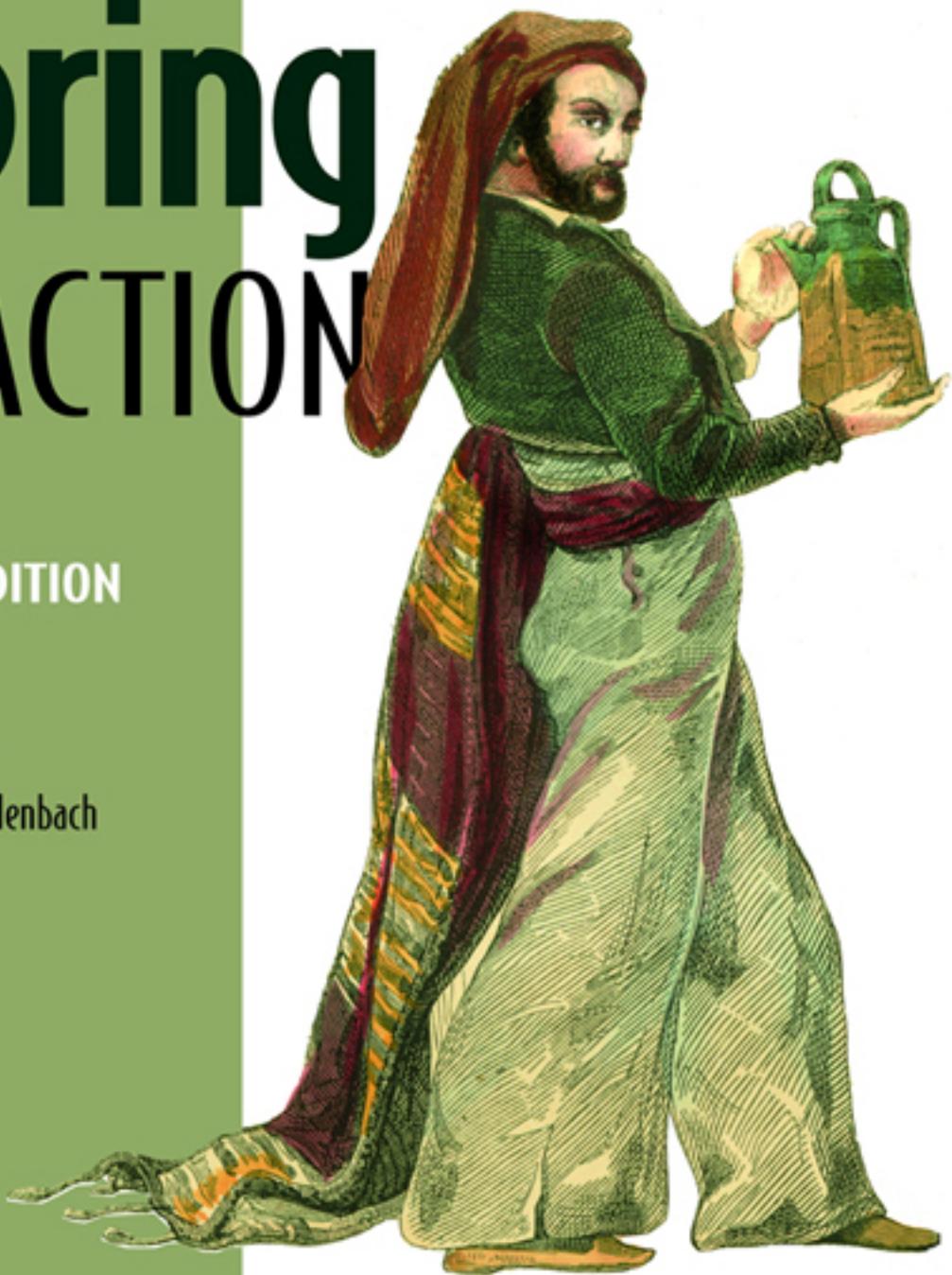


Updated for Spring 2.0

Spring IN ACTION

SECOND EDITION

Craig Walls
with Ryan Breidenbach



Praise for the First Edition

“This is one of those rare books that connect a tutorial for using a certain software product with a plethora of ideas on good software design and design patterns. I enjoyed this book very much...”

—Computing Reviews

“Covers all the bases with extensive examples and explicit instructions...a superbly organized and fluently written instruction and reference manual.”

—Internet Bookwatch

“...easy to read...and has just enough humor mixed in...”

—Books-On-Line

“While Spring’s reference documentation is high quality, this book makes learning Spring much more enjoyable. The book injects a fair amount of humor that keeps it entertaining. If you want to learn Spring, you can’t go wrong with this offering.”

—Bill Siggelkow’s Weblog
Author of *Jakarta Struts Cookbook*

“Truly a great resource... The book clearly defines the power that Spring brings to enterprise programmers and how Spring abstracts away many of the tougher J2EE services that most serious applications use. The book has been through a rigorous early access program, so thankfully grammar and code errors are all but non-existent. To me, there is nothing worse than trying to learn a new technology from a poorly written and edited technical book. Thankfully, Craig, Ryan, and the Manning team have paid attention to detail and produced a book that I highly recommend.”

—JavaLobby.org

“A complete reference manual that covers nearly every aspect of Spring. This doesn’t mean it is complicated: every explanation is clear and there are a lot of code examples. ...[it] explains clearly what “Inversion of Control” and AOP mean and how Spring makes them possible. ...how you can write services and Daos, and how you can simply implement transaction management and service remoting. ...the third part talks about the Web layer covering Spring MVC as well as other technologies and frameworks. ...Overall an excellent resource for any developer interested in using Spring in his project.”

—Java User Group Milano

Spring in Action

Second Edition

CRAIG WALLS
with Ryan Breidenbach

M
MANNING
Greenwich
(74° w. long.)

For online information and ordering of this and other Manning books, please visit www.manning.com. The publisher offers discounts on this book when ordered in quantity. For more information, please contact:

Special Sales Department
Manning Publications Co.
Sound View Court 3B Fax: (609) 877-8256
Greenwich, CT 06830 Email: orders@manning.com

©2008 by Manning Publications Co. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means electronic, mechanical, photocopying, or otherwise, without prior written permission of the publisher.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in the book, and Manning Publications was aware of a trademark claim, the designations have been printed in initial caps or all caps.

© Recognizing the importance of preserving what has been written, it is Manning's policy to have the books we publish printed on acid-free paper, and we exert our best efforts to that end.

 Manning Publications Co. Copyeditor: Liz Welch
Sound View Court 3B Typesetter: Dottie Marsico
Greenwich, CT 06830 Cover designer: Leslie Haimes

ISBN 1-933988-13-4
Printed in the United States of America
1 2 3 4 5 6 7 8 9 10 – MAL – 13 12 11 10 09 08 07

For my wife Raymie and my daughters Maisy and Madison

*I am endlessly mystified as to how I merit the love
of the world's three most beautiful girls.*

brief contents

PART 1 CORE SPRING	1
1 ■ Springing into action	3
2 ■ Basic bean wiring	31
3 ■ Advanced bean wiring	72
4 ■ Advising beans	116
PART 2 ENTERPRISE SPRING.....	153
5 ■ Hitting the database	155
6 ■ Managing transactions	220
7 ■ Securing Spring	247
8 ■ Spring and POJO-based remote services	305
9 ■ Building contract-first web services in Spring	343
10 ■ Spring messaging	384
11 ■ Spring and Enterprise JavaBeans	423
12 ■ Accessing enterprise services	441

PART 3 CLIENT-SIDE SPRING487

13	■ Handling web requests	489
14	■ Rendering web views	533
15	■ Using Spring Web Flow	580
16	■ Integrating with other web frameworks	623
<i>appendix A</i>	<i>Setting up Spring</i>	667
<i>appendix B</i>	<i>Testing with (and without) Spring</i>	678

contents

<i>preface</i>	xix
<i>preface to the first edition</i>	xxii
<i>acknowledgments</i>	xxv
<i>about this book</i>	xxvii
<i>about the title</i>	xxxiii
<i>about the cover illustration</i>	xxxiv

PART 1 CORE SPRING 1

1	<i>Springing into action</i>	3
1.1	What is Spring?	5
	<i>Spring modules</i>	6
1.2	A Spring jump start	11
1.3	Understanding dependency injection	14
	<i>Injecting dependencies</i>	14
	<i>Dependency injection in action</i>	15
	<i>Dependency injection in enterprise applications</i>	21
1.4	Applying aspect-oriented programming	24
	<i>Introducing AOP</i>	24
	<i>AOP in action</i>	26
1.5	Summary	30

