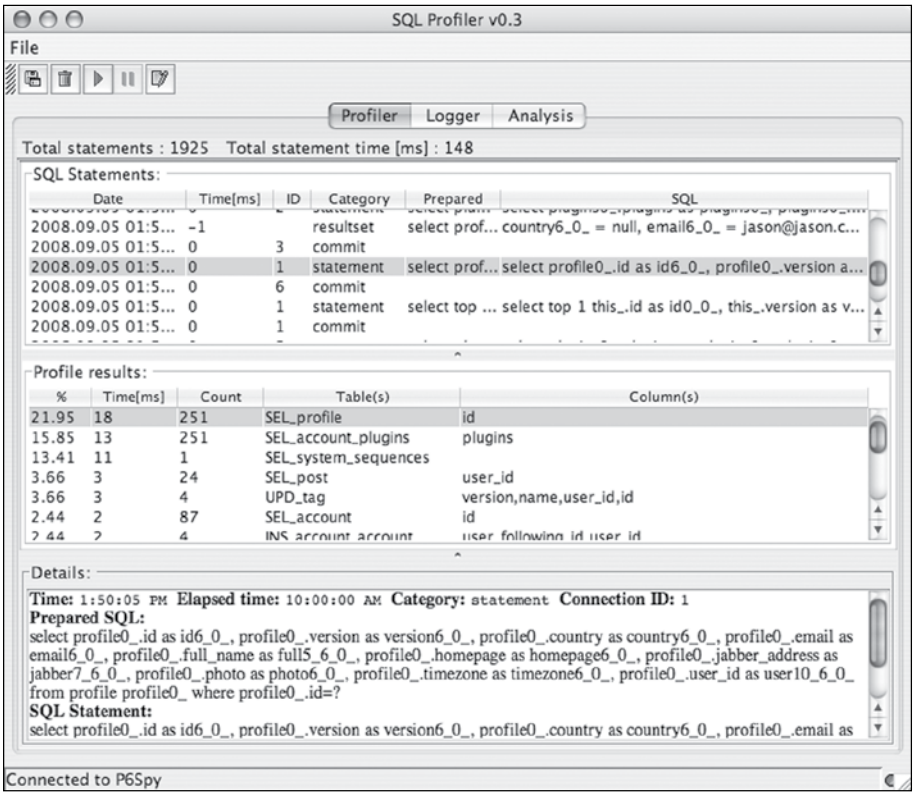


Рис. 13.1. *Профилировщик P6Spy в действии*

Для получения данных профилирования лишь нескольких запросов используйте кнопки Play (Запустить) и Pause (Пауза). Щелкните на кнопке Play (Запустить), выполните операции с определенной страницей приложения, щелкните на кнопке Pause (Пауза) и просмотрите результаты в средней панели. Средняя панель покажет вам, какие запросы выполнялись дольше всего. В крайней правой колонке будут перечислены поля таблицы, задействованные в запросе (эта информация может быть полезна при определении полей, которые желательно было бы включить в индекс).

Чтобы сымитировать производительность базы данных под реальной нагрузкой, можно воспользоваться инструментом генератора нагрузки, таким как Apache JMeter (*http://jakarta.apache.org/jmeter/*). Инструмент JMeter предлагает возможность сымитировать поток обращений к приложению сотен (или тысяч) пользователей и позволяет проверить поведение приложения под нагрузкой.

Расширение P6Spy поможет вам получить основную информацию о производительности базы данных и позволит определить, для каких запросов в приложении можно было бы получить наибольшую выгоду от их регулировки и использования кэширования. Как только вы определите узкие места, у вас может появиться желание добавить индексы. Рассмотрим, как это делается в платформе Grails.

13.2.5. Повышение производительности за счет индексирования полей

Как только ваши попытки профилирования дадут представление о том, какие запросы выполняются дольше всего, вам может потребоваться добавить индексы в базу данных, чтобы обеспечить более высокую скорость поиска. Платформа Grails позволяет реализовать собственную схему индексирования с помощью универсального замыкания `mapping` (вы уже видели это замыкание при настройке параметров кэширования).

Предположим, что процедура создания отчетов выполняет поиск в таблице `User` и создает некоторый отчет на базе поля `dateCreated` – дате создания учетной записи. Учитывая высокую популярность нашего вебсайта, имеет смысл создать индекс по полю `dateCreated`. Чтобы создать индекс, ему нужно присвоить имя (в нашем случае, чтобы осчастливить администратора базы данных, мы будем использовать имя `date_created_idx`) и затем присоединить его к полю класса предметной области, для которого предназначен этот индекс:

```
static mapping = {
    dateCreated index:'date_created_idx'
}
```

Создание индексов и выполнение профилирования с помощью расширения P6Spy может быть отличным способом идентификации запросов, кэширование которых даст наибольший эффект.

Теперь посмотрим, какие средства предлагаются платформой Grails для ускорения запросов с применением кэширования.

13.2.6. Как кэшировать запросы?

Мы много говорили о кэше второго уровня и о том, как увеличить производительность навигации по дереву объектов. Но как быть, когда нужно просто организовать кэширование запросов?

Как быть, если мне потребуется распределенное кэширование?

Кэш второго уровня – замечательный механизм, но по сложившейся традиции он не очень хорошо поддерживает возможность работы в кластере, что существенно снижает его полезность в системах с высокой степенью доступности. Технологии репликации, как правило, не очень производительны, тем не менее в этой области были достигнуты значительные успехи. В настоящее время платформа Grails прекрасно поддерживает с помощью расширений открытые реализации механизмов распределенного кэширования, не уступающие коммерческим аналогам, такие как Terracotta. Библиотека Ehcache теперь также обладает возможностью распределенного кэширования. Если ваше приложение выполняется в кластере, обязательно ознакомьтесь с этими механизмами.

Для этого имеются определенные возможности, но весьма маловероятно, что они будут полезны. Механизм GORM кэширует только результаты запросов, если ни одна из таблиц, участвующих в запросе, не изменилась с момента последнего запроса. Если механизм GORM обнаружит изменения в любом элементе в любой таблице, результаты будут удалены из кэша, и запрос будет передан базе данных. По сути здесь появляется лишняя операция проверки кэша.

Но если вы по-прежнему хотите использовать кэширование запросов, у вас есть для этого несколько возможностей. Первая из них заключается в использовании аргумента `cache` в динамических методах поиска, как показано ниже:

```
def entries = Post.findAllByUser(user, [cache: true])
```

Такой подход дает неплохие результаты, если используемый запрос может быть выражен в терминах динамических методов поиска. К сожалению, такое случается нечасто, поэтому у вас остаются две возможности: запросы с критериями и запросы HQL.

Запросы с критериями обеспечивают лучшее решение, и его мы рекомендуем. Представим, что необходимо кэшировать последние сообщения пользователя, для этого достаточно указать параметр `cacheable(true)`:

```
def entries = Post.createCriteria().list {
    eq('user', user)
    cacheable(true)
}
```

Но запомните, если изменится хотя бы одна таблица, `Posts` или `Users` (не обязательно для данного пользователя, но и для любого другого), этот запрос будет удален из кэша.

Критерии с запросами, использующие эту возможность, выглядят достаточно просто. Но если вы уже применяете запрос HQL, вы можете напрямую использовать библиотеку Hibernate:

```
def recentPosts = {
    def posts = Post.withSession { session ->
        session.createQuery(
            "select p from $Post.name p where p.user.id=:userid")
            .setCacheable(true)
            .setString("userid", params.userid)
            .list()
    }
    [posts : posts ]
}
```

Как видите, этот прием более мощный, но менее элегантный, чем динамические методы поиска или запросы с критериями, которые мы предпочитаем использовать для кэшируемых запросов (на практике).

По умолчанию все запросы сохраняются в едином кэше с именем `org.hibernate.cache.StandardQueryCache`. Вы можете указать кэш с этим именем в своем файле ehcache.xml, если вам потребуется определить времена существования объектов в кэше. Если этого не сделать, будут использоваться настройки `defaultCache`. В качестве альтернативы можно использовать различные *области кэширования* для каждого кэшируемого запроса.

Чтобы задействовать области кэширования, определите в файле ehcache.xml элемент `cache` с именем кэша (например, `hourlyCache`). После этого имя области кэширования можно передавать в запросы с критериями:

```
def entries = Post.createCriteria().list {
    eq('user', user)
    cacheable(true)
    cacheRegion('hourlyCache')
}
```

Если для выполнения запросов используется HQL, присоедините вызов метода `setCacheRegion('hourlyCache')`, чтобы получить тот же эффект.

Мы рассмотрели множество вариантов повышения производительности запросов. Но, прежде чем мы завершим этот раздел, рассмотрим еще одну возможность увеличения производительности запросов на уровне сервера, которую мы еще не исследовали. Источники данных JNDI, которые предоставляются большинством серверов, обеспечивают эффективный способ управления соединениями с базами данных. Пришло время заняться их исследованием и поговорить о преимуществах, которые они предлагают.

13.2.7. JNDI? Это что-то древнее...

Если вы всегда относили интерфейс языка Java для доступа к службам имен и каталогов (Java Naming and Directory Interface, JNDI) к категории «древний хлам J2EE, который остался в далеком прошлом», пришло время взглянуть на него по-новому. Существует несколько очевидных причин использовать источники данных JNDI в проектах на платформе Grails, особенно, когда имеются уже действующие источники данных:

- Вам не придется жестко определять в приложении имена пользователей, пароли, имена баз данных или даже типы, благодаря чему вы сможете хранить исходные тексты приложения в системах управления версиями даже на общедоступных сайтах, таких как GitHub, SourceForge и Google Code.
- Он предлагает достаточно эффективный способ объединения сервером ресурсов в единый пул, включая добавление подключений к пулу в случае необходимости и повторного подключения бездействующих экземпляров.
- Каждый разработчик может свободно использовать предпочитаемую им систему управления базами данных без использования дополнительных файлов настройки и библиотек.

Однако интерфейс JNDI имеет и некоторые недостатки:

- Формат именования, используемый службой JNDI, может отличаться от контейнера к контейнеру.
- Настройка для работы с Jetty сопряжена с определенными проблемами и требует создания собственного файла jetty-env.xml в каталоге /WEB-INF/.

Однако нет никаких существенных причин не применять интерфейс JNDI для работы с действующими у вас источниками данных, и мы рекомендуем изменить настройки базы данных для эксплуатационного режима на использование JNDI сразу же после выполнения команды `grails create-app`. Однажды по неосторожности мы запустили в эксплуатацию одно небольшое приложение, использующее встроенную базу данных HSQLDB, и мы не хотели бы, чтобы вы прошли через те же муки, через которые пришлось пройти нам.

В листинге 13.6 приводится пример настройки источников данных JNDI в файле DataSource.groovy для сервера приложений Glassfish. Эти настройки могут изменяться от сервера к серверу, поэтому обязательно ознакомьтесь с документацией.

Листинг 13.6. Настройка источников данных JNDI для использования в Glassfish

```
// специальные настройки окружения
environments {
```

```
    // ...

    production {
        dataSource {
            dbCreate = "update"
            jndiName = "jdbc/hubbub"  ← Настройка имени JNDI на сервере приложений
        }
    }
}
```

После внесения изменений в файл DataSource.groovy вам необходимо определить на сервере приложений сам источник данных JNDI. Обычно создание источника данных JNDI и настройка имени пользователя базы данных, пароля, размера пула подключений и других параметров выполняется с помощью консоли администрирования сервера.

На этом мы заканчиваем наш обзор советов и рекомендаций по настройке запросов и источников данных. Теперь мы займемся проблемами интеграции с уже имеющимся программным обеспечением. Интеграция с имеющимися базами данных – очень важная часть разработки корпоративных приложений, поэтому всю вторую половину главы мы посвятили исследованию типичных проблем, с которыми вам, возможно, придется столкнуться при интеграции своих новых приложений Grails в исторически сложившееся окружение.

13.3. Интеграция с имеющимся ПО: работа с несколькими источниками данных

Механизм платформы Grails для работы с источниками данных позволяет легко настроить единственный источник данных, но как быть, если приложению потребуется использовать несколько источников данных? Представим в качестве примера, что вам необходимо обратиться к таблице с названиями стран и их кодами, находящейся в другой базе данных.

Лучше всего для этого воспользоваться расширением Datasources, написанным Бартом Беквизом (Burt Beckwith); оно позволяет отобразить различные классы предметной области на различные базы данных. Вы по-прежнему можете использовать динамические методы поиска, а платформа Grails будет направлять запросы соответствующему источнику данных, опираясь на ваши настройки.

Первый шаг – установка раширения:

```
grails install-plugin datasources
```

Далее вам требуется указать, какие классы предметной области на какие источники данных отображаются. Для определения альтернативных источников данных и правил отображения классов предметной области на источники данных расширение использует специальный файл

с именем /grails-app/conf/Datasources.groovy (обратите внимание на множественное число слова Datasources).

В листинге 13.7 приводится пример определения в файле Datasources.groovy, которое загружает два класса предметной области из отдельной базы данных PostgreSQL. Большинство настроек будут вам знакомы, потому что вы уже видели их в стандартном файле DataSource.groovy.

Листинг 13.7. Настройка нескольких источников данных в файле /grails-app/conf/Datasources.groovy

```
datasources = {

    datasource(name: 'countries') {
        domainClasses([com.grailsinaction.CountryCodes,
                       com.grailsinaction.RegionCodes])
        readOnly(true)
        driverClassName('org.postgresql.Driver')
        url('jdbc:postgresql://localhost/reference_db')
        username('glen')
        password('password')
        dbCreate('update')
        logSql(true)
        dialect(org.hibernate.dialect.PostgreSQLDialect)
        hibernate {
            cache {
                use_second_level_cache(false)
                use_query_cache(false)
            }
        }
    }
}
```

Определяет классы, использующие источник данных

Помечает источник данных как доступный только для чтения

В этом файле вы можете настроить столько источников данных, сколько потребуется, и создать соответствующие записи в компоненте `domainClasses`, перечислив каждый класс. После добавления определения вы можете обращаться к своим объектам предметной области с помощью обычных динамических методов поиска, таких как `CountryCodes.findByCode('au')`. Операции сохранения и изменения также работают.

Расширение по-прежнему находится на стадии разработки, но оно обещает стать простым способом интеграции нескольких источников данных в одно приложение на платформе Grails. За дополнительной информацией об имеющихся параметрах настройки расширения обращайтесь к электронной документации, которую можно найти по адресу: *http://grails.org/plugin/datasources*.

Теперь, когда вы знаете, как обеспечить взаимодействие с несколькими источниками данных, займемся исследованием следующей проблемы – взаимодействия с базами данных, имеющими ужасную структуру таблиц.

13.4. Сложности взаимодействий с существующими базами данных

Было бы неплохо, если бы всякий раз у нас имелась возможность начинать создание модели данных с чистого листа, но жизнь редко бывает такой простой. Вам часто придется иметь дело с существующими базами данных, которые ваш администратор баз данных не собирается модифицировать. Как и все приличные платформы приложений, Grails прекрасно может действовать в существующем окружении.

В этом разделе мы исследуем два приема переноса существующих приложений баз данных на платформу Grails. Сначала мы посмотрим, как можно повторно использовать существующие отображения Hibernate и объекты предметной области на языке Java, и создадим пользовательский интерфейс на платформе Grails. После этого мы рассмотрим ситуацию, когда имеется давно существующая база данных, но нет отображений Hibernate. В таком случае вы просто можете пропустить этап создания отображений Hibernate и сразу использовать язык DSL механизма GORM для определения отображения объектов предметной области в схему существующей базы данных.

Но сначала мы познакомим вас с неудачно сконструированной базой данных. Эта модель данных была обнаружена нами на существующем в действительности предприятии (правда, мы изменили имена, чтобы никого не обижать). Данная модель содержит все неприятные сюрпризы, какие только можно ожидать: естественные строковые ключи; автоинкрементные ключи, не основанные на последовательностях; таблицы со ссылками «многие-ко-многим»; использование нескольких типов данных для создания ключей таблиц (включая char) и многие другие. Схематически структура модели данных изображена на рис. 13.2.

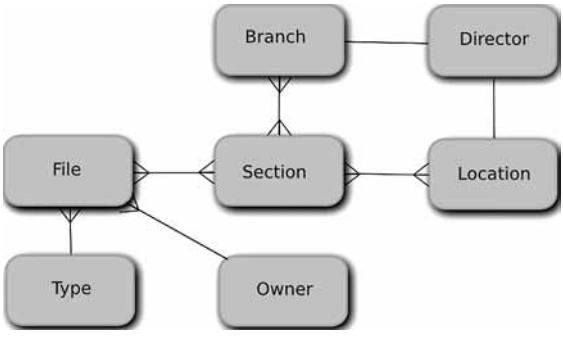


Рис. 13.2. *Пример существующей базы данных*

Рассмотрим процесс создания простого одностраничного пользовательского интерфейса, который позволит нам просматривать содержимое

этой модели данных. Если у вас имеются отображения Hibernate для базы данных, будет совсем неплохо повторно их использовать. Для начала исследуем этот вариант.

13.4.1. Повторное использование отображений Hibernate

Если у вас уже имеется длительный опыт работы с библиотекой Hibernate, следовательно, вы уже знакомы с файлами отображений и аннотациями. Вероятно, у вас уже имеются файлы .hbm.xml и соответствующие им классы предметной области. Нет никакой нужды отказываться от них.

Вы можете скопировать свои существующие файлы отображений в каталог grails-app/conf/hibernate, который предназначен специально для этого. Скопируйте свои классы предметной области в каталог /src/java/ (или, возможно, в виде файла JAR в каталог /lib), и вы сможете их использовать. Если ваш файл с настройками Hibernate имеет имя /grails-app/conf/hibernate/hibernate.cfg.xml, тогда вам ничего больше не нужно делать.

В листинге 13.8 приводится пример файла hibernate.cfg.xml для базы данных, структура которой изображена на рис. 13.2.

Листинг 13.8. Повторное использование существующих XML-файлов отображения Hibernate

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
      "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
      "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory>
        <mapping resource="org.hbm.xml"/>
    </session-factory>
</hibernate-configuration>
```

Обратите внимание, что здесь отсутствуют настройки источника данных, которые вы привыкли видеть в файле hibernate.cfg.xml, – они уже определены в файле DataSource.groovy. Но нам необходимо определить ссылку на существующий файл org.hbm.xml, содержащий определения отображений Hibernate классов предметной области для базы данных. Для определения этих отображений мы можем использовать аннотации, но это уже существовавшие ранее настройки, поэтому будем считать, что новые возможности из мира XML нам недоступны.

Наш файл org.hbm.xml – это стандартный файл отображения Hibernate. Он определяет отображение серии таблиц с причудливыми именами на множество стандартных классов Java. В листинге 13.9 приводится фрагмент файла отображения, чтобы вы могли получить представление о том, как выглядит его содержимое.

Листинг 13.9. Фрагмент существующего файла отображения Hibernate

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD//EN"
      "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping auto-import="true" default-lazy="false"
              package="com.grailsinaction.legacy.db">
    <class name="Branch"
          table="BK_BRANCH" mutable="false">
        <cache usage="read-write"/>
        <id name="name" column="BRANCH_NM">
            <generator class="assigned"/>
        </id>

        <set name="sections" table="BK_BRANCH_TO_SECTION"
              cascade="save-update">
            <key column="BRANCH_NM"/>
            <many-to-many class="Section" column="SECTION_ID"/>
        </set>

        <set name="files" table="BK_BRANCH_TO_FILE"
              cascade="save-update">
            <key column="BRANCH_NM"/>
            <many-to-many class="File" column="FILE_ID" />
        </set>

        <one-to-one name="manager" class="Manager" cascade="save-update"/>
    </class>
```

Синтаксис довольно неприглядный, но с его помощью выполняется значительная доля работы. Полный комплект файлов отображений для данной предметной области включен в исходные тексты примеров к книге.

Приукрасим мою модель данных: добавление интерфейса AJAX

Предположим, что мы получили свою действующую модель посредством импортирования существовавших отображений Hibernate, и у нас также имеется доступ к нашим объектам DTO языка Java. Напишем контроллер, чтобы создать простой пользовательский интерфейс для просмотра нашей модели предметной области.

У нас нет возможности задействовать механизм скаффолдинга, потому что сгенерированные им формы опираются на предположение о существовании полей идентификаторов, которые у нас отсутствуют. Мы могли бы изменить шаблоны скаффолдинга, но для этого нам придется потрудиться больше, чем нам хотелось бы. Создадим простой контроллер просмотра базы данных, опирающийся на использование технологии Ajax.

Ниже приводится реализация контроллера:

```
import com.grailsinaction.legacy.db.*

class BrowserController {

    def index = { redirect(action: 'list') }

    def list = {
        // Переход к форме с поддержкой Ajax
    }
}
```

Здесь нет ничего сложного – мы определили метод `list` для перехода к форме, обладающей поддержкой Ajax. В листинге 13.10 приводится реализация страницы list.gsp с раскрывающимся списком, заполняемым с помощью технологии Ajax.

Листинг 13.10. Реализация страницы list.gsp с применением технологии Ajax

```
<html>
<head>
    <title>Legacy DB Browser</title>
    <g:javascript library="prototype"/> </head> <body>
</head>
<body>
    <fieldset>
        <legend>Browse Branches</legend>
        <g:form action="showDetails">
            <label for="id">Branch:</label>
            <g:select name="id"                                  ← Заполняет
                from="${com.grailsinaction.legacy.db.            раскрывающийся список
                ➥ Branch.list()}"                               объектами из таблицы
                optionKey="name" optionValue="name"/>            Branch
            <g:submitToRemote value="Show Details"               ← Выбранный объект Branch
                update="details"                                 отправляется средствами
                url="[action: 'showDetails']"/>                  Ajax
        </g:form>
    </fieldset>
    <div id="details"></div> ← Элемент DIV, замещаемый содержимым ответа,
</body>                        который возвращается по запросу Ajax
</html>
```

Здесь нет ничего необычного. У нас имеется раскрывающийся список, заполненный названиями филиалов, и кнопка Show Details (Показать подробности), щелчок на которой приводит к отправке запроса контроллеру и замещению элемента DIV возвращаемым значением. Получившийся пользовательский интерфейс показан на рис. 13.3.

Добавим в контроллер логику, которая будет обрабатывать полученную форму и возвращать сформированную разметку HTML. Самый простой способ сгенерировать разметку HTML – воспользоваться построителем

разметки; реализуем программный код так, чтобы он возвращал информацию о директоре выбранного филиала. В листинге 13.11 приводится первый вариант программного кода.

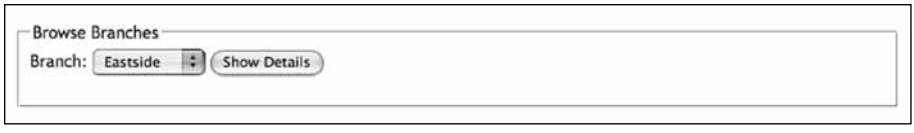


Рис. 13.3. *Интерфейс со списком филиалов отправляет запрос Ajax*

Листинг 13.11. Реализация метода, возвращающего информацию о филиале

```
def showDetails = {
    def branch = Branch.findByName(params.id) ← Поиск филиала в базе данных

    def writer = new StringWriter()
    def html = new groovy.xml.MarkupBuilder(writer)

    // Все это вполне возможно было бы реализовать непосредственно
    // в вызове метода render(), но это осложнило бы отладку
    html.div { ← Для отображения информации о филиале используется построитель разметки
        div(id: 'manager') {
            fieldset {
                legend('Manager Details')
                dl {
                    dt('Name:')
                    dd(branch.manager.name)
                    dt('Rating:')
                    dd(branch.manager.managementRating)
                }
            }
        }

    }

    render(writer.toString())

}
```

Обратите внимание, что в листинге 13.11 мы использовали динамический метод поиска, `Branch.findByName()`, вызвав его относительно класса DTO, унаследованного из прежней реализации. Платформа Grails снабдила объект DTO необходимыми ей механизмами сохранения и выполнения запросов, поэтому мы свободно можем использовать эти механизмы при работе с унаследованными классами.

Реализовав создание разметки при навигации по модели данных, попробуем воспользоваться нашим пользовательским интерфейсом, как показано на рис. 13.4.

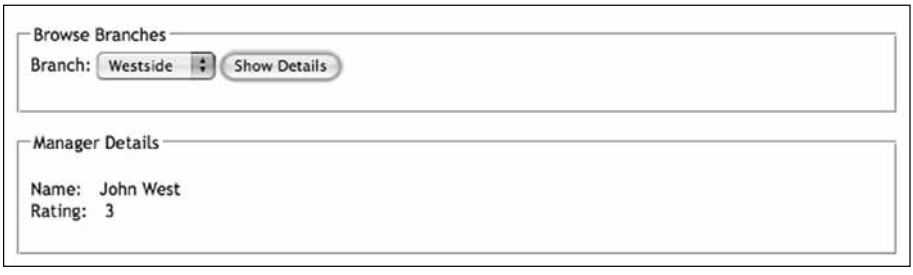


***Рис. 13.4.** Вывод основной информации о директоре филиала*

Фантастика! Наш интерфейс Ajax действует, но мы пока не задействовали все потенциальные отношения для просмотра связанных данных. Исправим этот недостаток прямо сейчас – создадим более полный построитель разметки. В листинге 13.12 приводится программный код, который позволяет получить информацию обо всей существующей модели данных.

Листинг 13.12. Более полный метод showDetails, который задействует все отношения

```
def showDetails = {
    def branch = Branch.findByName(params.id)

    def writer = new StringWriter()
    def html = new groovy.xml.MarkupBuilder(writer)

    // Все это вполне возможно было бы реализовать непосредственно
    // в вызове метода render(), но это осложнило бы отладку
    html.div {
        div(id: 'manager') {
            fieldset {
                legend('Manager Details')
                dl {
                    dt('Name:')
                    dd(branch.manager.name)
                    dt('Rating:')
                    dd(branch.manager.managementRating)
                }
            }
        }

        div(id: 'sections') {
            branch.sections.each { section ->
                fieldset {
                    legend('Section: ' + section.name)
                    dl {
                        dt('Start Date:')
                        dd(section.start)
```

```
                        dt('Files:')
                        dd(section.files.size())

                        section.files.each { sectToFile ->
                            dl(style:
                              'padding: 1em; border: 1px dotted black') {
                                dt('File Name: ')
                                dd(sectToFile.file.name)
                                dt('Type:')
                                dd(sectToFile.file.resourceType.name)
                                dt('Created:')
                                dd(sectToFile.start)
                                dt('Owner:')
                                dd(sectToFile.file.owner.name)
                            }
                        }
                        dt('Locations:')
                        dd(section.locations.size())
                        ul {
                            section.locations.each { sectToLoc ->
                                li(sectToLoc.location.name)
                            }
                        }
                    }
                }
            }
        }
    }
    render(writer.toString())
}
```

Построитель разметки получился более длинным, но теперь он отображает полную информацию о дереве объектов. Чтобы убедиться, что все действует правильно, взгляните на изменившийся пользовательский интерфейс, изображенный на рис. 13.5.

Мы знаем, что нам удалось задействовать все отношения, доставшиеся нам в наследство, но нам не хотелось бы ограничиваться возможностью доступа к данным в режиме «только для чтения». Поэтому сейчас мы посмотрим, какие существуют возможности сохранения новых объектов в имеющуюся базу данных; попутно мы также посмотрим, как можно усовершенствовать проверку некоторых ограничений.

Применение скаффолдинга к доставшимся в наследство классам предметной области Hibernate (POJO)

Может быть, вам будет интересно узнать, можно ли применить механизм скаффолдинга, рассмотренный в главе 4, к классам Java, полученным из унаследованных отображений Hibernate.

Самое удивительное, что это действительно возможно! Вы можете задействовать скаффолдинг в своем контроллере, воспользовавшись полным именем класса DTO:

```
def scaffold = com.grailsinaction.legacy.db.User
```

Платформа Grails создаст динамическую форму для него. Однако следует заметить, что этот прием практически никогда не работает без дополнительного вмешательства. Механизм скаффолдинга в платформе Grails делает некоторые предположения о структуре классов предметной области (включая предположение о наличии поля идентификатора с именем `id`). Эту проблему можно решить, создав с помощью механизма скаффолдинга реализацию шаблона и представления для контроллера (`grails generate-all`) и затем отредактировав ее вручную. Этот прием все еще способен помочь вам сэкономить свое время.

Рис. 13.5. *Вывод всей связанной информации*

Добавление ограничений в унаследованные классы POJO

Теперь, когда у нас имеется отображение унаследованных классов, можно рассмотреть некоторые возможности, позволяющие добавлять дополнительные ограничения. Даже хотя унаследованные классы являют-

ся аннотированными объектами POJO, платформа Grails по-прежнему обеспечивает нас средствами, основанными на соглашениях, для определения унаследованных отображений.

В классы Java нельзя добавить замыкания `constraints` с определениями ограничений (по меньшей мере не в JDK6), но платформа Grails позволяет определять ограничения за пределами классов Java в файлах с именами *ИмяКлассаПредметнойОбласти*Constraints.groovy. Ниже приводится пример файла BranchConstraints.groovy:

```
package com.grailsinaction.legacy.db

def constraints = {
    name(size: 4..30)
}
```

Важно отметить, что эти классы помещаются рядом с определениями объектов POJO в каталоге /src/java (а не в каталоге /src/groovy, как можно было бы ожидать). Обратите также внимание, что в данном случае замыкание `constraints` не является статическим, как в классах GORM.

Теперь напишем тест, чтобы подтвердить, что созданное ограничение действует, как ожидается. Реализация теста приводится в листинге 13.13.

Листинг 13.13. Тестирование проверки ограничений в унаследованных классах

```
import com.grailsinaction.legacy.db.*
import grails.test.*

class BranchTests extends GrailsUnitTestCase {

    void testBranchConstraints() {

        def branch = new Branch(name: 'a')
        assertFalse(branch.validate())
        branch.name = 'aaaa'
        assertTrue(branch.validate())

    }
}
```

← Проверка короткого имени 'a' потерпит неудачу

← Проверка имени с допустимой длиной будет успешно пройдена

Если теперь выполнить команду `grails test-app`, тест будет выполнен успешно, что подтверждает корректную работу ограничения.

После изучения возможности определения ограничений мы обладаем всем необходимым для успешной интеграции с отображениями Hibernate, доставшимися нам в наследство. Но как быть, если отображений Hibernate не существует? Давайте посмотрим, как можно использовать язык DSL механизма GORM для создания отображений базы данных без всех этих файлов XML.

13.4.2. Использование GORM DSL для доступа к существующей структуре таблиц базы данных

Интеграция отображений Hibernate позволяет получить фантастические результаты, если эти отображения Hibernate уже существуют. Но вам может потребоваться реализовать пользовательский интерфейс для базы данных, которая раньше управлялась приложением на языке PHP. В таких ситуациях для организации работы с существующими базами данных можно использовать язык DSL механизма GORM.

Добавляя в классы предметной области раздел `mapping` (который вы уже видели в разделе 13.2), можно изменять названия таблиц или полей, определять соединение таблиц, использовать собственные стратегии генерации ключей и многое другое. За кулисами определения на языке GORM DSL создают динамические отображения Hibernate, благодаря чему вы можете работать со своими классами предметной области и не иметь дела с файлами XML.

В этом разделе мы повторно реализуем пример из предыдущего раздела, но на этот раз с помощью GORM DSL. Сначала нам необходимо создать множество объектов классов предметной области в каталоге / grails-app/domains, потому что теперь мы будем иметь дело с обычными классами предметной области платформы Grails. Мы сохраним прежние имена пакетов, чтобы иметь возможность повторно использовать программный код контроллера, созданный ранее для просмотра новой объектной модели. На рис. 13.6 приводится список переделанных классов предметной области, которые теперь написаны на языке Groovy.

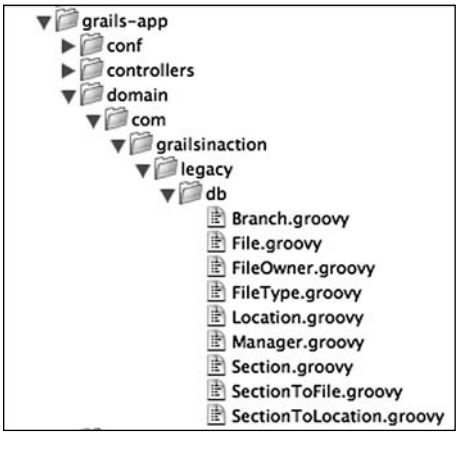


Рис. 13.6. *Новая структура предметной области*

После создания модели предметной области можно познакомиться с некоторыми возможностями отображения таблиц, доставшихся по наследству.

Основы: изменение имен таблиц и полей

У всякого администратора базы данных имеется своя долгая история выработки собственных причудливых стандартов по именованию объектов в базе данных, поэтому в первую очередь нам необходимо научиться создавать отображения в собственные имена полей и таблиц. Вы можете задействовать любые имена полей, какие вам только понравятся, воспользовавшись преимуществами блока `mapping`.

Займемся сначала чем-нибудь простым, таким как объект `FileOwner`, который не связан никакими отношениями. Переделанное определение класса предметной области приводится в листинге 13.14.

Листинг 13.14. Класс предметной области с блоком mapping, определяющим отображение на существующую таблицу

```
package com.grailsinaction.legacy.db

class FileOwner {

    int id ← ❶ Используется целочисленный ключ
    String name
    String description

    static mapping = {
        table 'BK_FILE_OWNER' ← ❷ Объявляется собственное имя таблицы
        version false ← ❸ Отключается поле версии, используемое библиотекой Hibernate

        id column: 'FILE_OWNER_ID', generator: 'increment' ← ❹ Определяется собственное имя поля первичного ключа и генератор значений для него
        name column: 'OWNER_NM'
        description column: 'OWNER_DESC'
    }

}
```

Наше замыкание `mapping` вводит несколько новых конструкций, с которыми вы еще не встречались прежде. Мы можем изменить имя таблицы ❷; и, определяя имена свойств в блоке, отобразить их на соответствующие поля в базе данных.

Обратите также внимание на строку `version false` ❸. Библиотека Hibernate использует поле `version` с целью отслеживания версий объектов для реализации оптимистической блокировки. Эта возможность не была реализована в унаследованных таблицах, поэтому мы ее отключили.

Мы также объявили поле `id` типа `int` (по умолчанию используется тип `long`), чтобы обеспечить соответствие с целочисленным полем `id` в базе данных ❶. Вместо этого можно было бы указать тип поля непосредственно в определении отображения (`id column: 'FILE_OWNER_ID', type: 'integer'`), что удобно при работе с полями, имеющими несколько необычный формат `timestamp` или `char` (такими как символьное поле `yes_no`

в базе данных DB2, используемое для представления логических значений).

Для поля `id` используется один из встроенных в библиотеку Hibernate генераторов для создания значений ключа. В данном случае мы задействовали стратегию `increment` ❹, которая обеспечивает увеличение числовых значений, но вы можете использовать любые другие генераторы (включая `sequences`, `hilo` и даже присваивание естественных ключей, о которых мы поговорим чуть ниже).

Все это основы отображения собственных имен полей и таблиц, но как быть с более тернистой проблемой отношений? Пришло время копнуть немного глубже.

Добавление отношений «один-ко-многим»

Одними из наиболее типичных отношений являются отношения «один-ко-многим» (или «многие-к-одному»). Наш объект `File` имеет несколько отношений «многие-к-одному».

В листинге 13.15 демонстрируется отображение этого сложного класса с использованием конструкций на языке DSL.

Листинг 13.15. Отображение сложных отношений с использованием конструкций DSL

```
package com.grailsinaction.legacy.db

class File {

    int id
    FileType resourceType
    String name
    short securityRating
    FileOwner owner
    Date start = new Date()
    Date end = new Date()
    String description

    static hasMany = [sections: SectionToFile]

    static mapping = {
        table 'BK_FILE'
        version false

        id column: 'FILE_ID', generator: 'increment'
        name column: 'FILE_NM'
        start column: 'START_DT', type: 'date'
        end column: 'END_DT', type: 'date'
        description column: 'FILE_DESC'
        securityRating column: 'SECURITY_RATING_VAL'
        owner column: 'FILE_OWNER_ID'
        resourceType column: 'FILE_TYPE_CD'
```

Моделируется отношение «многие-ко-многим» с помощью связанного объекта

Применяются собственные имена полей и типов

```
        sections column: 'SECTION_ID', joinTable: 'BK_FILE_SECTION_MAP'
    }
}
```

Определяется собственное соединение таблиц, реализующее отношение «многие-ко-многим»

Объекты `FileType` и `FileOwner` в этом классе связаны отношением «многие-ко-многим». Для его реализации достаточно отобразить их на имена полей в базе данных, которые являются внешними ключами. Определив отображение, вы получаете возможность манипулировать деревом объектов, как при использовании стандартного отношения в платформе Grails.

«Многие-ко-многим»: соединение таблиц

В нашем примере приложения необходимо явно смоделировать объекты соединения `SectionToFile`. Это обусловлено наличием дополнительных атрибутов в соединении (в нашем случае начальной и конечной дат владения файлом). Часто в соединении не участвуют дополнительные атрибуты, и в таких случаях получается классическое отношение «многие-ко-многим» с соединением таблиц, содержащим только два идентификатора, по одному для каждого объекта в отношении.

У нас имеется простое соединение таблиц без участия дополнительных атрибутов, связывающее таблицы `Section` и `Location`. В соединении таблиц `BK_LOCATION_SECTION_MAP` участвуют идентификаторы с обеих сторон и отсутствуют какие-либо дополнительные атрибуты. Для отображения такого отношения на языке GORM DSL необходимо определить блок `hasMany` и затем добавить соответствующее отображение `joinTable`. Отображение соединения таблиц демонстрируется в листинге 13.16.

Листинг 13.16. Отображение отношений, когда используется соединение таблиц

```
package com.grailsinaction.legacy.db

class Section {

    int id
    String name
    Date start = new Date()
    Date end = new Date()

    static hasMany = [files: SectionToFile,
                      locations: Location,
                      branches: BranchToSection ]

    static belongsTo = [ Branch ]

    static mapping = {
        table 'BK_SECTION'
        version false
```

```
        id column: 'SECTION_ID', generator: 'increment', type:'integer'
        name column: 'SECTION_NM'
        start column: 'START_DT', type: 'date'
        end column: 'END_DT', type: 'date'
        locations column:'SECTION_ID',
            joinTable:'BK_LOCATION_SECTION_MAP'
    }
}
```

← Отображение отдела на филиал через соединение таблиц

Обратите особое внимание на атрибут `locations`. В нем используется определение `joinTable`, в котором указывается имя таблицы связи для отношения «многие-ко-многим», и указывается поле (`SECTION_ID`) с этой стороны отношения.

Крайний случай: естественные и составные ключи

Нам необходимо рассмотреть еще несколько вопросов, связанных с ключами. Не во всех таблицах имеется возможность использовать суррогатные первичные ключи – во многих случаях уникальность обеспечивается естественным ключом или комбинацией полей, входящих в состав составного ключа. В языке GORM DSL имеется поддержка обоих типов ключей.

В нашем примере имеется объект связи `SectionToFile`, который моделирует атрибуты соединения таблиц `Section` и `File`. Так как это таблица связи, в ней нет ни естественного, ни суррогатного ключа, но на основе обоих полей мы можем смоделировать составной ключ. Данный прием демонстрируется в листинге 13.17. Обратите внимание, что библиотека Hibernate требует, чтобы эти объекты связи реализовали интерфейс `Serializable`.

Листинг 13.17. Обслуживание существующих таблиц связи в отношениях «многие-ко-многим»

```
package com.grailsinaction.legacy.db

class SectionToFile implements Serializable {

    Section section
    File file
    Date start
    Date end

    static mapping = {
        table 'BK_FILE_SECTION_MAP'
        version false

        id composite:['file', 'section']
        start column: 'START_DT', type: 'date'
        end column: 'END_DT', type: 'date'
        file column: 'FILE_ID'
        section column: 'SECTION_ID'
```

← Реализует интерфейс Serializable (согласно требованиям)

```
    }

}
```

Единственное, что мы потеряли, определив составной ключ на базе идентификаторов, – это возможность использовать метод `get()` на основе составного идентификатора. Вам придется использовать запрос, чтобы, например, вернуться обратно к оригинальному объекту.

Еще более сложным является случай использования естественных ключей. В классе `Branch` первичный ключ таблицы формируется свойством `name`. Уверяем вас, что при определенной сноровке механизм GORM способен решить и эту проблему, как показано в листинге 13.18.

Листинг 13.18. Обслуживание естественных ключей с использованием возможностей механизма GORM

```
package com.grailsinaction.legacy.db

class Branch {

    String id

    static transients = ['name']

    void setName(String name) {
        id = name                    ← Изменяет значение, лежащее в основе поля id
    }

    String getName() {
        return id                    ← Возвращает значение, лежащее в основе поля id
    }

    static hasMany = [sections:Section]

    static mapping = {
        table 'BK_BRANCH'
        version false

        id generator:'assigned', column: "BRANCH_NM", type:'string'
        sections column:'SECTION_ID',
                joinTable:'BK_BRANCH_TO_SECTION',
                type:'integer'

    }
}
```

Механизм GORM опирается на наличие поля `id`, и это постоянно проявляется в наших отображениях. В листинге 13.18 у нас имеется поле `name`, но нет поля `id`, поэтому мы создали методы чтения и записи для отображения нашего внутреннего поля `id` на общедоступное поле `name`. Выглядит не очень красиво, но решает нашу проблему.

Будьте внимательны при сохранении объектов, отображенных таким способом, потому что платформа Grails рассматривает непустое значение `id` как признак экземпляра, хранящегося в базе данных. При использовании такого подхода вы должны использовать параметр `branch.save(insert: true)`.

На этом мы заканчиваем исследование источников данных, доставшихся по наследству. Мы продемонстрировали вам один из худших примеров базы данных, поэтому теперь вы должны чувствовать в себе уверенность, что можете заняться базами данных с любой структурой, какая только достанется вам от администратора баз данных.

Когда следует использовать GORM DSL, а когда отображения Hibernate

Теперь, когда у вас имеется хорошее представление о возможностях GORM DSL и отображений Hibernate при работе с одной и той же ужасной базой данных, вас может заинтересовать вопрос, какой из этих двух вариантов предпочтительнее. Совершенно очевидно, если у вас уже имеются отображения Hibernate, есть смысл использовать их. Но как быть, если у вас имеется совершенно новое веб-приложение, которое необходимо объединить с сумасшедшей базой данных, доставшейся по наследству? Как обычно, ответ на этот вопрос такой: «все зависит от ситуации».

Если в базе данных широко используются естественные (не целочисленные) ключи или в соединениях таблиц используются дополнительные атрибуты, обычно проще применять отображения Hibernate. С другой стороны, если отображения Hibernate не были созданы ранее, практически всегда проблему можно решить с помощью GORM DSL (но это может потребовать пойти на некоторые компромиссы в вашей объектной модели – в конечном итоге, чтобы отобразить отношения, вам может потребоваться связывать объекты с помощью составных ключей).

Если существующая база данных была создана относительно недавно, содержит большое количество суррогатных ключей и не слишком сильно приправлена специальным соусом, предпочтительнее использование GORM DSL. В конечном итоге вы получите множество ясных классов объектов и сможете прожить свою жизнь, не утомляя себя работой со сложными файлами XML отображений Hibernate.

13.5. В заключение

В этой главе мы представили вам множество новых особенностей механизма GORM. Некоторые редко используемые типы отношений (включая наследование) способны помочь сэкономить ваше время (и привести программный код в порядок), но вы должны помнить достоинства и недостатки каждого возможного варианта.

Мы познакомили вас также с тонкостями кэширования запросов и организации кэша второго уровня и рассмотрели некоторые особенности настройки механизма кэширования.

Кроме того, мы показали некоторые приемы измерения производительности при работе с данными и исследовали инструменты профилирования, способные оказать помощь в определении узких мест. В заключение мы подробно обсудили диапазон возможностей интеграции с существующими базами данных (включая использование существующих файлов отображений Hibernate и создание отображений на языке DSL GORM).

Прежде чем перейти к исследованию платформы Spring и транзакций в главе 14, вспомним некоторые ключевые рекомендации из этой главы:

- *Разберитесь с наследованием*. При оценке возможности использовать механизм наследования постарайтесь определить, насколько дорого, с точки зрения производительности, придется заплатить за использование стратегии «одна-таблица-на-класс». Возможно, ослабленные требования к ограничениям при использовании стратегии «одна-таблица-на-иерархию» обойдутся вам дешевле.
- *Используйте источники данных JNDI*. Используйте механизм JNDI для организации доступа к действующим источникам данных (и изменяйте настройки базы данных для эксплуатационного режима на использование JNDI сразу же после выполнения команды `create app`).
- *Не опирайтесь на предположения, выполняйте профилирование*. Расширение P6Spy позволит вам получить четкое представление о времени выполнения запросов. Объединив его с инструментом создания нагрузки, таким как JMeter, вы сможете достаточно достоверно оценить, как будет вести себя ваше приложение под тяжелыми нагрузками.
- *Не переусердствуйте с кэшированием запросов, но используйте кэш второго уровня*. Иногда бывает лучше реализовать собственный механизм кэширования часто изменяющихся данных (таких, как сводные таблицы, которые обновляются регулярно, но получение которых основано на дорогостоящих запросах). Всегда включайте кэш второго уровня (хотя всегда можно найти доводы против этого, особенно при работе в кластерах).
- *Используйте существующие отображения Hibernate*. Если у вас уже имеются отображения Hibernate для существующей базы дан-

ных, используйте их. Использование GORM DSL требует идти на некоторые компромиссы, которые не стоят того, если вы уже проделали тяжелый труд по созданию действующих отображений Hibernate.

- *При работе с базами данных, имеющими удачно спроектированную структуру, отдавайте предпочтение GORM DSL; в противном случае выбирайте отображения Hibernate.* При разработке новых приложений, использующих существующие базы данных с удачной структурой, отдавайте предпочтение GORM DSL. В этом случае усилия на сопровождение будут меньше, и вашей команде не придется изучать библиотеку Hibernate. При работе с существующими базами данных, имеющими причудливую структуру, отдавайте предпочтение обычным отображениям Hibernate (особенно если у вас уже имеются созданные ранее файлы отображений Hibernate). Отображения Hibernate обладают более широкими возможностями, чем GORM DSL, и вы в конечном итоге получите более ясное решение.

В следующей главе мы нырнем в самую сердцевину платформы Grails и близко познакомимся с платформой Spring и транзакциями.

В этой главе:

- Введение в платформу Spring
- Максимальное использование платформы Spring
- Искусство транзакций

14

Платформа Spring и транзакции

После погружения в глубины механизма GORM и библиотеки Hibernate пришло время вернуться на уровень абстракции и посмотреть, как выглядит инфраструктура приложения. Механизм GORM представляет собой отличный уровень абстракции для работы с базами данных, и он позволяет вам использовать волшебство контроллеров и представлений GSP; но приложению, как и позвоночному животному, требуется скелет для скрепления всех его членов.

К счастью, платформа Grails построена на основе опробованной и проверенной платформы Spring, которая способна помочь наращивать возможности приложения. Мы уже упоминали эту платформу несколько раз; она имеет тенденцию оставаться невидимой в простых приложениях. Неприятность заключается в том, что пока она невидима, вы не замечаете мощные инструменты, имеющиеся в вашем арсенале. Мы не можем это так оставить, поэтому в данной главе мы расскажем о некоторых основах платформы Spring и о том, как платформа Grails позволяет напрямую взаимодействовать с ней.

Затем мы рассмотрим еще одну особенность, которая может оказаться весьма полезной после всех этих разговоров о базах данных в предыдущей главе: транзакции. Хотя они не входят в состав ядра платформы Spring, поддержка транзакций в Spring обеспечивает создание надежных приложений, ориентированных на работу с данными. Это очень важная тема, которая заслуживает вашего полного внимания.

Итак, посмотрим, что может нам предложить платформа Spring.

Примечание

Если вы уже являетесь поклонником платформы Spring, мы предлагаем вам бегло прочитать следующий раздел и ознакомиться с кратким введением в службы еще раз.

14.1. Spring и Grails

Некоторое время тому назад появилась новая парадигма проектирования приложений, которая получила название «Инверсия управления»[1] (Inversion of Control, IoC). Ее принцип прост: обязанности по управлению зависимостями объекта возлагаются не на сам объект, а на программный код, создающий его. Приложение делегирует ответственность за создание и инициализацию своих объектов третьей стороне – контейнеру IoC. Рис. 14.1 иллюстрирует различия между двумя подходами.

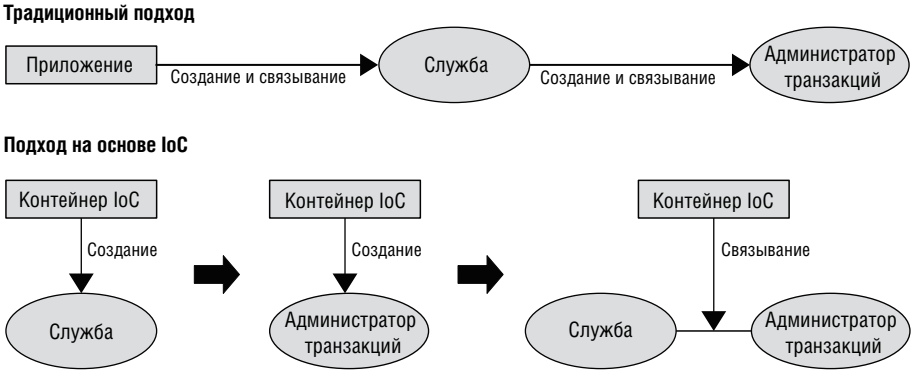


Рис. 14.1. *Парадигма инверсии управления в сравнении с традиционным подходом к управлению зависимостями: все объекты создаются контейнером IoC, а не самими объектами*

При взгляде на рис. 14.1 кажется очевидным, что традиционный подход проще для понимания, тогда зачем утруждать себя парадигмой IoC? Использование IoC имеет несколько важных преимуществ, но самое главное, что при следовании этой парадигме из класса изымается вся сложность управления зависимостями, и остается только основная логика его работы. Создание программного кода, реализующего создание или поиск соответствующих зависимостей, в классе может оказаться не самым приятным занятием, так почему не переложить эту проблему на кого-нибудь другого? Сами классы при этом становятся проще, и, как побочный эффект, их легче тестировать.

Самая распространенная на сегодняшний день форма парадигмы IoC известна под названием *«внедрение зависимостей»* (Dependency Injection), и именно этот шаблон проектирования реализован в платформе Spring. Детальное описание принципа внедрения зависимостей выходит далеко за рамки данной книги, но вы можете найти дополни-

[1] Иногда можно встретить термин «обращение контроля». – *Прим. перев.*

тельную информацию о нем в Википедии: *http://en.wikipedia.org/wiki/Dependency_injection*[1].

С помощью реализации принципа внедрения зависимостей платформа Spring управляет жизненным циклом объектов приложения, проще говоря, создает, инициализирует и разрушает их. Эти управляемые объекты называются компонентами платформы Spring (Spring beans) и являются обычными Java-объектами, которые даже не обязаны соответствовать спецификации JavaBeans, из-за чего их название вносит некоторую путаницу.

Следует заметить, что платформа Spring редко используется для управления *всеми* объектами в приложении. Обычно разработчики используют ее для управления одиночными объектами в своих приложениях. Например, следующий класс мог бы быть реализован в виде компонента платформы Spring без каких-либо дополнительных изменений:

```
class MyService {
    TransactionManager transactionManager

    void doTransaction() {
        ...
    }
}
```

Заметили, что в этом классе полностью отсутствуют ссылки на платформу Spring? Будет ли экземпляр этого класса компонентом платформы Spring, зависит лишь от того, будет ли он управляться платформой Spring.

Так как компонент платформы Spring – это «старый, добрый, простой объект Java» (Plain Old Java Object, POJO), ей необходимо сообщить, как соединять, или *связывать*, различные компоненты воедино и какими значениями инициализировать их свойства. Это достигается с помощью *дескриптора компонента* (bean descriptor), который содержит необходимые *определения компонента* (bean definitions) – информация, требующаяся для инициализации каждого объекта. Каждый компонент, управляемый платформой Spring, должен обладать соответствующим определением компонента.

Как же создаются эти дескрипторы компонентов? По традиции вы могли бы создать файл XML, следуя определенным правилам его оформления, но платформа Grails упрощает дело с помощью соглашения и замечательного языка DSL. Подробнее о файлах XML и конструкциях языка DSL мы поговорим в разделе 14.1.2 (каждый из этих двух подходов находит свое применение), но сначала мы посмотрим на простейшие компоненты Spring: службы платформы Grails.

---

[1] На русском языке: *http://ru.wikipedia.org/wiki/Внедрение_зависимости*. – *Прим. перев.*

14.1.1. Традиционный подход

Обычно разработка приложения на платформе Grails начинается с создания классов предметной области, после чего определяются контроллеры и представления. По мере продвижения вперед появляется искушение вместить в контроллеры еще больше логики, сделав их тяжеловесными и возложив на них более широкие обязанности, чем обслуживание пользовательского интерфейса. Затем вы обнаруживаете, что вам требуется повторно использовать некоторые из функциональных возможностей (например, организовать доступ к веб-службе), и приходите к открытию, что контроллеры не так-то просто вызывать из других частей приложения.

По множеству причин передовой опыт диктует, что логика работы, не связанная с пользовательским интерфейсом, должна выноситься в службы Grails. Этот опыт следует принципу разделения сфер влияния (Separation of Concerns, SoC), где под «сферами влияния» подразумеваются отдельные функциональные возможности или, возможно, более точно, зоны ответственности. Например, пользовательский интерфейс – это одна сфера влияния, доступ к базе данных – другая, а управление доступом может рассматриваться как третья. Если ограничить один класс единственной сферой влияния, программный код приложения будет проще для понимания и изменения. Обычно контроллеры реализуют обработку пользовательского интерфейса, а службы – основную логику работы приложения.

Это означает, что вся фактическая работа приложения должна выполняться службами. Главное правило, которого следует придерживаться, – контроллеры не должны обращаться к базе данных напрямую, за исключением случаев использования тривиальных запросов, ценность от многократного использования которых весьма невелика. К сожалению, трудно определить точные критерии, но эффект применения служб в общем виде можно наблюдать на рис. 14.2.

> **Подробнее о принципе SoC**
>
> Распределив свой программный код между контроллерами и службами, вы должны продолжать помнить о принципе разделения сфер влияния. Приложения сами по себе часто являются коллекцией сфер влияния, поэтому вы должны стремиться применять этот принцип ко всему программному коду. В приложении, например, управление отдельными сообщениями можно было бы интерпретировать как одну из сфер влияния, поиск – как другую, а управление друзьями и сторонниками – как третью. Кроме того, идеальным способом группировки классов по сферам влияния могут служить пакеты.

Вы уже видели службы в действии в главе 5, и вы знаете, как внедрять их в контроллеры и в другие элементы приложения на платформе Grails. Как же получается это волшебство? За кулисами платформа Grails создает для служб компоненты платформы Spring, имена которых совпадают с именами соответствующих классов, за исключением первого символа, который переводится в нижний регистр. Так, класс `PostService` превращается в компонент с именем `postService`.

Совет

Своим поведением замыкания очень похожи на методы, поэтому у вас может появиться искушение использовать их в службах вместо методов. Это не самая хорошая идея, потому что транзакции и другие особенности, опирающиеся на механизм перехвата обращений к методам, не будут действовать должным образом. Проблема в том, что ни платформа Spring, ни Java ничего не знают о замыканиях, поэтому они будут интерпретировать их как поля объекта. Только методы получают особую интерпретацию.

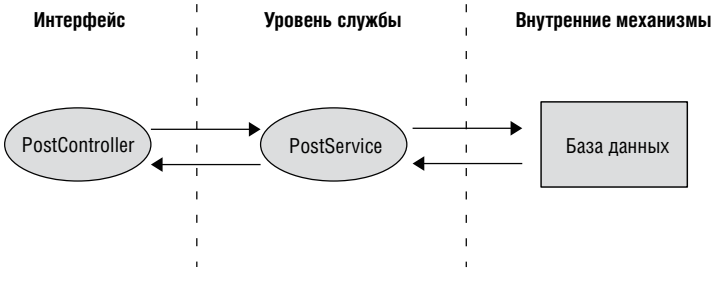


***Рис. 14.2.** Архитектура типичного приложения, использующего службы для инкапсуляции логики работы*

Имена компонентов – важная часть платформы Spring, потому что они используются для связывания объектов. В типичных приложениях на платформе Spring взаимозависимости между компонентами определяются в дескрипторах компонентов, но для внедрения компонентов вместе с их зависимостями в платформе Grails используется специальная особенность платформы Spring.

Волшебство управления зависимостями

Посмотрим еще раз на контроллер и службу из главы 5, реализующие операции с сообщениями. Эти два класса выглядят, как показано ниже:

```
class PostService {
    boolean transactional = true

    Post createPost(String userId, String content) {
        ...
    }
```

```
}

class PostController {
    def postService

    def scaffold = true
    ...
}
```

Теперь вы должны видеть, что имя свойства в контроллере совпадает с именем компонента Spring, полученного из службы `PostService`. Это основа автоматического внедрения зависимостей.

В платформе Spring имеется особенность под названием «автоматическое связывание», которая устраняет необходимость объявления зависимостей вручную; действует она следующим образом: если один из компонентов платформы Spring имеет свойство, имя которого совпадает с именем другого компонента, Spring автоматически связывает второй компонент с этим свойством. Поэтому свойство `postService` в контроллере инициализируется как ссылка на фактическую службу. Все это волшебство происходит до того, как компонент сможет использоваться, поэтому при создании методов служб и действий контроллеров можно смело предполагать, что соответствующие свойства уже были инициализированы.

Автоматическое связывание по типу

Опытные разработчики, использующие платформу Spring, знают, что существует также возможность выполнять автоматическое связывание компонентов по типу. Все компоненты, которыми управляет платформа Grails, автоматически связываются по имени, и у вы никак не можете это изменить, но когда мы будем обсуждать дескрипторы компонентов, определяемые пользователем, вы увидите, что существует возможность автоматического связывания компонентов по типу .

Пока мы говорим о типах, обратите также внимание на то, что свойство `postService` объявлено как свойство с динамическим типом. Отчасти это обусловлено тем, что повторная загрузка класса в настоящее время терпит неудачу, если в нем присутствует объявление свойства со статическим типом службы. Это проблема механизма повторной загрузки класса, которая может быть исправлена не так скоро, как хотелось бы, поэтому в настоящий момент для внедряемых служб лучше всего использовать объявление `def`.

Но минутку! Если возможность автоматического связывания действует только для компонентов Spring, как тогда осуществляется вне-

дрение контроллеров в службы? Все просто: контроллеры – это тоже компоненты Spring. Однако мы настоятельно не рекомендуем пытаться встраивать контроллеры в другие компоненты. Нет никаких гарантий, что имена компонентов и их типы в будущем не изменятся; и, кроме того, контроллеры не так-то просто использовать в другом программном коде.

Любой именованный компонент может быть внедрен таким способом. Платформа Grails сама определяет множество компонентов, выборочный перечень которых приводится в табл. 14.1. Некоторые из них могут оказаться полезными, когда окажется невозможным использовать привычные механизмы платформы Grails.

Компонент `grailsApplication` можно использовать для получения информации о приложении, такой как параметры настройки, контекст приложения на платформе Spring (рассматривается ниже) и перечень загруженных элементов. Если вам потребуется прямой доступ к возможностям библиотеки Hibernate, внедрите в свой класс компонент `sessionFactory`. Требуется выполнить парсинг страницы GSP? Воспользуйтесь компонентом `groovyPagesTemplateEngine`. И наконец, но не в последнюю очередь, компонент `messageSource` позволит вручную отыскать текст сообщения по его коду.

Таблица 14.1. Некоторые полезные компоненты Spring в платформе Grails

Имя компонента	Тип	Предоставляется
`grailsApplication`	`DefaultGrailsApplication`	Ядром Grails
`sessionFactory`	`org.hibernate.SessionFactory`	Расширением Hibernate
`groovyPagesTemplate-Engine`	`GroovyPagesTemplateEngine`	Расширением Controllers
`messageSource`	`ReloadableResourceBundle-MessageSource`	Расширением i18n

Это лишь часть того, что доступно; практически каждое расширение добавляет свои собственные компоненты. Главное препятствие к их использованию вы уже преодолели – вы знаете, что они существуют! Документация (как для платформы Grails, так и для расширений) может упоминать эти компоненты, но зачастую единственный способ обнаружить их – прибегнуть к почтовой рассылке проекта Grails или заглянуть в файлы дескрипторов расширения. Подробнее о последних будет рассказываться в главе 16. К счастью, вам редко придется обращаться к компонентам платформы Spring напрямую, и со временем соответствующая документация, вне всяких сомнений, будет улучшаться.

Механизм внедрения зависимостей действует и упрощает нам жизнь, но иногда может потребоваться задействовать компоненты внутри объектов, которые не управляются платформой Spring. Как тогда быть?

Контекст приложения на платформе Spring

В этом разделе мы покинем царство соглашений и посмотрим на другой способ извлечения компонентов Spring. Допустим, что у нас имеется сервлет, который требуется добавить в приложение и которому требуется обеспечить доступ к службам. Получить ссылку на службу можно только с помощью платформы Spring, но как получить доступ к Spring из объектов, которые ею не управляются?

Ответ на этот вопрос находится в контексте приложения на платформе Spring – в едином объекте, который управляет компонентами и загрузкой ресурсов. По сути это ядро контейнера IoC платформы Spring. Он обладает некоторыми полезными особенностями, но сейчас нас интересует только метод `getBean()`:

```
PostService postService = applicationContext.getBean("postService")
```

Все это достаточно просто; но для решения нашей проблемы сначала необходимо получить ссылку на контекст приложения, что можно сделать двумя способами. Контекст приложения можно получить из контекста сервлета, произведя следующие магические действия:

```
import org.springframework.context.ApplicationContext
import org.springframework.web.context.support.WebApplicationContextUtils

ApplicationContext appCtx =
      WebApplicationContextUtils.getWebApplicationContext(servletContext)
```

Выглядит, как фраза, произнесенная с набитым ртом! Честно говоря, такой подход можно использовать, если у вас уже имеется доступ к контексту сервлета, то есть внутри сервлета. В противном случае можно использовать экземпляр приложения на платформе Grails, как показано ниже:

```
import org.codehaus.groovy.grails.commons.ApplicationHolder

def grailsApp = ApplicationHolder.application
def appCtx = grailsApp.getMainContext()
```

Это более предпочтительный метод, если вам требуется заполучить контекст приложения, но контекст сервлета недоступен. За дополнительной информацией об данном замечательном объекте обращайтесь к справочному руководству платформы Spring и к документации с описанием API.

Традиционный подход, который мы рассмотрели в этом разделе, достаточно прост в использовании и обладает весьма широкими возможностями. Но на некотором этапе, когда речь заходит о создании и настройке компонентов Spring, соглашения превращаются в ограничивающий фактор. В следующем подразделе мы рассмотрим более мощный и гибкий способ выполнения такой работы.

14.1.2. Создание и определение собственных компонентов

Выше мы упомянули, что платформа Grails сама определяет множество компонентов Spring, и теперь вы знаете, что ваши службы также являются компонентами Spring. Но как быть с классами, которые не являются службами? Такие классы могут, например, предоставляться библиотекой Java и содержаться в файле JAR. Вам требуется какой-нибудь способ информировать платформу Spring о таких классах.

Как вы уже, наверно, догадались, платформа Grails имеет ответ на этот вопрос. В действительности у нее есть два ответа. Вы можете добавить определение компонента Spring в файл grails-app/conf/spring/resources.xml, в файл the grails-app/conf/spring/resources.groovy или в оба сразу. Они действуют примерно одинаково, но формат определения компонентов в этих двух файлах существенно отличается: в первом из них используется формат XML, а во втором – язык предметной области (Domain-Specific Language, DSL) .

> **Порядок загрузки**
>
> Оба файла ресурсов загружаются после того, как будут определены все компоненты ядра Grails и расширений. Это означает, что внутри этих файлов вы можете ссылаться на компоненты платформы Grails и расширений, но расширения не будут видеть компоненты, определяемые вами.
>
> Что особенно важно, вы можете переопределять компоненты, которые уже были определены каким-либо расширением. Например, если вы определите компонент с именем `messageSource`, он будет использоваться вместо одноименного компонента, определенного в расширении i18n.

Это вызывает у нас некоторое затруднение. Какой метод использовать, зависит от того, удобнее ли вам работать с форматом XML или DSL. Но учтите, что язык DSL платформы Spring позволяет использовать переменные, условные конструкции, циклы и все остальные конструкции языка Groovy. А XML – это всего лишь XML.

Здесь мы будем рассматривать только язык Spring DSL, но если вы захотите узнать больше об особенностях создания дескрипторов компонентов Spring на языке разметки XML, обращайтесь к справочному руководству для платформы Spring: *http://static.springframework.org/spring/docs/2.5.x/reference/*, оно даст вам базовое представление о многих особенностях платформы Spring и о том, как ими пользоваться.

Введение в Spring DSL

Признаемся честно, мы не очень любим работать с форматом XML. К счастью, этого и не требуется для определения компонентов Spring. Подобно другим диалектам DSL на базе языка Groovy для описания конфигурации компонента на языке Spring DSL используется нотация методов и замыканий. По сути любое обращение к несуществующему (или отсутствующему) методу рассматривается как начало определения нового компонента.

В листинге 14.1 приводится пример файла resources.groovy, который мы опишем ниже, чтобы вы могли понять, как он действует. Данный конкретный пример создает коллекцию статистик Hibernate посредством JMX (Java Management Extensions).

Листинг 14.1. Пример определения компонента в файле resources.groovy

```
import org.springframework.jmx.support.MBeanServerFactoryBean
import org.springframework.jmx.export.MBeanExporter
import org.hibernate.jmx.StatisticsService

beans = {

    hibernateStats(StatisticsService) { ← ❶ Начало определения нового компонента
        statisticsEnabled = true
        sessionFactory = ref("sessionFactory") ← ❷ Ссылка на именованный
    }                                              компонент

    mbeanServer(MBeanServerFactoryBean) {
        locateExistingServerIfPossible = true
    }

    exporter(MBeanExporter) {
        server = mbeanServer ← ❸ Альтернативный способ ссылки на компонент
        beans = ["org.hibernate:name=statistics": hibernateStats]
    }
}
```

Файл resources.groovy – это просто сценарий на языке Groovy, поэтому вы можете поместить сюда практически все, что пожелаете. Но, чтобы определить компоненты Spring, вы обязаны объявить переменную сценария `beans` и присвоить ей замыкание. Если определение какого-либо компонента окажется за пределами этого замыкания, оно не будет подхвачено платформой Grails и, скорее всего, вызовет исключение.

Из чего состоит определение компонента на языке DSL? Оно содержит имя компонента, за которым следует имя класса компонента в круглых скобках ❶; имя метода превращается в имя компонента, а аргумент – в класс компонента. В файле XML класс компонента определяется в виде строки, но, так как в данном случае мы имеем дело с языком

Groovy, допускается указывать имя класса в виде литерала. Это означает, что вы можете использовать конструкции импортирования, чтобы обеспечить ясность определений.

Если вам требуется настроить свойства компонента, вы должны поместить блок замыкания вслед за объявлением, как это сделано для всех трех определений в данном примере. Внутри этого блока допускается устанавливать свойства, используя простую форму записи `<имя свойства> = <значение>`. В отличие от файла XML, где все сущее является строками, значение в DSL должно иметь тип, соответствующий свойству. В листинге 14.1 мы определили значения для свойств типа `Boolean` и `Map`. Это означает, что такой синтаксис обеспечивает более краткий способ инициализации коллекций и отображений, чем в формате XML.

Использование ссылок на компоненты – это способ связывания компонентов вручную. Ссылки определяются так же, как и обычные свойства, но при этом у вас имеется два варианта записи правой части выражения присваивания. При использовании первого вы можете указать в свойстве имя компонента без кавычек ❸, но этот прием можно применять, только если требуемый компонент был определен выше в том же файле. Второй вариант заключается в использовании метода `ref()` ❷, который принимает имя компонента в виде строки. Из этих двух вариантов мы предпочитаем второй, поскольку он действует всегда независимо от того, где определяется требуемый компонент, и независимо от порядка следования определений компонентов в файле resources.groovy.

Это было довольно просто. Возможно, мы не показали в данном примере многие из имеющихся приемов, но позднее вы поймете, что продемонстрированных вполне достаточно в подавляющем большинстве случаев. Чтобы получить некоторое представление о более специфических особенностях, мы сравним форматы DSL и XML. Это позволит вам понять, что запутанность Spring имеет иной источник, и затем применить полученные знания DSL.

Сравнение форматов Spring XML и DSL

Язык Spring DSL поддерживает практически все возможности, которые вы можете найти в формате описания дескрипторов Spring XML. В табл. 14.2 приводятся конструкции в формате XML и эквивалентные им конструкции на языке DSL. Если какая-то конструкция окажется для вас непонятной или вы не сможете определить, что она делает, обращайтесь к справочному руководству для платформы Spring, где рассматриваются эти и многие другие конструкции.

Как видно из таблицы, некоторые конструкции на языке Spring DSL оказываются намного более краткими. Это обстоятельство наряду с возможностью использовать действительные типы данных при определении значений свойств и аргумента конструктора способно поколе-

бать уверенность даже самых горячих поклонников XML. Кроме того, как уже отмечалось выше, в определениях на языке DSL можно использовать конструкции языка Groovy управления потоком выполнения.

Таблица 14.2. Сравнение формата описания дескрипторов Spring XML и Spring DSL

Возможность	XML	Spring DSL

Атрибуты компонентов	`<bean id="ex" class="o.e.Ex"`
	`  scope="prototype"`
	`  autowire="byType"`
	`  init-method="init"`
	`  destroy-method="finish"/>`
		`ex(o.e.Ex) { b ->`
		`  b.scope = "prototype"`
		`  b.autowire = "byType"`
		`  b.initMethod = "init"`
		`  b.destroyMethod = "finish"`
		`}`

Списки	`<bean id="ex" class="o.e.Ex">`
	`  <property name="items">`
	`  <list>`
	`  <value>1</value>`
	`  <value>2</value>`
	`  <value>3</value>`
	`  </list>`
	`  </property>`
	`</bean>`
		`ex(o.e.Ex) {`
		`  items = [1, 2, 3]`
		`}`

Статические фабричные методы	`<bean id="ex" class="o.e.Ex"`
	`  factory-method="create"/>`
		`ex(o.e.Ex) { b ->`
		`  b.factoryMethod = "create"`
		`}`

Фабричные методы экземпляра	`<bean id="ex"`
	`  factory-bean="myFactory"`
	`  factory-method="create"/>`
		`ex(myFactory: "create")`

Рассмотрим следующий пример, где `serviceName` и `isTransactional` являются переменными, значения которых устанавливаются в другом месте:

```
def beanNames = [ "transactionInterceptor" ]

"${serviceName}Manager"(org.example.ServiceManager) {
    service = serviceName
    if (isTransactional) {
        interceptors = beanNames.collect { ref(it) }
    }
}
```

Этот пример демонстрирует, как можно создать компонент, имя которого не определяется на этапе выполнения, и как можно использовать условные конструкции, влияющие на настройку. Кроме всего прочего, мы использовали здесь метод `collect()`. Возможность смешивать обычный программный код Groovy с определением компонентов открывает перед нами целый мир новых возможностей.

Недавно в язык Spring DSL была добавлена поддержка пространств имен. Первоначально они были представлены как пространства имен XML, чтобы упростить настройку, но теперь вы можете использовать преимущества пространств имен в языке Groovy.

Пространства имен в Spring DSL

Мы продемонстрируем порядок использования пространств имен на простом примере, основанном на поддержке аспектно-ориентированного программирования (Aspect-Oriented Programming, AOP) в платформе Spring. Мы не будем погружаться в особенности аспектно-ориентированного программирования, но отметим, что в следующем примере при вызове каждого метода производится обращение к компоненту `sessionFactory`:

```
beans = {
    xmlns aop: "http://www.springframework.org/schema/aop"

    aop.config {
        aspect id: "profiling", ref: "profileInterceptor" {
            around method: "profile", pointcut: "bean(sessionFactory)"
        }
    }
    profileInterceptor(com.manning.gria.ProfilingInterceptor)
}
```

В первую очередь необходимо объявить пространство имен и присвоить ему префикс (в данном случае: `aop`), что делается с помощью встроенного метода `xmlns()`. Вы можете также объявить дополнительные пространства имен, добавив в вызов метода дополнительные именованные аргументы, как показано ниже:

```
xmlns aop: "...", context: "...", ...
```

После объявления пространства имен можно использовать свойство, имя которого совпадает с префиксом пространства имен, для доступа к различным особенностям, как мы сделали это для свойства `aop.config` в предыдущем примере. Обратите внимание, что методы `aspect()` и `around()` не предваряются префиксом, потому что они используются внутри метода с префиксом. Вы могли бы также использовать следующий синтаксис:

```
aop {
    config {
        aspect ...
    }
}
```

Нам нечего больше добавить к описанию пространств имен Spring. Любая документация с описанием определенного пространства имен от-

носится к формату XML, но теперь вы знаете, как перейти от формата XML к формату Spring DSL. Вы готовы идти дальше! За дополнительной информацией о языке Spring DSL мы рекомендуем обращаться к руководству пользователя платформы Grails, где вы найдете подробное описание этой темы, достойное вашего внимания.

Мы практически закончили обсуждение платформы Spring, однако один элемент в табл. 14.2 заслуживает отдельного рассмотрения. Одним из атрибутов, присущих компонентам, является атрибут `scope`, и он непосредственно поддерживается службами Grails.

Области видимости компонентов

Все компоненты платформы Spring по умолчанию являются единичными объектами, то есть в приложении присутствует только один экземпляр каждого компонента такого типа. Это делает любую информацию, хранящуюся в компоненте, такую как простое поле, эквивалентной глобальной переменной, что не всегда хорошо в среде, состоящей из нескольких потоков выполнения!

Одиночные объекты, не хранящие никакой информации, являются идеальными, но если вам потребуется сохранять в компоненте некоторую информацию, вы можете облегчить себе жизнь, реализовав управление видимостью совместно используемых данных. Допустим, что ваш компонент занимается обработкой запросов HTTP. Вы существенно обезопасили бы себя, если бы информация в компоненте была доступна только в пределах единственного запроса, особенно если учесть, что обработка запроса обычно производится единственным потоком выполнения. Для каждого запроса могла бы иметься отдельная копия данных, что позволило бы безопасно изменять эту информацию, не оказывая влияния на другие запросы.

Данная концепция известна как «область видимости» (scope), и вы можете изменять эту область видимости для служб, объявляя статическое поле:

```
class MyRequestService {
    static scope = "request"
    ...
}
```

Области видимости `singleton` и `request` являются не единственными доступными для вас. Ниже перечислены все области видимости с краткими описаниями:

- `singleton` – во всем приложении существует только один экземпляр компонента (по умолчанию)
- `prototype` – каждый раз, когда компонент извлекается из контекста приложения, создается новый экземпляр

- `request` – создается по одному экземпляру для каждого запроса HTTP
- `session` – существует по одному экземпляру для каждого сеанса HTTP
- `flow` – существует по одному экземпляру для каждого потока или подпотока (за дополнительной информацией обращайтесь к главе 9)
- `conversation` – существует по одному экземпляру для каждого диалога (за дополнительной информацией обращайтесь к главе 9)

Как видно в табл. 14.2, вы можете также указывать область видимости внутри определения компонента.

Теперь, когда вы знаете, как определять дополнительные компоненты в приложении, вам доступна вся мощь платформы Spring. Если вы обнаружите крутую библиотеку, которая позволяет интегрироваться с платформой Spring, вы будете знать, как использовать ее из приложения на платформе Grails. Здесь мы смогли лишь слегка прикоснуться к особенностям платформы Spring, поэтому для серьезного ее применения в разработке приложений стоит изучить документацию и книги, посвященные этой теме.

Имея серьезную подготовку, мы можем смело предстать перед еще одной особенностью платформы Spring, которая широко используется платформой Grails, но может вызывать недопонимание: транзакциями.

14.2. Использование транзакций в механизме GORM

В некоторых областях транзакции обеспечивают значительное преимущество. Корпорация Microsoft создала Microsoft Transaction Server (MTS), а затем разработала модель COM+ для упрощения разработки приложений, опирающихся на использование транзакций. Компоненты EJB и их серверы также разрабатывались с прицелом на транзакции. Так что же это такое? Начнем с определения:

Определение

Транзакция – это единица работы, в которой выполняются либо все операции, либо ни одной. Невозможно выполнить только часть транзакции.

Одним словом, либо все, либо ничего. В качестве практического примера рассмотрим транзакцию, которая изменяет несколько таблиц в базе данных. Если какое-либо из этих изменений терпит неудачу, не подтверждается ни одно из изменений, произведенных в базе данных. Транзакция завершается, только если все изменения прошли успешно. Поэтому транзакции являются гарантией сохранения непротиворечивости данных.