2 Basic bean wiring 31

- 2.1 Containing your beans 33
 - Introducing the BeanFactory* 34 • *Working with an application context* 35 • *A bean's life* 37
- 2.2 Creating beans 40
 - Declaring a simple bean* 40 • *Injecting through constructors* 42
- 2.3 Injecting into bean properties 46
 - Injecting simple values* 47 • *Referencing other beans* 48
 - Wiring collections* 52 • *Wiring nothing (null)* 58
- 2.4 Autowiring 58
 - The four types of autowiring* 59 • *Mixing auto with explicit wiring* 63 • *To autowire or not to autowire* 63
- 2.5 Controlling bean creation 64
 - Bean scoping* 65 • *Creating beans from factory methods* 66
 - Initializing and destroying beans* 68
- 2.6 Summary 71

3 Advanced bean wiring 72

- 3.1 Declaring parent and child beans 73
 - Abstracting a base bean type* 74 • *Abstracting common properties* 76
- 3.2 Applying method injection 79
 - Basic method replacement* 80 • *Using getter injection* 83
- 3.3 Injecting non-Spring beans 85
- 3.4 Registering custom property editors 88
- 3.5 Working with Spring's special beans 92
 - Postprocessing beans* 93 • *Postprocessing the bean factory* 95
 - Externalizing configuration properties* 96 • *Resolving text messages* 99 • *Decoupling with application events* 101
 - Making beans aware* 103
- 3.6 Scripting beans 106
 - Putting the lime in the coconut* 107 • *Scripting a bean* 108
 - Injecting properties of scripted beans* 111 • *Refreshing scripted beans* 112 • *Writing scripted beans inline* 113
- 3.7 Summary 114

4 *Advising beans* 116

- 4.1 Introducing AOP 118
 - Defining AOP terminology* 119 ▪ *Spring's AOP support* 122
- 4.2 Creating classic Spring aspects 125
 - Creating advice* 127 ▪ *Defining pointcuts and advisors* 132
 - Using ProxyFactoryBean* 136
- 4.3 Autoproxying 139
 - Creating autoproxies for Spring aspects* 140 ▪ *Autoproxying @AspectJ aspects* 141
- 4.4 Declaring pure-POJO aspects 145
- 4.5 Injecting AspectJ aspects 149
- 4.6 Summary 152

PART 2 ENTERPRISE SPRING 153

5 *Hitting the database* 155

- 5.1 Learning Spring's data access philosophy 157
 - Getting to know Spring's data access exception hierarchy* 158
 - Templating data access* 161 ▪ *Using DAO support classes* 163
- 5.2 Configuring a data source 165
 - Using JNDI data sources* 165 ▪ *Using a pooled data source* 167
 - JDBC driver-based data source* 168
- 5.3 Using JDBC with Spring 170
 - Tackling runaway JDBC code* 170 ▪ *Working with JDBC templates* 173 ▪ *Using Spring's DAO support classes for JDBC* 180
- 5.4 Integrating Hibernate with Spring 183
 - Choosing a version of Hibernate* 185 ▪ *Using Hibernate templates* 186 ▪ *Building Hibernate-backed DAOs* 190
 - Using Hibernate 3 contextual sessions* 192
- 5.5 Spring and the Java Persistence API 194
 - Using JPA templates* 194 ▪ *Configuring an entity manager factory* 197 ▪ *Building a JPA-backed DAO* 202

- 5.6 Spring and iBATIS 203
 - Configuring an iBATIS client template* 204 ▪ *Building an iBATIS-backed DAO* 207
- 5.7 Caching 208
 - Configuring a caching solution* 210 ▪ *Proxying beans for caching* 215 ▪ *Annotation-driven caching* 217
- 5.8 Summary 218

6 *Managing transactions* 220

- 6.1 Understanding transactions 222
 - Explaining transactions in only four words* 223
 - Understanding Spring's transaction management support* 224
- 6.2 Choosing a transaction manager 225
 - JDBC transactions* 226 ▪ *Hibernate transactions* 227
 - Java Persistence API transactions* 227 ▪ *Java Data Objects transactions* 228 ▪ *Java Transaction API transactions* 229
- 6.3 Programming transactions in Spring 229
- 6.4 Declaring transactions 232
 - Defining transaction attributes* 233 ▪ *Proxying transactions* 238 ▪ *Declaring transactions in Spring 2.0* 241 ▪ *Defining annotation-driven transactions* 243
- 6.5 Summary 245

7 *Securing Spring* 247

- 7.1 Introducing Spring Security 248
- 7.2 Authenticating users 252
 - Configuring a provider manager* 253 ▪ *Authenticating against a database* 256 ▪ *Authenticating against an LDAP repository* 264
- 7.3 Controlling access 271
 - Voting access decisions* 272 ▪ *Casting an access decision vote* 273 ▪ *Handling voter abstinen*ce 275

7.4 Securing web applications 275

Proxying Spring Security's filters 278 ▪ Handling the security context 285 ▪ Prompting the user to log in 286 ▪ Handling security exceptions 291 ▪ Enforcing web security 293 ▪ Ensuring a secure channel 294

7.5 View-layer security 297

Conditionally rendering content 298 ▪ Displaying user authentication information 299

7.6 Securing method invocations 300

Creating a security aspect 301 ▪ Securing methods using metadata 303

7.7 Summary 304

8 *Spring and POJO-based remote services 305*

8.1 An overview of Spring remoting 306

8.2 Working with RMI 309

Wiring RMI services 310 ▪ Exporting RMI services 312

8.3 Remoting with Hessian and Burlap 316

Accessing Hessian/Burlap services 317 ▪ Exposing bean functionality with Hessian/Burlap 318

8.4 Using Spring's HttpInvoker 322

Accessing services via HTTP 323 ▪ Exposing beans as HTTP Services 324

8.5 Spring and web services 326

Exporting beans as web services using XFire 326

Declaring web services with JSR-181 annotations 330

Consuming web services 333 ▪ Proxying web services with an XFire client 340

8.6 Summary 341

9 *Building contract-first web services in Spring 343*

9.1 Introducing Spring-WS 345

9.2 Defining the contract (first!) 347

Creating sample XML messages 348

9.3	Handling messages with service endpoints	353
	<i>Building a JDOM-based message endpoint</i>	355
	<i>Marshaling message payloads</i>	358
9.4	Wiring it all together	361
	<i>Spring-WS: The big picture</i>	361
	<i>Mapping messages to endpoints</i>	363
	<i>Wiring the service endpoint</i>	364
	<i>Configuring a message marshaler</i>	364
	<i>Handling endpoint exceptions</i>	367
	<i>Serving WSDL files</i>	369
	<i>Deploying the service</i>	373
9.5	Consuming Spring-WS web services	373
	<i>Working with web service templates</i>	374
	<i>Using web service gateway support</i>	381
9.6	Summary	382

10 *Spring messaging* 384

10.1	A brief introduction to JMS	386
	<i>Architecting JMS</i>	387
	<i>Assessing the benefits of JMS</i>	390
	<i>Setting up ActiveMQ in Spring</i>	392
10.2	Using JMS with Spring	393
	<i>Tackling runaway JMS code</i>	393
	<i>Working with JMS templates</i>	395
	<i>Converting messages</i>	402
	<i>Using Spring's gateway support classes for JMS</i>	405
10.3	Creating message-driven POJOs	407
	<i>Creating a message listener</i>	408
	<i>Writing pure-POJO MDPs</i>	412
10.4	Using message-based RPC	416
	<i>Introducing Lingo</i>	417
	<i>Exporting the service</i>	418
	<i>Proxying JMS</i>	420
10.5	Summary	422

11 *Spring and Enterprise JavaBeans* 423

11.1	Wiring EJBs in Spring	425
	<i>Proxying session beans (EJB 2.x)</i>	426
	<i>Wiring EJBs into Spring beans</i>	430
11.2	Developing Spring-enabled EJBs (EJB 2.x)	431

11.3 Spring and EJB3 434

Introducing Pitchfork 435 • *Getting started with Pitchfork* 436
Injecting resources by annotation 437 • *Declaring interceptors using annotations* 438

11.4 Summary 440

12 Accessing enterprise services 441

12.1 Wiring objects from JNDI 442

Working with conventional JNDI 443 • *Injecting JNDI objects* 446 • *Wiring JNDI objects in Spring 2* 449