А в чем, собственно, проблема? Классическим примером является банковское программное обеспечение. Представьте, что клиент банка желает перевести деньги с одного счета на другой. На первый взгляд эта операция состоит из двух этапов: снять деньги с одного счета и положить их на другой. Но как быть, если первый этап был выполнен успешно, а на втором по какой-либо причине возникла ошибка? Без поддержки транзакций в приложении клиент просто потеряет свои деньги! Это большая проблема.

Давайте теперь без лишней суеты посмотрим, как использовать механизм транзакций в приложениях на платформе Grails.

14.2.1. Простота поддержки транзакций в службах

Чтобы исследовать свойства транзакций, мы добавим их поддержку в приложение Hubbub. Это надуманный пример использования транзакций, но нам надо на чем-то учиться.

Допустим, что пользователи могут отвечать на сообщения других пользователей, включая строку `@<имя пользователя>` в свои сообщения. Для нас было бы желательно иметь возможность сохранять такие ответы вместе с именами пользователей, которым они направлены, поэтому добавим еще один класс предметной области:

```
class Reply {
    Post post
    User inReplyTo
}
```

Затем нам необходимо обеспечить создание экземпляра класса `Reply` всякий раз, когда пользователь отправляет ответ. Кроме того, если приложение Hubbub не распознает имя пользователя, указанное в сообщении, оно не должно сохранять это сообщение в базе данных. Будучи разумным человеком, вы сначала проверили бы имя пользователя и только потом сохранили бы сообщение, но мы поступим иначе. Мы сначала сохраним сообщение и *только потом* проверим имя пользователя, потому что именно так мы сможем продемонстрировать действие транзакций.

Примечание

Чтобы получить корректное поведение приложения, вы должны использовать базу данных, поддерживающую транзакции, такую как PostgreSQL, H2 или MySQL с механизмом InnoDB. Базы данных, не поддерживающие транзакции, такие как MySQL с таблицами MyISAM, не будут работать как должно.

Куда мы добавим необходимый программный код? Пришло время вернуться к нашему старому доброму другу – к классу `PostService`. Улучшенная реализация, в которой мы создаем ответы, приводится в листинге 14.2.

Листинг 14.2. Служба сообщений, позволяющая оставлять ответы

```
package com.grailsinaction

class PostService {
    Post createPost(String userId, String content) {
        def user = User.findByUserId(userId)
        def post
        if (user) {
            post = new Post(content: content)
            user.addToPosts(post)

            if (!user.save()) {
                throw new PostException(
                    message: "Invalid or empty post", post: post)
            }

            def m = content =~ /@(\w+)/ ← Отыскивает «@...» в сообщении
            if (m) {
                def targetUser =                    ← Проверяет наличие
                    User.findByUserId(m[0][1])        указанного пользователя

                if (targetUser) {
                    new Reply(post: post, inReplyTo: targetUser).save()
                }
                else {
                    throw new PostException(
                        message: "Reply-to user not found", post: post)
                }
            }
        }
        throw new PostException(message: "Invalid User Id")
    }
}
```

Как можно заметить, теперь служба сообщений сохраняет новое сообщение, отыскивает, кому предназначен ответ, и затем проверяет наличие этого пользователя. Если поиск увенчался успехом, новый экземпляр класса `Reply` сохраняется; в противном случае возбуждается исключение. Теперь, когда служба готова к работе, что требуется сделать, чтобы добавить в нее поддержку транзакций? Ничего!

По умолчанию все общедоступные методы в службах уже поддерживают транзакции. Если вы попробуете отправить сообщение, содержащее строку `@dilbert` (предполагается, что пользователь `dilbert` не зарегистрирован в приложении Hubbub), вы не только увидите в броузере сообщение об исключении с трассировочной информацией, но и сможете убедиться, что сообщение не было сохранено. Почему бы вам не попробовать это у себя? Попытайтесь отправить следующие два сообщения:

- `@glen hi there mate!`
- `@dilbert do you really exist?`

Первое сообщение появится в вашем списке, а второе – нет, как мы и надеялись. Здесь возникает интересный вопрос: как платформа Grails узнает, что следует откатить (не подтверждать) транзакцию? Метод возбуждает исключение, благодаря этому платформа Grails (точнее платформа Spring) понимает, что транзакция потерпела неудачу.

За кулисами

Поддержка транзакций в службах реализована с использованием механизма аспектно-ориентированного программирования платформы Spring: каждая служба обертывается промежуточным компонентом `TransactionProxyFactoryBean` платформы Spring. Когда вы обращаетесь к службе, фактически вы обращаетесь к этому промежуточному компоненту.

Сами по себе транзакции управляются компонентом `transactionManager`, который по умолчанию является экземпляром класса `HibernateTransactionManager` платформы Spring. Вы можете переопределить этот класс в одном из файлов resources.*, если вам потребуется реализовать специализированный администратор транзакций или указать собственные настройки.

Прежде чем двигаться дальше, мы должны избавиться от вывода трассировочной информации. И пока мы еще здесь, было бы полезно организовать возможность убедиться, что ответ был сохранен. Изменим действие `add` контроллера сообщений, как показано ниже:

```
def add = {
    def content = params.postContent
    if (content) {
        def post = null
        try {
            post = postService.createPost(session.user.id, content)
        }
        catch (e) {
        }

        if (post) {
            flash.message = "Added new post"
        }
        else {
            flash.message = "Failed to add new post"
        }
    }
    redirect(action: "list")
}
```

Самое важное, на что здесь следует обратить внимание, – мы перехватываем исключение. Мы ничего не должны делать с ним, мы просто

сбрасываем его и возвращаем значение `null`. В действующей системе сообщение об ошибке должно быть более информативным, но для целей демонстрации достаточно и таких. Кроме того, для класса предметной области `Replay` необходимо добавить контроллер, использующий механизм скаффолдинга:

```
class ReplyController {
    def scaffold = true
}
```

Это позволит нам проверить, сохраняются ли ответы. Сделав это, можно двигаться дальше.

Типичная проблема, с которой многие сталкиваются при использовании поддержки транзакций в платформе Grails, связана с обработкой исключений. Чтобы понять, что мы имеем в виду, измените в службе `PostService` строку

```
throw new PostException(message: "Reply-to user not found", ...)
```

как показано ниже:

```
throw new Exception("Reply-to user not found")
```

Все, что мы сделали, — изменили тип исключения, но если теперь попытаться послать ответ, содержащий строку `@dilbert`, новое сообщение появится в списке! Почему это произошло?

Тестирование транзакций

Интеграционные тесты по умолчанию выполняются в рамках транзакции, которая откатывается после выполнения теста. Этим гарантируется, что изменения в данных не будут оказывать влияния на другие тесты, но это также означает, что вы не сможете проверить, выполняется ли откат транзакций. Если вам требуется протестировать действие транзакций, вы должны добавить в тест статическое свойство `transactional`:

```
static transactional = false
```

Альтернативный способ заключается в использовании функциональных тестов.

По умолчанию платформа Spring имитирует поведение контейнеров EJB: автоматический откат транзакций производится только для исключений времени выполнения и ошибок. Она предполагает, что контролируемые исключения вы будете обрабатывать самостоятельно. Поскольку Java вынуждает нас иметь дело с контролируемыми исключениями (или объявлять их в методе), это не такая большая проблема. С другой стороны, язык Groovy позволяет интерпретировать контроли-

руемые исключения так, как если бы они были исключениями времени выполнения, поэтому не сразу очевидно, почему не откатываются транзакции. Незнание этой проблемы может привести к потере большого количества времени.

А как быть, если вам не требуется, чтобы служба поддерживала транзакции? Тогда установите статическое свойство `transactional` в значение `false`:

```
class PostService {
    static transactional = false
    ...
}
```

Не забудьте отменить предыдущее изменение (метод службы должен возбуждать исключение `PostException`), затем запустите приложение и попробуйте отправить ответ со строкой `@dilbert` еще раз. Заметили какие-нибудь изменения? Теперь сообщение появилось в списке! Даже хотя `dilbert` не является именем пользователя, известным приложению (и возбуждается исключение времени выполнения), сообщение все равно сохраняется в базе данных. Служба больше не поддерживает транзакции.

Самое интересное, что если удалить блок `try...catch` из контроллера, вы обнаружите, что сообщения больше не сохраняются. Что происходит теперь? Чтобы понять это, вам нужно узнать, как взаимодействуют транзакции и механизм сеансов библиотеки Hibernate.

14.2.2. Транзакции, сеанс и я

Принцип действия библиотеки Hibernate может показаться странным для новичков. Когда вы поверх всего этого кладете транзакции, пора доставать аспирин. Отчасти проблема заключается в том, что сеанс Hibernate сам проявляет некоторые признаки поведения транзакций, в чем вы только что убедились. Если вы пройдете через код приложения с помощью отладчика, вы заметите, что вызов метода `save()` ничего не сохраняет в базе данных. Так это транзакция или все-таки нет?

Сеанс Hibernate кэширует все, что отправляется в базу данных. Этот механизм известен вам как *кэш первого уровня*. Когда вы производите сохранение экземпляра предметной области, вы фактически сохраняете его в кэше. Данные не попадут в базу данных, пока не будет выполнена операция синхронизации кэша и базы данных, называемая «выталкиванием из кэша». Весь этот процесс схематически изображен на рис. 14.3.

Вспомните пример в конце предыдущего подраздела: когда мы перестали перехватывать исключение времени исполнения, возбуждаемое службой, это исключение продолжило свое распространение и предотвратило выполнение операции синхронизации в сеансе. Если вы верне-

те блок `try...catch` на место, действие завершится, как обычно, позволив платформе Grails синхронизировать кэш сеанса и отправить новое сообщение в базу данных. Как вы уже, вероятно, поняли, платформа Grails откроет сеанс для вас еще до того, как будет вызвано действие контроллера, и синхронизирует его, если действие завершится, как обычно.

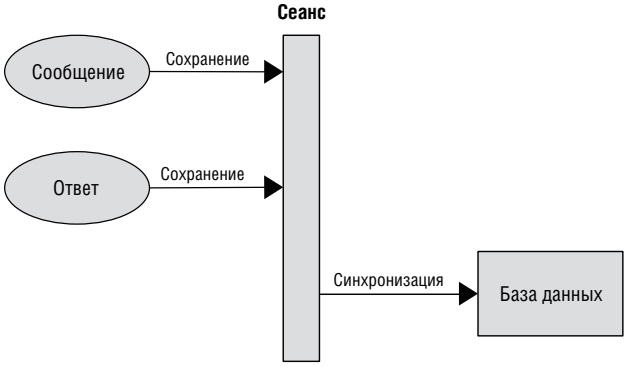


Рис. 14.3. *Сеанс Hibernate – это кэш в памяти, который сначала сохраняет изменения в данных локально, а затем переносит эти изменения в базу данных в процессе выполнения операции синхронизации*

За кулисами

Платформа Grails реализует собственную версию механизма перехвата веб-запросов платформы Spring, чтобы гарантировать, что сеанс будет открыт до того, как будет выполнено действие контроллера. В платформе Spring этот механизм реализован в виде класса `OpenSessionInViewInterceptor`, но платформа Grails немного модифицирует его, изменяя режим синхронизации сеанса на значение `manual` после того, как действие вернет управление, но перед тем, как будет отображено представление. Это дает возможность представлениям обращаться к коллекциям и отношениям с отложенной загрузкой, но препятствует выполнению ими каких-либо изменений, которые автоматически будут сохранены в базе данных.

В случае необходимости у вас есть возможность выполнить синхронизацию кэша вручную. Самый простой способ сделать это заключается в том, чтобы передать именованный аргумент `flush: true` либо методу `save()`, либо методу `delete()` класса предметной области. Все изменения, произведенные в сеансе до этого момента, будут сохранены в базе данных. Чтобы быстро проверить это, уберите из действия `add` контроллера конструкцию, перехватывающую исключение, и задействуйте аргумент `flush` в вызове метода `save()`, сохраняющего сообщение в службе:

```
...
post = new Post(content: content)
user.addToPosts(post)
user.save(flush: true)
```

Если теперь попытаться отправить ответ пользователю `dilbert`, сообщение будет сохранено в базе данных; достаточно лишь открыть страницу со списком сообщений, чтобы убедиться в этом.

Главное, что следует помнить, – сохранение изменений в базе данных делится на два этапа: сначала производится изменение объектов предметной области (или сохранение новых), а затем выполняется синхронизация кэша сеанса. Следующий вопрос таков: как в эту модель укладываются транзакции?

Когда создается сеанс Hibernate?

Всегда хорошо понимать, как действует сеанс Hibernate, но от этого мало проку, если не знать, когда он создается и как долго остается открытым. Ниже приводится краткий перечень наиболее типичных ситуаций:

- Сеанс открывается перед вызовом действия контроллера, а его кэш синхронизируется после завершения действия. Сеанс остается открытым до отображения представления или до тех пор, пока не будет отправлен ответ (например, с помощью метода `redirect()` или `error()`).
- Если сеанс еще не был открыт, транзакция открывает новый сеанс и сохраняет его открытым на протяжении всего своего времени жизни.
- Запросы и методы, такие как get(), автоматически создают сеанс, если он еще не был открыт. Сеанс закроется, как только метод вернет управление.

Включим снова поддержку транзакций в службе `PostService`, но оставим аргумент `flush: true`. Что произойдет теперь, если попытаться отправить ответ пользователю `dilbert`? Вы получите трассировочную информацию, как и прежде, но когда вы обновите страницу со списком сообщений, вы не увидите там этот ответ. Даже несмотря на то, что вы перехватываете исключение в контроллере, сообщение все равно не сохраняется в базе данных. Это уже начинает выглядеть, как черная магия. Разве платформа Grails не должна была сохранить сообщение в базе данных при явной синхронизации кэша? Нет, и если вы вернетесь к определению транзакции, вы поймете почему.

Любые изменения, произведенные в рамках транзакции, либо должны быть выполнены целиком, либо не должны быть выполнены вообще.

Даже явная синхронизация кэша не способна сохранить изменения, выполненные к этому моменту. Платформа Grails обеспечивает такое поведение, запуская транзакцию в *базе данных* в самом начале транзакции *приложения* (например, при входе в метод службы, поддерживающей транзакции). Когда платформа Grails синхронизирует кэш сеанса, изменения добавляются в транзакцию базы данных, но они не подтверждаются. Как только транзакция приложения завершается, платформа Grails автоматически синхронизирует кэш сеанса и подтверждает транзакцию базы данных, сохраняя изменения, выполненные к этому моменту. С отключенной поддержкой транзакций сеанс Hibernate действует в режиме автоматического подтверждения транзакций, что объясняет сохранение изменений в базе данных при синхронизации кэша.

Внимание

Когда транзакция закрывает сеанс, она подтверждает *все* изменения, связанные с этим сеансом, включая и те, что были выполнены перед запуском транзакции. Если вы предполагаете использовать транзакции, мы рекомендуем вам изменять или сохранять объекты предметной области только в пределах транзакций.

У вас может появиться вопрос: что произойдет, если запустить транзакцию внутри другой транзакции? Представим, что наша служба сообщений (с включенной поддержкой транзакций) вызывает метод *другой* службы, также поддерживающей транзакции. Запустит ли второй метод свою собственную транзакцию? Окажет ли это какое-либо влияние на первую транзакцию? Ответ на этот вопрос такой: вложенные транзакции объединяются с родительской транзакцией, поэтому в каждый момент времени существует только одна транзакция (родительская). Кэш сеанса Hibernate синхронизируется только по завершении родительской транзакции, а если родительская транзакция терпит неудачу, выполняется откат всех дочерних транзакций.

До сих пор мы рассматривали транзакции в контексте служб. Но как быть, если службы оказываются слишком мощным инструментом, или вам требуется более точное управление моментами запуска и остановки транзакций?

14.2.3. Точное управление транзакциями

Когда может возникнуть необходимость в более точном управлении моментом входа в транзакцию? Такая возможность может потребоваться при непосредственном изменении базы данных в контроллере, когда данное конкретное изменение должно производиться только в одном месте. Или вам может потребоваться выполнить множество транзакций внутри единственного метода службы.

Допустим, что мы решили перенести программный код создания сообщения обратно в контроллер, но при этом нам хотелось бы выполнять

эту операцию в рамках транзакции. Мы не рекомендуем использовать такой прием на практике, но мы применим его в целях демонстрации. Все, что мы должны сделать, – это вызвать статический метод `withTransaction()`, доступный во всех классах предметной области. Получившийся программный код приводится в листинге 14.3. Для простоты мы убрали некоторые проверки переменных.

Листинг 14.3. Использование метода withTransaction() для точного управления транзакциями

```
import org.springframework.transaction.TransactionStatus

class PostController {
    ...
    def add = {
        Post.withTransaction { TransactionStatus status ->  ❶ Запускает транзакцию
            def user = User.get(session.user.id)
            user.addToPosts(content: params.postContent)
            user.save()

            def m = content =~ /@(\w+)/
            if (m) {
                def targetUser = User.findByUserId(m[0][1])
                if (targetUser) {
                    new Reply(post: post, inReplyTo: targetUser).save()
                }
                else {
                    status.setRollbackOnly()  ❷ Откатывает изменения
                }
            }
        }
        ...
    }
    ...
}
```

Хотя мы и рекомендуем вам использовать службы с поддержкой транзакций, данный пример наглядно демонстрирует, как можно быстро добавить поддержку транзакций в блок программного кода. Основным здесь является вызов метода `withTransaction()` ❶, который принимает замыкание в качестве аргумента. Все, что находится внутри этого замыкания, выполняется внутри транзакции. Дополнительное преимущество такого подхода заключается в том, что вы получаете доступ к экземпляру класса `TransactionStatus` платформы Spring, с помощью которого можно вручную откатить транзакцию, не возбуждая исключение ❷.

Единственное, что может вас здесь озадачить, – это причина, по которой метод `withTransaction()` вызывается относительно класса предметной области `Post`, а не `User`. В действительности неважно, какой класс предметной области использовать – это не оказывает никакого влияния на работу программного кода в блоке. Вы могли бы использовать даже экзем-

пляр класса Tag, который вообще не используется в блоке транзакции. Не удивительно, что это обстоятельство часто смущает пользователей. Представляйте себе классы предметной области как нечто такое, к чему просто прикрепляется этот метод – своего рода владелец метода.

Так когда вместо служб с поддержкой транзакций следует использовать блоки транзакций? На этот вопрос трудно дать однозначный ответ; кроме того, он во многом зависит от конкретной ситуации. Службы с поддержкой транзакций – это рекомендуемый подход, они проще, чем блоки транзакций, и позволяют создавать компоненты многократного использования. Блок транзакции хорошо подходит для реализации одноразовых операций и в случаях, когда отсутствует открытый сеанс или нет возможности реализовать службу. Типичными примерами, где блоки транзакций имеют преимущество перед службами, могут служить класс BootStrap и интеграционные тесты.

Транзакции чрезвычайно важны для многих приложений, и вы видели, насколько платформа Grails упрощает их использование. Иногда может казаться, что они странно ведут себя, особенно в комбинации с сеансами Hibernate, но теперь, когда вы знаете, что творится за кулисами, вы способны решить любые проблемы, которые могут возникнуть.

14.3. В заключение

Как вы могли видеть, платформа Spring является фундаментальной частью платформы Grails, и хотя вы можете успешно разрабатывать простые приложения, даже не зная о ее существовании, вам потребуется воспользоваться ею при разработке более сложных приложений. К счастью, это легко сделать с помощью служб и механизма автоматического внедрения зависимостей, предоставляемого платформой Grails.

Когда вы начнете применять библиотеки Java и интегрировать свои приложения с другими системами на языке Java, вы часто будете сталкиваться с ситуациями, где службы будут не в состоянии помочь. Для успешной интеграции вам потребуется определить свои компоненты посредством файлов resources.xml и resources.groovy, чтобы их можно было легко добавлять в различные элементы платформы Grails. Если вам придется часто выполнять подобную интеграцию, мы рекомендуем вам поближе познакомиться с платформой Spring либо с помощью электронной документации, либо с помощью книг. Эта платформа содержит намного больше, чем набор классов, имеющих отношение к базовому контейнеру IoC.

Поддержка транзакций в платформе Grails – это еще один пример возможностей, предоставляемых платформой Spring, и вы имели возможность убедиться, насколько просто их использовать. Если вы пишете приложение, выполняющее изменения в базе данных, вы должны поближе познакомиться с транзакциями и эффектами, которые они оказывают на сохранение и изменение объектов предметной области.

Что же мы можем порекомендовать для ваших проектов?

- *Выносите логику работы в службы*. Мало того, что этот совет соответствует принципу разделения сфер влияния (Separation of Concerns) и позволяет получить программный код, который будет проще понять, сопровождать и тестировать, но он также поможет вам реализовать функциональные возможности, доступные для многократного использования, такие как шлюзы к веб-службам, интерфейсы удаленных взаимодействий и другие.
- *Очень важно оформлять единичные объекты в виде компонентов Spring*. Управление жизненным циклом и наличие механизма внедрения зависимостей подразумевают особую выгоду от превращения любых единичных объектов в компоненты Spring. Для определения компонентов на основе классов, находящихся в каталоге /src или в файлах JAR, используйте файлы resources.*.
- *Отдавайте предпочтение языку Spring DSL*. Независимо от того, являетесь ли вы поклонником XML или нет, преимущества определения компонентов Spring в виде программного кода неоспоримы благодаря поддержке условных конструкций, циклов и определений, характерных для того или иного окружения.
- *Выполняйте изменения в базе данных внутри транзакций*. Реализация внесения изменений в базу данных за пределами транзакции требует хорошего знания принципов действия сеансов Hibernate, в случае такой реализации вы легко можете нарушить непротиворечивость данных. Транзакции чрезвычайно просты в использовании, так почему бы не использовать их?

  Обратите внимание, что транзакции создают дополнительную нагрузку, поэтому при реализации массивных изменений вам может потребоваться использовать иное решение. Однако это слишком специфический случай, который едва ли коснется вас.

После учета этих рекомендаций в своем приложении вы обнаружите, что дальнейшее его развитие, тестирование и обслуживание стали намного проще, чем вы ожидали.

В этой главе:

- Упаковка приложения
- Расширение набора команд платформы Grails
- Сборка проектов на платформе Grails с использованием других инструментов

15

После компиляции, тестирования и запуска

До сих пор в этой книге мы рассматривали приложение на платформе Grails как нечто изолированное. Вы увидели, как пишется приложение, как оно тестируется и как запускается, и на этом все. Но что необходимо предпринять, чтобы сделать свое приложение доступным для своих клиентов или для всех желающих? Какие действия необходимо выполнить по оформлению дистрибутива приложения, чтобы другие могли установить его? А если приложение является лишь составной частью более крупного проекта, как быть в этом случае?

Все эти вопросы касаются общей картины разработки приложений. На рис. 15.1 выделены основные стадии цикла разработки, которые мы рассмотрим в этой главе.

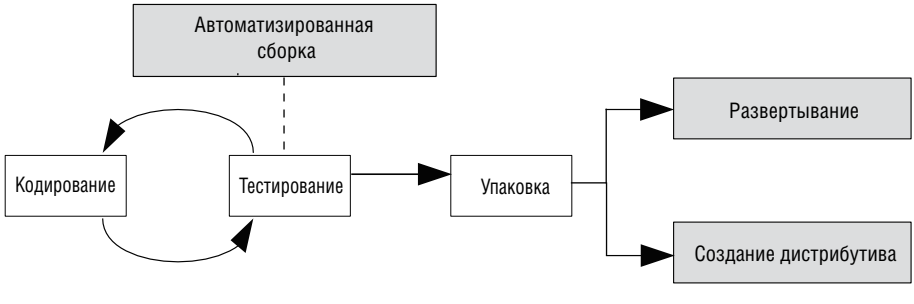


Рис. 15.1. *Типичный цикл разработки приложений на платформе Grails. Этапы программирования, тестирования и упаковки реализуются системой сборки Grails, остальные выполняются вручную или с помощью расширений*

История не заканчивается с «завершением» программирования: приложение еще необходимо сделать доступным для пользователей; и процесс выпуска приложения может состоять из нескольких шагов. Вне всяких сомнений, вы уже привыкли к системе сборки, которая автоматизирует компиляцию и тестирование приложения (интересно, когда в последний раз вы вызывали компилятор Java непосредственно?), но нередко вы можете автоматизировать и процедуру создания дистрибутива с помощью того же инструмента сборки. Такая автоматизация важна для снижения вероятности появления ошибок и улучшения воспроизводимости. К сожалению, людям свойственно совершать ошибки, особенно, когда они устают.

Возможно, вы и не знаете об этом, но платформа Grails включает в себя свою собственную систему сборки, которая охватывает весь спектр операций: компиляцию, тестирование, запуск и упаковку приложений. Каждый раз, когда вы запускаете одну из команд платформы Grails, вы инициируете систему сборки. Но система сборки не является чем-то неизменным: мы покажем вам, как можно расширить ее, чтобы обеспечить поддержку гипотетического процесса подготовки дистрибутива.

Автоматизированная сборка

Многие коммерческие и практически все открытые проекты используют методологию автоматизированной сборки (Continuous Integration, CI). Основная идея заключается в том, чтобы сервер постоянно проверял последние исходные тексты проекта, собирал его и производил тестирование, а при обнаружении ошибок на каком-либо этапе отправлял сообщения, чтобы разработчики могли быстро отыскать и исправить проблему.

Существует несколько продуктов, реализующих автоматизированную сборку, но ни один из них не имеет прямой поддержки приложений на платформе Grails. Поэтому приходится использовать сценарии командной оболочки, пакетные файлы, систему сборки Ant или нечто похожее, позволяющее выполнять соответствующие команды платформы Grails и сценарии. Какой способ использовать, зависит от продукта, с которым вы работаете, однако наиболее часто для реализации автоматизированной сборки используется инструмент Ant. Мы можем порекомендовать только один продукт – Hudson (*https://hudson.dev.java.net/*), который имеет расширение для сборки проектов на платформе Grails.

Затем мы рассмотрим ситуации, когда система сборки платформы Grails оказывается не самым лучшим выбором или даже вообще не может использоваться. Возможно, ваша компания требует использовать какой-то определенный инструмент сборки, такой как Maven, или, возможно, ваше приложение на платформе Grails является составной ча-

стью более крупного проекта и собирается в процессе сборки всего проекта. Каковы бы ни были причины, у вас может появиться потребность организовать сборку приложения на платформе Grails с помощью другого инструмента. Чтобы помочь вам в этом, мы обсудим доступные варианты. В частности, вы обнаружите, что сборка приложения на платформе Grails с помощью таких продуктов, как Apache Ant и Apache Maven, не только возможна, но и реализуется достаточно просто.

Но сначала рассмотрим систему сборки, имеющуюся в составе платформы Grails: как с ее помощью упаковать приложение, как она действует и как можно расширить и настроить ее.

15.1. Система сборки платформы Grails

В типичном проекте вы управляете различными фазами цикла разработки (изображенными на рис. 15.1), применяя для этого такой инструмент сборки, как Ant или Maven. Платформа Grails также могла бы использовать один из них и переложить на пользователя все хлопоты, связанные с установкой необходимого инструмента сборки, но это пошло бы вразрез с философией Grails, суть которой заключается в том, чтобы предоставить все необходимые инструменты «из коробки» (за исключением Java!). Поэтому она включает собственную систему сборки, с которой вы взаимодействуете с помощью командной строки.

Это не просто инструмент сборки. Вам не придется вручную настраивать файл сборки, потому что инструмент знает, как компилировать проект и как запускать тесты – все необходимые действия выполняются автоматически. Именно поэтому мы называем его *системой* сборки, а не *инструментом* сборки. Но платформе Grails необходима реализация некоторых команд, и для этого она использует специализированный инструмент, который может быть незнаком вам: Gant. Подробнее о нем мы поговорим в разделе 15.1.2.

Взгляните, как выглядит цикл разработки, изображенный на рис. 15.1, и вы заметите, что система сборки платформы Grails не предоставляет никаких инструментов для этапов создания дистрибутива и развертывания, поэтому вам придется засучить рукава и узнать, как расширить ее в этом направлении. Не волнуйтесь, мы сделаем из вас эксперта к концу этого раздела.

Но сначала рассмотрим фазу упаковки в цикле разработки, которой мы еще не касались в этой книге.

15.1.1. Упаковка приложения

Прежде чем мы сможем передать или развернуть приложение на платформе Grails, мы должны упаковать его в форму, пригодную для обеих ситуаций. Как вы уже, наверно, знаете, стандартным форматом пакетов веб-приложений в мире Java является WAR (сокращение от Web

Archive – веб-архив), который фактически является форматом JAR с четко определенной структурой файлов и каталогов. Файлы WAR могут копироваться в любой соответствующий стандартам контейнер сервлетов, такой как Apache Tomcat или Jetty, и приложения просто включатся в работу.

Развертывание приложения – это настолько фундаментальная часть цикла разработки, что в платформе Grails предусмотрена специальная команда для упаковки приложения:

```
grails war
```

Эта команда соберет все классы приложения, зависимости и ресурсы и поместит их в нужное место внутри архива WAR.

Почему бы не опробовать ее прямо сейчас? Предположим, что нам нужно развернуть приложение Hubbub в установленном нами экземпляре Tomcat. Команда `war` создаст в корневом каталоге проекта файл hubbub-0.1.war, который затем можно будет скопировать в каталог webapps сервера Tomcat. Запустите Tomcat и введите в броузере адрес URL приложения, обычно *http://localhost:8080/hubbub-0.1/*.

Внимание

Платформа Grails имеет также команду `package`, которую можно увидеть в списке, если запустить команду `grails help`. Но она не создает файлы WAR, и вообще не делает ничего полезного для конечного пользователя. Вероятно, со временем она будет объявлена устаревшей и удалена.

Нужен ли нам этот номер версии в адресе URL? Он будет доставлять только лишние неудобства при вводе адреса вручную. Существует два способа избавиться от номера версии, но самый простой состоит в том, чтобы изменить имя создаваемого файла WAR:

```
grails war hubbub.war
```

Нет никаких ограничений на имя создаваемого файла. А возможно ли создать файл WAR из приложения, находящегося в каталоге webapps сервера Tomcat? Передайте команде полный путь и имя файла, как показано ниже (где `$TOMCAT_HOME` – это переменная окружения, хранящая путь к корневому каталогу Tomcat):

```
grails war $TOMCAT_HOME/webapps/hubbub.war
```

По умолчанию файл WAR будет создан для действующего окружения, но при желании можно изменить тип окружения. Например, чтобы создать пакет приложения с настройками для окружения разработки, запустите следующую команду:

```
grails dev war
```

Из других доступных окружений можно указать `test` и `prod` (от «production» – «готовый», или «действующий»).

За кулисами

Вас может заинтересовать вопрос, как приложение, упакованное в файл WAR, узнает, в каком окружении оно выполняется. Обычно данная информация определяется такими командами, как `test-app` и `run-app`. Но в контейнере сервлета это невозможно. Поэтому платформа Grails добавляет информацию об окружении в файл application.properties, который затем упаковывается в архив WAR. После развертывания приложение прочитает настройки окружения из этого файла.

Мы еще не достигли конца повествования об упаковке приложений, но той информации, которую мы уже предоставили, более чем достаточно для большинства ситуаций. На случай, если вам потребуется более полное управление процессом создания файла WAR, платформа Grails предлагает несколько параметров настройки. Допустим для примера, что приложение требует иную версию библиотеки, чем та, что поставляется в составе платформы Grails, такую как последняя и лучшая версия библиотеки Hibernate. В идеале вы могли бы поместить необходимую версию библиотеки в каталог lib приложения, и все было бы замечательно, но в действительности сгенерированный файл WAR будет содержать обе версии библиотеки Hibernate, что, с большой вероятностью, вызовет проблемы. Как можно исключить версию платформы Grails из файла WAR?

Прежде чем будет создан сам архив WAR, все файлы, входящие в него, копируются во временный каталог, называемый «подготовительной площадкой» (staging area). Мы можем удалить ненужный файл JAR из такого временного каталога до того, как он будет заархивирован в файл WAR – существует специальный метод, который позволит нам сделать это. Создайте файл grails-app/conf/BuildConfig.groovy и добавьте в него следующие строки:

```
grails.war.resources = { String stagingDir ->
    delete(file: "${stagingDir}/WEB-INF/lib/hibernate3-3.2.6.jar")
}
```

Указанное замыкание будет вызвано непосредственно перед началом создания файла WAR, и, как можно видеть из примера, платформа Grails передает ему местоположение временного каталога. Делегатом является экземпляр класса `AntBuilder` языка Groovy, о котором подробнее рассказывается ниже, а пока просто поверьте, что строка в замыкании вызывает выполнение задачи `delete` инструмента сборки Ant и удаляет архив JAR с библиотекой Hibernate из временного каталога.

В файле BuildConfig.groovy используется тот же синтаксис, что и в файле Config.groovy, который вы уже использовали, но тогда как последний из них содержит информацию о настройках для запущенного при-

ложения, файл BuildConfig.groovy используется во время его сборки. Вмешательство в процесс создания файла WAR является функцией не приложения, а системы сборки, поэтому параметр `grails.war.resources` находится там, где он находится.

Другие доступные вам параметры настройки (все они должны находиться в файле BuildConfig.groovy) перечислены в табл. 15.1, но так как они имеют весьма специфическое назначение, мы не будем подробно останавливаться на них. Вам достаточно знать об их существовании – вам едва ли придется столкнуться с ситуациями, где они могут оказаться полезными.

Получив файл WAR, можно подумать об этапах создания дистрибутива и развертывания приложения. Какой из этих этапов вы выберете, зависит от вашей ситуации, но вам не стоит волноваться – мы опишем оба. Вопрос в том, что делать дальше. Похоже, что мы достигли конечной остановки, дальше которой команды платформы Grails не идут.

Таблица 15.1. Параметры настройки, влияющие на процесс создания файла WAR

Параметр настройки	Тип	Описание
`grails.war.destFile`	Строка	Указывает путь к создаваемому файлу WAR, включая имя самого файла. Значение этого параметра переопределяется явным аргументом команды `grails war`.
`grails.war.copyToWebApp`	Замыкание	Все наборы файлов с применением синтаксиса Ant, описанные в этом замыкании, копируются в корень временного каталога сборки. Этот параметр идеально подходит для выбора файлов из каталога web-app, которые должны включаться в файл архива WAR. Делегатом замыкания является AntBuilder; ему передается аргумент команды `grails war`, если таковой имеется.
`grails.war.dependencies`	Замыкание или список	Если этому параметру передается замыкание, оно определяет наборы файлов с применением синтаксиса Ant, которые должны быть скопированы в каталог WEB-INF/lib. Его делегатом является AntBuilder, и он не имеет аргументов. Если параметру передается список, в него включаются шаблоны Ant файлов, определяющие, какие файлы JAR должны копироваться из каталога lib платформы Grails.

grails.war.resources	Замыкание	Это замыкание, принимающее один или два аргумента, позволяет выполнить последние подготовительные операции перед созданием файла WAR. Первый аргумент – это путь к временному каталогу сборки, а необязательный второй аргумент – это аргумент команды grails war. Делегатом замыкания является AntBuilder.

15.1.2. Действуем самостоятельно: как создать команду dist

Сколько бы вы ни искали, вы не найдете команду платформы Grails, выходящую за стадию упаковки. Система сборки оставляет вас наедине с самим собой, как только файл WAR готов. Справедливости ради следует отметить, что как только речь заходит о создании дистрибутива и развертывании приложения, требования пользователей начинают существенно разниться, поэтому платформе Grails было бы сложно обеспечить что-либо стоящее в этой области.

Многие скажут, что копирование файла WAR в контейнер сервлета не самая сложная задача, и все же любой, кто имеет опыт сборки и развертывания приложений, знает, что даже здесь могут появляться ошибки. Да и само развертывание приложения не такая простая операция. То же относится к сборке и передаче дистрибутивов в общее пользование. В обоих случаях автоматизация в состоянии оказать существенную помощь заинтересованному лицу.

Идеальное решение заключается в том, чтобы расширить систему сборки платформы Grails и включить в нее команды, автоматизирующие указанные процессы. Сделаем это.

Делаем шаг вперед: Gant

Каждая команда платформы Grails реализована в виде сценария на языке Groovy. Однако это необычные сценарии: в них используются синтаксические конструкции, которые могут интерпретироваться специальным инструментом сборки Gant.

Заглянув в каталог $GRAILS_HOME/scripts, можно увидеть множество таких сценариев, имена большинства из них будут вам знакомы (они соответствуют командам ядра платформы Grails). Вы также можете создавать свои собственные сценарии в нескольких каталогах (где $HOME – это ваш домашний каталог, а <app> – это каталог приложения):

- $HOME/.grails/scripts
- <app>/scripts
- plugins

Любые сценарии в вашем домашнем каталоге превращаются в постоянно доступные команды Grails. Вы можете использовать их из любых проектов или вообще не использовать их. В свою очередь, сценарии в каталоге приложения доступны, только когда платформа Grails запускается из каталога приложения. В следующей главе, где рассказывается об особенностях разработки расширений, вы узнаете, как можно увеличивать набор команд платформы Grails с помощью расширений.

Что такое Gant? В прошлом Ant был самым распространенным инструментом, использовавшимся для сборки проектов на языке Java. Но времена изменились, и появились новые инструменты, потеснившие Ant. Некоторые из них основаны на языке Groovy, и одним из таких инструментов является Gant. Согласно утверждениям его автора Gant главным образом является средством описания задач Ant, однако в нем присутствуют некоторые атрибуты полноценного инструмента сборки, такие как задания и зависимости.

Поскольку система сборки платформы Grails использует инструмент Gant, вам придется научиться писать сценарии Gant, чтобы реализовать дополнительные команды. Вместо того чтобы описывать все кровавые подробности создания таких сценариев (на самом деле не такие уж и кровавые), мы покажем, как написать сценарий, создающий пакет дистрибутива для приложения Hubbub, и тем самым убьем двух зайцев одним выстрелом. В конце этого процесса у вас появится совершенно новая команда (`grails dist`).

Сначала разберемся с тем, чего мы ждем от команды. Не у всех пользователей имеется под рукой контейнер сервлетов, поэтому будет совсем не лишним включить в дистрибутив Jetty, а также некоторые сценарии для запуска приложения Hubbub внутри него. Мы добавим также в дистрибутив проектную документацию для полноты. Все это должно быть упаковано в виде архива zip, который можно будет выложить на веб-сайте или переслать по электронной почте. Ниже показано, как будет выглядеть содержимое итогового файла zip:

```
+-- bin
|   |
|   +-- startHubbub
|   +-- startHubbub.bat
|
+-- lib
|   |
|   +-- jetty-6.1.12.jar
|
+-- docs
|   |
|   +-- ...
|
+-- hubbub.war
```

Выяснив объем работ, можно начинать писать сценарий, который выполнит все необходимые действия. Можно смело утверждать, что описываемая здесь процедура создания дистрибутива достаточно универсальна и может применяться к любым приложениям на платформе Grails, но исключительно в целях демонстрации мы создадим сценарий в самом проекте.

Дистрибутивы и источники данных

Самой большой проблемой при создании дистрибутивов приложений на платформе Grails является настройка источников данных. Настройки соединения с базой данных вкомпилированы в класс `DataSource`, поэтому конечный пользователь не может изменить их!

Одно из решений состоит в том, чтобы указать в настройках источника данных имя JNDI. Пользователь сможет тогда связать настройки соединения с базой данных с этим именем.

Другой вариант – объявить в файле Config.groovy набор внешних файлов свойств, из которых платформа Grails будет загружать информацию о настройках:

```
grails.config.locations = ["file:.../custom.properties"]
```

За дополнительной информацией об этом параметре обращайтесь к руководству пользователя платформы Grails.

Файл должен иметь имя Dist.groovy. Почему? Потому что платформа Grails определяет имя команды исходя из имени файла сценария, то есть файл Dist.groovy станет командой `grails dist`. Точнее говоря, имя команды формируется из имени файла, в котором все заглавные символы, образующие начала отдельных «слов», преобразуются в строчные и слова отделяются символами дефиса. Например, имени файла CreateDomainClass.groovy соответствует команда `create-domain-class`. В случае сомнений создайте сценарий и выполните команду `grails help`, чтобы увидеть, что платформа Grails думает об имени соответствующей команды.

Эта команда не просто покажет вам, какое влияние оказывает имя файла сценария: с ее помощью вы сможете определить, доступен ли сценарий как команда и можно ли его использовать за пределами проекта. Добавьте символ подчеркивания (_) в начало имени файла, и платформа Grails будет интерпретировать его как *внутренний* сценарий – у вас не будет команды для запуска этого сценария. Добавьте подчеркивание в конец имени файла, и сценарий сможет быть запущен за пределами проекта на платформе Grails как команда `create-app`. Символ подчеркивания не становится частью имени команды.

Сохранив файл Dist.groovy, вы сможете выполнить команду `dist`, даже если вы оставите файл пустым! Он ничего не будет делать, но мы вскоре это исправим. В листинге 15.1 приводится программный код нашего нового сценария.

Листинг 15.1. Сценарий dist

```
includeTargets << grailsScript("_GrailsWar")
includeTargets << grailsScript("_GrailsDocs")

target(createDist: "Creates Hubbub distribution") {
    depends(docs, war)

    zipFile = "${basedir}/hubbub.zip"

    createZip()
}

target(createZip: "Packages the required files into a zip") {
    def warFile = new File(warName)

    ant.zip(destfile: zipFile) {
        zipfileset(dir: "${basedir}/src/templates/bin", prefix: "bin")
        zipfileset(dir: "${grailsHome}/lib", prefix: "lib") {
            include(name: "jetty*.jar")
        }
        zipfileset(dir: "${basedir}/docs", prefix: "docs")
        fileset(file: warFile)
    }
}

setDefaultTarget("createDist")
```

❶ Загрузка сценариев платформы Grails

❷ Указывает зависимости для задания

❸ Явный вызов задания

❹ Вызывается задача zip системы Ant

❺ Задание сценария по умолчанию

Нам необходимо убедиться, что файл документации и архив WAR будут созданы до того, как мы попытаемся создать дистрибутив, включающий их. Нет смысла делать это своими руками, так как существуют соответствующие команды. Наличие команд подразумевает наличие соответствующих сценариев, благодаря чему мы можем их использовать. Мы подключаем задания из интересующих нас сценариев ❶, и эти задания тотчас же становятся доступными для нас.

Синтаксическая конструкция `includeTargets <<` является типичной для инструмента Gant, но метод `grailsScript()` характерен только для сценариев Grails и должен использоваться только для включения сценариев, предоставляемых самой платформой Grails. Вы можете подключать задания из своих собственных сценариев (или из сценариев, предоставляемых расширениями), но при этом вам необходимо указывать справа имя файла, как показано ниже:

```
includeTargets << new File("${basedir}/scripts/Other.groovy")
```

Обратите внимание, что вы должны указать полный путь к файлу сценария, а также расширение .groovy. В методе `grailsScript()` расширение *не* должно указываться.

Задание при использовании инструмента Gant (соответствует понятию задания Ant) определяется с помощью метода `target()`, вызов которого в общем представлении выглядит, как показано ниже:

```
target(<name> : <description>) {
    ...
}
```

где `<name>` – строка, представляющая имя задания, а `<description>` – краткое описание, что это задание делает (которое также отображается командой `grails help`). Реализация задания, как вы уже могли предположить, находится в фигурных скобках. В данном случае задание называется `createDist`.

Сценарий и локальные переменные

Помимо локальных переменных, полей класса и аргументов методов в сценариях на языке Groovy имеется еще один тип переменных – *переменные сценария*. Они создаются всякий раз, когда выполняется инициализация переменных, которые не были объявлены прежде. Так, инструкция

```
def var = "Test"
```

создаст локальную переменную, а инструкция

```
var = "Test"
```

создаст переменную сценария, если она не была объявлена в другом месте. Самая важная особенность переменных сценария заключается в том, что они являются глобальными. Если локальная переменная подчиняется обычным правилам области видимости, то переменная сценария будет доступна после ее создания в любой точке сценария независимо от того, в какой области видимости она была создана.

И последнее замечание: переменные сценария являются глобальными не только внутри сценария, при использовании инструмента Gant они будут доступны также всем подключенным сценариям. Данное правило справедливо и в обратном направлении: переменные сценария, созданные в подключаемых сценариях, доступны и подключающему сценарию. Это единственный способ передачи данных между заданиями, но вы должны знать, что глобальное пространство имен легко можно «засорить» такими переменными и тем самым увеличить вероятность конфликта имен.

В теле задания с помощью метода `depends()` можно указать, что предварительно необходимо выполнить другие задания ❷. Обратите внимание, что вы не обязаны заключать в кавычки имена заданий в вызове `depends()`, однако вы можете добавить их, если пожелаете. В нашем сценарии эта одна строка гарантирует, что файлы docs и war будут созданы до того, как начнется создание файла zip.

После объявления зависимостей мы инициализируем переменную сценария `zipFile`, записывая в нее местоположение и имя файла zip дистрибутива. В некоторой степени это излишне, потому что мы легко могли бы указать путь в задании `createZip`, но мы хотели показать, как можно передавать информацию из одного задания в другое. Кроме всего прочего, такой подход подразумевает, что другой сценарий или задание могли бы вызвать `createZip` и указать иной путь к файлу.

Мы почти закончили разбираться со сценарием, осталось рассказать лишь еще о нескольких элементах. Вы можете не только объявлять зависимости, используя метод `depends()`, но и вызывать задания непосредственно, как если бы они были простыми методами ❸. Одно из преимуществ такого приема заключается в том, что он позволяет точно определять момент выполнения требуемого задания. С другой стороны, использование метода `depends()` гарантирует, что в процессе сборки задание будет выполнено всего один раз, даже если от него зависят несколько других заданий.

Внутри нашего второго задания мы извлекаем имя созданного файла WAR из переменной сценария `warName` и, используя задачу `zip` системы Ant, создаем архив с дистрибутивом ❹. Имя и путь к файлу zip определяются из переменной сценария `zipFile`, значение которой мы указали в теле главного задания. Как определяется местоположение файла WAR? Значение переменной `warName` устанавливается заданием `war` в сценарии _GrailsWar. Обратите также внимание на свойство `ant`: эта неявная переменная предоставляется инструментом Gant и является экземпляром класса `AntBuilder` языка Groovy. Любая задача Ant может быть выполнена вызовом метода с тем же именем относительно объекта `ant`; например вызов `ant.echo()` запустит задачу `echo`. Каждый именованный аргумент отображается в атрибут задачи, а внутри замыкания можно добавлять вложенные элементы. Например, эквивалентом конструкции ❹ является следующий фрагмент XML в инструменте Ant:

```
<zip destfile="${zipFile}">
    <zipfileset dir="${basedir}/src/templates/bin" prefix="bin"/>
    <zipfileset dir="${grailsHome}/lib" prefix="lib">
        <include name="jetty*.jar"/>
    </zipfileset>
    <zipfileset dir="${basedir}/docs" prefix="docs"/>
    <fileset file="${warName}"/>
</zip>
```

В действительности соответствие между форматом XML и программным кодом Groovy определяется обычными правилами, согласно которым элементы XML отображаются в вызовы методов, атрибуты – в именованные аргументы, а вложенные элементы – в замыкания.

В конце сценария ❺ мы определяем задание по умолчанию, которое будет вызываться платформой Grails при выполнении соответствующей команды. В данном случае при вызове команды `grails dist` платформа Grails будет вызывать задание `createDist`. Вызов метода `setDefaultTarget()` должен производиться *после* того, как будет определено указанное в нем задание; в противном случае будет выведено сообщение об ошибке.

Уф! Слишком много подробностей для одного раза. Ключевыми особенностями, о которых следует помнить, являются задания, зависимости, включения и класс `AntBuilder`. Немного практики, и вы научитесь пользоваться всем этим; кроме того, существует множество открытых расширений, которые могут служить источником идей. Что касается класса `AntBuilder`, теперь, когда вы знаете, как отображается синтаксис XML системы Ant в конструкции языка Groovy, лучшим источником информации для вас будет справочное руководство для системы сборки Ant, которое можно найти по адресу: *http://ant.apache.org/manual/*[1].

А как быть с существующими заданиями? Как узнать, какие сценарии платформы Grails следует подключать и какие задания вызывать? Это зависит от того, в каком сейчас состоянии находится руководство пользователя платформы Grails. Сначала вы можете ознакомиться со списком наиболее полезных сценариев и с их главными заданиями в табл. 15.2.

Таблица 15.2. Некоторые полезные сценарии платформы Grails и их задания

Сценарий	Задания	Описание
`_GrailsSettings`	Нет	Устанавливает значения некоторых полезных переменных и выполняет другие важные настройки, но не имеет заданий. Если вы не подключаете никакие другие сценарии, подключите этот – практически все сценарии платформы Grails подключают его.
`_GrailsArgParsing`	`parseArguments`	Анализирует аргументы командной строки, переданные сценарию. Эти аргументы доступны сценарию в виде переменной `argsMap`.

[1] Неплохое учебное руководство на русском языке можно найти по адресу: *http://www.ibm.com/developerworks/ru/edu/j-apant/. – Прим. перев.*

Таблица 15.2 (продолжение)

Сценарий	Задания	Описание
_GrailsClean	cleanAll, clean	Действия, выполняемые этим сценарием, явствуют из его имени. Задание cleanAll удаляет отчеты, полученные в процессе тестирования, а задание clean не удаляет их.
_GrailsCompile	compile	Компилирует исходный программный код приложения.
_GrailsTest	compileTests, testApp	Компилирует и запускает тесты.
_GrailsRun	runApp, runWar	Запускает приложение.
_GrailsWar	war	Создает файл архива WAR с приложением.

По мере накопления опыта в создании сценариев вы поймете, что самый лучший способ узнать, какие сценарии и задания доступны, состоит в том, чтобы обратиться к первоисточнику: к самим сценариям платформы Grails! Если у вас платформа Grails установлена локально, вы найдете эти сценарии в каталоге $GRAILS_HOME/scripts в распакованном и готовом к просмотру виде.

Одним из сценариев, заслуживающих особого упоминания, является _GrailsEvents, в котором находится реализация механизма событий для системы сборки.

События системы сборки

Платформа Grails предоставляет сценариям механизм событий, позволяющий запускать и реагировать на именованные события. Например, этап компиляции платформы Grails возбуждает одно событие до того, как все файлы с исходными текстами будут скомпилированы, и другое – после. Каждое задание возбуждает событие начала и конца, как изображено на рис. 15.2.

Чтобы возбудить событие вручную, ваш сценарий должен подключить сценарий _GrailsEvents (прямо или косвенно через другой сценарий) и затем вызвать метод event():

```
event("DistCreated", [zipFile])
```

Первый аргумент – это имя события, а второй – список параметров, которые передаются приемникам событий. Поскольку данный вызов фактически является определением конкретного события, вы должны описать где-нибудь, что означает это событие и какие параметры передаются вместе с ним. Это упростит интеграцию ваших сценариев для разработчиков приложений и авторов расширений.

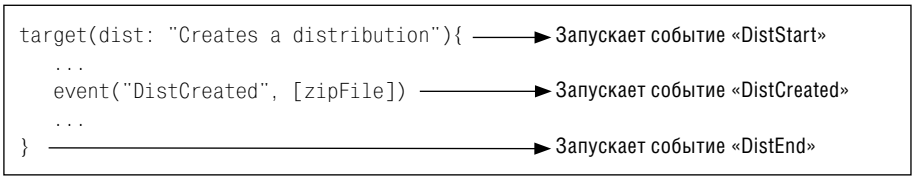


Рис. 15.2. *Задания в системе сборки платформы Grails автоматически возбуждают события начала и конца, а метод event() позволяет возбуждать события вручную*

От событий мало проку, если программный код не сможет получить и отреагировать на них, поэтому давайте рассмотрим, как реализовать прием событий. Сначала необходимо подключить файл с именем _Events.groovy, находящийся в каталоге scripts. Затем необходимо определить обработчики событий, создав переменные сценария с соответствующими именами и замыканиями в качестве значений:

```
eventDistCreated = { String zip ->
    println "Created distribution zip ${zip)"
}
```

Имя переменной сценария – это имя события с префиксом «event». К сожалению, события, такие как это, обычно плохо документируются, поэтому нередко приходится самостоятельно просматривать сценарии, чтобы узнать количество параметров, передаваемых вместе с событием, и что это за параметры. Мы надеемся, что эта ситуация улучшится в будущем, а пока, чтобы помочь вам, мы перечислили несколько полезных событий в табл. 15.3. Обратите внимание, что аргументы события передаются методу `event()` в виде списка, тем не менее замыкание обработчика события получает их в виде отдельных аргументов.

Таблица 15.3. Некоторые полезные сценарии событий

Имя сценария	Аргументы	Описание
`StatusUpdate`	Сообщение о состоянии (строка)	Извещает приемник о текущем состоянии процесса сборки. В основном используется для нужд журналирования.
`CompileStart`, `CompileEnd`	Строка	Возбуждаются до и после компиляции исходных текстов приложения. Аргумент указывает – что именно компилируется. Например, значение `source` свидетельствует о том, что компилируются исходные тексты приложения, а значение `tests` – о том, что компилируются испытательные тесты.

Таблица 15.3 (продолжение)

Имя сценария	Аргументы	Описание
`CreateWarStart`, `CreateWarEnd`	Два аргумента: имя файла WAR (включая путь) и местоположение временного каталога сборки. Оба аргумента являются строками.	Возбуждаются до и после создания файла WAR приложения.
`CleanStart`, `CleanEnd`	Нет аргументов	Возбуждаются до и после очистки проекта.

Возможно, сценарии не самая захватывающая часть платформы Grails, но они настоящие рабочие лошадки и способны облегчить жизнь пользователям. Если вы имеете опыт работы с Ant, тогда у вас не должно возникать проблем со сценариями, и мы уверены, что вам понравится свобода, которую дает настоящий язык программирования – вне всяких сомнений, в сценариях Gant гораздо проще реализовать условное ветвление и циклы, чем в Ant.

Если вы не знакомы с инструментом Ant или не разбираетесь в его сценариях, в этом нет ничего плохого. Платформа Grails уже предоставляет вполне исчерпывающий набор команд, поэтому у вас может не быть никаких причин создавать дополнительные. Но, прежде чем вы выбросите их из головы, мы приведем еще один пример, который может изменить ваше мнение: автоматическое развертывание приложения на платформе Grails в Tomcat.

15.1.3. Развертывание

Создание дистрибутива приложения Hubbub помогло нам взглянуть на сценарии Gant с одной стороны: от нас не требовалось думать о чем-либо другом, кроме того, что мы хотели включить в архив zip.

Теперь пришло время посмотреть на фазу развертывания, последний прямоугольник на рис. 15.1. Развертывание веб-приложения непосредственно в контейнер сервлетов или на сервере приложений Java – гораздо более типичная процедура, чем создание дистрибутива, поэтому оставайтесь с нами!

Автоматизация развертывания – хорошая идея, потому что она уменьшает вероятность ошибки и степень напряженности человека, выполняющего развертывание. К сожалению, в этой области также невозможно предложить решение «на все случаи жизни», поэтому все сценарии, которые мы будем рассматривать в качестве примеров, могут использоваться только для конкретного приложения (если не внести в них необходимые изменения). Тем не менее сценарий, автоматизирующий процедуру развертывания приложения в сервере Tomcat, пригодится многим и послужит поучительным примером.

Автоматическое развертывание в Tomcat 5.5

Когда вы загружаете сервер Apache Tomcat, вы получаете не только контейнер сервлетов. Вы дополнительно получаете веб-приложение Tomcat Manager, которое обслуживает удаленное развертывание пользовательских приложений в контейнере. Нам осталось только узнать, как получить доступ к веб-приложению и затем пользоваться его прикладным интерфейсом.

Прежде всего необходимо получить доступ к веб-приложению Tomcat Manager. Несмотря на то, что оно устанавливается вместе с сервером Tomcat и готово к использованию, во время установки не создаются учетные записи пользователей, имеющих право доступа к нему. Поэтому сначала необходимо настроить учетную запись, под которой мы *сможем* использовать приложение.

Откройте в редакторе файл <tomcat>/conf/tomcat-users.xml (где <tomcat> – каталог установки сервера Tomcat) и добавьте следующую строку:

```
<tomcat-users>
    ...
    <user username="overlord" password="Kr@kat04" roles="manager"/>
</tomcat>
```

Вообще вы должны стремиться выбирать необычное имя пользователя и как можно более замысловатый пароль, потому что это приложение дает в руки пользователя большие возможности. Любой пользователь сервера Tomcat, которому присвоена роль `manager`, обладает правом доступа к веб-приложению Tomcat Manager.

Проверить настройки можно, запустив сервер Tomcat (обычно это делается запуском сценария startup.sh или пакетного файла startup.bat в операционной системе Windows) и открыв в броузере страницу *http://localhost:8080/manager/html*. После ввода имени пользователя и пароля вы должны увидеть страницу, как изображенную на рис. 15.3.

Вы можете поиграть немного с веб-приложением, но это не совсем то, что нас интересует в данный момент. Вспомните, мы хотели автоматизировать развертывание своего веб-приложения с помощью сценария; значит, нам не придется взаимодействовать с интерактивным пользовательским интерфейсом. К счастью, мы можем посылать запросы `GET` и `POST` по определенным адресам URL приложения для выполнения операций установки (`deploy`), удаления (`undeploy`), запуска (`start`) или остановки (`stop`) приложения. Более того, эти операции можно делать даже из броузера. Например, если открыть страницу *http://localhost:8080/manager/deploy?war=file:/tmp/hubbub.war*, приложение Tomcat Manager попытается развернуть файл /tmp/hubbub.war.

Теперь, когда вы получили знания о том, как пользоваться приложением Tomcat Manager, можно попробовать применить их на практике и написать сценарий развертывания приложения Hubbub на сервере Tomcat.

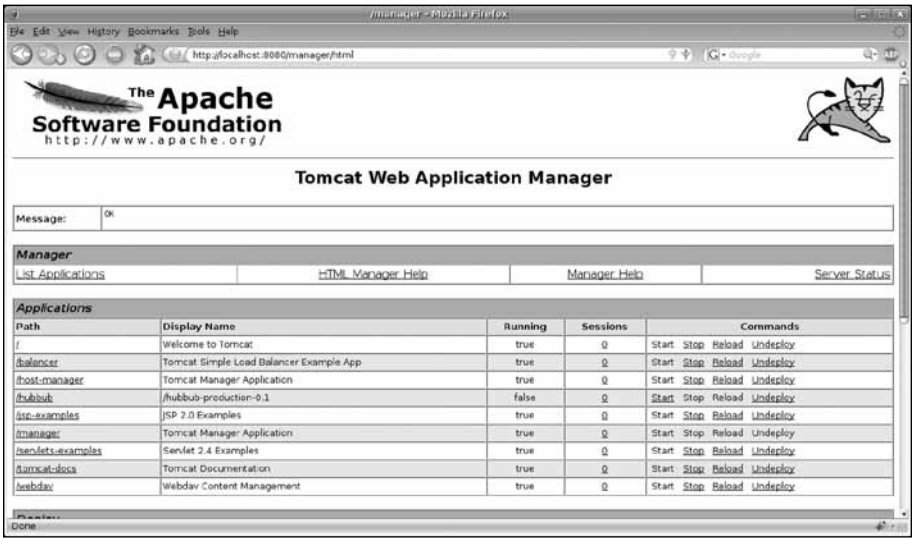


Рис. 15.3. *Приложение Tomcat Manager*

Разверни его, крошка!

Что должен делать этот сценарий? Как и прежде, когда мы писали сценарий Dist, необходимо убедиться в наличии файла WAR. Если файл присутствует, мы должны послать запрос `GET` приложению Tomcat Manager и передать в составе адреса URL местоположение файла WAR. Все просто! Программный код сценария приводится в листинге 15.2.

Листинг 15.2. Сценарий tomcat-deploy

```
scriptEnv = "production"  ← ❶ Устанавливает окружение по умолчанию

includeTargets << grailsScript("_GrailsWar")

target(deploy: "Deploys Hubbub's WAR to Tomcat") {
    depends(war)

    def username = buildConfig.tomcat.mgr.username  ← ❷ Чтение данных из
    def password = buildConfig.tomcat.mgr.password      файла с настройками

    Authenticator.setDefault([  ← Выполняет аутентификацию пользователя
        getPasswordAuthentication: {->      в приложении Tomcat Manager
            new PasswordAuthentication(username, password)
        }
    ] as Authenticator)

    def url =
        "http://localhost:8080/manager/deploy?war="  ← Сборка адреса URL
    url += warName                                       команды deploy
    url += "&path=/hubbub&update=true"
```

```
    def response = new URL(url).text
    if (response.startsWith("OK")) {
        println "Application deployed successfully!"
        return 0 ← ❸ Код возврата
    }
    else {
        println "Application failed to deploy: $response"
        return 1
    }
}

setDefaultTarget("deploy")
```

Данный сценарий создает файл WAR (эта операция возлагается на задание `war`), отправляет запрос `GET` приложению Tomcat Manager и читает ответ. Здесь имеется несколько новых особенностей, первая из которых – переменная `scriptEnv` ❶.

При запуске какой-либо команды Grails вы обычно не указываете окружение явно – платформа Grails по умолчанию использует наиболее подходящее значение. Так, команда `grails run-app` по умолчанию выбирает окружение разработки, а команда `grails war` по умолчанию устанавливает рабочее окружение. В случае нестандартных сценариев, платформа Grails не знает, каким должно быть значение по умолчанию, поэтому она использует окружение разработки. Однако это не совсем то, что нам требуется в сценарии развертывания. Нам необходимо, чтобы по умолчанию устанавливалось действующее окружение, как и в случае команды `grails war`, именно это и делается в строке ❶.

Следует сделать два важных замечания о переменной `scriptEnv`. Во-первых, в качестве значения этой переменной должно использоваться полное название окружения, а не короткое имя, используемое в командной строке, то есть вместо `prod` следует использовать `production`. Во-вторых, эту переменную *необходимо* инициализировать перед подключением заданий из сценария _GrailsSettings. Обратите внимание, что сценарий _GrailsSettings автоматически подключается практически через любые сценарии Grails, единственное заметное исключение – сценарий _GrailsProxy.

Следующая особенность – это переменная сценария `buildConfig` ❷. Ранее вы уже видели, что пользователи могут сохранять параметры настройки не только в файле Config.groovy, но и в файле BuildConfig.groovy. Параметры из обоих файлов доступны в сценарии в виде переменных `config` и `buildConfig` соответственно. Чтобы получить значение свойства `tomcat.mgr.username`, мы используем синтаксис `buildConfig.tomcat.mgr.username`. Как определить, в каком файле с настройками должны находиться данные? Как правило, любые параметры настройки, необходимые сценариям, должны помещаться в файл BuildConfig.groovy, поэтому наш сценарий развертывания использует переменную `buildConfig`.

Еще одной новой особенность является инструкция `return` ❸. Если задание возвращает целое число, Gant интерпретирует его как код возврата. Это важно, потому что многие инструменты, особенно это относится к серверам автоматизированной сборки (о которых мы поговорим чуть ниже), часть используют код возврата, чтобы определить, насколько успешно прошла сборка. Значение 0 свидетельствует об успехе, тогда как любое ненулевое значение говорит о том, что произошла какая-то ошибка. Обычно, когда при выполнении задания обнаруживаются какие-нибудь проблемы, в качестве кода возврата используется значение 1 (как это сделали мы в сценарии развертывания), потому что оно используется как универсальный код ошибки. Другие значения имеет смысл использовать, только если вам требуется обозначить различия между проблемами.

И еще одно замечание касается свойства `text`, которое добавляется к объекту URL интерпретатором Groovy. При обращении к этому свойству открывается соединение с указанным адресом URL, читается ответ и возвращается содержимое ответа в виде строки.

Чего же вы ждете? Опробуйте свою новую команду `tomcat-deploy`, и вы поймете, как за счет небольших усилий, направленных на создание сценария, удалось снять бремя развертывания приложения вручную.

Заключительный совет

Содержимое файла WAR имеет большое значение, поэтому мы рекомендуем перед тем, как создавать файл WAR, который планируется развернуть в действующем окружении (или просто передать кому-либо), получить из репозитория самые последние версии файлов проекта с исходными текстами. При выемке исходных текстов вы должны отдавать предпочтение версии, которая была протестирована. Создание файлов WAR из копий, с которыми работают разработчики, практически всегда будет приводить к проблемам, вызванным локальными изменениями, дополнительными файлами и так далее.

Поздравляем, теперь вы можете считать себя специалистом по созданию сценариев! Мы показали, как легко написать пару полезных сценариев, и мы уверены, что теперь у вас начнут появляться собственные идеи. При определенных навыках вы сможете даже оформить свои сценарии в виде расширения, о чем мы поговорим в следующей главе.

Хотя мы уделили системе сборки платформы Grails немало времени, мы рассмотрели ее лишь со стороны командной строки Grails. Для многих пользователей командная строка – это все, что им требуется, но, как мы говорили в начале главы, иногда для сборки приложений на

платформе Grails необходимо использовать различные инструменты. Однако, настройка таких инструментов для успешной сборки проектов Grails – это не прогулка по парку. Как обращаться с расширениями, которые внедряются в процесс сборки? Нам необходим некоторый способ, который позволил бы использовать существующую систему сборки из других инструментов сборки.

15.2. Автоматизированная сборка – не для любителей

Платформа Grails обладает гибкой и расширяемой системой сборки, но иногда она может не соответствовать вашим требованиям. Что если ваша компания предписывает использовать инструмент Maven как стандартное средство сборки проектов, или ваше приложение на платформе Grails является частью крупного проекта, для сборки которого используется инструмент Ant? Для людей, отвечающих за сборку, сообщение о том, что они должны использовать что-то другое для сборки вашего приложения, будет выглядеть как неподъемный груз. Так что готовьтесь к шквалу возмущений!

Существование этой проблемы признается разработчиками платформы Grails, поэтому уже сейчас имеются механизмы гладкой интеграции системы сборки платформы Grails с обоими наиболее популярными в мире Java инструментами сборки, Ant и Maven. Для начала поговорим об инструменте Ant, который продолжает пользоваться феноменальной популярностью.

15.2.1. Ant

Это предок всех инструментов сборки для платформы Java. Несмотря на почтенный возраст, по сравнению с его более молодыми конкурентами, он достаточно надежен, гибок и по-прежнему находит широкое применение. Возможно, самый большой комплимент, который можно отпустить в его сторону, – уже существует несколько других инструментов сборки, способных напрямую выполнять задачи, написанные для Ant. Но достаточно ли просто будет с помощью этого инструмента реализовать сборку проектов на платформе Grails? К счастью, да.

Представьте, что у нас имеется простой файл сборки для инструмента Ant, в котором присутствуют задания для компиляции классов приложения, для компиляции тестов, для запуска тестов и сборки файла WAR. Обычно эти задания должны содержать соответствующие задачи Ant, такие как `javac` и `junit`, но сборка приложения на платформе Grails – не самое простое занятие, и поэтому лучше как можно больше работы переложить на плечи системы сборки Grails. Кроме того, следует помнить, что некоторые расширения внедряются в систему сборки платформы Grails с целью обеспечить дополнительные особенности,

поэтому, если сборку целиком и полностью производить только средствами инструмента Ant, эти особенности окажутся недоступными.

Платформа Grails включает в себя специализированную задачу для Ant, которая активирует систему сборки Grails и позволяет запускать любые сценарии Grails. В листинге 15.3 показано, как настроить эту задачу и как использовать ее.

Листинг 15.3. Файл инструмента Ant для сборки приложения на платформе Grails

```
<project name="my-app" basedir=".">
    <property environment="env"/>
    <property name="grailsHome" value="env.GRAILS_HOME"/>   <- Местоположение Grails

    <taskdef name="grails"  <- ❶ Определение задачи Grails для Ant
        classname="grails.ant.GrailsTask"
        classpath="${grailsHome}/dist/grails-bootstrap-1.1.jar"/>

    <target name="compile">
        <grails home="${grailsHome}" script="Compile"/>  <- ❷ Вызов задачи
    </target>

    <target name="test">
        <grails home="${grailsHome}"                     <- Вызов сценария
            script="TestApp" args="-unit"/>                 с аргументами
    </target>
</project>
```

Первая хитрость состоит в том, чтобы при определении задачи гарантировать, что JAR-файл grails-bootstrap будет находиться в одном из каталогов, включенных в путь поиска классов ❶. В примере мы явно устанавливаем значение пути поиска классов (classpath) в определении задачи, указывая местоположение JAR-файла относительно каталога установки Grails. Вы могли бы скопировать файл JAR в один из каталогов пути поиска классов самого инструмента Ant, но такой подход используется реже. Важно отметить, что класс `grails.ant.GrailsTask` содержится в файле grails-bootstrap.

Как только задача будет определена, ею можно воспользоваться. Мы вызываем сценарий `Compile` инструмента Gant, чтобы скомпилировать классы приложения ❷. Приложение ссылается на большое количество классов платформы Grails, поэтому сценарий компиляции должен знать, где их искать. По этой причине мы передаем задаче Ant путь к каталогу установки платформы Grails в виде атрибута `home`. Обязательный атрибут `script` содержит имя запускаемого сценария, а атрибут `args` содержит список разделенных пробелами аргументов, передаваемых сценарию. Сценарий `Compile` не использует аргументы, но сценарий `TestApp` может принимать несколько аргументов, и мы демонстрируем один из них в листинге 15.3 (аргумент `-unit` сообщает платформе Grails, что требуется выполнить только модульные тесты).

Такой подход отлично работает и весьма прост, но он имеет один недостаток: он предполагает, что платформа Grails была установлена локально. Вам может показаться, что в этом нет никакой проблемы; в конце концов, он неплохо действует. Но подумайте: как много программных продуктов, реализующих сборку, требуют установки других инструментов сборки? Вероятно, не много (если таковые вообще существуют). Определенно можно сказать, что большинство платформ веб-приложений на Java не предъявляют подобных требований. Ваши коллеги едва ли с охотой откликнутся на предложение установить еще один программный продукт. К счастью, имеется возможность использовать платформу Grails, не устанавливая ее, но для этого придется выполнить дополнительные настройки.

Без Grails?

Когда выполняется установка Grails, все ее JAR-файлы в каталогах lib и dist добавляются в пути поиска классов этапов компиляции и тестирования, что существенно упрощает жизнь разработчику. Но как быть, если платформа Grails не была развернута? Ее файлы JAR должны попасть в путь поиска классов, в этом и заключается ваша задача.

В случае задачи Ant мы можем использовать некоторые вложенные элементы, предназначенные специально для этого. Но сначала мы должны рассмотреть проблему зависимостей *задачи*. Вам лишь нужно, чтобы в пути поиска классов задачи находился JAR-файл grails-bootstrap, но для запуска самой задачи требуется еще несколько библиотек, таких как Groovy и Gant. В предыдущем примере задача получала доступ к этим библиотекам благодаря наличию установленной платформы Grails (через атрибут `home`), но мы также можем передать местоположение этих библиотек с помощью атрибута `classpathref`

```
<grails script="..." classpathref="grails.classpath">
```

или с помощью вложенного элемента `classpath`:

```
<grails scripts="...">
    <classpath>
        <pathelement location="..."/>
        ...
    </classpath>
</grails>
```

Оба приема должны быть знакомы по некоторым основным задачам Ant.

Это самое простое. Сложнее определить, какие файлы JAR должны добавляться в путь поиска классов и где они должны храниться. Вы могли бы скопировать JAR-файл grails-bootstrap и все JAR-файлы из каталога lib платформы Grails в каталог lib *своего приложения*. После этого достаточно было бы просто добавить все файлы из этого каталога в путь поиска классов. Недостаток такого подхода заключается в том, что вам

придется добавлять слишком много совершенно ненужных библиотек. Существует ли иная возможность?

Платформа Grails и Ivy

Создав проект с помощью платформы Grails, вы можете обнаружить в корневом каталоге проекта готовый к использованию файл сборки Ant и несколько файлов для инструмента Ivy. Вы можете сразу же начать использовать этот файл сборки и задействовать его для автоматической сборки.

Для всех файлов JAR платформы Grails, в том числе и для grails-bootstrap, имеются соответствующие определения объектной модели проекта Maven (Maven Project Object Model, POM). Они включают описания зависимостей элементов. Выполнять разбор определений POM вручную – не самое простое занятие, но вы можете воспользоваться инструментом Apache Ivy (*http://ant.apache.org/ivy/*) или задачей Ant для Maven (*http://maven.apache.org/ant-tasks/index.html*) и с их помощью извлекать необходимые зависимости.

Предположим, что вы решили использовать задачи Ant для Maven и загрузить их в пространство имен `artifact` в файле сборки. Следующий фрагмент загрузит все зависимости, необходимые задаче Ant для Grails, и создаст путь к ним с идентификатором `grails.classpath`:

```
<project ...>
    ...
    <target name="dependencies">
        <artifact:dependencies pathId="grails.classpath">
            <dependency groupId="org.grails" artifactId="grails-bootstrap"
                version="1.1" scope="runtime"/>
            <dependency groupId="org.grails" artifactId="grails-scripts"
                version="1.1" scope="runtime"/>
        </artifact:dependencies>
    </target>
    ...
    <target name="compile" depends="dependencies">
        <grails script="Compile" classpathref="grails.classpath"/>
    </target>
    ...
</project>
```

Полное описание задач Ant для Maven вы найдете на веб-сайте. Наиболее важными элементами в предыдущем фрагменте являются две зависимости: чтобы получить возможность вызывать любые сценарии платформы Grails из задач Ant, вам необходимы grails-bootstrap и grails-scripts (а также их зависимости).

Компиляция, тестирование, запуск...

Путь поиска классов, необходимый задаче Ant, отличается от необходимого для компиляции, тестирования и запуска приложения. В конце концов, системе сборки требуются только библиотеки grails-bootstrap и grails-scripts, но самому приложению часто требуется механизм GORM, поддержка контроллеров и представлений, поддержка тестирования и другие компоненты платформы Grails. Как упоминалось выше, платформа Grails выбирает необходимые библиотеки из каталога установки, но когда платформа вообще не установлена, необходим иной подход.

Платформа Grails помещает все зависимости в каталог lib, однако внутри система сборки поддерживает три различных пути поиска файлов: один – для компиляции классов приложения, другой – для компиляции и выполнения тестов, и третий – для запуска самого приложения. Эти пути редко совпадают, но как узнать, какие файлы JAR должны входить в каждый из них? Вы легко можете определить необходимые библиотеки, которые используются непосредственно вашим программным кодом, но элементы платформы Grails – совсем другое дело.

Как и в случае с определением пути к классам для задачи Ant, одна из возможностей состоит в том, чтобы скопировать все библиотеки платформы Grails в каталог lib приложения и затем использовать, например, такое определение:

```
<project ...>
    <path id="grails.classpath">
        <fileset dir="${basedir}/lib" includes="*.jar"/>
    </path>
    ...
    <target name="compile">
        <grails script="Compile" classpathref="grails.classpath">
            <compile-classpath>
                <path refid="grails.classpath"/>
            </compile-classpath>
            <test-classpath>
                <path refid="grails.classpath"/>
            </test-classpath>
            <runtime-classpath>
                <path refid="grails.classpath"/>
            </runtime-classpath>
        </grails>
    </target>
    ...
</project>
```

Каждый элемент `*-classpath` интерпретируется так же, как элемент `classpath`, благоря чему вы можете встраивать в них элементы `fileset`, `path` и `pathelement`. Как и в предыдущем случае, этот подход страдает не-

которой избыточностью, и для вас было бы предпочтительнее использовать задачи Ant для Maven или Ivy.

Это ведет нас к инструменту сборки, в котором управление зависимостями поставлено во главу угла: Maven.

15.2.2. Maven

Нравится вам это или нет, но Maven по-прежнему широко используется. Основные его принципы соглашений по сборке и управление зависимостями могут упростить процедуру сборки и обеспечить непротиворечивость множества проектов. Однако интеграция приложения на платформе Grails с инструментом сборки Maven вручную подобна мазохизму, потому что каждый из этих продуктов имеет свое мнение и эти мнения существенно отличаются! К счастью, интеграция существенно упрощается благодаря расширению Maven и архетипам.

Самый простой путь объединить инструмент Maven и платформу Grails состоит в создании нового проекта с использованием архетипа Grails:

```
mvn archetype:generate -DarchetypeGroupId=org.grails \
    -DarchetypeArtifactId=grails-maven-archetype \
    -DarchetypeVersion=1.0 \
    -DarchetypeRepository=http://snapshots.repository.codehaus.org \
    -DgroupId=example -DartifactId=my-app
```

В результате будет создано новое приложение с именем `my-app` и с идентификатором группы `example`. Обратите внимание, что для этого вам не потребуется устанавливать платформу Grails. Все, что вам нужно, – это версия Maven 2.0.9 (или выше).

Посмотрим, что создала эта команда:

```
my-app
|
+--- pom.xml
|
+--- src - main - webapp - WEB-INF - web.xml
```

Файл pom.xml – это модель проекта Maven (POM), которая включает в себя все, что вам необходимо для сборки приложения на платформе Grails с помощью Maven. Она содержит стандартный набор зависимостей платформы Grails и параметры настройки для расширения Grails Maven. Файл web.xml пустой и необходим только для того, что расширение Maven WAR не генерировало сообщение об ошибке во время упаковки приложения. Оставим его в покое.