12.2 Sending email 450

Configuring a mail sender 451 • *Constructing the email* 453

12.3 Scheduling tasks 456

Scheduling with Java's Timer 457 • *Using the Quartz scheduler* 460 • *Invoking methods on a schedule* 464

12.4 Managing Spring beans with JMX 466

Exporting Spring beans as MBeans 467 • *Remoting MBeans* 477 • *Handling notifications* 482

12.5 Summary 485

PART 3 CLIENT-SIDE SPRING 487

13 Handling web requests 489

13.1 Getting started with Spring MVC 490

A day in the life of a request 491 • *Configuring DispatcherServlet* 492 • *Spring MVC in a nutshell* 495

13.2 Mapping requests to controllers 502

Using SimpleUrlHandlerMapping 503 • *Using ControllerClassNameHandlerMapping* 504 • *Using metadata to map controllers* 505 • *Working with multiple handler mappings* 505

13.3 Handling requests with controllers 506

Processing commands 509 • *Processing form submissions* 512
Processing complex forms with wizards 520 • *Working with throwaway controllers* 528

- 13.4 Handling exceptions 531
- 13.5 Summary 532

14 *Rendering web views* 533

- 14.1 Resolving views 534
 - Using template views* 535 ▪ *Resolving view beans* 537
 - Choosing a view resolver* 540
- 14.2 Using JSP templates 542
 - Binding form data* 542 ▪ *Rendering externalized messages* 544
 - Displaying errors* 547
- 14.3 Laying out pages with Tiles 549
 - Tile views* 550 ▪ *Creating Tile controllers* 554
- 14.4 Working with JSP alternatives 556
 - Using Velocity templates* 557 ▪ *Working with FreeMarker* 564
- 14.5 Generating non-HTML output 569
 - Producing Excel spreadsheets* 570 ▪ *Generating PDF documents* 573 ▪ *Developing custom views* 576
- 14.6 Summary 578

15 *Using Spring Web Flow* 580

- 15.1 Getting started with Spring Web Flow 582
 - Installing Spring Web Flow* 584 ▪ *Spring Web Flow essentials* 589 ▪ *Creating a flow* 591
- 15.2 Laying the flow groundwork 591
 - Flow variables* 591 ▪ *Start and end states* 593 ▪ *Gathering customer information* 594 ▪ *Building a pizza order* 601
 - Completing the order* 605 ▪ *A few finishing touches* 608
- 15.3 Advanced web flow techniques 611
 - Using decision states* 612 ▪ *Extracting subflows and using substates* 614
- 15.4 Integrating Spring Web Flow with other frameworks 619
 - Jakarta Struts* 619 ▪ *JavaServer Faces* 620
- 15.5 Summary 622

16 *Integrating with other web frameworks* 623

- 16.1 Using Spring with Struts 624
 - Registering the Spring plug-in with Struts* 626 • *Writing Spring-aware Struts actions* 627 • *Delegating to Spring-configured actions* 629 • *What about Struts 2?* 632
- 16.2 Working Spring into WebWork 2/Struts 2 633
- 16.3 Integrating Spring with Tapestry 636
 - Integrating Spring with Tapestry 3* 637 • *Integrating Spring with Tapestry 4* 641
- 16.4 Putting a face on Spring with JSF 643
 - Resolving JSF-managed properties* 644 • *Resolving Spring beans* 646 • *Using Spring beans in JSF pages* 646
Exposing the application context in JSF 648
- 16.5 Ajax-enabling applications in Spring with DWR 648
 - Direct web remoting* 650 • *Accessing Spring-managed beans DWR* 659
- 16.6 Summary 664

appendix A Setting up Spring 667

appendix B Testing with (and without) Spring 678

index 707

web content

- web chapter Building portlet applications
- appendix C Spring XML configuration reference
- appendix D Spring JSP tag library reference
- appendix E Spring Web Flow definition reference
- appendix F Customizing Spring configuration

preface

It was December 7, 2005. I was standing at the side of a large hotel meeting room in Miami Beach, Florida. The room was filled with developers from all over the world who had descended upon the beautiful sandy beaches of southern Florida for a single purpose: to talk about Spring.

What can I say? It was a room full of nerds. Rather than soak in the sun and surf, we all gathered inside to bask in the warm glow of our laptop screens to learn more about our beloved framework from those who know it best.

On that particular night, we were hanging on the words of Spring's creator, Rod Johnson, as he presented the opening keynote for the conference. He spoke of Spring's origins and the successes it had enjoyed. Then he invited a few members of the Spring team to the podium to introduce new features that were to be in the next version.

He wasn't far into his presentation when Rod made an announcement that caught everyone's attention. We were all expecting these great new features to be available in Spring 1.3, the supposed next version of Spring. Much to our surprise, Rod informed us that there would be no Spring 1.3; the next version would be Spring 2.0.

The decision to bump up the major version number of the next release isn't made lightly. Such an action connotes a significant advance in Spring. If the next version of Spring would be 2.0, then we could expect major enhancements. Indeed, ten months later, Spring 2.0 would be released with an abundance of new capabilities, including:

- Simplified XML configuration and the option to create custom configuration elements
- Greatly simplified AOP and transactions
- Support for Java 5 annotations for declaring aspects, transactions, and required bean properties
- The ability to create beans from scripts written in JRuby, Groovy, or BeanShell
- New JDBC templates to support named parameters and Java 5 features
- Improved JMS support, including receiving messages asynchronously (for creating message-driven POJOs)
- A new form-binding JSP tag library
- Several convention-over-configuration improvements to reduce the amount of XML required to configure Spring
- Support for the Java Persistence API (JPA)
- Enhanced bean scoping, including request and session scoping of beans for web applications
- The ability to perform dependency injection on objects that Spring doesn't create (such as domain objects)

At one point in his keynote, Rod said that if the wealth of new features being introduced didn't justify a jump to 2.0, then how would they ever be able to justify a 2.0 release?

That's not all. In addition to the work being done on the core Spring Framework, several interesting Spring-related projects were underway to provide additional capabilities on top of Spring. Among them:

- Spring Web Flow, which is based on Spring MVC and enables development of flow-based web applications
- XFire, for exporting your Spring beans as SOAP web services
- Spring-WS for creating contract-first web services
- Spring Modules, which provides (among other things) declarative caching and validation
- Direct Web Remoting (DWR) for Ajax-enabling Spring beans
- Lingo, which makes it possible to asynchronously invoke methods on remote beans

Then it occurred to me: if all of these new advances in Spring didn't justify a second edition of *Spring in Action*, then what would? As it turned out, Manning was thinking the same thing.

And now, well over a year later, here's the long-awaited update to *Spring in Action* that covers many of the new features of Spring 2.0. It has taken me a lot longer to finish than I had planned, but I hope that it was worth the wait. My goal for this edition is the same as with the first: to share the joy of developing in Spring. I hope this book will serve to enhance your enjoyment of Spring.

preface to the first edition

Software developers need to have a number of traits in order to practice their craft well. First, they must be good analytical thinkers and problem solvers. A developer’s primary role is to create software that solves business problems. This requires analyzing customer needs and coming up with successful, creative solutions.

They also need to be curious. Developments in the software industry are moving targets, always evolving. New frameworks, new techniques, new languages, and new methodologies are constantly emerging. Each one is a new tool that needs to be mastered and added to the toolbox, allowing the developer to do his or her job better and faster.

Then there is the most cherished trait of all, “laziness.” The kind of laziness that motivates developers to work hard to seek out solutions with the least amount of effort. It was with curiosity, a good dose of “laziness,” and all the analytical abilities we could muster that the two of us struck out together four years ago to find new ways to develop software.

This was the time when open source software was reaching critical mass in the Java community. Tons of open source frameworks were blossoming on the Java landscape. In order to decide to adopt one, it had to hit the sweet spot of our needs—it had to do 80% of what we needed right out of the box. And for any functionality that was not right out of the box, the framework needed to be easily extendible so that functionality too would be included. Extending didn’t mean

kludging in some hack that was so ugly you felt dirty afterwards—it meant extending in an elegant fashion. That wasn't too much to ask, right?

The first of these frameworks that gained immediate adoption on our team was Ant. From the get-go, we could tell that Ant had been created by another developer who knew our pain in building Java applications. From that moment on, no more `javac`. No more `CLASSPATH`. All this with a straightforward (albeit sometimes verbose) XML configuration. Huzzah! Life (and builds) just got easier.

As we went along, we began adopting more and more tools. Eclipse became our IDE of choice. Log4J became our (and everybody else's) default logging toolkit. And Lucene supplanted our commercial search solution. Each of these tools met our criteria of filling a need while being easy to use, understand, and extend.

But something was lacking. These great tools were designed to help develop software, like Ant and Eclipse, or to serve a very specific application need, like searching in the case of Lucene and logging for Log4J. None of them addressed the needs at the heart of enterprise applications: persistence, transactions, and integration with other enterprise resources.

That all changed in the last year or so when we discovered the remarkable one-two enterprise punch of Spring and Hibernate. Between these two frameworks nearly all of our middle- and data-tier needs were met.

We first adopted Hibernate. It was the most intuitive and feature-rich object/relational mapping tool out there. But it was by adopting Spring that we really got our code to look good. With Spring's dependency injection, we were able to get rid of all our custom factories and configurers. In fact, that is the reason we first integrated Spring into our applications. Its wiring allowed us to streamline our application configurations and move away from homegrown solutions. (Hey, every developer likes writing his own framework. But sometimes you just have to let go!)

We quickly discovered a nice bonus: Spring also provided very easy integration with Hibernate. This allowed us to ditch our custom Hibernate integration classes and use Spring's support instead. In turn, this led us directly to Spring's support for transparent persistence.

Look closely and you will see a pattern here. The more we used Spring, the more we discovered new features. And each feature we discovered was a pleasure to work with. Its web MVC framework worked nicely in a few applications. Its AOP support has been helpful in several places, primarily security. The JDBC support was quite nice for some smaller programs. Oh yeah, we also use it for scheduling. And JNDI access. And email integration. When it comes to hitting development sweet spots, Spring knocks the ball out of the park.

We liked Spring so much, we decided somebody should write a book about it. Fortunately, one of us had already written a book for Manning and knew how to go about doing this sort of thing. Soon that “somebody who should write a book” became us. In taking on this project we are trying to spread the gospel of Spring. The Spring framework has been nothing but a joy for us to work with—we predict it will be the same for you. And, we hope this book will be a pleasant vehicle for you to get to that point.

acknowledgments

Wow! It took a lot longer to get this book done than I thought it would. But there's no way you would be holding it in your hands if it weren't for the help, inspiration, and encouragement of all of the great folks behind the scenes.

First, I'd like to acknowledge the hard-working souls at Manning who miraculously turned my sloppily written manuscript into the fine piece of programming literature that is now before you: Marjan Bace, Mary Pierges, Cynthia Kane, Dotie Marsico, Karen Tegtmeier, Leslie Haimes, Liz Welch, Gabriel Dobrescu, Ron Tomich, Kerri Bonasch, Jackie Carter, Frank Blackwell, Michael Stephens, and Benjamin Berg.

I'd also like to thank the reviewers who took the time to provide feedback and criticism needed to shape the book: Doug Warren, Olivier Jolly, Matthew Payne, Bill Fly, Jonathon Esterhazy, Philip Hallstrom, Mark Chaimungkalanont, Eric Raymond, Dan Allen, George M. Jempty, Mojahedul Hasanat, Vlad Kofman, Ashik Uzzaman, Norman Richards, Jeff Cunningham, Stuart Caborn, Patrick Dennis, Bas Vodde, and Michael Masters. In addition, Erik Weibust and Valentin Crettaz did a second technical review of the manuscript, just before it went to press.

Then there are those people who didn't work on the book directly but had no less of an impact on me or on how this book turned out.

To my best friend, loving wife, and most beautiful woman in the world, Raymie. Thank you so much for your enduring patience another seemingly never-ending book project. I'm sorry that it took so long. Now that it's over, I owe you more flowers and date nights. And maybe some yard work.

My sweet and adorable little girls, Maisy and Madison: Thanks for your hugs and laughs and playtime that gave me a pleasant break from the book.

To Ryan Breidenbach, my coauthor on the first edition: Many thanks for helping me get this started and for your feedback on the second edition.

To the Spring team: No part of this book would be possible (or even necessary) without your vision and drive to create such an awesome framework. I'd especially like to thank Rod Johnson and Colin Sampaleanu for their comments on my blog and IM sessions that helped guide my thinking, as well as Arjen Poutsma for reviewing the Spring-WS chapter and keeping me in check.

To all of my coworkers over the past couple of years: I've learned many valuable things working alongside you and couldn't thank you more for your professionalism, dedication, and friendship: Jeff Hanson, Jim Wallace, Don Beale, Van Panyanouvong, James Tikalsky, Ryan Breidenbach, Marianna Krupin, Tonji Zimmerman, Jeff Wellen, Chris Howard, Derek Lane, Tom McGraw, Greg Vaughn, Doug Warren, Jon West, Peter Presland-Byrne, Ravi Varanasi, Srinivasa Penubothu, Gary Edwards, Greg Helton, Jacob Orshalick, Valerie Crowley, Tyler Osborne, Stephanie Co, Maggie Zhuang, Tim Sporcic, William Johnson, John Moore, Brian Eschbach, Chris Morris, Dave Sims, Andy Cline, Bear Cahill, Greg Graham, and Paul Nelson.

A shout-out to all of my other friends, colleagues, fellow nerds, people I've met at conferences, members of my LinkedIn list, and those who bribed me to put their name in the acknowledgments: James Bell, Daniel Brookshier, Scott Davis, Ben Galbraith, Bill Fly, Justin Gehtland, Pete Gekas, Robert Gleaton, Stu Holloway, Erik Hatcher, Rick Hightower, Ramnivas Laddad, Guillaume Laforge, Crazy Bob Lee, Ted Neward, Matt Raible, Leo Ramirez, Arun Rao, Norman Richards, Chris Richardson, James Strachan, Bruce Tate, Glenn Vanderburg, Becca Wheeler, and Jay Zimmerman.

And finally, my endless gratitude to Jack Bauer...for saving the world, 24 hours at a time.

about this book

The Spring Framework was created with a very specific goal in mind—to make developing JEE applications easier. Along the same lines, *Spring in Action* was written to make learning how to use Spring easier. My goal is not to give you a blow-by-blow listing of Spring APIs. Instead, I hope to present the Spring Framework in a way that is most relevant to a JEE developer by providing practical code examples from real-world experiences.

Since Spring is a modular framework, this book was written in the same way. I recognize that not all developers have the same needs. Some may want to learn the Spring Framework from the ground up, while others may want to pick and choose different topics and go at their own pace. That way, the book can act as a tool for learning Spring for the first time as well as a guide and reference for those wanting to dig deeper into specific features.

Roadmap

Spring in Action Second Edition is divided into three parts, plus two appendices. Each of the three parts focuses on a general area of the Spring Framework: the core framework, the business and data layers, and the presentation layer. While each part builds on the previous section, each is also able to stand on its own, allowing you to dive right into a certain topic without starting from the beginning.

In part 1, you'll explore the two core features of the Spring framework: dependency injection (DI) and aspect-oriented programming (AOP). This will give you a

good understanding of Spring's fundamentals that will be utilized throughout the book.

In chapter 1, you'll be introduced to DI and AOP and how they lend themselves to developing loosely coupled Java applications.

Chapter 2 takes a more detailed look at how to configure and associate your application objects using dependency injection. You will learn how to write loosely coupled components and wire their dependencies and properties within the Spring container using XML.

Once you've got the basics of bean wiring down, you'll be ready to look at some of the more advanced features of the Spring container in chapter 3. Among other things, you'll learn how to hook into the lifecycle of your application components, create parent/child relationships among your bean configurations, and wire in scripted components written in Ruby and Groovy.

Chapter 4 explores how to use Spring's AOP to decouple cross-cutting concerns from the objects that they service. This chapter also sets the stage for later chapters, where you'll use Spring AOP to provide declarative services such as transactions, security, and caching.

Part 2 builds on the DI and AOP features introduced in part 1 and shows you how to apply these concepts in the data and business tiers of your application.

Chapter 5 covers Spring's support for data persistence. You'll be introduced to Spring's JDBC support, which helps you remove much of the boilerplate code associated with JDBC. You'll also see how Spring integrates with several popular persistence frameworks such as Hibernate, iBATIS, and the Java Persistence API (JPA).

Chapter 6 complements chapter 5, showing you how to ensure integrity in your database using Spring's transaction support. You will see how Spring uses AOP to give simple application objects the power of declarative transactions.

In chapter 7 you will learn how to apply security to your application using Spring Security. You'll see how Spring Security secures application both at the web request level using servlet filters and at the method level using Spring AOP.