Первое, что сразу же бросается в глаза, – все это не очень похоже на приложение Grails. Однако ситуацию легко исправить командой:

```
mvn initialize
```

Инструмент Maven загрузит все необходимые составляющие платформы Grails (и их зависимости) и создаст стандартную, знакомую вам структуру проекта. Теперь все готово, чтобы двигаться дальше.

За кулисами

Даже хотя расширение Maven для платформы Grails внедряется в стандартный цикл сборки, он использует дым и зеркала, чтобы создать обманчивое ощущение более тесной интеграции, чем она есть на самом деле. Для сборки приложения на платформе Grails не используется ни одно стандартное расширение для инструмента Maven. Выполнение каждого этапа сборки делегируется соответствующему сценарию платформы Grails, например этап `compile` делегируется сценарию `Compile`, `test` – сценарию `TestApp`, а `package` – сценарию `War`. Фактически вся интеграция опирается на стандартные расширения, не делающие ничего, потому что структура проекта не соответствует соглашениям.

Следует также отметить, что зависимости, присутствующие в сценариях платформы Grails по-прежнему принимаются во внимание, поэтому все предыдущие этапы (такие как `compile`) выполняются не только пакетом `mvn`, сценарий `War` также вызывает компиляцию приложения на платформе Grails. Другими словами, задание `compile` платформы Grails вызывается более одного раза. Это не самое оптимальное решение, но оно действует.

Расширение Maven для платформы Grails внедряется в стандартный цикл сборки, реализованный в инструменте Maven, поэтому у вас имеется возможность вызывать различные этапы сборки, такие как `compile`, `test` и `package`. Кроме того, как и в любом другом проекте Maven, зависимости приложения объявляются в описании модели POM, и вам больше не нужно копировать необходимые файлы JAR в каталог lib. Расширение гарантирует, что библиотеки будут добавлены в соответствующие пути к файлам, и вам не придется беспокоиться об этом, как при использовании инструмента Ant, о котором рассказывалось в предыдущем разделе. Расширение само позаботится обо всем, что нужно.

В дополнение к стандартным этапам сборки расширение добавляет множество заданий, таких как `run-app` и `install-templates`. Фактически расширением реализованы задания, эквивалентные большинству команд платформы Grails; полный их перечень вы найдете в руководстве пользователя платформы Grails. Для запуска команд используется следующий синтаксис:

```
mvn org.grails:grails-maven-plugin:1.0:run-app
```

Этот прием действует, но у вас вряд ли есть желание постоянно вводить такие длинные команды. Данная многословность необходима, чтобы избежать конфликтов с расширениями, реализующими задания с идентичными именами. К счастью, вы можете исключить необходимость ввода большого числа символов с клавиатуры, добавив следующий фрагмент в файл $HOME/.m2/settings.xml:

```
<settings>
    ...
    <pluginGroups>
        <pluginGroup>org.grails</pluginGroup>
    </pluginGroups>
</settings>
```

Этот маленький фрагмент XML способен превратить монстроподобные команды в более короткие эквиваленты, например:

```
mvn grails:run-app
```

Так намного проще для глаз и для пальцев, и мы думаем, что вы согласитесь с нами.

Теперь вы знаете, как создать новый проект на платформе Grails с поддержкой Maven, но как быть, если потребуется встроить поддержку инструмента Maven в существующий проект? Просто:

```
mvn grails:create-pom
```

Эта команда создаст модель POM и некоторые другие файлы проекта, после чего вы сразу же можете выполнить сборку с помощью инструмента Maven.

И еще одно последнее замечание: из-за особенностей интеграции с инструментом Maven некоторые его расширения, такие как Jetty, не будут работать с проектами на платформе Grails. Все, что опирается на стандартную структуру проекта Maven, скорее всего, не будет действовать без дополнительной настройки под структуру проекта Grails. Но, несмотря на это, имеющейся интеграции с Maven должно быть вполне достаточно для большинства проектов, и у нее есть дополнительное преимущество – простота установки и настройки.

Если вам потребуется интеграция с каким-либо другим инструментом, отличным от Ant или Maven, те приемы, которые мы только что обсудили, едва ли помогут вам. Вам может потребоваться поработать непосредственно с классами платформы Grails, реализующими начальную загрузку и настройку. Эта тема выходит далеко за рамки книги, но совсем нелишне будет знать, что это возможно. Мы надеемся, что в будущем и другие инструменты сборки приобретут поддержку проектов на платформе Grails.

Мы много говорили об инструментах сборки и о системе сборки самой платформы Grails, но у нас остался еще один важный аспект, имеющий отношение к развертыванию приложений, который мы еще не рассмат-

ривали. Упростить развертывание новых версий приложения очень важно, но как быть со структурой базы данных? По мере развития приложения структура базы данных, скорее всего, будет изменяться, поэтому очень важно предусмотреть возможность управления этими изменениями.

15.3. Обслуживание изменений в модели данных

Большинство приложений на платформе Grails построено на моделях данных, хранящихся в базе данных. В идеальном мире первая же попытка спроектировать эту модель была бы удачной, и вам никогда не пришлось бы изменять ее. Но, как все мы знаем, действительность далека от идеала. Модели данных постоянно развиваются, а это требует внесения изменений в структуру базы данных. Часто такое происходит уже *после* того, как действительные данные начнут накапливаться в базе данных.

Управление этими изменениями составляет важную часть не только процесса разработки приложения, но и его развертывания. Как можно улучшить структуру и внедрить в нее новые возможности, не потеряв существующие данные? Сначала мы рассмотрим, возможно ли выработать подход на основе настроек `dbCreate` источника данных, а затем рассмотрим расширение, созданное специально для решения этой проблемы.

15.3.1. Изменение структуры с помощью Hibernate

При разработке приложения Hubbub вы использовали преимущество использования особенности библиотеки Hibernate, называемой «автоматическим экспортом структуры базы данных». Для создания соответствующих таблиц в базе данных эта особенность использует модель предметной области. На рис. 15.4 изображено, как она действует в процессе запуска приложения.

Управление автоматическим экспортом структуры производится с помощью параметра настройки источника данных `dbCreate`, с которым вы познакомились в главе 1 (раздел 1.4.1). В ходе разработки этому параметру обычно присваивается значение `create` или `create-drop`. Эти значения гарантируют, что в базе данных не будут сохраняться устаревшие таблицы и поля, но ценой потери всех данных при каждом запуске приложения. Если вам необходимо постоянно иметь в приложении некоторые справочные или тестовые данные, их следует создавать классом `BootStrap` приложения.

Как только модель данных начинает вырисовываться, такая постоянная потеря данных начинает раздражать, особенно, если вы создали неплохой набор данных в процессе тестирования приложения вручную. С этого момента бывает желательно переключить параметр `dbCreate` в значе-

ние update. Такой прием обеспечит сохранение информации в базе данных и обеспечит синхронизацию структуры базы данных в соответствии с моделью данных.

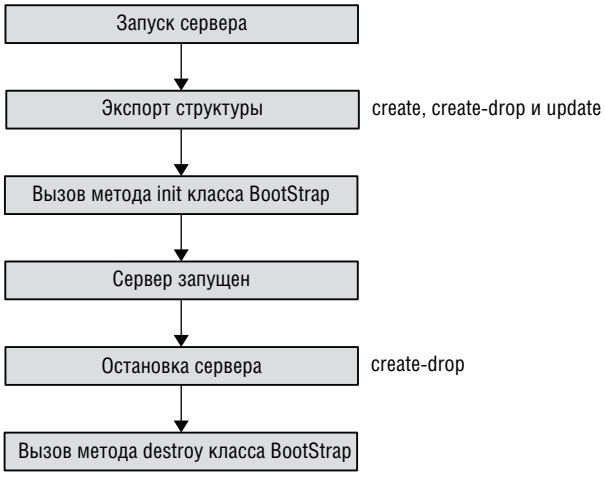


Рис. 15.4. *Место механизма экспорта структуры библиотеки Hibernate в процессе запуска и остановки приложения. Строки справа указывают активные настройки dbCreate на каждом этапе*

Значение update отлично подходит для этапа разработки, но с ним сопряжены некоторые серьезные проблемы, из-за которых его нежелательно использовать в действующих системах. Во-первых, это значение не обеспечивает некоторые виды перемещения данных. Классическим примером является переименование поля таблицы: библиотека Hibernate создаст новое поле, но она не перенесет в него данные и не удалит старое поле. Это влечет за собой вторую проблему: старые таблицы и поля остаются в базе данных, собирая пыль и вызывая путаницу. Наконец, библиотека Hibernate не следит за изменениями в структуре базы данных, то есть она не обеспечивает возможность откатиться к одной из предыдущих версий.

Традиционное в таких ситуациях решение для управляемого перемещения данных заключается в использовании написанных вручную сценариев SQL, которые обновляют структуру базы данных и выполняют перенос информации. Вы, конечно, можете пойти этим путем, но для этого вы должны полностью удалить параметр настройки dbCreate. Это предотвратит вмешательство в структуру базы данных со стороны библиотеки Hibernate.

Если вы решите использовать сценарии SQL, вам будет интересно познакомиться со следующей командой платформы Grails:

```
grails schema-export initial-schema.sql
```

Она создаст SQL-сценарий на языке определения данных (Data Definition Language, DDL), который можно будет использовать для повторного создания структуры в пустой базе данных. Вместо того чтобы полагаться на механизм автоматического экспортирования данных из библиотеки Hibernate, вы можете выполнить сценарий SQL на пустой базе данных, подготовив ее тем самым к работе с приложением. Если вызвать команду без аргумента, она создаст файл с именем ddl.sql.

Создание и сопровождение сценариев SQL – довольно тяжелый труд, но мы к нему не сильно привыкли вследствие использования платформы Grails. Поэтому сейчас мы познакомимся с расширением, которое обещает упростить нам жизнь, когда дело дойдет до перемещения данных.

15.3.2. Интеллектуальное перемещение данных с расширением Autobase

Расширение Autobase пытается решить две проблемы: перемещение данных, которое не может быть выполнено библиотекой Hibernate, и перемещение данных после изменения структуры базы данных. Обе проблемы уже решены в библиотеке Java с именем LiquiBase, и именно на этой библиотеке основано расширение.

Первым шагом на пути к обретению возможности интеллектуального перемещения данных является установка расширения:

```
grails install-plugin autobase
```

В процессе установки расширение создаст в корневом каталоге проекта новый каталог – migrations. Здесь будет храниться информация об изменениях в структуре и в данных.

Что дальше? Допустим, что вам потребовалось в приложении Hubbub переименовать поле `userId` в классе предметной области `User` и присвоить ему новое имя `username`. Если параметр `dbCreate` установлен в значение `update`, мы можем выполнить необходимые изменения и перезапустить приложение. Библиотека Hibernate создаст новое поле `username`, как и ожидалось, но старое поле останется на месте, и в нем будут храниться все существовавшие ранее идентификаторы пользователей. Очевидно, это проблема.

Вместо того чтобы использовать значение `update`, нам следует вообще удалить параметр `dbCreate`. Это означает, что мы принимаем на себя ответственность за *все* изменения в базе данных и, как минимум, понимаем, что делаем – это важно. Затем мы должны описать перемещения, которые расширение Autobase сможет применить. Создание нового сценария перемещения производится выполнением простой команды

```
grails create-migration RenameUserId
```

которая создаст файл migrations/<login>/RenameUserId.groovy, где <login> – это имя вашей учетной записи.

Вновь созданный файл практически пуст, поэтому мы должны добавить в него описание перемещения для поля `user_id`. Обратите внимание, что мы ссылаемся на имя поля в базе данных, а не на имя поля класса предметной области.

```
changeSet(id:'RenameUserId', author:'peter') {
    renameColumn(tableName:"account",
                 oldColumnName:"user_id",
                 newColumnName:"username")
}
```

Мы определили единственный набор изменений (changeset), который производит переименование поля, выполняя то, что на языке библиотеки LiquiBase называется «структурным перемещением» (structural migration). Другими словами, мы вызываем операцию, которая модифицирует структуру таблицы или представления в базе данных. На рис. 15.5 изображен эффект нашего воздействия.

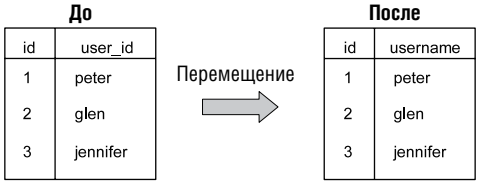


Рис. 15.5. *Результат переименования поля user_id в username в таблице с учетными записями без потери данных*

Откуда взялись методы `changeSet()` и `renameColumn()`? Использованный выше синтаксис DSL отображается в эквивалентный формат XML описания изменений библиотеки LiquiBase. Это упрощает использование существующей документации для библиотеки LiquiBase, где упоминается формат XML, поскольку вы уже знаете, как выполняются преобразования между этими двумя форматами. О других доступных видах преобразований можно узнать по адресу: *http://www.liquibase.org/manual/home*.

Теперь, когда вы запустите приложение Hubbub, расширение Autobase автоматически выполнит перемещение данных. Более того, оно настолько интеллектуально, что пропустит описания изменений, которые уже были выполнены в базе данных.

Это был пример единственной операции перемещения, но вы, вне всяких сомнений, будете применять множество подобных перемещений в своих приложениях. Вы можете добавлять новые блоки `changeSet()` в существующий файл перемещения или создать новый. Какой способ выберете вы, зависит от ваших личных предпочтений, но легче управлять небольшим числом файлов, содержащих по нескольку наборов изменений, чем большим числом файлов с один набором изменений в каждом.

За кулисами

Библиотека LiquiBase фиксирует, какой набор изменений был применен к базе данных, в своей собственной таблице. Всякий раз, когда выполняется перемещение данных, библиотека LiquiBase выполняет только те наборы изменений, которые следуют за последним сохраненным в таблице.

Наборы изменений применяются в том порядке, в каком они следуют в сценарии. Однако по умолчанию не гарантируется какой-то определенный порядок применения самих сценариев. Изменить этот порядок можно, отредактировав файл migrations/changelog.groovy и заменив вызов метода `includeAll()` серией вызовов метода `include()` для каждого отдельного сценария, расположив их в требуемом порядке. В качестве аргумента последний метод принимает путь к единственному сценарию.

Что включать в наборы изменений? Это хороший вопрос, но на него нет простого ответа. Одна из стратегий состоит в том, чтобы создавать новый набор изменений всякий раз, когда потребуется изменить структуру базы данных. Если это крупное изменение, его желательно разбить на несколько небольших наборов. Например, добавление нового отношения может потребовать создания нового поля или двух и добавления внешнего ключа, при этом все эти изменения могут быть оформлены в одном наборе изменений.

Хотя вам предоставляется большая свобода в выборе, что помещать в наборы изменений, единственное, чего делать не следует, – это оставлять базу данных в «сломанном» состоянии. Например, удалить поле и оставить внешний ключ, который ссылается на него.

Такие изменения, как переименования полей, не являются единственными возможными. Расширение Autobase добавляет новую команду, которая позволяет выполнять более сложные операции:

```
grails create-groovy-migration UpdateData
```

Эта команда создаст не только обычный сценарий перемещения, но и сценарий на языке Groovy в каталоге migrations/<login>/.scripts. Программный код сценария обеспечивает доступ к некоторым важным объектам, таким как экземпляр класса `groovy.sql.Sql` (за информацией о других объектах обращайтесь к сценарию, созданному этой командой), позволяющий напрямую манипулировать данными в базе данных.

Это все, что мы хотели сказать! Просто запомните, что как только вы удаляете параметр настройки `dbCreate`, вы должны создавать наборы изменений для каждого изменения в модели данных, требующего изменения в структуре базы данных.

Внимание

В настоящее время расширение Autobase не может воссоздавать структуру базы данных с нуля, поэтому оно не может работать с совершенно пустой базой данных. Остается надеяться, что эта проблема будет решена в ближайшем будущем. А пока рассмотрите возможность использования команды `schema-export` перед применением каких-либо перемещений. После запуска сценария SQL, сгенерированного этой командой, в пустой базе данных, вы сможете запустить само приложение (с установленным расширением Autobase).

Управляемое изменение структуры базы данных с использованием такого инструмента, как расширение Autobase, первое время может казаться неудобным, особенно в сравнении с механизмом автоматического экспорта структуры базы данных, реализованном в библиотеке Hibernate, но в конечном счете этот подход убережет вас от множества разочарований. Когда известно, какие изменения применялись, достаточно просто связать их с особенностями запросов и с сообщениями об ошибках. Кроме того, вы легко можете откатиться к предыдущим состояниям базы данных. Эти возможности гарантируют, что любом случае в вашем распоряжении будут необходимые инструменты и информация, которые помогут развернуть новую версию приложения при уже существующей базе данных.

15.4. В заключение

Как вы могли видеть в этой главе, разработка программного обеспечения предусматривает не только создание программного кода. Вы должны быть в состоянии запускать тесты, производить упаковку приложения и, возможно, развертывать его. Платформа Grails предоставляет полноценную систему сборки, которая принимает на себя все бремя по созданию системы, но позволяет при этом вмешиваться в ее действия и расширять ее. И все же платформа не обеспечивает полную поддержку всего цикла разработки, поэтому вам может потребоваться создавать свои сценарии Grails.

Кроме того, у вас может сложиться такая ситуация, когда невозможно использовать систему сборки платформы Grails непосредственно, и вы вынуждены пользоваться альтернативным инструментом сборки. Вы видели, что платформа Grails обеспечивает возможность интеграции с такими инструментами, как Ant и Maven, поэтому у вас нет никаких причин для беспокойства. Однако вопрос заключается скорее не в том, есть ли у вас возможность выбора, а в том, нужно ли вам использовать один из этих инструментов.

Ниже приводятся несколько рекомендаций, которые помогут вам решить проблемы, связанные с разработкой и сборкой приложения:

- *Создавайте сценарии для часто повторяющихся задач.* Автоматизация – важный прием, способствующий снижению вероятности по-

явления ошибок и повышению воспроизводимости результатов. Более того, вложенные в самом начале небольшие инвестиции времени могут позднее принести большие дивиденды. Когда время будет поджимать, вы будете рады тому, что для выполнения любой задачи достаточно ввести единственную команду.

- *Используйте автоматическую сборку.* Этот прием обязателен почти для любого коллектива, куда входит более чем один человек, так как он позволяет выявлять ошибки при объединении изменений, полученных от разных людей. Автоматическая сборка будет полезна даже для единственного разработчика, так как она будет выполнять полный комплект тестов, если вы вдруг забудете выполнить их локально или если их выполнение занимает продолжительное время.
- *Используйте тот инструмент сборки, который для вас более удобен.* В конце концов, вы должны использовать тот инструмент сборки, который отвечает вашим требованиям и который для вас является наиболее удобным. Вам нравится система сборки платформы Grails – используйте ее, если это возможно. Она поможет вам сэкономить немало усилий. Если ваше приложение является частью более крупного проекта, рассмотрите возможность использования другого инструмента сборки.
- *Используйте управляемый способ перемещения данных.* Несмотря на удобство механизма экспорта структуры базы данных из библиотеки Hibernate, его полноценность, в конечном счете, несколько ограничена. Возьмите на вооружение такой инструмент, как расширение Autobase, как только модель данных начнет приобретать четкие очертания или, самое позднее, после первого развертывания приложения.

Что бы вы ни выбрали, вы можете быть уверены, что платформа Grails обеспечит поддержку вашего решения и позволит использовать выбранный вами инструмент сборки без лишних хлопот.

В этой главе:
- Как действуют расширения
- Создание собственных расширений
- Управление репозиториями расширений

16

Разработка расширений

В предыдущей главе мы изучили проблемы интеграции с системой сборки и управления проектом, теперь мы рассмотрим способ разделения приложения на модули и добавления в него возможностей за счет создания собственных расширений. Мы познакомили вас с системой расширений платформы Grails в главе 8, где вы могли видеть, насколько удобными могут быть расширения. Они являются одной из самых фундаментальных и мощных частей инфраструктуры платформы Grails, и даже если вам никогда не придется писать свои собственные расширения, понимание того, как действует эта система, пригодится вам в будущем.

Когда может возникнуть необходимость написать свое собственное расширение? Мы не хотели бы ограничивать вас какими-либо рамками, тем не менее ниже приводятся три наиболее часто встречающихся типа ситуаций:

- Интеграция платформы Grails с существующей библиотекой или инструментом
- Реализация специальных возможностей
- Реализация модульной архитектуры приложения

Вы уже видели множество примеров ситуаций первого типа, в них мы использовали такие расширения, как Searchable (обеспечивающее интеграцию с библиотекой Compass), Spring Security и другие. Примерами для второго типа ситуаций могут служить расширения Authentication и Functional Test, которые обеспечивают управление доступом и возможность функционального тестирования соответственно. Различия между первыми двумя типами носят скорее технический, чем практический характер, поэтому нет большой необходимости различать их.

Последний в списке тип ситуаций имеет отношение к организации модульной архитектуры приложения. Для построения небольших прило-

жений, таких как Hubbub, нет никакого смысла использовать модульную архитектуру; однако крупные приложения будут сложны для понимания и сопровождения, если оставить их монолитными. Разбиение функциональных возможностей приложения на отдельные расширения поможет противостоять этим проблемам; кроме того, эти расширения можно будет повторно использовать в других приложениях. Такой подход особенно хорошо использовать для разделения обязанностей. Например, представьте, что у вас имеется приложение, которое обрабатывает счета, составляет платежные ведомости, рассчитывает налоги и так далее. Каждая из этих «функций» могла бы быть реализована в виде отдельного расширения, которые затем можно было бы установить в общий проект, чтобы создать приложение. Причем расширения могут разрабатываться отдельными коллективами.

Теперь, когда вы знаете, зачем создавать собственные расширения, можно посмотреть, как это делается. В данной главе в качестве примера мы напишем расширение security для нашего приложения Hubbub. Такая идея может показаться глупой, потому что у нас уже имеется расширение Spring Security, однако это будет удобный способ продемонстрировать наиболее важные особенности, которые нам хотелось бы обсудить.

16.1. Создание нашего первого расширения

Как всегда с чего-то надо начинать, и расширения для платформы Grails не являются исключением. В данном разделе мы создадим проект для нашего расширения security и заложим в его фундамент первый камень: мы добавим в него некоторые элементы, почти так же, как мы делали при создании нового приложения на платформе Grails.

16.1.1. Вы уверены, что это не приложение?

Итак, как же нам создать проект расширения? Все просто – запустим волшебную команду:

```
grails create-plugin security
```

Она создаст каталог с именем security и сохранит в нем заготовку проекта расширения, как показано на рис. 16.1. Это не случайное совпадение, что проект расширения сильно напоминает проект приложения на платформе Grails. Вы можете запустить его, как приложение, воспользовавшись обычной командой `run-app`, что очень удобно для тестирования.

Самое заметное отличие между расширением и приложением – наличие файла дескриптора расширения SecurityGrailsPlugin.groovy (выделен на рис. 16.1). Дескриптор содержит не только информацию о расширении, такую как его автор и описание решаемых им задач, но и программный код, позволяющий внедряться в работу платформы Grails и влиять на поведение приложений во время выполнения.

Имя класса дескриптора имеет большое значение, поскольку оно определяет официальное имя самого расширения. Вместо того чтобы хранить имя в поле класса, платформа Grails определяет его, отсекая окончание «GrailsPlugin», преобразуя заглавные буквы в строчные и отделяя слова символом дефиса. Например, имя `SecurityGrailsPlugin` превращается в `security`, а `MyTagsGrailsPlugin` превращается в `my-tags`.

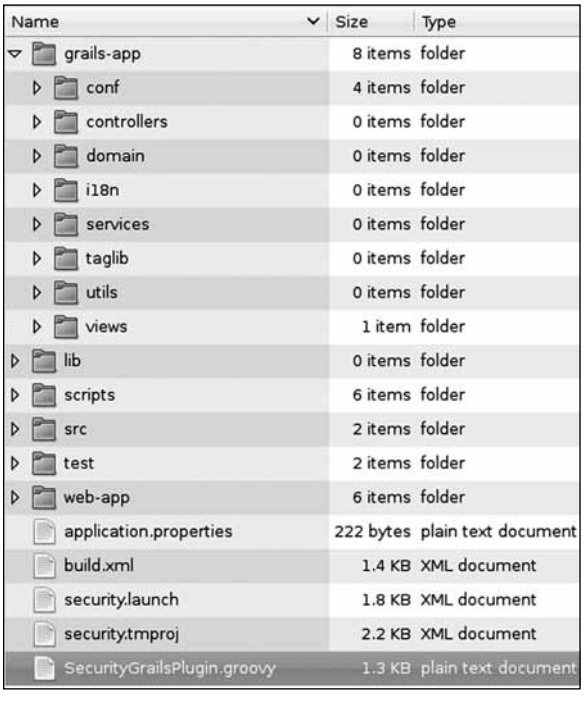


Рис. 16.1. *Структура каталогов проекта расширения. Обратите внимание, насколько близко эта структура напоминает структуру каталогов проекта приложения Grails. Файл дескриптора расширения выделен цветом*

Теперь рассмотрим созданный дескриптор. В листинге 16.1 приводится немного сокращенная версия первоначального содержимого.

Листинг 16.1. Начальное содержимое дескриптора расширения

```
class SecurityGrailsPlugin {
    def version = 0.1                          ← ❶ Объявление версии расширения
    def grailsVersion = "1.1 > *"                  и его зависимостей
    def dependsOn = [:]

    def pluginExcludes = [
        "grails-app/views/error.gsp"           ← ❷ Файлы, исключаемые из расширения
    ]
```

```
    def author = "Your name"
    def authorEmail = ""
    def title = "Plugin summary/headline"
    def description = '''\
Brief description of the plugin.
'''

    def documentation =
        "http://grails.org/Security+Plugin"

    def doWithSpring = { ... }
    def doWithApplicationContext = { ctx -> ... }
    def doWithWebDescriptor = { xml -> ... }
    def doWithDynamicMethods = { ctx -> ... }
    def onChange = { event -> ... }
    def onConfigChange = { event -> ... }
}
```

❸ Объявление информации о расширении

❹ Определение местоположения документации для расширения

❺ Объявление обработчиков

Начнем с информации о версии и о зависимостях расширения ❶. Поле `version` содержит текущий номер версии расширения. Хотя начальное значение является числом, вы можете также использовать строку. То есть такие значения, как `"0.1-SNAPSHOT"` и `"merlin"`, являются допустимыми. С другой стороны, платформа Grails ожидает, что нумерация версий расширения будет следовать соглашению `major.minor.patch`, где каждая часть является целым числом, причем заключительная часть `patch` является необязательной. Например, оба номера, 1.0.4 и 2.1, соответствуют этому соглашению. Нередко при выпуске версий, находящихся в разработке, добавляется окончание «-SNAPSHOT» (например, когда вы хотели бы получить отзывы об изменениях, внесенных в расширение).

Поле `grailsVersion` определяет версии платформы Grails, с которыми данное расширение будет работать. Это может быть либо единственная версия, либо диапазон в виде:

```
lowerBound > upperBound
```

Такое выражение следует читать как: «любая версия от `lowerBound` по `upperBound` включительно». Для любой из границ диапазона допускается использование шаблонного символа (`*`). Выражение ниже означает: «любая версия до 1.1 включительно»:

```
* > 1.1
```

Следующее выражение означает: «любая версия от 1.0 включительно»:

```
1.0 > *
```

Аналогичные диапазоны могут использоваться в поле `dependsOn`, которое содержит отображение имен расширений в номера их версий (или диапазоны версий). Например, если данное расширение зависит от расширений Quartz и Remoting, эти зависимости могут быть описаны, как показано ниже:

```
def dependsOn = [ quartz: "0.4", remoting: "0.3 > *" ]
```

Обратите внимание, что в этом поле не допускается использовать диапазоны с использованием символа <.

Далее следует поле `pluginExcludes` ❷. При создании пакета расширения платформа Grails упаковывает файлы, составляющие его, в архив zip. Некоторые из этих файлов может быть нежелательным помещать в архив, особенно, если эти файлы используются только для нужд тестирования. Вы можете предотвратить включение этих файлов в архив, указав пути к ним в поле `pluginExcludes`. Упаковку расширений мы будем рассматривать ниже в разделе 16.2.

Дополнительная информация о расширении ❸ указывается, главным образом, для конечных пользователей. Команда `list-plugins` отобразит название расширения и его заголовок (поле `title`), а команда `plugin-info` отобразит содержимое всех этих полей.

Что касается поля `documentation` ❹, это адрес URL, указывающий, где находится документация для расширения. Для большинства расширений, находящихся в центральном репозитории платформы Grails, на веб-сайте проекта Grails имеется одна или более страниц.

Ниже поля `documentation` следует нечто более интересное. Все эти замыкания `doWith*` и `on*` ❺ являются обработчиками, которые позволяют расширению воздействовать на поведение приложения во время выполнения. Они обеспечивают тесную интеграцию с такими составляющими платформы Grails, как механизм GORM. В табл. 16.1 коротко описывается, какие возможности предоставляет каждый из обработчиков, и перечисляются аргументы, передаваемые платформой Grails соответствующим замыканиям. Первые три приводятся в том порядке, в каком они вызываются платформой Grails во время запуска приложения.

Таблица 16.1. Обработчики, обеспечивающие тесную интеграцию расширения с приложением на платформе Grails

Свойство	Аргументы	Описание
`doWithSpring`	нет	Это замыкание позволяет определять и настраивать компоненты Spring.
`doWithDynamicMethods`	`applicationContext`	Это замыкание позволяет добавлять динамические свойства и методы к классам и другим элементам платформы. Единственный аргумент – полностью настроенный контекст приложения на платформе Spring.

Свойство	Аргументы	Описание
doWithApplicationContext	applicationContext	Это замыкание позволяет выполнять операции над полностью настроенным контекстом приложения на платформе Spring. Из этого замыкания можно управлять только экземплярами компонентов, но не их определениями.
doWithWebDescriptor	webXml	Это замыкание позволяет изменять веб-дескриптор приложения. Ему передается объект GPathResult (созданный классом XmlSlurper), который можно использовать для добавления таких элементов, как определения сервлетов и фильтров.
onChange	event	Это замыкание позволяет реагировать на изменения в любых элементах платформы, за которыми следит данное расширение.
onConfigChange	event	Это замыкание позволяет реагировать на любые изменения в файле Config.groovy приложения.
onShutdown	event	Это замыкание позволяет выполнить определенные действия в случае остановки приложения. Контекст приложения на платформе Spring все еще доступен в этот момент.

Обратите внимание, что все эти обработчики являются замыканиями; если вы попытаетесь вместо них использовать методы, они не будут вызываться. Мы еще вернемся к этим обработчикам в следующем разделе, но они составляют лишь часть нашего повествования. Сначала мы рассмотрим, как подготовить модели предметной области и веб-страницы.

16.1.2. Контроллеры, представления и другие артефакты

Самый простой способ расширить приложение – реализовать с помощью расширения готовые к употреблению артефакты, такие как классы предметной области и контроллеры. Вернемся к нашему расшире-

нию security: нам требуется обеспечить присваивание ролей пользователям и сохранение этой информации, поэтому нам могут пригодиться некоторые классы предметной области. Кроме того, регистрация пользователя в приложении считается стандартной операцией, тогда почему бы нам не реализовать собственные контроллер и представления, обеспечивающие ее выполнение? Наконец, мы могли бы расширить некоторые теги так, чтобы они отображали части веб-страницы, только если пользователь обладает определенной ролью или является зарегистрированным пользователем. Например, кнопка Remove Post (Удалить сообщение) в приложении Hubbub должна быть доступна только администраторам.

> **Святые артефакты!**
>
> Вы часто будете видеть термин «артефакт» на протяжении оставшейся части главы, но что он означает? Некоторые типы классов интерпретируются платформой Grails особым образом, например классы предметной области, контроллеры и библиотеки тегов. Все они объединяются под общим названием «артефакты» и располагаются в каталоге grails-app проекта.

Большинство артефактов платформы Grails, с которыми вы встречались до сих пор, могут включаться в состав расширения, и мы начнем с того, что посмотрим, какие классы предметной области могут быть реализованы в нашем расширении security.

Классы предметной области

Поскольку нам требуется реализовать возможность присваивания ролей пользователям, нам нужен будет один класс предметной области, представляющий пользователей, и один, представляющий роли.

Наш класс `Role` выглядит очень просто и содержит только поле с названием роли:

```
class Role {
    String name

    static constraints = {
        name(blank: false, unique: true)
    }
}
```

Данные ограничения гарантируют уникальность имен ролей. Другой класс предметной области порождает определенную проблему: в приложении Hubbub уже имеется класс `User`, но он содержит поля, никак не связанные с простым управлением доступом, такие как `signupDate` и `followers`. Они являются характерными только для приложения Hub-

bub, поэтому они не должны включаться в класс предметной области расширения.

В данной ситуации невозможно обойтись без наследования классов предметной области: мы можем выделить поля `userId` и `password` в новый класс и изменить существующий класс `User`, чтобы он наследовал класс из расширения. Новый класс, который мы назовем `Account`, мы включим в состав расширения, а класс `User` останется в приложении. Новая реализация выглядит, как показано ниже:

```
// В расширении
class Account {
    String userId
    String password

    static hasMany = [ roles: Role ]

    static constraints = {
        // Те же ограничения для полей userId и password, что и прежде
        ...
    }
}

// В приложении
class User extends Account {
    ...
}
```

Теперь у нас имеются файлы Account.groovy и Role.groovy в каталоге security/grails-app/domain и User.groovy в каталоге hubbub/grails-app/domain. В данный момент это означает, что приложение Hubbub не будет компилироваться, пока не будет установлено расширение security.

Классы предметной области, предоставляемые расширением, существенным образом будут затрагивать модель предметной области любого проекта, куда будет устанавливаться это расширение. У пользователя нет никакой возможности отключить, изменить или удалить их, не отредактировав сами файлы расширения, поэтому надо быть очень осторожными с классами, которые включаются в состав расширения.

В нашем случае расширение зависит от классов предметной области `Account` и `Role`, поэтому имеет смысл предоставить их в том виде, в каком они есть. Альтернативный подход заключается в том, чтобы скопировать поведение различных команд `create-*` и `generate-*` и написать сценарии создания модели предметной области. Вы наблюдали, как используется этот прием расширением Spring Security в виде его команды `generate-manager`. Как реализовать такой подход, мы расскажем в разделе 16.3.5.

Следующий шаг состоит в том, чтобы предоставить пользователям возможность регистрироваться в приложении.

Контроллеры

Нам требуется реализовать прием имени пользователя и пароля и их проверку на соответствие значениям, хранящимся в базе данных. Если процедура регистрации завершается успехом, мы должны поместить соответствующий экземпляр класса `User` в область видимости сеанса. В противном случае следует перенаправить посетителя обратно на страницу регистрации. То есть нам необходимы: страница регистрации, действие контроллера, обслуживающее регистрацию, и действие контроллера для выхода из приложения. Кроме того, так как мы собираемся обеспечить поддержку ролей, нам потребуется действие контроллера, к которому будут перенаправляться пользователи, не имеющие права доступа к той или иной странице, – то есть страница «доступ запрещен». Ниже приводится схема нашего контроллера `LoginController`:

```
package com.grailsinsaction.security

class LoginController {
    def index = {}
    def signIn = { ... }
    def signOut = { ... }
    def unauthorized = { ... }
}
```

Конкретная реализация этих действий является несущественной для нашего обсуждения, но вы сможете увидеть ее в исходных текстах примеров для этой главы.

Самое главное здесь состоит в том, что существует возможность поместить этот контроллер в каталог расширения grails-app/controllers, а связанные с ним представления – в каталог grails-app/views/login. Как только расширение будет установлено в приложение, контроллер и представления будут вести себя так, как если бы они были частью приложения. Например, в случае с приложением Hubbub вы могли бы ввести в броузере адрес URL *http://localhost:8080/hubbub/login/index* и получить страницу регистрации, предоставляемую расширением. При этом вам не придется выполнять какие-либо дополнительные настройки.

После установки расширения вы сможете переопределить контроллер и представления в своем приложении. Например, если вам не понравится, как выглядит стандартная страница регистрации, вы сможете поместить свой собственный файл index.gsp в каталог grails-app/views/login приложения. Или вы можете полностью изменить поведение процедуры регистрации, предоставив собственную реализацию контроллера `LoginController`. Единственное, чего вы не сможете сделать, – это переопределить только контроллер и использовать представления расширения. Доступ к представлениям расширения имеют только контроллеры расширения.

За кулисами

Чтобы упростить возможность переопределения контроллера, мы поместили его в пакет. Поэтому все будет действовать безупречно, при условии что контроллер в приложении будет иметь то же самое имя, что и контроллер в расширении.

Будьте осторожны: все артефакты, как внутри приложения, так и внутри устанавливаемых расширений, собираются в одном и том же каталоге, поэтому классы с одинаковыми именами и в одноименных пакетах будут конфликтовать между собой.

Библиотеки тегов

Библиотеки тегов действуют практически так же, как контроллеры, и часто используются в мире расширений. Мы настоятельно рекомендуем в библиотеках тегов расширений использовать пространства имен, которые были бы уникальными для данных расширений. Например, для расширения security мы использовали пространство имен `hsec` (банальная комбинация символа «h» от названия приложения «Hubbub» и «sec» от названия расширения «security»). Нет никакой гарантии, что некоторые другие расширения не будут использовать такое же пространство имен, но внимательный подход к выбору поможет минимизировать риск.

Зачем может потребоваться реализовать теги в расширении? Собственные теги могут упростить доступ к функциональным возможностям расширения из представлений GSP. Возможно, такой подход полезен не для всех расширений, но наше расширение security, например, предоставляет теги, которые отображают или скрывают свое содержимое в зависимости от того, зарегистрировался пользователь или нет.

Представления

Мы уже упоминали представления при обсуждении контроллеров; представления в расширении действуют так же, как и любые другие представления. Единственная сложность возникает, когда тег ссылается на другой ресурс в расширении. Когда вы запускаете команду `run-app` из каталога проекта расширения, путь к ресурсам расширения (таким как файлы JavaScript и CSS) другой по сравнению с путем, когда расширение установлено в приложение. В последнем случае путь включает имя расширения во избежание конфликтов имен между разными расширениями.

Чтобы обеспечить использование соответствующего пути, требуется прибегнуть к некоторой магии. Представьте, что наше расширение содержит файл JavaScript, и нам необходимо подключить его к одному из

представлений. Вы должны сконструировать путь к файлу JavaScript, используя динамическое свойство `pluginContextPath`:

```
<g:resource dir="${pluginContextPath}/js" file="security.js" />
```

При непосредственном запуске расширения ссылка будет выглядеть, как: /myapp/js/security.js, а когда расширение будет установлено в приложение, она приобретет вид: /myapp/plugins/security-0.1/js/security.js.