Chapter 8 explores how to expose your application objects as remote services. You'll also learn how to seamlessly access remote services as though they were any other object in your application. Remoting technologies explored will include RMI, Hessian/Burlap, SOAP-based web services, and Spring's own `HttpInvoker`.

Although chapter 8 covers web services in Spring, chapter 9 takes a different look at web services by examining the Spring-WS project. In this chapter, you'll learn how to use Spring-WS to build contract-first web services, in which the service's contract is decoupled from its implementation.

Chapter 10 looks at using Spring to send and receive asynchronous messages with JMS. In addition to basic JMS operations with Spring, you'll also learn how to use the open source Lingo project to expose and consume asynchronous remote services over JMS.

Even though Spring eliminates much of the need for EJBs, you may have a need to use both Spring and EJB together. Therefore, chapter 11 explores how to integrate Spring with EJB. You'll learn how to write Spring-enabled EJBs, how to wire EJB references into your Spring application context, and even how to use EJB-like annotations to configure your Spring beans.

Wrapping up part 2, chapter 12 will show you how to use Spring to schedule jobs, send e-mails, access JNDI-configured resources, and manage your application objects with JMX.

Part 3 moves the discussion of Spring a little closer to the end user by looking at the ways to use Spring to build web applications.

Chapter 13 introduces you to Spring's own MVC web framework. You will discover how Spring can transparently bind web parameters to your business objects and provide validation and error handling at the same time. You will also see how easy it is to add functionality to your web applications using Spring's rich selection of controllers.

Picking up where chapter 13 leaves off, chapter 14 covers the view layer of Spring MVC. In this chapter, you'll learn how to map the output of a Spring MVC controller to a specific view component for rendering to the user. You'll see how to define application views using JSP, Velocity, FreeMarker, and Tiles. And you'll learn how to create non-HTML output such as PDF, Excel, and RSS from Spring MVC.

Chapter 15 explores Spring Web Flow, an extension to Spring MVC that enables development of conversational web applications. In this chapter you'll learn how to build web applications that guide the user through a specific flow.

Finally, chapter 16 shows you how to integrate Spring with other web frameworks. If you already have an investment in another web framework (or just have a preference), this chapter is for you. You'll see how Spring provides support for several of the most popular web frameworks, including Struts, WebWork, Tapestry, and JavaServer Faces (JSF).

Appendix A will get you started with Spring, showing you how to download Spring and configure Spring in either Ant or Maven 2.

One of the key benefits of loose coupling is that it makes it easier to unit-test your application objects. Appendix B shows you how to take advantage of dependency injection and some of Spring's test-oriented classes for testing your applications.

Additional web content

As I was writing this book, I wanted to cover as much of Spring as possible. I got a little carried away and ended up writing more than could fit into the printed book. Just like with many Hollywood movies, a lot of material ended up on the cutting room floor:

- “*Building portlet applications*” This chapter covers the Spring Portlet MVC framework. Spring Portlet MVC is remarkably similar to Spring MVC (it even reuses some of Spring MVC’s classes), but is geared for the special circumstances presented by portlet applications.
- *Appendix C, “Spring XML configuration reference”* This appendix documents all of the XML configuration elements available in Spring 2.0. In addition, it includes the configuration elements for Spring Web Flow and Direct Web Remoting (DWR).
- *Appendix D, “Spring JSP tag library reference”* This appendix documents all of the JSP tags, both the original Spring JSP tags and the new form-binding tags from Spring 2.0.
- *Appendix E, “Spring Web Flow definition reference”* This appendix catalogs all of the XML elements that are used to define a flow for Spring Web Flow.
- *Appendix F, “Customizing Spring configuration”* This appendix, which was originally part of chapter 3, shows you how to create custom Spring XML configuration namespaces.

There’s some good stuff in there and I didn’t want that work to be for naught. So I convinced Manning to give it all of the same attention that it would get if it were to be printed and to make it available to download for free. You’ll be able to download this bonus material online at <http://www.manning.com/SpringinAction>.

Who should read this book

Spring in Action Second Edition is for all Java developers, but enterprise Java developers will find it particularly useful. While I will guide you along gently through code examples that build in complexity throughout each chapter, the true power of Spring lies in its ability to make enterprise applications easier to develop. Therefore, enterprise developers will most fully appreciate the examples presented in this book.

Because a vast portion of Spring is devoted to providing enterprise services, many parallels can be drawn between Spring and EJB. Therefore, any experience you have will be useful in making comparisons between these two frameworks.

Finally, while this book is not exclusively focused on web applications, a good portion of it is dedicated to this topic. In fact, the final four chapters demonstrate how Spring can support the development your applications' web layer. If you are a web application developer, you will find the last part of this book especially valuable.

Code conventions

There are many code example throughout this book. These examples will always appear in a fixed-width code font. If there is a part of example we want you to pay extra attention to, it will appear in a **bolded code** font. Any class name, method name, or XML fragment within the normal text of the book will appear in code font as well.

Many of Spring's classes and packages have exceptionally long (but expressive) names. Because of this, line-continuation markers (→) may be included when necessary.

Not all code examples in this book will be complete. Often we only show a method or two from a class to focus on a particular topic.

Complete source code for the application found throughout the book can be downloaded from the publisher's website at www.manning.com/walls3 or www.manning.com/SpringinAction.

About the author

Craig Walls is a software developer with more than 13 years' experience and is the coauthor of *XDoclet in Action* (Manning, 2003). He's a zealous promoter of the Spring Framework, speaking frequently at local user groups and conferences and writing about Spring on his blog. When he's not slinging code, Craig spends as much time as he can with his wife, two daughters, six birds, four dogs, two cats, and an ever-fluctuating number of tropical fish. Craig lives in Denton, Texas.

Author Online

Purchase of *Spring in Action* includes free access to a private web forum run by Manning Publications where you can make comments about the book, ask technical questions, and receive help from the authors and from other users. To access the forum and subscribe to it, point your web browser to www.manning.com/walls3 or www.manning.com/SpringinAction. This page provides information on how to get on the forum once you are registered, what kind of help is available, and the rules of conduct on the forum.

Manning's commitment to our readers is to provide a venue where a meaningful dialogue between individual readers and between readers and the author can take place. It is not a commitment to any specific amount of participation on the part of the author, whose contribution to the book's forum remains voluntary (and unpaid). We suggest you try asking the author some challenging questions, lest his interest stray!

The Author Online forum and the archives of previous discussions will be accessible from the publisher's website as long as the book is in print.

about the title

By combining introductions, overviews, and how-to examples, the *In Action* books are designed to help learning and remembering. According to research in cognitive science, the things people remember are things they discover during self-motivated exploration.

Although no one at Manning is a cognitive scientist, we are convinced that for learning to become permanent it must pass through stages of exploration, play, and, interestingly, retelling of what is being learned. People understand and remember new things, which is to say they master them, only after actively exploring them. Humans learn in action. An essential part of an *In Action* guide is that it is example-driven. It encourages the reader to try things out, to play with new code, and explore new ideas.

There is another, more mundane, reason for the title of this book: our readers are busy. They use books to do a job or to solve a problem. They need books that allow them to jump in and jump out easily and learn just what they want just when they want it. They need books that aid them *in action*. The books in this series are designed for such readers.

about the cover illustration

The figure on the cover of *Spring in Action Second Edition* is a “Le Caraco,” or an inhabitant of the province of Karak in southwest Jordan. Its capital is the city of Al-Karak, which boasts an ancient hilltop castle with magnificent views of the Dead Sea and surrounding plains.

The illustration is taken from a French travel book, *Encyclopedie des Voyages* by J. G. St. Saveur, published in 1796. Travel for pleasure was a relatively new phenomenon at the time and travel guides such as this one were popular, introducing both the tourist as well as the armchair traveler to the inhabitants of other regions of France and abroad.

The diversity of the drawings in the *Encyclopedie des Voyages* speaks vividly of the uniqueness and individuality of the world’s towns and provinces just 200 years ago. This was a time when the dress codes of two regions separated by a few dozen miles identified people uniquely as belonging to one or the other. The travel guide brings to life a sense of isolation and distance of that period and of every other historic period except our own hyperkinetic present.

Dress codes have changed since then and the diversity by region, so rich at the time, has faded away. It is now often hard to tell the inhabitant of one continent from another. Perhaps, trying to view it optimistically, we have traded a cultural and visual diversity for a more varied personal life. Or a more varied and interesting intellectual and technical life.

We at Manning celebrate the inventiveness, the initiative, and the fun of the computer business with book covers based on the rich diversity of regional life two centuries ago brought back to life by the pictures from this travel guide.

Part 1

Core Spring

Spring does a lot of things, but when you break it down to its core parts, Spring’s primary features are dependency injection (DI) and aspect-oriented programming (AOP). Starting in chapter 1, “Springing into action,” you’ll be given a quick overview of DI and AOP in Spring and see how they can help you to decouple application objects.

In chapter 2, “Basic bean wiring,” we’ll take a more in-depth look at how to keep all your application objects loosely coupled using DI. You’ll learn how to define your application’s objects and then wire them with dependencies in the Spring container using XML.

Turning it up a notch in chapter 3, “Advanced bean wiring,” we’ll explore some of the more advanced features of the Spring container and see how to use some of Spring’s more powerful configuration techniques.

Chapter 4, “Advising beans,” explores how to use Spring’s AOP features to decouple systemwide services (such as security and auditing) from the objects they service. This chapter sets the stage for chapters 6 and 7, where you’ll learn how to use Spring AOP to provide declarative transaction and security.

Springing into action

1

This chapter covers

- Exploring Spring's core modules
- Decoupling application objects
- Managing cross-cutting concerns with AOP

It all started with a bean.

In 1996, the Java programming language was still a young, exciting, up-and-coming platform. Many developers flocked to the language because they had seen how to create rich and dynamic web applications using applets. But they soon learned that there was more to this strange new language than animated juggling cartoon characters. Unlike any language before it, Java made it possible to write complex applications made up of discrete parts. They came for the applets, but they stayed for the components.

It was in December of that year that Sun Microsystems published the JavaBeans 1.00-A specification. JavaBeans defined a software component model for Java. This specification defined a set of coding policies that enabled simple Java objects to be reusable and easily composed into more complex applications. Although JavaBeans were intended as a general-purpose means of defining reusable application components, they were primarily used as a model for building user interface widgets. They seemed too simple to be capable of any “real” work. Enterprise developers wanted more.

Sophisticated applications often require services such as transaction support, security, and distributed computing—services not directly provided by the JavaBeans specification. Therefore, in March 1998, Sun published the 1.0 version of the Enterprise JavaBeans (EJB) specification. This specification extended the notion of Java components to the server side, providing the much-needed enterprise services, but failed to continue the simplicity of the original JavaBeans specification. In fact, except in name, EJB bears little resemblance to the original JavaBeans specification.

Despite the fact that many successful applications have been built based on EJB, EJB never achieved its intended purpose: to simplify enterprise application development. It is true that EJB’s declarative programming model simplifies many infrastructural aspects of development, such as transactions and security. However, in a different way, EJBs complicate development by mandating deployment descriptors and plumbing code (home and remote/local interfaces). Over time, many developers became disenchanted with EJB. As a result, its popularity has started to wane in recent years, leaving many developers looking for an easier way.

Today, Java component development has returned to its roots. New programming techniques, including aspect-oriented programming (AOP) and dependency injection (DI), are giving JavaBeans much of the power previously reserved for EJBs. These techniques furnish plain-old Java objects (POJOs) with a declarative programming model reminiscent of EJB, but without all of EJB’s complexity.

No longer must you resort to writing an unwieldy EJB component when a simple JavaBean will suffice.

In all fairness, even EJBs have evolved to promote a POJO-based programming model. Employing ideas such as DI and AOP, the latest EJB specification is significantly simpler than its predecessors. For many developers, though, this move is too little, too late. By the time the EJB 3 specification had entered the scene, other POJO-based development frameworks had already established themselves as de facto standards in the Java community.

Leading the charge for lightweight POJO-based development is the Spring Framework, which we'll be exploring throughout this book. In this chapter, we're going to explore the Spring Framework at a high level, giving you a taste of what Spring is all about. This chapter will give you a good idea of the types of problems Spring solves and will set the stage for the rest of the book. First things first—let's find out what Spring is.

1.1 What is Spring?

Spring is an open source framework, created by Rod Johnson and described in his book *Expert One-on-One: J2EE Design and Development*. It was created to address the complexity of enterprise application development. Spring makes it possible to use plain-vanilla JavaBeans to achieve things that were previously only possible with EJBs. However, Spring's usefulness isn't limited to server-side development. Any Java application can benefit from Spring in terms of simplicity, testability, and loose coupling.

NOTE To avoid ambiguity, I'll use the word "bean" when referring to conventional JavaBeans and "EJB" when referring to Enterprise JavaBeans. I'll also throw around the term "POJO" (plain-old Java object) from time to time.

Spring does many things, but when you strip it down to its base parts, Spring is a lightweight dependency injection and aspect-oriented container and framework. That's quite a mouthful, but it nicely summarizes Spring's core purpose. To make more sense of Spring, let's break this description down:

- *Lightweight*—Spring is lightweight in terms of both size and overhead. The bulk of the Spring Framework can be distributed in a single JAR file that weighs in at just over 2.5 MB. And the processing overhead required by Spring is negligible. What's more, Spring is nonintrusive: objects in a

Spring-enabled application often have no dependencies on Spring-specific classes.

- *Dependency Injection*—Spring promotes loose coupling through a technique known as dependency injection (DI). When DI is applied, objects are passively given their dependencies instead of creating or looking for dependent objects for themselves. You can think of DI as JNDI in reverse—instead of an object looking up dependencies from a container, the container gives the dependencies to the object at instantiation without waiting to be asked.
- *Aspect-oriented*—Spring comes with rich support for aspect-oriented programming (AOP) that enables cohesive development by separating application business logic from system services (such as auditing and transaction management). Application objects do what they're supposed to do—perform business logic—and nothing more. They are not responsible for (or even aware of) other system concerns, such as logging or transactional support.
- *Container*—Spring is a container in the sense that it contains and manages the lifecycle and configuration of application objects. In Spring, you can declare how each of your application objects should be created, how they should be configured, and how they should be associated with each other.
- *Framework*—Spring makes it possible to configure and compose complex applications from simpler components. In Spring, application objects are composed declaratively, typically in an XML file. Spring also provides much infrastructure functionality (transaction management, persistence framework integration, etc.), leaving the development of application logic to you.

To restate: When you strip Spring down to its base parts, what you get is a framework that helps you develop loosely coupled application code. Even if that were all that Spring did, the benefits of loose coupling (maintainability and testability) would make Spring a worthwhile framework to build applications on.

But Spring is more. The Spring Framework comes with several modules that build on the foundation of DI and AOP to create a feature-filled platform on which to build applications.

1.1.1 *Spring modules*

The Spring Framework is made up of several well-defined modules (see figure 1.1). When taken as a whole, these modules give you everything you need to develop enterprise-ready applications. But you don't have to base your

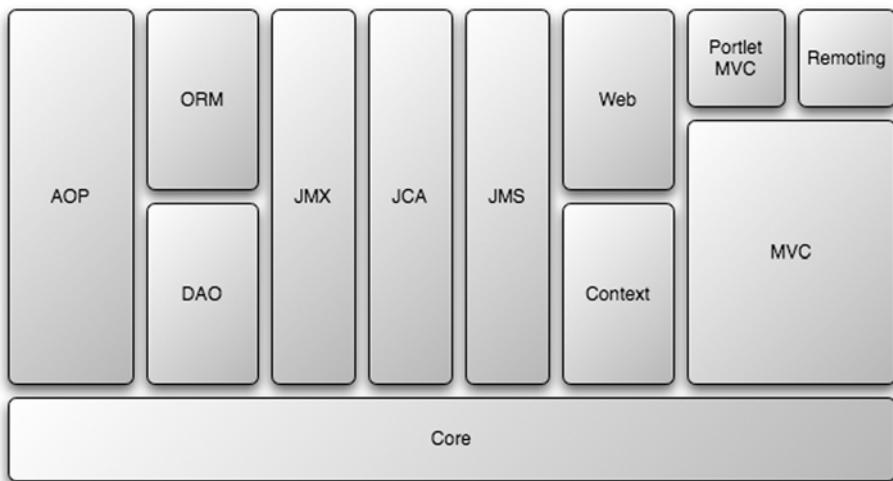


Figure 1.1 The Spring Framework is composed of several well-defined modules built on top of the core container. This modularity makes it possible to use as much or as little of the Spring Framework as is needed in a particular application.

application fully on the Spring Framework. You are free to choose the modules that suit your application and look to other options when Spring doesn't fit the bill. In fact, Spring offers integration points with several other frameworks and libraries so you won't have to write them yourself.