Когда представления ссылаются на шаблоны, требуется использовать иной подход. Тег `render` принимает атрибут `plugin`, который обеспечивает то, что нам необходимо:

```
<g:render template="roleList" plugin="security"/>
```

Он отобразит шаблон `roleList` расширения независимо от того, используется ли этот тег в представлении приложения или расширения.

На рис. 16.2 можно заметить, что артефакты сохраняются в расширении в тех же каталогах, что и в приложении. Все необходимое, чтобы сделать эти артефакты доступными приложениям, в которые устанавливается расширение, делает сама платформа Grails.

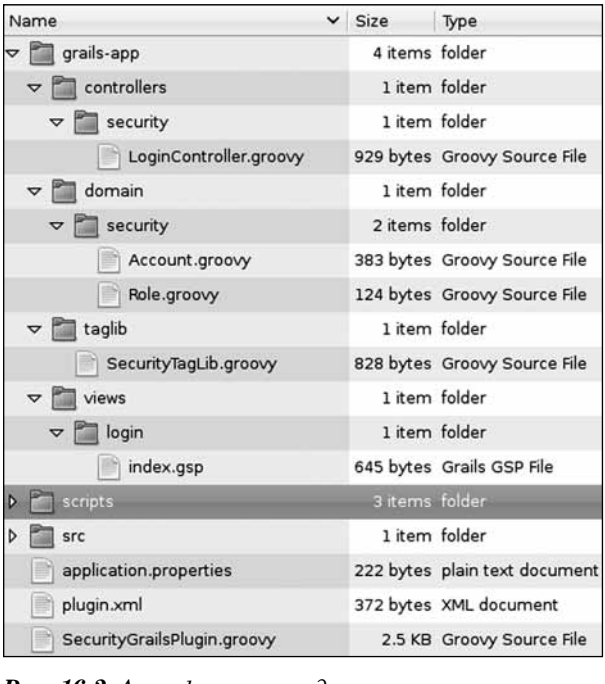


Рис. 16.2. *Артефакты, предоставляемые расширением security*

Теперь у нас имеется модель предметной области и функциональность, обеспечивающая возможность входа и выхода из приложения, но у нас нет механизма управления доступом к страницам приложения. Как до-

бавить его, мы покажем в разделе 16.3, а пока рассмотрим вопрос публикации нашего расширения.

16.2. Публикация расширений

От расширения самого по себе мало проку. Он не имеет самостоятельного значения без приложения, куда его можно было бы установить. Как устранить этот недостаток, вы уже видели в главе 8, где рассказывалось об установке расширений в приложение либо из файлов zip, либо из центрального репозитория расширений для платформы Grails. Теперь мы покажем, как создать этот файл zip и как добавить расширение в центральный репозторий.

В первую очередь перед публикацией расширения необходимо убедиться, что он действует должным образом. Поэтому сейчас мы снова обратимся к нашим инструментам тестирования.

16.2.1. Тестирование расширений

Проект расширения имеет структуру, практически идентичную структуре проекта приложения, а это означает, что все приобретенные вами навыки модульного и интеграционного тестирования можно применить и к расширениям. Тесты обоих типов будут полезны для проверки корректной работы артефактов; тесты пишутся так же, как и для приложения, и сохраняются в каталогах расширения test/unit и test/integration. Что недоступно ни модульным, ни интеграционным тестам, так это проверка обработчиков в дескрипторе расширения.

Следовательно, особую важность в проверке работоспособности расширения после его установки в приложение приобретает функциональное тестирование. Это единственный тип тестирования, когда расширение проверяется в окружении, близком к ожидаемому. Платформа Grails предоставляет два способа: выполнять функциональные тесты, как если бы расширение было приложением, или создавать отдельные проекты, куда устанавливать расширение.

Тестирование расширения как приложения

При использовании первого способа расширение функционального тестирования устанавливается непосредственно в проект, после чего тестирование выполняется, как обычно. Вопрос состоит в том, что тестировать? В случае расширения security нам потребуется несколько страниц, где мы могли бы организовать управление доступом и затем проверить его. Однако в самом расширении нет ничего похожего, поэтому нам придется добавить несколько испытательных контроллеров и представлений и, возможно, несколько классов предметной области. Вся хитрость состоит в том, чтобы эти дополнительные артефакты не включались в состав расширения при упаковке.

Допустим, что мы добавили в расширение классы предметной области Post и User. Мы можем исключить их из «официального» дистрибутива расширения, воспользовавшись свойством pluginExcludes в дескрипторе расширения:

```
pluginExcludes = [
    "grails-app/domain/Post.groovy",
    "grails-app/domain/User.groovy"
]
```

Этот простой прием гарантирует, что после установки расширения пользователи не обнаружат в базе данных своих приложений неожиданные таблицы Post и User.

Тестирование расширения как приложения – хороший способ, пока список исключений остается достаточно коротким, но при увеличении полноты охвата тестами этот список наверняка начнет раздуваться, как воздушный шар. Кроме того, довольно сложно проверить работу расширения при различных значениях параметров настройки, потому что в проекте может использоваться только одно значение для каждого параметра. Решить эту проблему нам поможет второй способ, упоминавшийся выше.

Использование нескольких тестовых приложений

Создание тестовых приложений для проверки расширения позволяет обеспечивать более полные наборы тестов. Например, можно создать отдельное приложение для каждого варианта настроек расширения. Проблема в том, что сам процесс упаковки и установки расширения оказывается слишком неуклюжим. Всякий раз, когда вам потребуется протестировать изменение в расширении, вы будете вынуждены упаковывать его и устанавливать в приложении. К счастью, существует иной способ: вы можете настроить приложение так, чтобы оно загружало расширение из его (расширения) каталога проекта. Просто добавьте в файл BuildConfig.groovy приложения такую строчку:

```
grails.plugin.location.security = "/path/to/plugin/dir"
```

Любые изменения в исходных файлах расширения будут доступны приложению при следующем его запуске. Обратите внимание, что при таких настройках вы не должны устанавливать расширение в приложение.

На рис. 16.3 изображены типичные настройки приложения, содержащего два установленных расширения и ссылку на каталог проекта нашего расширения security.

Теперь вы можете создать столько испытательных, сколько потребуется для проверки всех особенностей вашего расширения. Мы предпочитаем сохранять такие проекты в каталоге test/projects рядом с каталогом test/unit, где каждый проект имеет следующий параметр настройки в своем файле BuildConfig.groovy:

```
grails.plugin.location.security = "../../.."
```

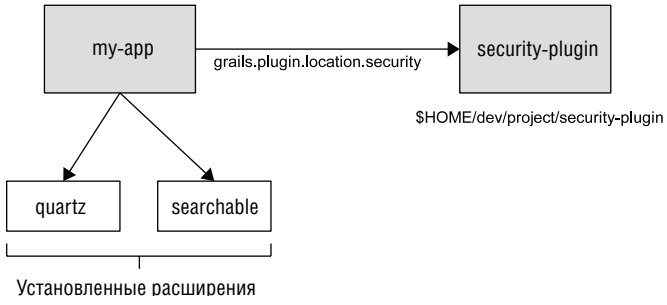


Рис. 16.3. *Приложение на платформе Grails включает одно расширение по ссылке (security-plugin) и имеет два расширения, установленные обычным способом. Каталог security-plugin содержит исходные тексты расширения*

Теперь любой сможет получить исходные тексты расширения и выполнить функциональные тесты внутри каждого испытательного проекта.

Как только все сложности тестирования остаются позади, можно начинать размышлять об упаковке и выпуске расширения.

16.2.2. Выпускаем расширение на волю

Чтобы сделать расширение доступным для всеобщего использования, его сначала необходимо упаковать в файл zip, чтобы потом можно было передать команде `install-plugin`. После упаковки расширение можно отправить либо в центральный репозиторий расширений платформы Grails либо сохранить в собственном репозитории, чтобы пользователи легко могли установить его. Механизм публикации мы обсудим чуть ниже, а пока рассмотрим процедуру упаковки.

Упаковка расширения

Упаковка – основной этап распространения программного обеспечения независимо от того, создается ли при этом выполняемый файл JAR, файл WAR или что-то иное. В случае расширения для платформы Grails в результате упаковки создается обычный файл zip, и платформа Grails предоставляет для этой цели специальную команду:

```
grails package-plugin
```

Эта команда создаст файл zip с именем grails-<name>-<version>.zip, где <name> и <version> – это имя и версия расширения.

Вы можете скопировать этот файл в общедоступную файловую систему на веб-сервере или передать его каким-либо другим, удобным для вас способом. После этого пользователи смогут забрать файл zip и установить его:

```
grails install-plugin grails-security-0.1.zip
```

За кулисами

Когда выполняется упаковка расширения, платформа Grails создает файл plugin.xml, содержащий метаинформацию (такую как версия, автор и так далее) и список артефактов и других ресурсов, предоставляемых расширением. Этот файл добавляется в файл архива zip и позднее используется платформой Grails при установке расширения и при запуске приложения.

Почему мы упомянули об этом? Потому что если файл plugin.xml будет поврежден или будет содержать информацию, не соответствующую ресурсам, упакованным в расширение, это может нарушить работу платформы Grails (самым причудливым образом). Мы не думаем, что вам когда-нибудь придется беспокоиться об этом файле, но мы понимаем, что порою такое может случиться.

Не волнуйтесь, команда `package-plugin` добавит в архив только те файлы, которые необходимы для использования расширения. Все тесты, файлы с настройками и все, что объявлено в поле `pluginExcludes` дескриптора, будет исключено. Точно узнать, какие файлы включаются и какие исключаются, можно, просмотрев содержимое переменных сценария `pluginIncludes` и `pluginExcludes` в сценарии _GrailsDev.groovy.

Хотя способ установки расширения вполне функционален, он не такой дружественный по отношению к пользователю, как следующий:

```
grails install-plugin security
```

Чтобы такая команда действовала, расширение security должно быть опубликовано в центральном репозитории расширений платформы Grails.

Публикация в центральном репозитории

Центральный репозиторий реализован на базе системы управления версиями Subversion, как и все репозитории расширений. Вы не можете сохранять в репозитории свои расширения без соответствующих привилегий, но можете запросить разрешение, выполнив шаги, описанные здесь: *http://www.grails.org/Creating+Plugins*. После получения привилегий разработчика всего один шаг будет отделять вас от передачи расширения в общее пользование. В корневом каталоге проекта выполните следующую команду:

```
grails release-plugin
```

Она добавит проект расширения в центральный репозиторий и сохранит в другом каталоге копию исходных текстов, извлеченных из репозитория. Например, если исходные тексты расширения находятся в ка-

талоге .../dev/projects/grails-security, то свежая копия из Subversion будет сохранена в каталоге .../dev/projects/checkout/grails-security.

Мы рекомендуем замещать прежний каталог проекта каталогом, извлеченным из репозитория. После того как исходные тексты расширения будут добавлены в репозиторий, ожидается, что команда `release-plugin` будет выполняться из каталога с рабочей копией проекта, полученной из репозитория Subversion (дополнительное замечание приводится во врезке). Когда эта команда будет запущена в следующий раз (например, когда будет готова версия 0.2), она обновит рабочую копию, добавив в нее все изменения, которые были отправлены в репозиторий другими разработчиками, отправит ваши изменения и затем выпустит новую версию.

Не нравится Subversion?

Несмотря на то, что команда `release-plugin` приспособлена для взаимодействия с системой управления версиями Subversion, вы можете пользоваться клиентскими приложениями других систем, таких как Git или Bazaar. Вам просто нужно вручную отправить свои изменения в репозиторий Subversion, прежде чем запускать команду.

Публикация в собственных репозиториях

Процедура, которая была описана выше, производит публикацию расширения в центральном репозитории расширений, но как быть, если расширение потребуется опубликовать в собственном репозитории? Например, возможно, ваше расширение является частной собственностью вашей компании, и оно должно быть опубликовано во внутреннем репозитории. В этом случае вам следует выполнить следующие действия:

1. Создать псевдоним для URL репозитория.
2. Передать команде `release-plugin` аргумент `--repository=<псевдоним>`.

Первый шаг связан с добавлением соответствующих строк в файл BuildConfig.groovy расширения. Например, следующие строки добавляют псевдоним `internalRepository` репозитория:

```
grails.plugin.repos.discovery.internalRepository =
    "http://.../grails-plugins/trunk"
grails.plugin.repos.distribution.internalRepository =
    "https://.../grails-plugins/trunk"
```

Обратите внимание: вы должны добавить адреса URL для обоих параметров, `discovery` и `distribution`. После добавления этих строк в файл BuildConfig.groovy выполните следующую команду, чтобы опубликовать расширение во внутреннем репозитории:

```
grails release-plugin --repository=internalRepository
```

Чтобы убедиться, что все прошло успешно, можно попробовать установить расширение в проект:

```
grails install-plugin --repository=internalRepository security
```

В данном случае аргумент `--repository` является необязательным. В его отсутствие платформа Grails выполнит поиск во *всех* настроенных репозиториях, включая центральный.

Собственные репозитории могут использоваться для хранения ваших частных расширений или изменения общедоступных расширений для своих нужд. Они могут также использоваться для контроля за тем, какие версии общедоступных расширений по умолчанию устанавливаются членами вашей компании. Если вам потребуется создать собственный репозиторий, узнать, как это делается, вы сможете в руководстве пользователя платформы Grails.

Теперь вы знаете, как опубликовать расширение security, но оно еще не готово к публикации. У нас имеется модель предметной области, а также реализация операций входа и выхода, но у нас пока отсутствует управление доступом. Чтобы добавить его, нам необходимо начать использовать обработчики расширения, упоминавшиеся выше.

16.3. Интеграция с платформой Grails

Упаковка и запуск приложений на платформе Grails связана с выполнением за кулисами большого числа операций, которые часто остаются незаметными для разработчика приложений. Однако как разработчик расширений вы должны знать о том, какие процессы протекают внутри, если вы хотите обеспечить тесную интеграцию своих расширений с платформой Grails.

В этом разделе вы узнаете, как:

- Добавлять свои собственные динамические методы, как это делают основные расширения платформы Grails
- Обрабатывать перезагрузку классов во время выполнения
- Настраивать собственные компоненты Spring
- Добавлять записи в веб-дескриптор приложения
- Реализовать дополнительные команды

Все эти действия, за исключением последнего, выполняются с помощью специальных обработчиков в дескрипторе расширения, которые были перечислены в табл. 16.1. Теперь без лишней суеты начнем работать над динамическими особенностями.

16.3.1. Расширение артефактов динамическими методами

Для управления доступом к функциональным возможностям приложения необходим своего рода привратник. Наше расширение security должно реализовать эту роль, но *как* это сделать, зависит от того, что требуется обезопасить. Это может быть доступ к определенным классам предметной области или к методам служб, но для простоты наше расширение security будет управлять доступом только к определенным адресам URL. Идеальным местом для реализации такого вида управления являются фильтры платформы Grails, потому что здесь можно предотвратить возможность доступа, просто возвращая значение `false` из обработчика `before`.

Это порождает важный вопрос: как сообщить расширению, какие адреса URL должны быть доступны только зарегистрированным пользователям, а какие требуют, чтобы пользователь обладал определенной ролью? Было бы совсем неплохо иметь нечто подобное:

```
class SecurityFilters {
    def filters = {
        all(controller: "*", action: "*") {
            before = {
                authcRequired()
            }
        }

        posts(controller: "post", action: "*") {
            before = {
                roleRequired("user")
            }
        }
    }
}
```

Идея состоит в том, чтобы дать возможность обращаться к страницам приложения только аутентифицированным пользователям (которые выполнили вход). Кроме того, доступ ко всем страницам, обслуживаемым контроллером сообщений, должны иметь только пользователи с ролью «user». Представленный способ достаточно краток, понятен и информативен. Но возможно ли реализовать его, и если да, то как?

Одна из возможностей заключается в том, чтобы реализовать методы `authcRequired()` и `roleRequired()` в абстрактном классе фильтров. Тогда любой класс, наследующий его, мог бы вызывать эти методы, как это только что было продемонстрировано на примере класса `SecurityFilters`. Однако при таком подходе возникает несколько проблем. Во-первых, языки программирования Groovy и Java не допускают множе-

ственное наследование классов, поэтому своим решением вы заставите разработчиков наследовать свой класс, ограничивая их свободу маневра. Во-вторых, появляется явная зависимость от вашего расширения, а это означает, что класс фильтров не будет компилироваться, пока не будет установлено расширение. Это прием *навязывания*.

Неужели нет других способов? Конечно, есть! Вспомните, что классы фильтров написаны на языке Groovy, поэтому мы можем использовать ненавязчивый способ, с которым вы уже сталкивались: *внедрение динамических методов*. Мы можем внедрить методы `authcRequired()` и `roleRequired()` во все классы фильтров, не вынуждая разработчиков изменять ни строчки программного кода.

Платформа Grails позволяет использовать этот прием, предоставляя расширениям специальный обработчик, где они могут добавлять любые динамические методы и свойства к любым классам. Все, что требуется от расширения, – это реализовать замыкание `doWithDynamicMethods`. Вы должны знать, *как* добавлять такие методы, в противном случае от вашего замыкания не будет никакой пользы.

Сейчас самое подходящее время прочитать раздел главы 2, посвященный динамическим методам, если вы этого еще не сделали. Он позволит вам получить некоторое представление о том, что такое динамические методы и свойства и как они действуют. В случае с замыканием `doWithDynamicMethods` динамические методы и свойства добавляются с помощью приема, который называется *программированием метаклассов*.

Программирование метаклассов

В языке Groovy каждый класс связан с метаклассом, который определяет поведение этого класса во время выполнения. Когда в языке Groovy производится вызов метода, он передается метаклассу, который выполняет выбор вызываемого метода. Именно поэтому динамические методы добавляются в метакласс, а не в действующий класс, и по этой же причине у вас имеется возможность расширять классы JDK. Обычно эта сторона вещей невидима для вас как для программиста, но она выходит на передний план, как только вы вступаете в мир программирования метаклассов.

Однако достаточно теории, опробуем этот прием на практике! Синтаксис определения динамического метода достаточно прост:

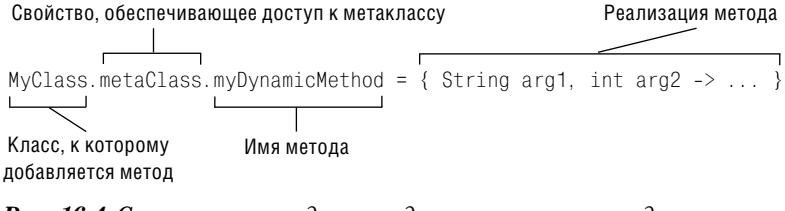


Рис. 16.4. *Синтаксис определения динамического метода*

Объявление начинается с имени класса, к которому требуется добавить метод. В этом примере мы использовали литерал класса, но можно было бы использовать также переменную, ссылающуюся на класс. Затем выполняется доступ к метаклассу с помощью свойства `metaClass` и добавляется определение метода с использованием замыкания.

Добавление свойства выполняется аналогичным способом, потому что в этом случае необходимо добавить методы чтения и записи:

```
MyClass.metaClass.getMyProp = {-> ... }
MyClass.metaClass.setMyProp = { String newValue -> ... }
```

Вы можете реализовать метод записи, но вам необходимо где-то хранить новое значение, при этом желательно обеспечить безопасность при работе со значением из нескольких потоков выполнения. Нельзя добавить новое поле прямо в класс, так что его реализация может оказаться весьма хитрым делом. По этой причине динамические свойства часто доступны только для чтения – определен только метод чтения.

Обратите внимание, что замыкание, реализующее метод чтения, явно объявляет нулевое количество аргументов. Следующее объявление *не* будет действовать как свойство:

```
MyClass.metaClass.getMyProp = { ... }
```

В этом случае вы получите метод `getMyProp()`, который принимает единственный необязательный аргумент.

Теперь вас может заинтересовать вопрос, как получить доступ к экземпляру класса, относительно которого был произведен вызов динамического метода. Допустим, что у нас имеется класс, содержащий коллекцию элементов, и вам требуется добавить свойство, возвращающее их количество. Как бы то ни было, но реализации вашего метода необходимо получить доступ к этому списку. Как вы могли догадаться, язык Groovy уже позаботился об этом: замыкание имеет доступ к свойству `delegate`, которое является ссылкой на экземпляр класса. Учитывая это обстоятельство, реализация метода могла бы выглядеть, как показано ниже:

```
MyClass.metaClass.getTotal = {-> return delegate.items.size() }
```

В действительности все замыкания имеют свойство `delegate`, но его содержимое зависит от контекста.

Это были самые основы. Нам осталось лишь продемонстрировать вам пару синтаксических конструкций для работы с метаклассами, которые могут оказаться полезными для вас:

```
MyClass.metaClass.static.myMethod = { ... }
MyClass.metaClass.myMethod << { ... }
```

Первая из них позволяет добавлять статические методы класса. В этом случае свойство `delegate` в замыкании будет ссылаться на сам класс. Во второй форме вместо оператора присваивания (=) используется опера-

тор сдвига влево (<<). Он действует так же, но если метод с таким именем уже существует, платформа Grails возбудит исключение. Если вы планируете переопределять существующие методы, используйте оператор =; в противном случае – оператор <<, чтобы при запуске приложения дать пользователю знать о конфликте имен.

Как видите, программирование метаклассов в своей основе не содержит ничего сложного. Однако реализация методов может оказаться весьма непростым делом, особенно когда дело доходит до разрешения имен. В целом же, все, что, по вашему мнению, должно работать, будет работать; но по мере углубления в программирование метаклассов вы неизбежно столкнетесь с некоторыми проблемами, и тогда, вероятно, стоит прочитать книгу, посвященную языку программирования Groovy, где описываются особенности разрешения имен в Groovy, в частности в замыканиях.

В самом начале, когда вы только приступаете к программированию метаклассов, у вас может появиться искушение использовать этот прием повсюду. Этот прием кажется новым, захватывающим и, как говорится, «когда у вас в руках оказывается молоток, все окружающие предметы становятся похожи на гвозди». Но будьте сдержанны. Как упоминалось в главе 2, динамические возможности языка Groovy могут быть неудобными в отладке, и они осложняют жизнь интегрированным средам разработки. Не забывайте, что существуют другие варианты: статические вспомогательные методы, службы Grails, наследование и так далее. Но если вы думаете, что ясность и простота в понимании (и написании) программного кода перевешивают недостатки, тогда используйте динамические особенности.

Внимание

Будьте осторожны, принимаясь за модификацию стандартных классов JDK и других общих классов, особенно, если вы планируете переопределить существующие методы или свойства. Изменения будут доступны всему программному коду на языке Groovy во всех потоках выполнения, поэтому даже самое незначительное изменение может привести к нежелательным последствиям в других частях программного кода, и причины таких ошибок будет очень трудно отыскать.

Теперь вы знаете, как реализовать динамические методы, но к каким классам их следует добавить? В случае нашего расширения security нам требуется добавить их ко всем фильтрам приложения, но как это сделать? Мы не знаем заранее, какие классы фильтров будут использоваться в приложении.

Поиск информации об артефактах в приложении

Представьте себе, что произойдет, когда расширение будет установлено в приложение на платформе Grails. Пока вы пишете его, вы знаете только о тех классах, что определяются в самом расширении, но как толь-

ко расширение будет установлено в приложение, оно внезапно окажется в среде, содержащей большее количество классов, чем оно использует, – в окружении классов, предоставляемых другими установленными расширениями, и классов, являющихся частью самого приложения. Существует ли способ определить, какие еще классы имеются, не обращаясь к загрузчику классов? Действительно, такой способ существует.

У программного кода в любом из обработчиков, перечисленных в табл. 16.1, имеется доступ к свойству с именем `application`. Это свойство представляет экземпляр приложения на платформе Grails, и мы можем использовать его для поиска доступных артефактов. Помните динамические методы поиска механизма GORM? Экземпляр приложения имеет похожие динамические методы и свойства, но в данном случае они опираются на типы доступных артефактов, а не на поля в классах предметной области.

Допустим, что вам требуется отыскать все контроллеры в приложении. Сделать это просто:

```
application.controllerClasses
```

Вы могли бы также отыскать все службы, обратившись к свойству `application.serviceClasses`. Фактически аналогичные операции можно выполнять для всех типов артефактов. Доступные для этого методы и свойства перечислены в табл. 16.2.

В табл. 16.2 вы увидите упоминание *дескрипторов артефактов*. Каждый артефакт в приложении имеет свой дескриптор, содержащий дополнительную информацию о нем; ключевые свойства дескрипторов перечислены в табл. 16.3.

Таблица 16.2. Средства получения информации об артефактах приложения

Шаблон метода	Описание
`<artifact>Classes`	Возвращает все дескрипторы артефактов приложения с типом `<artifact>`. Обратите внимание, что имя типа артефакта должно начинаться с маленькой буквы, потому что это свойство.
`get<Artifact>Class-(String artifactName)`	Возвращает дескриптор артефакта с типом `<Artifact>` для артефакта с именем `artifactName`. Например, вызов `getControllerClass("LoginController")` вернет дескриптор артефакта `LoginController`. Обратите внимание, что `artifactName` должно быть полным именем класса, включающим имя пакета, если класс определяется в пакете.
`is<Artifact>Class-(Class artifactClass)`	Возвращает `true`, если класс имеет тип `<Artifact>`, в противном случае возвращается значение `false`. Например, вызов `isControllerClass(LoginController)` вернет значение `true`.

Таблица 16.3. Свойства дескрипторов артефактов

Свойство	Тип	Описание	Пример
clazz	Class	Класс артефакта	org.example.LoginController
metaClass	MetaClass	Обеспечивает доступ к метаклассу артефакта; то же, что и clazz.metaClass	-
name	String	Имя артефакта, за исключением любых окончаний и имени пакета	Login
shortName	String	Имя артефакта, за исключением имени пакета	LoginController
propertyName	String	Краткое имя артефакта в виде свойства	LoginController

За кулисами

Динамические методы артефактов реализуются классом DefaultGrailsApplication с использованием метода invokeMethod(). Они отображаются на фактические методы, такие как getArtefacts(), а это означает, что у вас есть возможность получать ту же информацию из программного кода на языке Java, если это потребуется.

Дескрипторы артефактов являются экземплярами интерфейса GrailsClass. Имеет смысл поближе познакомиться с его методами, чтобы иметь возможность создавать более сложные расширения.

Вернемся к расширению security. Мы говорили, что хотим добавить в фильтры пару динамических методов, но фильтры – это особый случай. Поэтому сначала мы покажем, как добавить метод hasRole() во все контроллеры приложения. В действительности мы будем добавлять метод только к тем контроллерам, в которых будет объявлено статическое свойство с именем secure и значением true. Результат приводится в листинге 16.2.

Листинг 16.2. Внедрение метода в контроллеры

```
package com.grailsinaction.security

import org.codehaus.groovy.grails.commons.GrailsClassUtils as GCU
...
class SecurityGrailsPlugin {
    ...
    def doWithDynamicMethods = {
```

❶ Импорт вспомогательного класса

```
        application.controllerClasses.each { cClass ->  ← Обход контроллеров
            def isSecure = GCU.getStaticPropertyValue(  ← ❷ Получение
                    cClass.clazz,                           значения свойства
                    "secure")                               secure
            if (isSecure) {
                cClass.metaClass.hasRole = { name ->  ← ❸ Добавляет метод
                    def u = session.user                  в контроллер
                    if (u) {
                        u.merge()  ← Повторное подключение к сеансу Hibernate
                        return u.roles.any { it.name == name }
                    }
                    else {
                        return false
                    }
                }
            }
        }
        ...
    }
    ...
}
```

`GrailsClassUtils` ❶ – удобный класс для авторов расширений, и мы используем его здесь для чтения значения статического свойства `secure` ❷. Получив значение свойства, мы проверяем его и добавляем метод `hasRole()` ❸ только к тому контроллеру, в котором этому свойству присвоено значение `true`. Данный динамический метод возвращает `true`, только если пользователь обладает указанной ролью.

Это было достаточно просто, а что можно сказать о добавлении методов к фильтрам? В отличие от большинства других артефактов, экземпляр фильтра не имеет тот же тип, что и класс `*Filters`, который добавляется в определение. Например, контроллер сообщений является экземпляром класса `PostController`, но фильтр безопасности не является экземпляром класса `SecurityFilters`. В действительности все фильтры являются экземплярами класса `FilterConfig`, поэтому добавление динамических методов к ним выглядит очень просто:

```
import org.codehaus.groovy.grails.plugins.web.filters.FilterConfig
...
FilterConfig.metaClass.authcRequired = {-> session.user != null }
```

Предыдущая строка должна находиться внутри замыкания `doWithDynamicMethods` – мы опустили остальную часть класса, чтобы не загромождать описание лишним кодом.

Теперь расширение security может блокировать доступ к страницам приложения дружественным по отношению к пользователю способом (полный программный код реализации вы найдете в исходных текстах примеров к главе 16). И всего лишь за счет добавления нескольких строк кода!

Мы практически закончили, но нам осталось позаботиться еще об одном. Что произойдет, если разработчик изменит класс контроллера в процессе работы приложения?

16.3.2. Обработка перезагрузки класса

Возможность платформы Grails перезагружать измененные файлы приложения во время выполнения является одним из самых заметных достоинств. К сожалению писателей расширений, пользователи ожидают, что расширения будут учитывать эту возможность и использовать ее в своих интересах. Например, при перезагрузке контроллера все динамические методы контроллера будут утрачены, поэтому на расширение возлагается обязанность восстановить такие методы. В этом нам помогут замыкания `onChange` и `onConfigChange`.

Платформа Grails поддерживает списки так называемых *отслеживаемых ресурсов*, напоминающие шаблоны путей к файлам, которые вы могли использовать вместе с инструментом сборки Ant. Всякий раз, когда изменяется файл, соответствующий шаблону, платформа Grails сначала перезагружает его (если он содержит определение класса или сценарий на языке Groovy), а затем извещает все заинтересованные расширения вызовом их замыканий `onChange`.

Расширения ядра и их имена

Большая часть функциональных возможностей платформы Grails реализована в виде расширений, которые упакованы в файлы JAR платформы. Мы называем их *расширениями ядра*. Существует также расширение с именем `core` (ядро), но если мы и будем ссылаться на него, мы обязательно отметим это явно. Примерами расширений ядра могут служить controllers, domainClass, i18n и services.

Очевидно, что первым шагом наше расширение должно обозначить свою заинтересованность в подобных извещениях. Существует два способа сделать это: организовать «наблюдение» за существующим расширением или определить список отслеживаемых ресурсов. Допустим, вам требуется, чтобы платформа Grails извещала вас о любых изменения в контроллерах. Поскольку расширение ядра controllers уже отслеживает изменения в контроллерах, наблюдение за этим расширением означает, что ваше расширение тоже будет извещаться об этих изменениях. В действительности ваше расширение будет получать те же извещения, что и расширение, за которым установлено наблюдение, поэтому вам следует знать, например, что расширение controllers также наблюдает за изменениями в библиотеках тегов.

Организовать наблюдение за расширением очень просто, достаточно лишь добавить в дескриптор расширения свойство `observe`, как показано ниже:

```
def observe = ["controllers"]
```

Каждый элемент списка – это имя расширения, за которым ведется наблюдение. Если какое-либо из этих расширений не установлено в приложении, оно просто будет игнорироваться. В табл. 16.4 приводится список наиболее интересных расширений и ресурсов, изменения в которых они отслеживают.

Таблица 16.4. Файлы, за изменениями в которых следят некоторые расширения ядра

Имя расширения	Отслеживаемые ресурсы
`controllers`	Контроллеры и библиотеки тегов
`services`	Службы
`filters`	Классы фильтров
`urlMappings`	Отображения адресов URL
`hibernate`	Файлы с параметрами настройки Hibernate
`i18n`	Пакеты ресурсов i18n (например, `messages.properties`)

Другой способ заключается в том, чтобы определить свойство `watchedResources`, содержащее один или более шаблонов имен файлов, которые определяют файлы, представляющие интерес для расширения. Например, чтобы организовать слежение за контроллерами, мы могли бы добавить такое определение:

```
def watchedResources = "file:./grails-app/**/*Controller.groovy"
```

или такое:

```
def watchedResources = ["file:./grails-app/**/*Controller.groovy"]
```

Прежде чем перейти непосредственно к обработке извещений, взгляните на пример шаблона файла, изображенный на рис. 16.5.

***Рис. 16.5.** Определение шаблона файла*

Схему указывать необязательно; она определяет порядок доступа к файлу. Вторая часть – это шаблон пути к файлу, который оформляется в соответствии с соглашениями, принятыми в инструменте сборки Ant. Шаблону «двойная звездочка» (`**`) соответствует любое количество вложен-

ных подкаталогов (в том числе и нулевое), тогда как одиночная звездочка (*) может использоваться для определения соответствия одному уровню вложенности каталогов или любой части имени файла или каталога. Например, шаблон, представленный на рис. 16.5, кроме прочего будет соответствовать файлам ./grails-app/MyController.groovy и ./grails-app/org/example/MyOtherController.groovy в числе других.

Если вам действительно придется использовать свойство `watchedResources`, мы рекомендуем вам всегда использовать схему `file:` и начинать пути к файлам с символов `./`. За дополнительной информацией об оформлении шаблонов путей к файлам обращайтесь к разделу о ресурсах в справочном руководстве для платформы Spring.

Теперь, когда вы знаете, как организовать получение извещений об изменениях, рассмотрим, как их обрабатывать. Вашему замыканию `onChange` будет передан объект события со следующими свойствами:

- `source` – Изменившийся файл или артефакт. Если изменился класс, свойство `source` будет представлять экземпляр класса (`java.lang.Class`); в противном случае это будет экземпляр ресурса Spring (`org.springframework.core.io.Resource`).
- `application` – Экземпляр приложения на платформе Grails, который удобно использовать для получения информации об артефактах. Это свойство можно считать избыточным, так как замыкание `onChange` уже имеет доступ к динамическому свойству `application`.
- `ctx` – Контекст приложения на платформе Spring, с помощью которого можно получить доступ к любым компонентам.
- `manager` – Администратор расширения.
- `plugin` – Экземпляр дескриптора расширения.

Из всех этих свойств наибольший интерес для нас представляет свойство `source`. Вооруженные этой информацией, вы сможете сделать все, что необходимо, чтобы обеспечить нормальную работу расширения.

В листинге 16.3 мы вновь вернулись к расширению security и реализовали повторное добавление динамического метода `hasRole()` к любым контроллерам, которые изменились, потому что динамические свойства и методы утрачиваются при перезагрузке класса.

Листинг 16.3. Реакция на изменение контроллера

```
class SecurityPlugin {
    ...
    def observe = ["controllers"] ← Наблюдение за расширением controllers
    ...
    def onChange = { event ->
        if (application.isControllerClass(
                event.source)) { ← ❶ Проверяем, что свойство source
                                    представляет контроллер
```

```
                event.source.metaClass.hasRole =
                        { String name ->          <-- Повторно добавляет
                    ...                               метод в контроллер
                }
            }
        }
    }
```

Здесь мы организовали наблюдение за расширением controllers, поэтому нам необходимо игнорировать изменения в библиотеках тегов. Мы делаем это с помощью объекта, представляющего приложение на платформе Grails, определяя тип изменившегося артефакта ❶. Метод `isControllerClass()` является динамическим, так что в нашем распоряжении имеются похожие методы для всех зарегистрированных типов артефактов. Например, вы могли бы использовать метод `isDomainClass()`.

Этот пример достаточно прост, но теперь вы в состоянии реализовать обработку большинства случаев перезагрузки класса или файла. Следует также помнить, что замыкание `onChange` вызывается не только при изменении артефактов, но и при создании нового артефакта. Обычно это не сказывается на вашем программном коде, но на всякий случай вы должны знать об этой особенности.

Мы рассмотрели проблему обработки изменений в артефактах, но как быть с изменениями в файлах с настройками? К счастью, реализовать обработку этих изменений достаточно просто. Все, что от вас требуется, – это добавить замыкание `onConfigChange`, которое принимает в качестве аргумента объект события. Вот и все – вам даже не придется указывать отслеживаемые ресурсы или устанавливать наблюдение за определенным расширением. Основное отличие заключается в том, что свойство `source` объекта события в этом случае представляет новый объект с настройками, который можно использовать для доступа к любым необходимым настройкам обычным способом.

Кода же желательно добавлять поддержку перезагрузки в свое расширение? Вам определенно следует делать это тогда, когда расширение добавляет в артефакты динамические свойства или методы; в противном случае эти методы растают как облако дыма после перезагрузки. Другой типичный случай – когда расширение читает значения из артефакта, например из свойства. Разработчик может изменить это значение, поэтому расширение должно гарантировать, что новое значение тут же вступит в силу. Кроме этого нет никаких жестких правил. Все сводится к тому, чтобы определить, оказывают ли изменения в артефактах какое-либо влияние на поведение вашего расширения.

У нас уже имеется действующее расширение security, которое способно управлять доступом к страницам приложения, и все же мы использовали лишь часть возможностей, доступных в расширениях. Вы не должны стремиться реализовать все подряд только потому, что вы это

можете, и на практике большинство расширений не нуждаются или не используют все пункты интеграции. Тем не менее мы должны рассмотреть остальные особенности, поэтому далее мы добавим в наше расширение некоторые дополнительные функциональные возможности.

16.3.3. Использование платформы Spring

Помните, как мы настраивали компоненты Spring с помощью файлов resources.* в главе 14? Нечто подобное можно делать и с расширениями. Рассмотрим, как в настоящее время расширение сохраняет пароли: оно преобразует текстовые пароли в хеши по жестко заданному алгоритму SHA-1. Было бы неплохо предоставить пользователям возможность самим выбирать алгоритм хеширования, но для этого нам потребуется заняться хранением паролей и сравнением их в процессе процедуры аутентификации.

В типичном приложении на языке Java соответствующее решение заключалось бы в том, чтобы создать интерфейс, реализующий сохранение и сравнение паролей. Различные реализации этого интерфейса могли бы использовать различные алгоритмы хеширования. Ниже приводится пример, как это может выглядеть:

```
public interface PasswordHandler {
    String encode(String plain)
    boolean passwordsMatch(String plain, String encoded)
}
```

Метод `encode()` преобразует пароль из текстовой формы представления в хеш, а метод `passwordsMatch()` сравнивает простой текстовый пароль с его зашифрованной версией.

Теперь нам осталось реализовать способ настройки обработчика паролей по умолчанию с алгоритмом хеширования по умолчанию. В этом нам поможет замыкание `doWithSpring`. В данном замыкании вы можете определять компоненты, используя язык Spring DSL, как показано в листинге 16.4.