As you can see, all of Spring's modules are built on top of the core container. The container defines how beans are created, configured, and managed—more of the nuts and bolts of Spring. You will implicitly use these classes when you configure your application. But as a developer, you will most likely be interested in the other modules that leverage the services provided by the container. These modules will provide the frameworks with which you will build your application's services, such as AOP and persistence.

Let's take a look at each of Spring's modules in figure 1.1, one at a time, to see how each fits into the overall Spring picture.

The core container

At the very base of figure 1.1, you'll find Spring's core container. Spring's core container provides the fundamental functionality of the Spring Framework. This module contains the `BeanFactory`, which is the fundamental Spring container and the basis on which Spring's DI is based.

We'll be discussing the core module (the center of any Spring application) throughout this book, starting in chapter 2, when we examine bean wiring using DI.

Application context module

Spring's application context builds on the core container. The core module's BeanFactory makes Spring a container, but the context module is what makes it a framework. This module extends the concept of BeanFactory, adding support for internationalization (I18N) messages, application lifecycle events, and validation.

In addition, this module supplies many enterprise services such as email, JNDI access, EJB integration, remoting, and scheduling. Also included is support for integration with templating frameworks such as Velocity and FreeMarker.

Spring's AOP module

Spring provides rich support for aspect-oriented programming in its AOP module. This module serves as the basis for developing your own aspects for your Spring-enabled application. Like DI, AOP supports loose coupling of application objects. With AOP, however, applicationwide concerns (such as transactions and security) are decoupled from the objects to which they are applied.

Spring's AOP module offers several approaches to building aspects, including building aspects based on AOP Alliance interfaces (<http://aopalliance.sf.net>) and support for AspectJ. We'll dig into Spring's AOP support in chapter 4.

JDBC abstraction and the DAO module

Working with JDBC often results in a lot of boilerplate code that gets a connection, creates a statement, processes a result set, and then closes the connection. Spring's JDBC and Data Access Objects (DAO) module abstracts away the boilerplate code so that you can keep your database code clean and simple, and prevents problems that result from a failure to close database resources. This module also builds a layer of meaningful exceptions on top of the error messages given by several database servers. No more trying to decipher cryptic and proprietary SQL error messages!

In addition, this module uses Spring's AOP module to provide transaction management services for objects in a Spring application.

We'll see how Spring's template-based JDBC abstraction can greatly simplify JDBC code when we look at Spring data access in chapter 5.

Object-relational mapping (ORM) integration module

For those who prefer using an object-relational mapping (ORM) tool over straight JDBC, Spring provides the ORM module. Spring's ORM support builds on the DAO support, providing a convenient way to build DAOs for several ORM solutions. Spring doesn't attempt to implement its own ORM solution, but does provide hooks into several popular ORM frameworks, including Hibernate, Java Persistence API, Java Data Objects, and iBATIS SQL Maps. Spring's transaction management supports each of these ORM frameworks as well as JDBC.

In addition to Spring's template-based JDBC abstraction, we'll look at how Spring provides a similar abstraction for ORM and persistence frameworks in chapter 5.

Java Management Extensions (JMX)

Exposing the inner workings of a Java application for management is a critical part of making an application production ready. Spring's JMX module makes it easy to expose your application's beans as JMX MBeans. This makes it possible to monitor and reconfigure a running application.

We'll take a look at Spring's support for JMX in chapter 12.

Java EE Connector API (JCA)

The enterprise application landscape is littered with a mishmash of applications running on an array of disparate servers and platforms. Integrating these applications can be tricky. The Java EE Connection API (better known as JCA) provides a standard way of integrating Java applications with a variety of enterprise information systems, including mainframes and databases.

In many ways, JCA is much like JDBC, except where JDBC is focused on database access, JCA is a more general-purpose API connecting to legacy systems. Spring's support for JCA is similar to its JDBC support, abstracting away JCA's boilerplate code into templates.

The Spring MVC framework

The Model/View/Controller (MVC) paradigm is a commonly accepted approach to building web applications such that the user interface is separate from the application logic. Java has no shortage of MVC frameworks, with Apache Struts, JSF, WebWork, and Tapestry among the most popular MVC choices.

Even though Spring integrates with several popular MVC frameworks, it also comes with its own very capable MVC framework that promotes Spring's loosely coupled techniques in the web layer of an application.

We'll dig into Spring MVC in chapters 13 and 14.

Spring Portlet MVC

Many web applications are page based—that is, each request to the application results in a completely new page being displayed. Each page typically presents a specific piece of information or prompts the user with a specific form. In contrast, portlet-based applications aggregate several bits of functionality on a single web page. This provides a view into several applications at once.

If you’re building portlet-enabled applications, you’ll certainly want to look at Spring’s Portlet MVC framework. Spring Portlet MVC builds on Spring MVC to provide a set of controllers that support Java’s portlet API.

Spring’s web module

Spring MVC and Spring Portlet MVC require special consideration when loading the Spring application context. Therefore, Spring’s web module provides special support classes for Spring MVC and Spring Portlet MVC.

The web module also contains support for several web-oriented tasks, such as multipart file uploads and programmatic binding of request parameters to your business objects. It also contains integration support with Apache Struts and Java-Server Faces (JSF).

Remoting

Not all applications work alone. Oftentimes, it’s necessary for an application to leverage the functionality of another application to get its work done. When the other application is accessed over the network, some form of remoting is used for communication.

Spring’s remoting support enables you to expose the functionality of your Java objects as remote objects. Or if you need to access objects remotely, the remoting module also makes simple work of wiring remote objects into your application as if they were local POJOs. Several remoting options are available, including Remote Method Invocation (RMI), Hessian, Burlap, JAX-RPC, and Spring’s own HTTP Invoker.

In chapter 8, we’ll explore the various remoting options supported in Spring.

Java Message Service (JMS)

The downside to remoting is that it depends on network reliability and that both ends of the communication be available. Message-oriented communication, on the other hand, is more reliable and guarantees delivery of messages, even if the network and endpoints are unreliable.

Spring’s Java Message Service (JMS) module helps you send messages to JMS message queues and topics. At the same time, this module also helps you create

message-driven POJOs that are capable of consuming asynchronous messages. We'll see how to use Spring to send messages in chapter 10.

Although Spring covers a lot of ground, it's important to realize that Spring avoids reinventing the wheel whenever possible. Spring leans heavily on existing APIs and frameworks. For example, as we'll see later in chapter 5, Spring doesn't implement its own persistence framework—instead, it fosters integration with several capable persistence frameworks, including simple JDBC, iBATIS, Hibernate, and JPA.

Now that you've seen the big picture, let's see how Spring's DI and AOP features work. We'll get our feet wet by wiring our first bean into the Spring container.

1.2 A Spring jump start

Dependency injection is the most basic thing that Spring does. But what does DI look like? In the grand tradition of programming books, I'll start by showing you how Spring works with the proverbial "Hello World" example. Unlike the original Hello World program, however, this example will be modified a bit to demonstrate the basics of Spring.

The first class that the "Springified" Hello World example needs is a service class whose purpose is to print the familiar greeting. Listing 1.1 shows the `GreetingService` interface, which defines the contract for the service class.

Listing 1.1 The interface for a greeting service

```
package com.springinaction.chapter01.hello;
public interface GreetingService {
    void sayGreeting();
}
```

`GreetingServiceImpl` (listing 1.2) implements the `GreetingService` interface. Although it's not necessary to hide the implementation behind an interface, it's highly recommended as a way to separate the implementation from its contract.

Listing 1.2 `GreetingServiceImpl`, which prints a friendly greeting

```
package com.springinaction.chapter01.hello;
public class GreetingServiceImpl implements GreetingService {
    private String greeting;
    public GreetingServiceImpl() {}
    public GreetingServiceImpl(String greeting) {
        this.greeting = greeting;
    }
}
```

```
public void sayGreeting() {  
    System.out.println(greeting);  
}  
public void setGreeting(String greeting) {  
    this.greeting = greeting;  
}  
}
```

The GreetingServiceImpl class has a single property: greeting. This property is simply a String that holds the message that will be printed when the sayGreeting() method is called. You may have noticed that greeting can be set in two different ways: by the constructor or by the property's setter method.

What's not apparent just yet is who will make the call to either the constructor or the setGreeting() method to set the property. As it turns out, we're going to let the Spring container set the greeting property. The Spring configuration file (hello.xml) in listing 1.3 tells the container how to configure the greeting service.