Листинг 16.4. Простой пример определения на языке Spring DSL

```
import security.HashPasswordHandler
import security.Sha1Hasher
...
def doWithSpring = {
    hashMachine(Sha1Hasher) {      ← Настройка алгоритма хеширования по умолчанию
        salt = "jfhw4h942hvkb2jh"
    }

    passwordHandler(HashPasswordHandler) {
        hasher = ref("hashMachine") ← Связывание двух компонентов
        iterationCount = 5
    }
}
```

В этом примере мы определили компонент, занимающийся обработкой паролей (passwordHandler), и настроили его на использование алгоритма хеширования SHA-1 при шифровании паролей. Но как пользователь сможет переопределить алгоритм хеширования в этом примере?

За кулисами

Вы можете использовать язык Spring DSL потому, что делегатом для замыкания doWithSpring является экземпляр grails.spring.BeanBuilder. Все остальные обработчики (такие как doWithDynamicMethods) выполняются с делегатом: org.codehaus.groovy.grails.plugins.DefaultGrailsPlugin.

Когда запускается платформа Grails, она вызывает все замыкания doWithSpring всех установленных расширений и заполняет контекст приложения предоставленными определениями компонентов. После того как будут обработаны все расширения, платформа Grails загружает определения компонентов из файлов resources.xml и resources.groovy. Чтобы задействовать иной алгоритм хеширования, пользователю достаточно определить компонент с именем hashMachine в одном из этих файлов. Например, его можно определить в файле resources.groovy:

```
beans = {
    hashMachine(security.Md5Hasher)
}
```

Если компонент, определяемый пользователем, будет иметь то же имя, что и компонент, определяемый в замыкании doWithSpring, пользовательский компонент переопределит компонент в замыкании. Все очень просто!

Мы практически закончили разбираться с этой темой, и нам осталось лишь объяснить назначение замыкания doWithApplicationContext. Оно вызывается платформой Grails после того, как все расширения получат возможность добавить свои определения компонентов и будет настроен контекст приложения. В единственном аргументе замыканию передается сам контекст приложения, полностью инициализированный и готовый к использованию. Данное замыкание не часто реализуется расширениями, но если вам когда-либо потребуется выполнить некоторые операции с контекстом приложения и его компонентами (кроме определения компонентов), этот обработчик именно то, что вам нужно.

Последние обработчики уводят нас от платформы Spring и переносят в мир сервлетов и фильтров сервлетов, которые составляют важную часть веб-приложения.

16.3.4. Экспериментируем с сервлетами и фильтрами

Платформа Grails в значительной степени ориентирована на разработку веб-приложений, и хотя приложение на платформе Grails легко можно написать, даже не задумываясь о веб-дескрипторе, не стоит все-таки о нем забывать. Помните: это единственный способ настроить сервлеты и фильтры сервлетов.

В расширении security, разрабатываемом в этой главе, нет никакой необходимости настраивать сервлеты или фильтры сервлетов, поэтому в демонстрационных целях мы обратимся к одному из существующих расширений для платформы Grails. Расширение JSecurity обеспечивает приложения на платформе Grails возможностью управления доступом, и так получилось, что ему требуется настроить фильтр сервлета. Мы могли бы вынудить пользователя вручную настраивать этот фильтр, но тогда расширение быстро потеряло бы своих пользователей. Поэтому оно самостоятельно настраивает фильтр.

Чтение документа XML, его модификация и последующее сохранение может быть весьма утомительным занятием, поэтому мы рады сообщить, что вам не придется этого делать. Вам просто нужно реализовать замыкание `doWithWebDescriptor` и изменить представление веб-дескриптора, находящееся в памяти. Как это сделать? Сначала платформа Grails выполняет парсинг шаблона веб-дескриптора, используя класс `XmlSlurper`, а затем результат передается замыканию `doWithWebDescriptor` расширения в виде аргумента. Используя его, мы можем изменить веб-дескриптор.

Мы знаем, что аргумент, передаваемый замыканию `doWithWebDescriptor`, является результатом парсинга шаблона веб-дескриптора. Мы также знаем, как пользоваться этим объектом, чтобы получить необходимую нам информацию. Но как мы сможем изменить документ XML? Мы должны сделать это каким-либо способом, если собираемся настроить фильтр сервлета. Объект с результатами парсинга документа XML предоставляет такую возможность, и в листинге 16.5 демонстрируется, как выполняется настройка фильтра JSecurity.

Листинг 16.5. Настройка фильтра сервлета во время выполнения

```
def doWithWebDescriptor = { webXml ->
    def contextParam = webXml.'context-param' ← ❶ Извлекает все элементы <context-param>

    contextParam[contextParam.size() - 1] + { ← ❷ Добавляет новые настройки фильтра
        'filter' {
            'filter-name'('securityContextFilter')
            'filter-class'('org.jsecurity.spring.SpringJSecurityFilter')
            'init-param' {
                'param-name'('securityManagerBeanName')
                'param-value'('jsecSecurityManager')
            }
        }
```

```
    }

    def filter = webXml.'filter-mapping'.find {                    ← ❸ Отыскивает
        it.'filter-name'.text() == 'charEncodingFilter'              определенное
    }                                                                отображение фильтра

    filter + {
        'filter-mapping' {
            'filter-name'('securityContextFilter')
            'url-pattern'('/*')
        }
    }
}
```

Порядок следования определений фильтра не имеет значения, поэтому расширение JSecurity добавляет свой фильтр сразу же за последним элементом `context-param` ❶ и ❷. Вы заметили, что имя элемента заключено в кавычки ❶? Это обусловлено тем, что имена методов и свойств не могут содержать дефисы, так что мы воспользовались свободой, предоставляемой языком Groovy, использовать строки вместо имен. Переменная `contextParam` содержит список элементов, поэтому для получения последнего элемента мы использовали синтаксис обращений к элементам массива. Почему мы не использовали индекс -1? По некоторым причинам этот прием не действует с классом `XmlSlurper`, и нам пришлось воспользоваться старомодным способом.

Ключевая особенность, обеспечивающая модификацию документа XML, находится чуть ниже. Как только у нас появляется элемент, который требуется добавить в конец содержимого, мы можем использовать оператор сложения (+) с замыканием ❷. Содержимое замыкания достаточно очевидно, если вы знакомы синтаксисом определения фильтров в веб-дескрипторах. Имя каждого метода соответствует имени элемента, а новое замыкание отмечает начало вложенных элементов. На практике это определение эквивалентно следующему фрагменту:

```
<filter>
    <filter-name>securityContextFilter</filter-name>
    <filter-class>org.jsecurity.spring.SpringJSecurityFilter</filter-class>
    <init-param>
        <param-name>securityManagerBeanName</param-name>
        <param-value>jsecSecurityManager</param-value>
    </init-param>
</filter>
```

Хотя имя `filter` в листинге заключено в кавычки, они не являются обязательными. Кавычки использованы лишь для того, чтобы обеспечить непротиворечивость внешнего вида элементов.

Мы говорили выше, что порядок следования определений фильтров не имеет значения, но это не относится к *отображениям* фильтров. Порядок их определения соответствует порядку, в каком они будут выпол-

няться. По этой причине нам требуется быть немного более внимательными к тому, куда вставляются отображения. В строке ❸ мы отыскиваем отображение фильтра `charEncodingFilter`, который практически гарантированно будет присутствовать там. Почти все фильтры должны следовать за ним, что и делает программный код в листинге с отображением фильтра JSecurity. Единственный случай, когда такой подход не будет действовать, – когда пользователь создаст свой шаблон веб-дескриптора и удалит из него отображение `charEncodingFilter`. В таких ситуациях лучше всего будет вставить свое отображение фильтра в начало списка (в элемент с индексом 0).

> **А где другие фильтры?**
>
> Платформа Grails реализует несколько собственных фильтров сервлетов. Один используется для отображения адресов URL, а другой, кроме всего прочего, добавляет некоторую дополнительную информацию в запрос. Можно, конечно, попробовать, но вам не удастся отыскать их определения в в веб-дескрипторе, передаваемом замыканию. Все, что добавляется расширением, недоступно расширениям, загружаемым позже.
>
> Это несколько ограничивает ваши возможности по управлению местоположением фильтров в цепочке, но не все потеряно. При разумном использовании свойства `loadAfter` вы сможете гарантировать, что ваши фильтры будут следовать за некоторыми фильтрами ядра. К сожалению, в настоящее время нет возможности вставлять один элемент перед другим.

Как только вы начнете разбираться в синтаксисе парсера XML, вы фактически овладеете замыканием `doWithWebDescriptor`. Нам нечего больше добавить к сказанному. Вы сможете добавлять параметры контекста, устанавливать предельное время существования сеанса, настраивать сервлеты, то есть выполнять все операции, поддерживаемые веб-дескриптором.

К настоящему моменту мы рассмотрели все точки интеграции в дескрипторе расширения. Их не так много, но они довольны мощны, и мы надеемся, что вы самостоятельно сможете определить еще целую массу возможностей, которые они могут предложить. Последняя точка интеграции, которую мы рассмотрим, находится в каталоге scripts.

16.3.5. Добавление новых команд Grails

До настоящего момента мы рассматривали способы, которыми расширение может оказывать влияние на поведение приложения Grails во время выполнения, а что можно сказать об этапе разработки? В предыдущей главе вы видели, что существует возможность расширения

набора команд платформы Grails за счет создания своих сценариев для вашего приложения; то же можно реализовать и в своем расширении. Платформа Grails упаковывает сценарии расширения так, что после его установки соответствующие команды становятся доступны приложению.

Какие команды можно реализовать в расширениях? Практически любые, какие только существуют под солнцем. Можно реализовать команды для запуска тестов, для изменения структуры базы данных, для создания файлов приложения из шаблонов, для выполнения специальных видов компиляции и много другого. Например, в листинге 16.6 приводится сценарий, создающий контроллеры и представления интерфейса управления пользователями, которые взаимодействуют с нашим расширением security.

Листинг 16.6. Сценарий, создающий интерфейс управления пользователями для расширения security

```
includeTargets << grailsScript("_GrailsEvents")

target(installMgmtUI: "Installs the user management UI") {
    depends(checkVersion)

    templateDir = "${securityPluginDir}/src/templates" <-- ❶ Получает местоположение шаблонов

    ant.copy(todir: "${basedir}/grails-app/controllers") {
        fileset(dir: "${templateDir}/controllers", includes: "*.groovy")
    }

    ant.copy(todir: "${basedir}/grails-app/views") {
        fileset(dir: "${templateDir}/views", includes: "**/*.gsp")
    }
}

setDefaultTarget("installMgmtUI")
```

Здесь нет ничего сложного. Этот сценарий копирует шаблоны контроллера и представления из каталога расширения в каталог приложения. Важно обратить внимание на то, как определяется местоположение шаблонов ❶. Платформа Grails добавляет в сценарий переменную с именем `<pluginName>PluginDir` для связи с каждым расширением, установленным в приложение. Она содержит абсолютный путь к каталогу, куда установлено расширение, поэтому для определения местоположения шаблонов мы можем использовать переменную `securityPluginDir`. При желании можно даже определить каталог установки другого расширения, хотя такая потребность возникает достаточно редко.

Вдобавок к вашим собственным сценариям для каждого расширения создаются три внутренних сценария, готовых к использованию: _Install.groovy, _Uninstall.groovy и _Upgrade.groovy. Первый автоматически вызывается при установке расширения в приложение. Второй вы-

зывается при удалении расширения. И последний вызывается при обновлении приложения с одной версии платформы Grails до другой командой `grails upgrade`. Этот последний сценарий в значительной степени избыточен, потому что он не имеет никакого отношения к обновлению самого расширения. Сценарии составляют неотъемлемую часть разработки расширения и могут существенно облегчить жизнь пользователям расширения. А так как одним из этих пользователей можете быть вы сами, стоит потратить некоторое время на воплощение каких-либо идей в виде сценариев.

К настоящему моменту мы представили вам все основные особенности расширений и показали, как можно использовать их для расширения возможностей платформы Grails. Это было трудное путешествие, но вы сделали это. Теперь можно отдышаться, пока мы коротко напомним обо всем, что вы узнали в этой главе.

16.4. В заключение

Расширения – сложная тема, потому что они в состоянии оказывать ощутимое влияние практически на все аспекты платформы Grails. Но они являются настолько фундаментальной частью платформы Grails, что вам необходимо понять, как они действуют, если вы хотите использовать ее потенциал полностью. Даже в самых простых приложениях, скорее всего, потребуется использовать одно или более расширений.

На протяжении книги вы познакомились со множеством расширений, но это лишь малая часть из того, что доступно. Их разнообразие демонстрирует, насколько гибкой и мощной системой расширений обладает платформа Grails. Общедоступные расширения могут служить также источником вдохновения и идей.

Используете ли вы расширения для реализации модульной архитектуры приложения или для упаковки отдельных функциональных возможностей, в любом случае они могут существенно упростить процесс разработки приложений для пользователей расширений и способствовать многократному использованию программного кода. В этой главе мы показали, насколько просто начать разработку расширений, а также познакомили вас с некоторыми инструментами, способными помочь в дальнейшей их разработке.

В качестве учебного материала вы можете также использовать другие расширения. Например, расширение JSecurity имеет весьма интересный дескриптор расширения, в котором реализованы многие обработчики. Кроме того, в его состав входят несколько сценариев, с которыми вы можете ознакомиться. Расширение GWT имеет замечательный набор сценариев, в которых используются более сложные приемы программирования, чем в расширении JSecurity.

В заключение приведем некоторые рекомендации по проектированию и разработке расширений:

- *Стремитесь к простоте*. При реализации собственных расширений используйте минимальное количество возможностей и обработчиков. Например, если нет смысла добавлять динамические свойства или методы, то и не делайте этого.

  Важно понять также, что расширения не должны быть сложными и громоздкими. Они могут быть чрезвычайно простыми и легкими в разработке. Не забывайте, что расширение – это всегда независимый программный код, который может использоваться несколькими проектами независимо от того, насколько он тривиален.

- *Реализуйте обработчик* `onChange`, *когда это необходимо*. Если ваше расширение добавляет динамические методы или свойства, обязательно реализуйте обработчик `onChange`, чтобы не потерять эти методы и свойства после перезагрузки. Пользователи весьма тесно привязаны к возможности перезагрузки классов, поэтому расширения, которые не поддерживают ее, не пользуются популярностью.

- *Для нужд своего коллектива используйте локальные репозитории расширений*. Локальные репозитории расширений служат нескольким целям. Во-первых, они позволяют контролировать, какие версии расширений по умолчанию устанавливаются членами вашего коллектива. Во-вторых, они позволяют модифицировать общедоступные расширения для своих нужд. Наконец, они могут быть полезны для организации модульной архитектуры приложений.

- *Используйте модульную архитектуру для крупных или сложных приложений*. Шаблон разделения сфер влияния (Separation of Concerns) обеспечивает широкие возможности, а расширения позволяют применять его в разработке приложений на платформе Grails. Для управления модульной архитектурой приложения можно использовать механизм подключения расширений (с помощью параметра настройки `grails.plugin.location.*`) или локальный репозиторий расширений.

- *Пишите функциональные тесты для своих расширений*. Для обеспечения надежности расширений вы должны писать функциональные тесты. Для этого особенно удобно использовать механизм подключения расширений, который позволяет создавать множество проектов испытательных приложений, загружающих расширение их исходных текстов.

Алфавитный указатель

Цифры и символы

1:1, отношения, 107, 123
1:m, отношения, 107, 123, 127
200, код состояния, 448, 449
201, код состояния, 449
301, код состояния, 449
302, код состояния, 449
400, код состояния, 449
401, код состояния, 449
403, код состояния, 449
404, код состояния, 448, 449
500, код состояния, 449% , шаблонный символ, 157, 168
*, шаблонный символ, 618
**, шаблонный символ, 617
${}, инструкция интерпретации, 180
${}, синтаксис подстановки значений переменных, 40
?., безопасный оператор разыменования, 67
?:, оператор, 68
[:], оператор, 39
*., оператор, 68
+, оператор, 623
<<, оператор, 76, 235, 612
<=>, оператор, 69
==, оператор, 69
==~, оператор, 81
=~, оператор, 81
, (запятая), в циклах for, 85
() (круглые скобки) в языке Groovy, 79
; (точка с запятой) в языке Groovy, 78
\ (обратный слэш), 80
_ (символ подчеркивания) в именах сценариев, 565

A

AbstractConverter, класс, 428
abs(), метод, 96
Accept-Language, заголовок, 145
Accept, заголовок протокола HTTP, 439, 440, 442
Activation, библиотека, 411
ActiveMQConnectionFactory, класс, 468
ActiveMQ, сервер сообщений, 466, 470
  консоль, 467
addPost, действие, 182
addTo(), метод, 128
Ajax (Asynchronous JavaScript and XML асинхронный JavaScript и XML), 56
  g:submitToRemote, тег, 252, 254
  введение, 56
  выбор библиотеки, 251
  изображение-индикатор, 253
  основы удаленных взаимодействий, 252
all, как название формата данных, 439
AnonymousAuthenticationProvider, 416
AntBuilder, класс, 561, 568
Ant, инструмент сборки, 558, 564, 568
ant, переменная, 568
Apache Ant, инструмент сборки, 559
  интеграция, 577
Apache Axis 2, библиотека, 457
Apache Ivy, инструмент сборки, 580
Apache Maven, инструмент сборки, 559
Apache Tomcat, сервер, 560
API
  изменение, 422
  стабильность, 443
app.grails.version, свойство, 242
ApplicationContext, класс, 30, 538
ApplicationHolder, класс, 538
application/JSON, тип содержимого, 260
application.properties, файл, 242, 561
application/xhtml+xml, тип MIME, 430
application/xml, тип MIME, 430
application, свойство, 613, 618
app.version, свойство, 242
argsMap, переменная, 569
around(), метод, 543
artifacts, каталог, 155
aspect(), метод, 543
assertCurrentStateEquals(), метод, 374
assertFlowExecutionActive(), метод, 374
assertFlowExecutionEnded(), метод, 374

assertFlowExecutionOutcomeEquals(), метод, 374
assertForwardUrlMapping(), метод, 292
AssertionError, исключение, 65
assertRedirectUrl(), метод, 295
assertReverseUrlMapping(), метод, 292
assertUrlMapping(), метод, 291
assert, инструкция, 65
assert*(), методы, 269
asType(), метод, 74, 428
as, оператор
  JSON, 428
  XML, 428
  в символьных литералах, 73
  описание, 73
AuthenticateService, служба, 399
Authentication, расширение, 396, 592
Autobase, расширение, 587
avg(), проекция, 170
Axis2, расширение, 457

B

Bazaar, система управления версиями, 607
BeanBuilder, класс, 621
beans, переменная сценария, 540
before, обработчик, 609
belongsTo, поле, 124, 127
  влияние на отношения «многие ко многим», 134
  две формы определения, 128
  двунаправленная связь, 128
  и каскадные операции, 125
  отношение наследования, 495
BigDecimal, класс
  сравнивание, 70
BigDecimal, тип
  в языке Groovy, 71
bindData(), метод, 197, 384, 386
blank, ограничение, 120
BLOB (Binary Large Object большой двоичный объект), 125
Blowfish, алгоритм, 392
body(), метод, 237
BootStrap.groovy, файл, 173
  destroy, замыкание, 173
  init, замыкание, 173
BootStrap, класс, 286, 289, 298, 387, 398, 411, 555, 585
Bouncy Castle, сайт, 392
BuildConfig.groovy, файл, 36, 561, 575, 607
buildConfig, переменная, 575
Burlap, протокол удаленных взаимодействий, 455
byte[], тип данных
  хранение изображений, 143

C

cacheable(), метод, применение в запросах с критериями, 508
cache, свойство, 509
CAPTCHA, 411
Certificate Authority, компания, выдающая заверенные сертификаты, 390
changeSet(), метод, 588
charEncodingFilter, класс, 624
charset, параметр, 430
char, тип данных, 85
CleanEnd, сценарий, 572
CleanStart, сценарий, 572
collect(), метод, 95, 130, 542
com.grailsinaction, пакет, 113
Comparable, интерфейс, 69
Comparator, интерфейс, 69
compareTo(), метод, 70, 95
Compass, библиотека, 30
CompileEnd, сценарий, 571
CompileStart, сценарий, 571
Compile, сценарий, 578, 583
COM+, модель, 545
Config.groovy, файл, 321, 561, 565
ConfigurationHolder, объект, 321
config, переменная, 575
constraints, замыкание, 117
  введение, 117
contentAsString, свойство, 281
Converters, расширение, 428
Content-Type, заголовок протокола HTTP, 440
ControllerUnitTestCase, класс, 278, 281, 431, 439
conversation, область видимости, 545
CONVERT_URL_TO_LOWERCASE_BEFORE_COMPARISON, параметр настройки, 401
cookies, 450
countBy*(), метод, 164
count(), метод, 47, 158, 163, 166
count(), проекция, 170
count, свойство, 96
create-drop, значение параметра настройки dbCreate, 585
create-drop, значение свойства источника данных, 147
create-functional-test, команда, 293

create-integration-test, команда, 285
createLinkTo, тег, 42
createMock(), метод, 277
CreateWarEnd, сценарий, 572
CreateWarStart, сценарий, 572
create, значение параметра настройки dbCreate, 585
cron, планировщик, 479
CRUD-операции, 31, 49
CRUD, операции, 139
CSS
  атрибуты class и id, 248
  использование для оформления, 248
  платформы CSS, 244
ctx, свойство , 618
curl, утилита, 430
CXF, библиотека, 457

D

DataSource.groovy, файл, 45, 162, 500, 504, 512
Datasources, расширение, 511
DataSource, класс, 565
dateCreated, специальное поле, 111
DateTagLib.groovy, файл, 234
dbCreate, параметр настройки, 45, 585
dbCreate, свойство источника данных, 147
defaultAction, свойство, 190
defaultCache, параметр, 509
DefaultGrailsApplication, класс, 537, 614
DefaultGrailsPlugin, класс, 621
defaultTargetUrl, параметр настройки, 407
def, инструкция
  и внедрение служб, 536
delegate, свойство, 611
delete(), метод, 116, 274, 295, 551
DELETE, метод протокола HTTP, 423, 435
  и броузеры, 430
demand, свойство, 275, 277
dependsOn, свойство, 595
depends(), метод, 568
destroy(), метод, негарантированный вызов, 174
DES, алгоритм, 391
Digg, веб-приложение, 461
discard(), метод, 197
display: none
  свойство CSS, 256
documentation, свойство, 596
Dojo, библиотека JavaScript, 56
do…while, цикл, 85
doWithApplicationContext, свойство, 621
doWithDynamicMethods, свойство, 610, 615
doWithSpring, свойство, 620, 621
doWithWebDescriptor, свойство, 622, 624
-Dserver.port, аргумент командной строки, 36
DSL и отображения Hibernate, 528

E

each(), метод, 94
eastwood-chart, расширение, 320
Eastwood, проект, 320
EasyMock, библиотека, 275
Effect.BlindDown(), метод, 257
Effect.BlindUp(), метод, 257
Effect.Fade(), метод, 256
Effect.toggle(), метод, 258
EhCacheProvider, класс, 501
ehcache.xml, файл, 501, 503, 509
Ehcache, библиотека, 501, 509
email, ограничение, 120
embedded, свойство, 497
encodeAsHTML(), метод, 388
encodeAsURL(), метод, 389
encodePassword(), метод, 405
equals(), метод, как оператор, 69
Errors, интерфейс, 267
errors, объект, 118
errors, свойство
  в имитациях управляющих объектов, 282
  в тестах, 267
ESB (Enterprise Service Bus), сервисная шина предприятия, 461
evals(), метод, библиотека Prototype, 260
event(), метод, 570
exists(), метод, 116
expose, свойство, 453, 457, 473
Expression Language (язык выражений), 40

F

Facebook Connect, технология аутентификации, 417
fetch, параметр, 164

для динамических методов поиска, 162
fieldError, объект, 119
File, класс, 96
FilterConfig, класс, 615
final, ключевое слово, 84, 88
findAllBy*(), метод, 160
findAll(), метод, 94
findBy*(), метод, 160
find(), метод, 94, 163
Firefox, броузер и тестирование интерфейса REST, 430
Fixtures, расширение, 298
flash.message, свойство, 184
flow, область видимости, 545
flush: true, аргумент, 114
followRedirect(), метод, 295
format, параметр запроса в согласовании формата, 440
format, свойство запроса, 437
form, тег, 227
for, цикл, 77, 85
from, атрибут, 228
FunctionalTestCase, класс, 293
functional-tests, команда, 295
Functional Test, расширение, 293, 433, 592

G

g:actionSubmit, тег, 230
g:checkbox, тег, 229
g:createLinkTo, тег, 224
g:createLink, тег, 223
g:datePicker, тег, 230
g:eachError, тег, 202
g:each, тег, 158, 222
g:elseif, тег, 222
g:else, тег, 222
g:form, тег, 156, 182, 227
g:form, тег, отправка формы текущему действию, 227
g:hasErrors, тег, 200, 362
g:hiddenField, тег, 229
g:if, тег, 222
g:javascript, тег, 252
g:layoutBody, тег, 242
g:layoutHead, тег, 242
g:layout, тег, 245
g:link, тег, 223
g:message, тег, 202
g:meta, тег, 242
g:paginate, тег, 172, 233
g:passwordField, тег, 227
g:pieChart, тег, 315
g:radioGroup, тег, 229
g:radio, тег, 229
g:renderErrors, тег, 200
g:render, тег, 246
g:select, тег, 228
g:submitToRemote, тег, 252, 254, 258
g:textArea, тег, 182, 227
g:textField, тег, 156, 227
g:uploadForm, тег, 208, 227
Gant, инструмент сборки, 559, 563, 564, 579
getArtefacts(), метод, 614
getBean(), метод, 538
getConversationScope(), метод, 373
getFlowScope(), метод, 373
getMainContext(), метод, 538
getResourceAsStream(), метод, 188
getWebApplicationContext(), метод, 538
get(), метод, 113, 166, 293, 294
GET, метод протокола HTTP, 423, 435
реализация технологии REST, 428
ge, параметр, 166
Git, система управления версиями, 607
-global, параметр командной строки, 311
Google Chart, расширение, 313
типы данных, таблица, 316
Google Guice, веб-платформа, 54
GORM, механизм, 105
1:1, отношение через встраивание, 496
afterDelete, событие, 498
afterInsert, событие, 498
afterUpdate, событие, 498
beforeDelete, событие, 498
beforeInsert, событие, 498
beforeUpdate, событие, 498
cache, свойство, 502
dateCreated, свойство, 499
hilo, генератор, 524
index, свойство, 507
lastUpdated, свойство, 499
onLoad, событие, 498
sequences, генератор, 524
tablePerHierarchy, свойство, 496
version, свойство, 523
введение, 110
включение кэша второго уровня, 500
выбор механизма кэширования, 501
доступ к кэшу только для чтения, 502
журналирование запросов, 505
значение текущего времени, 499

и естественные ключи, 524
изменение имен таблиц, 523
индексирование полей, 507
и несколько источников данных, 511
использование DSL для доступа к существующей базе данных, 522
кэш второго уровня, 499
кэширование, 499
максимальный период хранения в кэше, 503
обособленные объекты, 188
одновременные обращения к кэшу, 502
профилирование, 504
распределенное кэширование, 509
события, 498
соответствие имени кэша и класса предметной области, 503
стратегии кэширования, 504
существующие базы данных, 513
типы отношений, 123
GPathResult, класс, 99, 429
Grails
интеграция с инструментом сборки Maven, 582
интерактивная оболочка, 62
использование нестандартного порта, 36
квалифицированные услуги, 30
компоненты, 580
консоль, 46, 63, 161, 175
контроллеры, 176
макеты, 240
нестандартные параметры настройки, 321
окружение в интеграционных тестах, 286
платформа нового поколения, 25
поддержка, 30
подкаст, 32
превращение разработки в увлекательное занятие, 26
репозиторий расширений, 603, 605
семь основных доводов, 28
система расширений, 303
система сборки, 559, 563
сценарии
местоположение, 563
предоставляемые инструментом сборки Gant, 571
фильтры, 211
экосистема, 28
grailsApplication, компонент, 537
grailsApplication, объект, 321
grails-app, каталог, 598
_GrailsArgParsing, сценарий, 569
grails-bootstrap.jar, файл, 578, 580
GrailsClassUtils, класс, 615
GrailsClass, интерфейс, 614
_GrailsClean, сценарий, 570
_GrailsCompile, сценарий, 570
grails.config.locations, параметр настройки, 565
grails console, команда, 161
grails create-app, команда, 109
grails createapp, команда, 31
grails create-controller, команда, 140
grails create-controller, контроллер, 31
grails create-domain-class, команда, 110, 123, 127
grails create-filters, команда, 211
grails create-integration-test, команда, 159
grails create-job, команда, 477
grails create-plugin, команда, 593
grails create-service, команда, 52, 192
grails create-tag-lib, команда, 234
grailscrowd.com, сайт, 32
GrailsDaoAuthenticationProvider, 416
_GrailsEvents, сценарий, 570
grails generate-all, команда, 152, 520
grails generate-registration, команда, 411
grails generate-views, команда, 152
grails install-plugin global, команда, 311
grails install-plugin, команда, 249, 308, 397, 451, 608
grails install-quartz-config, команда, 477
grails install-searchable-config, команда, 334
grails install-templates, команда, 154
grails list-plugins, команда, 305
grails.mime.file.extensions, параметр настройки, 441
grails.mime.types, параметр настройки, 441
GrailsMockHttpServletRequest, класс, 280
GrailsMockHttpServletResponse, класс, 280
grails package-plugin, команда, 605
grails plugin-info, команда, 306
grails.plugin.location.*, параметр настройки, 604
grails.plugin.repos.discovery.*, параметр настройки, 607

grails.plugin.repos.distribution.*, параметр настройки, 607
_GrailsProxy, сценарий, 575
grailspodcast.com, сайт, 32
grails release-plugin, команда, 606
grails run-app, команда, 31, 109
_GrailsRun, сценарий, 570
grails-scripts.jar, файл, 580
_GrailsSettings, сценарий, 569, 575
grailsScript(), метод, 566
grails.server.port.http, параметр, 36
grails set-proxy, команда, 306
grails setversion, команда, 242
grails stats, команда, 58
GrailsTask, класс, 578
_GrailsTest, сценарий, 570
grails test-app -integration, команда, 114
grails test-app, команда, 114
GrailsUI, расширение, 338
  gui:autoComplete, тег, 344
  gui:resources, тег, 340
  gui:richEditor, тег, 342
  gui:tooltip, тег, 340
  jsonResult, объект, 346
  resultName, атрибут, 346
  yui-skin-sam, класс CSS, 343
  автодополнение, 338, 344
  автодополнение, возвращаемое значение в формате JSON, 345
  всплывающие подсказки, 338
  заголовок редактора форматированного текста, 342
  календарь, 342
  редактирование форматированного текста, 338
  установка, 339
grails uninstall-plugin, команда, 312
GrailsUnitTestCase, класс, 267, 272
grails upgrade, команда, 626
GrailsUrlMappingsTestCase, класс, 291
grailsVersion, свойство, 595
grails.views.default.codec, параметр настройки, 388
grails.war.copyToWebApp, параметр настройки, 562
grails.war.dependencies, параметр настройки, 562
grails.war.destFile, параметр настройки, 562
grails.war.resources, параметр настройки, 562, 563
grails war, команда, 58
_GrailsWar, сценарий, 568, 570
Grails, консоль, 386
Groovy
  вычисления выражений, 63
  сценарии, 64
  эксперименты с языком Groovy, 62
groovyblogs.org, сайт, 32
Groovy JDK, 93
GroovyPagesTemplateEngine, класс, 537
groovyPagesTemplateEngine, компонент, 537
GroovyTestCase, класс, 269
Groovy, язык программирования
  использование строк в качестве имен методов, 623
GSP
  и экранирование, 388
  страницы с сообщениями об ошибках, 394
GSP (Grails ServerPage), 40
GString, тип данных, 74
  экранирование символа $, 72
GUID (Globally Unique Identifier глобально уникальный идентификатор), 425
GWT, веб-платформа, 25

H

hasErrors(), метод, 201
hasMany, поле, 127, 133
Hessian, протокол удаленных взаимодействий , 455
hibernate.cfg.xml, файл, 514
HibernateTransactionManager, класс, 548
Hibernate, библиотека, 27, 110
  Restrictions, класс, 166
  аннотации, 514
  использование существующих отображений, 514
  кэш второго уровня, 500
  кэш первого уровня, 499
  сеанс, 114
  стратегии кэширования, 502
HQL, язык запросов, 171, 175
  кэширование, 509
  параметры запроса, 172
  уязвимость, 387
HSQLDB, база данных, 45
HtmlUnit, библиотека, 294
HTML, отказ от внедрения тегов в программный код, 39
HTTP INVOKER, протокол удаленных взаимодействий, 454

HttpServletResponse, класс, 428
HTTPS, протокол, 450
  аутентификация, 450
  и проверка, 384
  и протокол защищенных соединений (Secure Sockets Layer, SSL), 390
  как основа реализации REST, 423
  коды состояния, 429, 449
    список, 449
  отображение методов в действия по умолчанию, 436
Hubbub, приложение
  адреса URL, 424
  введение, 105
  добавление вывода сообщений в хронологическом порядке, 178
  добавление новых сообщений, 181
  модель данных, 109
  модель сущность-связь, 106
  проект пользовательского интерфейса, 108
  простая форма поиска, 155
  создание, 109
Hudson, инструмент сборки, 558

I

i18n, 144, 145
  Templates, расширение, 146
  имитация в броузере, 146
  каталог, 145
IBM WebSphere MQ, сервер сообщений, 466
id, параметр, 178, 184
id, поле, 113
id, свойство, 384
  в модульных тестах, 274
ifAllGranted(), метод, 404
ifAnyGranted(), метод, 405
ifNotGranted(), метод, 404
if…then…else, условный оператор, 68
ignoreCase, параметр, 164
ilike, параметр, 166
img, тег
  создание динамических ссылок, 223
includeAll(), метод, 589
includeTargets, инструкция, 566
include(), метод, 589
index(), действие, 38, 190
init(), метод, 387
inList, ограничение, 120
InnoDB, механизм, 546
_Install, сценарий, 625
install-plugin, команда, 293
Integer.parseInt(), метод, 73
IoC, контейнеры, 532
isAjax(), метод, 405
IS_AUTHENTICATED_ANONYMOUSLY, правило, 402
IS_AUTHENTICATED_FULLY, роль, 403, 414
IS_AUTHENTICATED_REMEMBERED, роль, 403, 414
isControllerClass(), метод, 619
isDomainClass(), метод, 619
isLoggedIn(), метод, 404
isNotEmpty, параметр, 166
is(), метод, 70
is…(), методы, 84
it, специальная переменная, 87

J

JabberService, класс, 470, 474
Jabber, расширение, 474
Jabber, система обмена мгновенными сообщениями, 462
  шлюз, 474
Java
  библиотеки, повторное использование, 31
  особенности языка отсутствующие в Groovy, 85
  экосистема, 31
JavaBean, компоненты, 84
Java Cryptography Extension (JCE), библиотека, 392
javac-version, команда, 33
JavaMail, библиотека, 411
Java Message Service (JMS), сервер сообщений, 462
JavaScript, язык программирования, 253, 385
  анимация, 255
  эффекты, 255
Java Server Pages, 39
Java, язык программирования, 26
JBoss Messaging, сервер сообщений, 466
JDK, 33
  расширение, 93
  расширение классов, 612
JDK, библиотека, 454
jetty-env.xml, файл, 510
Jetty, сервер, 560
JFreeChart, библиотека, 320
JMeter, инструмент имитации нагрузки, 507

JMS, расширение, 468
connectionFactory(), метод, 468
destination, свойство, 474
expose, свойство, 473
onMessage(), метод, 473, 476
настройка, 466
ошибка компиляции, 466
преобразование типов, 472
установка, 466
чтение из очереди, 473
JNDI, интерфейс, 565
источники данных, 510
формат именования, 510
joinTable, атрибут, 525
join(), метод, 95
jQuery, библиотека, 252, 255
JSecurity, расширение, 396, 416, 622
JSF, веб-платформа, 25
JSON, формат представления данных и технология REST, 435, 444
конвертер, 428
JsUnit, расширение, 298
JVM, виртуальная машина Java, 27

L

lang, параметр, 146
lastUpdated, специальное поле, 111
layoutBody, тег, 43
layoutHead, тег, 43
LDAP, протокол, 416
length(), метод, 94
length, свойство, 94
LinkedIn, веб-приложение, 461
link, тег, 57
LiquiBase, библиотека, 587
listOrderBy*(), метод, 164
List, класс, 76
list(), метод, 99, 163, 166
loadAfter, свойство, 624
Location, заголовок протокола HTTP, 449
Log4j, библиотека, 195
loginFormUrl, параметр настройки, 407
loginUserDomainClass, класс, 399
log, свойство, 272
Lucene, библиотека, 30

M

Mail, расширение, 320
bcc, поле, 323
body, поле, 323
cc, поле, 323
sendMail(), метод, 322
subject, поле, 323
to, поле, 323
адрес отправителя, 321
использование CSS, 323
использование представлений для отображения, 323
нестандартный порт, 322
поддержка SSL, 322
main(), метод, в языке Java, 64
manager, свойство, 618
mapping, замыкание, 126, 522
порядок сортировки, 131
Map, класс, 76
MarkupBuilder, класс, 98
Matcher, класс
как логический тип, 82
как логическое значение, 66
matches, ограничение, 120
Maven, инструмент сборки, 577, 582
задачи Ant, 580
Maven, расширение, 582
команды
compile, 583
package, 583
test, 583
maxSize, ограничение, 51, 120
max, параметр, 164
max(), проекция, 170
MD5, алгоритм, 391
messageSource, компонент, 537, 539
messages.properties, файл, 144
MetaClass, свойство, 161, 195, 384, 611
methodMissing, метод, 161
method, свойство, 426
Microsoft Transaction Server (MTS), 545
MIME, типы, 430
minSize, ограничение, 120
min(), проекция, 170
MissingMethodException, исключение, 93
MissingPropertyException, исключение, 93
m:n, отношения, 107, 123, 132
mockCommandObject(), метод, 282
mockController(), метод, 281
mockDomain(), метод, 273
mockForConstraintsTests(), метод, 267, 268
MockFor, класс, 276
mockFor(), метод, 275, 278, 285
MockHttpSession, класс, 280
mockLogging(), метод, 273, 285
ModelAndView, класс, 280

Mojo, 25
MOM (Message-Oriented Middleware), программное обеспечение для рассылки сообщений, 461
MultipartHttpServletRequest, класс, 209
MyISAM, механизм, 546

N

Navigation, расширение, 249
navigation, свойство, 249
nav:render, тег, 250
nav:resources, тег, 250
noSelection, атрибут, 228
nullable, ограничение, 118, 120
    наследование, 496
NullPointerException, исключение, 67
null, значение, 66
    как значение false, 65

O

Object, класс, 91
observe, свойство, 617
offset, параметр, 164
onChange, свойство, 616, 618, 619
onComplete, элемент, 253
onConfigChange, свойство, 616, 619
onJabberMessage(), метод, 475
onLoading, элемент, 253
onSuccess, элемент, 253
OPENID, технология аутентификации, 417
Open MQ, сервер сообщений, 466
OpenPGP, библиотека, 392
OpenSessionInViewInterceptor, класс, 551
optionKey, атрибут, 228
optionValue, атрибут, 228
order, параметр, 164
org.hbm.xml, файл, 514
OSCacheProvider, класс, 500
OSCache, библиотека, 501
out, свойство, 285
OWASP, проект, 394

P

P6SpyDriver, класс, 505
P6Spy, расширение, 504
    настройка, 504
    установка, 504
@page, директива, 440
page, объект, 294
page, свойство, 294
params, объект, 168
    привязка, 183
params, свойство, 290, 385
    в модульных тестах, 280
    и технология REST, 435
parseRequest, параметр отображения, 436
PATTERN_TYPE_APACHE_ANT, параметр настройки, 401
PHP, язык программирования, 26
pluginContextPath, свойство, 602
pluginExcludes, переменная, 606
pluginExcludes, свойство, 596, 604, 606
pluginIncludes, переменная, 606
plugins-list.xml, файл, 306
plugin.xml, файл, 606
plugin, свойство, 618
PostController, класс, 536
PostController, контроллер, 182
Poster, расширение для броузера Firefox, 430
PostService, класс, 535, 546
PostService, служба, 192, 194
Post, класс предметной области, 107, 127
post(), метод, 294
POST, метод протокола HTTP, 423
    реализация технологии REST, 428
    сравнение с методом PUT, 427
Profile, класс предметной области, 107, 124
Prototype, библиотека JavaScript, 56, 252
prototype, область видимости, 544
put(), метод, 294
PUT, метод протокола HTTP, 423
    и броузеры, 430
    сравнение с методом POST, 427
Python, язык программирования, 26

Q

Quartz, расширение, 30
    concurrent, свойство, 481
    group, атрибут, 483
    jobContext, объект, 483
    jobDataMap, атрибут, 482
    startDelay, свойство, 478
    StatefulJob, интерфейс, 482
    timeout, свойство, 477
    введение, 477
    внедрение служб, 478
    выражения cron, 479
    задания с сохранением состояния, 481

запуск нескольких копий, 481
настройка, 477
программное управление заданиями, 483
реализация рассылки ежедневных обзоров, 477
справочное руководство, 480
типы данных, 483
установка, 477
хранение данных, 486
Query by Example (QBE механизм запроса по образцу), 162

R

&raquo, метасимвол языка разметки HTML, 43
read(), метод, 113
redirectArgs, свойство, 280
redirectEnabled, свойство, 295
redirect(), метод, 190, 280, 449
ref(), метод, 541, 542
rejectedValue, объект, 119
ReloadableResourceBundleMessageSource, класс, 537
RememberMeAuthenticationProvider, 416
RemoteException, исключение, 454
remoteLink, тег, 57
Remote, класс, 454
Remoting, расширение, 451
removeFrom(), метод, 128
renameColumn(), метод, 588
renderArgs, свойство, 280
renderErrors(), метод, 201
render(), метод, 37, 181, 444
в веб-потоках, 355
в конвертерах, 428
в модульных тестах, 280
и данные в формате JSON, 259
render, тег
plugin, атрибут, 602
Reply, класс, 546
Requestmap, класс, 400
request, область видимости, 545
resources.groovy, файл, 467, 539
resources.xml, файл, 539
resource, параметр отображения, 436
response.outputStream(), свойство, 210
response.setContentLength(), метод, 210
REST, технология, 423
и свойство params, 435
определение, 423
тестирование, 431
return, инструкция, 80, 576
RFC-3023, 430
RMI, протокол удаленных взаимодействий, 454
ROLE_USER, роль, 414
Ruby on Rails, веб-платформа, 26, 37
Ruby, язык программирования, 26, 27
RuntimeException, исключение, и службы с поддержкой транзакций, 193
Runtime.exec(), метод, 389
runtimeLogging, расширение, 195, 372

S

SANS, организация, 394
save(), метод, 47, 113, 114, 116, 183, 274, 551
действия, выполняемые за кулисами, 114
scaffold, свойство, 520
Scarab, инструмент, 394
scope, атрибут, 363
в компонентах Spring, 544
Scriptaculous, библиотека, 252, 255, 260
scriptEnv, переменная, 575
Seam, веб-платформа, 25
Searchable.groovy, файл, 334
Searchable, расширение, 325, 592
component, параметр, 336
except, предложение, 335
Luke, приложение, 339
max, свойство, 328, 332
offset, свойство, 328
only, предложение, 335
results, свойство, 328
scores, свойство, 328
search(), метод, 327
spellCheck, параметр, 335
suggestQuery, параметр, 335
total, свойство, 328
withHighlighter, замыкание, 331
введение, 325
выделение найденных совпадений, 330
добавление многоточий, 331
максимальное число совпадений на странице, 332
местоположение файлов индекса в файловой системе, 334
настройка индексирования полей, 334
определение доступности объектов для поиска, 326
отладка индексов, 339

оформление стилями CSS, 330
подсказки по запросу, 335
поиск с учетом отношений, 336
постраничный просмотр, 332
собственный контроллер поиска, 327
страница поиска по умолчанию, 326
SearchBuilder DSL, язык запросов, 338
searchResult, объект, доступные поля, 328
SecurityContextHolder, класс, 404
Selenium, расширение, 297
sendError(), метод, 448
sendJabberMessage(), метод, 475
sendJMSMessage(), метод, 470
sendPubSubJMSMessage(), метод, 470
sendQueueJMSMessage(), метод, 470
sendTopicJMSMessage(), метод, 470
Serializable, интерфейс, 357, 483
SessionFactory, класс, 537
sessionFactory, компонент, 537
session, область видимости, 545
setCurrentState(), метод, 373
setDefaultTarget(), метод, 569
setRollbackOnly(), метод, 554
settings.groovy, файл, 36
setXmlRequestContent(), метод, 433
SHA-1, алгоритм, 391
SHA-256, алгоритм, 392
SHA-512, алгоритм, 392
shouldFail(), метод, 269
signalEvent(), метод, 373
SimpleMessageConverter, класс, 472
singleton, область видимости, 544
SiteMesh, библиотека, 240
SiteMesh, механизм шаблонов, 30, 42
size(), метод, 94
size, ограничение, 117, 121
SOAP (Simple Object Access Protocol простой протокол доступа к объектам), 421
SOA (Service-Oriented Architecture), архитектура предоставления услуг, 461
sort(), метод, 95
sort, параметр, 164
source, свойство, 618
Spring DSL, 621
Spring MVC, веб-платформа, 25
Spring Security (Acegi), расширение, 396, 592, 599
SpringSource, компания, 30
Spring, веб-платформа, 27, 54
DSL, 539
сравнение с XML, 541
введение, 532
использование в расширениях, 620
компоненты
дескрипторы, 533
доступ из-за пределов классов Spring, 538
области видимости, 544
определение, 533, 539
переопределение, 539
соглашения, 535
spy.log, файл, 505
spy.properties, файл, 505
SQL, 29
SQL Profiler, приложение, 505
startFlow(), метод, 373
StatusUpdate, сценарий, 571
StreamingMarkupBuilder, класс, 98
Stripes, веб-платформа, 25
Struts, веб-платформа, 25
Subversion, система управления версиями, 607
sum(), проекция, 170
Swing, библиотека, 430

T

@taglib, директива, 232
TagLibUnitTestCase, класс, 283
tagLib, свойство, 284
Tag, класс предметной области, 107, 133
Tapestry, веб-платформа, 25
target(), метод, 567
telnet, утилита, 385
templates, каталог, 154
Terracotta, реализация механизма кэширования, 509
test-app, команда, 268, 269
TestApp, сценарий, 578, 583
TestCase, класс, 269
text/xml, тип MIME, 430
text, свойство, 96, 576
times(), метод, 96
Tomcat Manager, приложение, 573, 575
Tomcat, сервер, 573
TopLink, библиотека, 110
toString(), метод, 95
transactional, свойство, 549
TransactionManager, компонент, 548
TransactionProxyFactoryBean, компонент, 548
TransactionStatus, класс, 554
try…catch…finally, конструкция, 66
Twitter, веб-приложение, 106, 461

U

_Uninstall, сценарий, 625
unique, ограничение, 117, 121
  при тестировании, 268
update, значение параметра настройки dbCreate, 586
_Upgrade, сценарий, 625
url
  ограничение доступа, 121, 400
UrlMappings.groovy, файл, 215
US-ASCII, кодировка, 430
userDomain(), метод, 405
UserRegistrationCommand, управляющий объект, 203
User, класс предметной области, 107, 110
User, объект, скаффолдинг, 140
UUID (Universally Unique Identifier универсальный уникальный идентификатор), 425

V

validate(), метод
  в модульных тестах, 282
  при тестировании, 267
validate(), ограничение, 118
validator, замыкание, 121
validator, ограничение, 121
verify(), метод, 277
version, свойство, 384, 595
  в модульных тестах, 274

W

War, сценарий, 583
WAR, файл, 27
  создание, 58
watchedResources, свойство, 617, 618
WebFlowTestCase, класс, 371
  доступные методы, 373
WebTest, расширение, 293
WebWork, веб-платформа, 25
web.xml, файл, 155
while, цикл, 85
Wicket, веб-платформа, 25
withCriteria, замыкание, 166
withFormat, синтаксическая конструкция, 438
  модульное тестирование, 439
withTransaction(), метод, 554
withWriter(), метод, 96

X

XFire, библиотека, 457
Xfire, расширение, 457
XHTML, язык разметки, 430
XML, формат представления данных
  дескрипторы компонентов, 539
  документы
    парсинг, 97
    создание, 97
  конвертер, 428
  настройки, 37
  привязка данных, 435
  типы содержимого, 430
xmlns(), метод, 543
XmlSlurper, класс, 99, 429, 432, 444, 622

Y

YahooUI, библиотека, 338
YAJWF (Yet Another Java Web Framework еще одна веб-платформа на языке Java), 25
yes_no, поле, DB2, 523
YUI Grids Builder, приложение, 244
YUI Grids, библиотека, 244
YUI, библиотека JavaScript, 56, 255

А

автоинкремент, 425
автоматизация, 563, 572
  функционального тестирования, 292
автоматизированная сборка, 558, 577
автоматизированное тестирование, 112
автоматическая сборка, 580
автоматический экспорт структуры базы данных, 585
автоматическое подтверждение транзакций, 553
автоматическое связывание, 536
автоматическое экранирование, 388
авторизация, 395
адреса URL
  уникальные, 424
аннотирование, с помощью расширения Spring Security, 400
анонимные классы, 85
анонимные функции, 87, 89
артефакты
  определение, 598
  поиск, 612
архетипы, 582

архитектура
предоставления услуг (Service-Oriented Architecture, SOA), 461
приложения, 535
управляемая данными, 138
управляемая предметной областью, 108
асинхронная обработка данных, 462
аспектно-ориентированное программирование, 90, 543, 548
атрибуты
при парсинге XML, 99
аутентификация, 395, 416
базовая аутентификация, 450
и технология REST, 450
аутентифицированные пользователи, 400

Б

базы данных
возврат к исходному состоянию после каждого цикла тестирования, 114
в интеграционных тестах, 286
транзакции, 545
безопасная передача сообщений, 456
безопасность
защита от злоумышленников, 382
значение, 381
как процесс, 382
контекст, 404
параметры настройки, 399
расширения, 396
служба аутентификации, 404
усиление ограничений, 413
библиотеки тегов, 222, 616
введение, 42
в модульных тестах, 283
в расширениях, 601
вызов одного тега из другого, 239
расширения Spring Security, 406
быстрота итераций, 59

В

валидаторы, 120
веб-архив (WAR), 560
веб-броузеры
и SSL, 390
и согласование формата представления данных, 443
участие в тестировании, 297
веб-дескриптор, 622
веб-потоки
DSL, 350
error(), метод, 359
on(), метод, 353
render(), метод, 355
scope, атрибут, 363
subflow(), метод, 369
to(), метод, 353
WebFlowTestCase, класс, 371
введение, 349
действие по умолчанию, 351
диалоги, 366
журналирование, 372
завершившиеся, 353
имена кнопок, 354
именование подпотоков, 369
имитация, 371
и постоянные ссылки, 353
и управляющие объекты, 358
кнопка «Назад» броузера, 353
когда неприменимы, 350
конечное состояние, 355
конечные состояния подпотока, 370
механизм сохранения состояния, 350
начальное состояние, 353
необычные ошибки, 372
область видимости
диалога, 368
потока, 356
отладка, 372
отображение собственных представлений, 355
параметр execution, 353
подпотоки, 366
преимущества, 350
привязка данных, 358
принятие решений программным способом, 361
проверка данных, 358
службы с сохранением состояния, 363
соглашения по именованию, 351
сообщения об ошибках, 360
состояние, 353
действия, 361
с переходом, 361
тестирование, 371
входных параметров, 374
завершения потока, 377
ограничений, 374
подпотоков, 375
типичные области применения, 350

веб-службы, 421, 451, 456
функциональная совместимость, 456
версии расширений, 595
верхняя граница диапазона, 77
вложенные классы, 85
вложенные транзакции, 553
внедрение
динамических методов, 610
зависимостей (Dependency Injection, DI), шаблон проектирования, 54, 532
внешние данные, 383
внутренние сценарии, 565
воспроизводимость, 292
временный каталог, во время упаковки, 561
вспомогательные объекты, определение, 275
встраивание классов предметной области, 496
вывод сообщений
в хронологическом порядке, 178
отладочных, в модульных тестах, 273
выгрузка файлов, 207
transferTo(), метод, 209
в файловую систему, 209
ограничение размера, 209
вызов
удаленных методов (Remote Method Invocation, RMI), 451
удаленных процедур (Remote Procedure Calls, RPC), 451
выполнение запросов, 155

Г

глобальное пространство имен, 567

Д

данные в интеграционных тестах, 286
двойная проверка, 384
действия
в отображениях адресов URL, 426
и конструкция withFormat, 439
передача параметров между действиями, 190
деление, значений типа BigDecimal, 71
дескрипторы артефактов, 613
дефисы
в именах свойств и методов, 623
диаграммы
apiString, свойство, 320
dataType, поле, 315
библиотека тегов, 315
в Интранете, 319
гистограммы, 316
демонстрационная страница, 313
значение extended для поля dataType, 316
значение simple для поля dataType, 316
значение text для поля dataType, 316
координатная сетка, 319
круговые, 315
легенда, 318
линейные, 318
метки осей, 317
с несколькими наборами данных, 318
создание, 313
тип bvs, 317
цвет фона, 317
диапазоны
версий расширений, 595
определение, 77
динамические методы
в контролерах, 384
и перезагрузка классов, 616
поиска, 47, 155, 157, 175, 388, 613
в интеграционных тестах, 158
в модульных тестах, 273
внутренняя реализация, 161
кэширование, 508
ограничение поиска двумя полями, 162
с двумя критериями, 157
стратегии немедленной и отложенной загрузки, 161
тестирование, 158
тестирование в консоли Grails, 161
динамические свойства контроллеров, доступные в модульных тестах, 280
динамические языки программирования, 27
динамический вызов, 91
динамическое программирование, 90
в расширениях, 609
дистрибутивы, 565
доверие, 390
долгоживущие адреса URL, 424

дополнительные синтаксические особенности, 78
Дэвид Хейнмейер Ханссон (David Heinemeier Hansson), основатель платформы Rails, 27

Ж

жизненный цикл объекта, 533
журналирование, 195
  в модульных тестах , 273
  настройка, 195

З

зависимости
  задач Grails Ant, 579
  между заданиями, 568
  проекта, 579
  расширений, 595
задания
  инструмента сборки Gant, 564
  по умолчанию, 569
задачи Ant
  для платформы Grails, 577
замыкания
  в Spring DSL, 540
  в службах, 535
  вызов, 89
  вызов с единственным аргументом, 87
  делегаты, 611
  как аргументы методов, 80, 87
  как обработчики в расширениях, 596
  как реализация динамических методов, 610
  как реализация имитации, 276
  определение, 86
  отличие от методов, 89
  синтаксис, 87
  с нулевым количеством аргументов, 611
запрос на получение подписанного сертификата (Certificate Signing Request, CSR), 391
запросы
  кэширование, 507
  область видимости в компонентах Spring, 544
  порядок следования элементов, 274
  с критериями, 165, 175, 388
    between, ограничение, 166
    eq, ограничение, 166
    groupBy, 170
    в интеграционных тестах, 290
    в модульных тестах, 273
    динамические, 167
    и динамические методы поиска, 171
    кэширование, 508
    логические операции, 167
    ограничения, 166
    проекции, 169
запуск тестов, 268
защита адресов URL, 400

И

идентификация пользователя, 396
извлечение данных, 155
изменение структуры базы данных с помощью Hibernate, 585
изменение шаблонов, 154
изображения
  отображение в контроллерах, 209
  работа с изображениями, 206
имена
  окружений, 575
  очередей, 468
  расширений, 594
именованные аргументы, в конструкторе, 84
имитация
  динамических свойств и методов, 272
  зависимостей, 272
  записей в базе данных, 273
  и метапрограммирование, 276
  определение, 266
  принцип действия, 267
  статических методов, 277
  управляющие объекты, 275
имитация объектов, 266, 272, 275
  и интеграционное тестирование, 289
импортирование, в языке Groovy, 64
инверсия управления (Inversion of Control, IoC), 54
  парадигма, 532
инициализация компонентов JavaBean в языке Groovy, 84
интеграционное тестирование, 285
  когда использовать, 290
  отличия от модульного тестирования, 285
  расширений, 603
интеграционные тесты, 112, 265, 549, 555
интеграция с Java, 31
интерактивная оболочка, 62

интернационализация, 144, 145
интерфейсы, 86
инъекция
SQL, 387
атаки, 172
команд операционной системы, 389
программного кода, 389
JavaScript, 388
путей к файлам, 389
исключение артефактов из расширений, 604
исключения
времени выполнения и контролируемые, 549
в транзакциях, 548
в языке Groovy, 66
и сокрытие сведений о платформе Grails, 393
испытательные данные, 290
источники данных, 565
введение, 45
итерации
с использованием замыканий, 87
с помощью тега g:each, 158
через диапазоны, 77

K

каскадирование, 136
каскадные операции изменения и удаления, 125
классы, определение в сценариях, 64
классы предметной области
Post, 127
Profile, 124
Tag, 133
User, 110
в расширениях, 597
идентификаторы объектов в технологии REST, 446
изменение свойств, 114
модульное тестирование, 265
поддерживаемые типы данных, 111
привязка параметров, 385
сериализация, 428
требования расширения Spring Security, 397
удаление, 115
ключи отображений, 85
кодеки, в GSP, по умолчанию, 389
кодировка символов
и XML, 430
коды
возврата, 576
сообщений, 537
состояния, 393
коллекции
в компонентах Spring, 541
и механизм определения истины в Groovy, 66
команды, 559
console, 63
create-app, 62, 565
create-auth-domains, 398, 399
create-domain-class, 565
create-groovy-migration, 589
create-migration, 587
create-plugin, 593
dist, 563, 569
generate-manager, 411, 599
help, 560, 567
install-plugin, 397, 605, 608
keytool, 390
list-plugins, 596
package, 560
package-plugin, 605
plugin-info, 596
quit, 62
release-plugin, 606
run-app, 390, 561, 575, 593, 601
run-war, 390
schema-export, 586
shell, 62, 63
test-app, 561
tomcat-deploy, 576
upgrade, 626
war, 560, 575
добавление с помощью расширений, 624
создание своих собственных, 563
компоненты
имена, 536
для служб, 535
конвертеры
введение, 428
конкатенация строк, 72
контейнер сервлета
и интеграционное тестирование, 286
при функциональном тестировании, 292
контейнер сервлетов, 560, 573
контекст приложения, 538
в интеграционных тестах, 286
и расширения, 621
контекст сервлета, и контекст приложения, 538
контролируемые исключения, 66

контроллеры, 176
в расширениях, 597, 600
действия по умолчанию, 189, 220
и доступ к базе данных, 534
и технология REST, 427
как компоненты Spring, 537
наш первый контроллер, 35
передача отображения в представление, 178
повторное использование, 534, 537
преобразование в адреса URL, 177
собственные имена представлений, 181
соответствие имен действий и представлений, 178
ссылки на действия, 223
тестирование, 278
явная инструкция return, 178
круглые скобки
в языке Groovy, 79
кэширование
запросов, 507
отдельных классов предметной области, 502
кэш первого уровня, 550

Л

литералы
апострофы, 72
диапазонов, 77
классов, 540
массивов, 75
отображений, 75
регулярных выражений, 80
символов, 72, 85
списков
как аргументы методов, 79
строк, 72, 80
чисел, 71
шаблонов, 80
логика работы, 534
логические значения
в языке Groovy, 65
локальные переменные
в сценариях, 567

М

макеты
боковых колонок, 246
введение, 40
макеты по соглашению
применение, 242
таблица, 243
общие элементы title, 242
объединение с содержимым, 241
макеты страниц
стандартизация, 243
фрагментирование с помощью шаблонов, 246
массивы
в языке Groovy, 73
межсайтовый сриптинг, 388
метаклассы, 27
методы
доступа, 83
отличие от замыканий, 89
методы проверки
для испытательных тестов, 271
в функциональных тестах, 296
механизм определения истины в Groovy, 66
и проверки, 65
миллиметровка, 108
«многие ко многим», отношение, 525
многострочные версии строковых литералов, 72
многострочные инструкции, 78
множественное наследование, 610
модели данных, 44
изменение, 585
отношения «многие ко многим», 132
отношения «один к одному», 123
отношения один ко многим, 127
типы отношений, 123
модели предметной области
введение, 44
модульное тестирование, 263, 265
иерархия классов, 270
конструкции withFormat, 439
контроллера, реализующего интерфейс REST, 432
преимущества, 265
расширений, 603
модульные тесты
структура, 278

Н

наблюдение, за расширениями, 617
наборы изменений, 588
навигационные вкладки, 249
навигация, порядок следования элементов, 250
надежная передача сообщений, 456
надежность
достигаемая через тестирование, 262
определяемая проектом, 262

наследование, 494
в классах предметной области, 599
настройки
для этапа сборки, 562
начальная загрузка данных по условию, 175
недействующие ссылки, 424
незашифрованные данные, 382
немедленная загрузка, 161
стратегия, 126
непротиворечивость данных, 545
нетипизированные переменные, 91
неявные аргументы, 87
нижняя граница диапазона, 77

О

обзоры программного кода, 394
области видимости, 185
public, 66
в компонентах Spring, 544
в пределах пакета, 66
область видимости
диалога, 350, 368
запроса (request scope), 186
кадра (flash scope), 186, 187, 219
полей, 83
потока, 350, 356
по умолчанию, 66
приложения (application scope), 188
сеанса (session scope), 187
сервлета (servletContext scope), 188
обмен данными
при использовании технологии REST, 434
обмен сообщениями, 461
мгновенными, 468
обособленный объект, пр использовании расширения Spring Security, 405
обработка ошибок, 201
вывод сообщений об ошибках рядом с полем формы, 201
обработчики событий, 571
обратная совместимость
API, 443
обратное отображение, 292
обратный слеш, 80
объединения ресурсов в единый пул, 510
объектная модель проекта Maven (Maven Project Object Model, POM), 580
объектно-реляционное отображение, 110
объекты переноса данных (Data Transfer Objects, DTO), 453
объявление типов, аргументов методов, 93
ограничения, 51
влияние на определение полей базы данных, 122
для обеспечения безопасности, 384
добавление в классы Java, 520
«один-ко-многим», отношение
и привязка данных, 437
одна таблица на иерархию, 494
ожидания
проверка, 277
окружение разработки, 575
операторы
?:, 68
[:], 39
*., 68
+, 623
<<, 76, 235, 612
<=>, 69
==, 69
==~, 81
=~, 81
новые в языке Groovy, 67
Организация по обеспечению совместимости веб-служб (Web Services Interoperability Organization, WS-I), 456
основные теги форм, теблица, 231
особенности языка Groovy, 64
отказ в обслуживании (Denial of Service, DoS), 383
отладка
в языке Groovy, 93
интерфейса REST, 446
отложенная загрузка, 161
стратегия, 126
отношения
«многие ко многим», 132
«один к одному», 123
«один ко многим», 127
связь объекта с самим собой, 136
отображение адресов URL, 215, 393
в представления, 218
коды ответов, 218
краткая форма определения, 216
необязательные переменные, 217
ошибка 404, 218
ошибка 500, 218
поддержка шаблонных символов, 218

порядок следования правил имеет значение, 217
при интеграционном тестировании, 286
проверка ограничений, 217
собственные страницы с сообщениями об ошибках, 218
соглашения для контроллеров, 37
создание постоянных ссылок, 216
тестирование, 291
отображений литералы
как аргументы методов, 79
синтаксис, 75
отображения
в компонентах Spring, 541
в языке Groovy, 75
доступ к элементам, 76
и механизм определения истины в Groovy, 66
как аргументы конструкторов, 85
отрицательные индексы в массивах, 76
отслеживаемые ресурсы, 616
очереди сообщений, 464
ошибки
исправление, 264
пакеты
группировка классов по сферам влияния, 534
импортируемые автоматически, 64
пакеты ресурсов, 219
параметризованные запросы, 388
параметры настройки, при тестировании расширений, 604
первичные ключи, в адресах URL, 425
«первым пришел, первым ушел», 464
передача сообщений
архитектуры, 462
введение, 461
двухточечная модель, 464
когда использовать, 463
масштабируемость, 462
поставщики, 464
потребители, 464
темы, 464
темы и очереди, 476
терминология, 464
широковещательная модель, 464
перезагрузка классов
и расширения, 616, 619
переменные сценария, 567
перемещение данных
с расширением Autobase, 587
перенаправление
в другое действие, 190
на внешний ресурс, 191
на страницу регистрации, 403
по абсолютному адресу URL, 191
после регистрации, 407
с параметрами, 191
переопределение
контроллеров расширения, 600
представлений расширения, 600
плавный переход цвета, 43
планирование заданий, 461
пограничные условия
при интеграционном тестировании, 288
подключение расширения из его исходного каталога, 604
подсветка синтаксиса, 63
подписки, 78
подсчет количества вызовов метода в модульных тестах, 277
подтверждения о регистрации, 411
поиск фильтров, 624
полиморфные запросы, 495
полномочия, 413
полнотекстовый поиск, 325
в сравнении с SQL, 325
имитация, 158
с помощью расширения, 325
пользовательский интерфейс и службы, 535
поля ввода, префиксы в именах, 437
порядок вызова методов в модульных тестах, 277
порядок загрузки, компонентов Spring, 539
порядок следования
полей, 141
правил доступа к адресам URL в настройках Spring Security, 402
результатов, 130
форматов, в конструкции withFormat, 439
порядок сортировки
возрастанию и по убыванию, 131
постраничный просмотр, 232
max, параметр, 233
offset, параметр, 233
использование коллекции params, 233
при полнотекстовом поиске, 332
построитель разметки, 254, 433, 517

представления, 39
в расширениях, 601
преимущество соглашений перед настройками, 26, 27, 28, 29, 39, 55, 59, 105
преобразование типов
простых типов, 74
с использованием механизма определения истины в языке Groovy, 66
приведение типов, 73
привязка данных, 196, 220, 386
введение, 183
в модульных тестах, 267
из формата XML, 435
исключение полей, 384
к вложенным объектам, 198
ко множеству объектов, 198
параметров, 197
посредством передачи параметров конструктору, 196
префиксы в именах полей формы, 199
привязка параметров, 197
пробелы
в охвате тестами, 287
в многострочных литералах строк, 72
проверка, 51
введение, 116
для безопасности, 384
на стороне клиента, 385
ограничений в управляющих объектах, в модульных тестах, 282
специальные приемы, 122
с помощью регулярных выражений, 121
проверки, 65
наличия исключений, 270
отключение, 65
соответствия типов, 263
программирование метаклассов, 610
продуктивность платформы, 59
проектирование для упрощения тестирования, 264
простая аутентификация, 417
простой фильтр безопасности, 211
пространства имен
в библиотеках тегов расширений, 601
в определениях компонентов Spring, 543
в файлах сборки Ant, 580
пространства имен тегов при имитации тегов, 283
простые типы данных, 71
протокол защищенных соединений (Secure Sockets Layer, SSL), 390
протоколы
свойство expose, 453
удаленных взаимодействий
сравнение, 454
процедура создания дистрибутива, 558
публикация расширений, 603

Р

работа с изображениями, 206
рабочее окружение, 575
развертывание приложения, 559, 563, 572, 574
разделение сфер влияния (Separation of Concerns, SoC), принцип, 534
разработка приложений, 557
разработка, управляемая тестированием, 263
разрешение имен на этапе выполнения, 92
разрешения, 398
разрывы строк, в языке Groovy, 78
расширение файлов
в согласовании формата, 440
и отображение адресов URL, 440, 441
расширения
Authentication, 396, 592
Autobase, 587
Fixtures, 298
Functional Test, 293, 409, 592
JSecurity, 396, 416, 622
JsUnit, 298
Searchable, 592
Selenium, 297
Spring Security, 592, 599
Spring Security (Acegi), 396
WebTest, 293
глобальные, 310
дескриптор, 594
интеграция с платформой Grails, 608
тестирование, 603
добавление динамических методов, 609
зависимости, 311
имена, 594
кэширование, 308
местоположение, 625
в файловой системе, 310

местоположение электронной документации, 307
нумерация версий, 305
обработчики, 596
поиск, 305
получение информации о версии, 306
представления, 600, 601
проект, 593
репозиторий, 305
система, 592
соглашения о нумерации версий, 595
создание, 593
список по категориям, 307
тестирование, как приложений, 603
удаление, 312
упаковка, 605
управление версиями, 310
установка, 308
определенной версии, 308
расширения, как модули, 592
регистрация новых пользователей, 411
регулярные выражения
в языке Groovy, 80
группы, 82
проверка, 121
результаты тестирования, 268
ресурсы
в расширениях, 601
в технологии REST, 423
выбор идентификаторов ресурсов, 425
загрузка из файловой системы, 188
и адреса URL, 424
рефакторинг, 108
рефлексия в динамическом программировании, 90
роли, 398, 413

C

самонастройка приложения, 59
свертка сообщений, 390
свободный режим, при имитации объектов, 278
свойства
в языке Java, 83
изменение, 114
компонентов Spring, инициализация, 541
логического типа, 84
определение, 83
связывание
объектов, 535
сеансы
и транзакции, 550
серверные страницы Java (Java Server Pages, JSP), 222
сервер приложений, 27
сервисная шина предприятия (Enterprise Service Bus, ESB), 461
сервлеты
настройка в расширениях, 622
сертификаты SSL, 390
самоподписанные, 390
сетевые черви, 382
символ подчеркивания, в именах сценариев, 565
синтаксис методов, 79
системы управления версиями, 310
скаффолдинг (scaffolding), 27, 31, 109, 139, 140
Ajax, 151
валидаторы, 141
введение, 49
в готовых приложениях, 149
динамический, 151
за пределами возможностей, 151
интернационализация, 145
и проверка, 142
настройка внешнего вида с помощью CSS, 149
настройка с помощью метода toString(), 148
порядок следования полей, 141
создание пользовательского интерфейса, 139
создание статических файлов, 152
стандартные методы, 153
статический, 152
шаблоны, 149
слабо связанные архитектуры, 462
службы, 534
введение, 191
внедрение, 194
и замыкания, 535
как единичные объекты (singleton) или прототипы, 195
области видимости потока, 363
поддержка транзакций, 546
тестирование, 271
транзакции, 192
собственная страница регистрации, 407
собственные операции приведения типов, 74
собственные репозитории расширений, 607

собственные теги и экранирование, 388
события
в инструменте сборки Gant, 570
и расширения, 618
список наиболее часто используемых, 571
системы сборки, 570
согласование формата представления, 434
соглашения
вместо настроек, 29
по именованию классов тестов, 279
по именованию файлов GSP, 439
содержимое ответа, в модульных тестах, 281
создание дистрибутива, 563
сообщения об ошибках
интернационализация, 144
настройка, 144
по умолчанию, 144
формат, 144
шаблонные поля, 144
сообщество Grails, 32
состояние
действия, 361
с переходом, 361
сохраняющие группы, в регулярных выражениях, 82
спам-боты, 411
специальные приемы проверки полей, 122
списки
в языке Groovy, 75
доступ к элементам, 76
литералов
как аргументы методов, 79
управления доступом (Access Control Lists, ACL), 415
сравнивание значение, 69
«старые, добрые, простые объекты» Java (Plain Old Java Object, POJO), 533
статический скаффолдинг, 152
статическое управление доступом, с помощью расширения Spring Security, 400
страницы с сообщениями об ошибках, 394
строгий режим, при имитации объектов, 277
строки
в языке Groovy, 72
как имена свойств и методов, 91
окруженные символами /, 73
сравнивание, 70
структура базы данных
изменение, 585
существующие базы данных, 513
GORM DSL, 522
поля идентификаторов, 515
примерение механизма скаффолдинга, 519
структура таблиц, 513
схема базы данных
автоматическое создание, 44
сценарии
вызов из инструмента сборки Maven, 583
запуск из задач Ant, 578
и расширения, 625
предоставляемые платформой Grails, 569
сценарии на языке Groovy, 63
запуск, 63
определения, 64

Т

теги
attrs, элемент, 234
g:isLoggedIn, 406
g:loggedInUserInfo, 406
g:, префикс, 236
теги GSP, основные, 222
в расширениях, 598, 601
итеративные, 234, 238
как вызовы методов, 224
логические, 234, 237
передача атрибутов HTML, 228
пространства имен, 236
простые, 234
расширение представлений, 234
ссылок, таблица, 224
типы, 234
форм, 222, 224
библиотеки тегов JSP, 232
экранирование значений атрибутов, 389
тело тега
как замыкание, 285
темы оформления, 247
использование фильтров, 248
темы сообщений, 464
тестирование, 54
библиотек тегов, 283
взаимодействий между объектами, 287
внутри полноценного окружения платформы Grails, 286

действий, использующих управляющие объекты, 281
завершения потока, 377
классов предметной области, 265
контроллеров, 278
ограничений, 265
отображения адресов URL, 291
расширений, 603
сверху вниз, 264
служб, 271
снизу вверх, 265
соглашение об именовании, 266
содержимое ответа, 281
тегов с атрибутами, 284
транзакции, 115
транзакций, 549
управления доступом, 408
управляющих объектов, 281
тестовые приложения, 604
тесты
запуск, 268
из интегрированный среды разработки, 265
модульные и интеграционные, 112
просмотр результатов, 268
удобство при проведении экспериментов, 112
типы данных
в языке Groovy, 70
строки, 72
типы содержимого
в GSP, 439
запроса, 441
точки с запятой, в языке Groovy, 78
транзакции, 545
автоматическое подтверждение, 553
базы данных, 553
вложенные, 553
и замыкания, 535
и исключения времени выполнения, 193
и сеансы, 550
определение, 545
откат изменений, 548
приложения, 553
тестирование, 549
функциональное тестирование, 549
трехместный оператор, 68
тройные кавычки, 72

У

удаленное развертывание приложений
в сервере Tomcat, 573
удаленные взаимодействия
с использованием веб-служб, 451
упаковка приложения, 559
управление
данными, 26
доступом, 395
терминология, 395
зависимостями, 532
автоматическое внедрение, 535
пользователями, 411
управляющие объекты, 202, 219
в веб-потоках, 358
введение, 202
в модульных тестах, 281
для обеспечения безопасности, 384
и принцип DRY, 204
и проверка уникальности, 205
как контейнеры бизнес-логики, 206
привязка к действию контроллера, 204
реализация выгрузки фотографий, 207
собственные проверки, 204
участие во внедрении, 206
условные операторы
сравнение Groovy и Java, 65
устранение помех, 59
утиная типизация, 92
уязвимости
в криптографических алгоритмах, 392
тестирование, 394

Ф

фабричные методы в определениях компонентов Spring, 542
философия опоры на самое лучшее, 30
фильтры, 220, 609
afterView, обработчик, 213
after, обработчик, 213
before, обработчик, 213
внедрение служб, 214
в управлении доступом, 415
доступные переменные, 213
модификация существующих значений модели данных в фильтрах, 214
настройка в расширениях, 622
при интеграционном тестировании, 286
применение к адресам URL, 214
применение к нескольким действиям сразу, 214

создание, 211
фазы жизненного цикла, 213
фильтры сервлетов, 408
фильтры
применение шаблонных символов при определении имен контроллеров и действий, 214
форма записи в виде массивов, 76
форма регистрации нового пользователя, 225
форма регистрации пользователя, 203
фрагменты списков, 78
функциональное тестирование, 263, 292
расширений, 603
технологии REST, 433
управления доступом, 408

Х

хранилище ключей, 391

Ц

центральный репозиторий, 603, 605
цикл разработки, 557
цифровые подписи, 390

Ч

частные ключи, 390
частные репозитории расширений, 607
черный список, 384
числа
вещественные, 71
в языке Groovy, 71

Ш

шаблонный подход, 31
шаблонные символы, 617, 618
шаблоны, 149
шифрование, 390
шифрование паролей, 399

Э

экранирование выходных данных, 386

Я

язык запросов библиотеки Hibernate (Hibernate Query Language, HQL)
в модульных тестах, 273
в интеграционных тестах, 290
язык описания веб-приложения (Web Application Description Language, WADL), 445
язык определения веб-служб (Web Service Definition Language, WSDL), 445, 457
язык предметной области (Domain-Specific Language, DSL), 539
языки программирования со статической типизацией, 27

По договору между издательством «Символ-Плюс» и Интернет-магазином «Books.Ru – Книги России» единственный легальный способ получения данного файла с книгой ISBN 978-5-93286-185-1, название «Grails. Гибкость Groovy и надежность Java» – покупка в Интернет-магазине «Books.Ru – Книги России». Если Вы получили данный файл каким-либо другим образом, Вы нарушили международное законодательство и законодательство Российской Федерации об охране авторского права. Вам необходимо удалить данный файл, атакже сообщить издательству «Символ-Плюс» (piracy@symbol.ru), где именно Вы получили данный файл.