Listing 1.3 Configuring Hello World in Spring

```
<?xml version="1.0" encoding="UTF-8"?>  
<beans xmlns="http://www.springframework.org/schema/beans"  
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
       xsi:schemaLocation="http://www.springframework.org/schema/beans  
                           http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">  
  
    <bean id="greetingService"  
          class="com.springinaction.chapter01.hello.GreetingServiceImpl">  
        <property name="greeting" value="Buenos Dias!" />  
    </bean>  
</beans>
```

The XML file in listing 1.3 declares an instance of a GreetingServiceImpl in the Spring container and configures its greeting property with a value of “Buenos Dias!” Let's dig into the details of this XML file a bit to understand how it works.

At the root of this simple XML file is the <beans> element, which is the root element of any Spring configuration file. The <bean> element is used to tell the Spring container about a class and how it should be configured. Here, the id attribute is used to name the bean greetingService and the class attribute specifies the bean's fully qualified class name.

Within the <bean> element, the <property> element is used to set a property, in this case the greeting property. As shown here, the <property> element tells

the Spring container to call `setGreeting()`, passing in “Buenos Dias!” (for a bit of Spanish flair) when instantiating the bean.

The following snippet of code illustrates roughly what the container does when instantiating the greeting service based on the XML definition in listing 1.3:

```
GreetingServiceImpl greetingService = new GreetingServiceImpl();
greetingService.setGreeting("Buenos Dias!");
```

Alternatively, you may choose to have Spring set the greeting property through `GreetingServiceImpl`'s single argument constructor. For example:

```
<bean id="greetingService"
      class="com.springinaction.chapter01.hello.GreetingServiceImpl">
    <constructor-arg value="Buenos Dias!" />
</bean>
```

The following code illustrates how the container will instantiate the greeting service when using the `<constructor-arg>` element:

```
GreetingServiceImpl greetingService =
    new GreetingServiceImpl("Buenos Dias");
```

The last piece of the puzzle is the class that loads the Spring container and uses it to retrieve the greeting service. Listing 1.4 shows this class.

Listing 1.4 The Hello World main class

```
package com.springinaction.chapter01.hello;
import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.FileSystemResource;

public class HelloApp {
    public static void main(String[] args) throws Exception {
        BeanFactory factory =
            new XmlBeanFactory(new FileSystemResource("hello.xml"));

        GreetingService greetingService =
            (GreetingService) factory.getBean("greetingService");

        greetingService.sayGreeting();
    }
}
```

The `BeanFactory` class used here is the Spring container. After loading the `hello.xml` file into the container, the `main()` method calls the `getBean()` method on the `BeanFactory` to retrieve a reference to the greeting service. With this

reference in hand, it finally calls the `sayGreeting()` method. When you run the Hello application, it prints (not surprisingly)

```
Buenos Dias!
```

This is about as simple a Spring-enabled application as I can come up with. Despite its simplicity, however, it does illustrate the basics of configuring and using a class in Spring. Unfortunately, it is perhaps too simple because it only illustrates how to configure a bean by injecting a `String` value into a property. The real power of Spring lies in how beans can be injected into other beans using DI.

1.3 ***Understanding dependency injection***

Although Spring does a lot of things, DI is at the heart of the Spring Framework. It may sound a bit intimidating, conjuring up notions of a complex programming technique or design pattern. But as it turns out, DI is not nearly as complex as it sounds. In fact, by applying DI in your projects, you'll find that your code will become significantly simpler, easier to understand, and easier to test.

But what does “dependency injection” mean?

1.3.1 ***Injecting dependencies***

Originally, dependency injection was commonly referred to by another name: inversion of control. But in an article written in early 2004, Martin Fowler asked what aspect of control is being inverted. He concluded that it is the acquisition of dependencies that is being inverted. Based on that revelation, he coined the phrase “dependency injection,” a term that better describes what is going on.

Any nontrivial application (pretty much anything more complex than `HelloWorld.java`) is made up of two or more classes that collaborate with each other to perform some business logic. Traditionally, each object is responsible for obtaining its own references to the objects it collaborates with (its dependencies). This can lead to highly coupled and hard-to-test code.

When applying DI, objects are given their dependencies at creation time by some external entity that coordinates each object in the system. In other words, dependencies are injected into objects. So, DI means an inversion of responsibility with regard to how an object obtains references to collaborating objects (see figure 1.2).

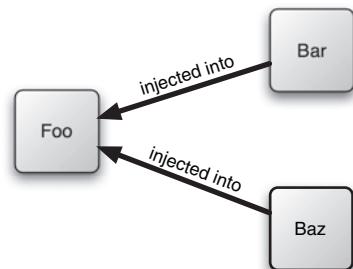


Figure 1.2 Dependency injection involves giving an object its dependencies as opposed to an object having to acquire those dependencies on its own.

The key benefit of DI is loose coupling. If an object only knows about its dependencies by their interface (not their implementation or how they were instantiated) then the dependency can be swapped out with a different implementation without the depending object knowing the difference.

For example, if the `Foo` class in figure 1.2 only knows about its `Bar` dependency through an interface then the actual implementation of `Bar` is of no importance to `Foo`. `Bar` could be a local POJO, a remote web service, an EJB, or a mock implementation for a unit test—`Foo` doesn't need to know or care.

If you're like me, you're probably anxious to see how this works in code. I aim to please, so without further delay...

1.3.2 Dependency injection in action

Suppose that your company's crack marketing team culled together the results of their expert market analysis and research and determined that what your customers need is a knight—that is, they need a Java class that represents a knight. After probing them for requirements, you learn that what they specifically want is for you to implement a class that represents an Arthurian knight of the Round Table who embarks on brave and noble quests to find the Holy Grail.

This is an odd request, but you've become accustomed to the strange notions and whims of the marketing team. So, without hesitation, you fire up your favorite IDE and bang out the class in listing 1.5.

Listing 1.5 A Knight of the Round Table bean

```
package com.springinaction.chapter01.knight;

public class KnightOfTheRoundTable {
    private String name;
    private HolyGrailQuest quest;

    public KnightOfTheRoundTable(String name) {
        this.name = name;
        quest = new HolyGrailQuest();
    }

    public HolyGrail embarkOnQuest()
        throws GrailNotFoundException {
        return quest.embark();
    }
}
```

In listing 1.5, the knight is given a name as a parameter of its constructor. Its constructor sets the knight's quest by instantiating a `HolyGrailQuest`. The implementation of `HolyGrailQuest` is fairly trivial, as shown in listing 1.6.

Listing 1.6 A query for the Holy Grail bean that will be given to the knight

```
package com.springinaction.chapter01.knight;
public class HolyGrailQuest {
    public HolyGrailQuest() {}

    public HolyGrail embark() throws GrailNotFoundException {
        HolyGrail grail = null;
        // Look for grail
        ...
        return grail;
    }
}
```

Satisfied with your work, you proudly check the code into version control. You want to show it to the marketing team, but deep down something doesn't feel right. You almost dismiss it as the burrito you had for lunch when you realize the problem: you haven't written any unit tests.

Knightly testing

Unit testing is an important part of development. Not only does it ensure that each individual unit functions as expected, but it also serves to document each unit in the most accurate way possible. Seeking to rectify your failure to write unit tests, you put together the test case (listing 1.7) for your knight class.

Listing 1.7 Testing the knight

```
package com.springinaction.chapter01.knight;
import junit.framework.TestCase;
public class KnightOfTheRoundTableTest extends TestCase {
    public void testEmbarkOnQuest() throws GrailNotFoundException {
        KnightOfTheRoundTable knight =
            new KnightOfTheRoundTable("Bedivere");
        HolyGrail grail = knight.embarkOnQuest();
        assertNotNull(grail);
        assertTrue(grail.isHoly());
    }
}
```

After writing this test case, you set out to write a test case for `HolyGrailQuest`. But before you even get started, you realize that the `KnightOfTheRoundTableTest` test case indirectly tests `HolyGrailQuest`. You also wonder if you are testing all contingencies. What would happen if `HolyGrailQuest`'s `embark()` method returned `null`? Or what if it were to throw a `GrailNotFoundException`?

Who's calling whom?

The main problem so far with `KnightOfTheRoundTable` is with how it obtains a `HolyGrailQuest`. Whether it is instantiating a new `HolyGrail` instance or obtaining one via JNDI, each knight is responsible for getting its own quest (as shown in figure 1.3). Therefore, you have no way to test the knight class in isolation. As it stands, every time you test `KnightOfTheRoundTable`, you will also indirectly test `HolyGrailQuest`.

What's more, you have no way of telling `HolyGrailQuest` to behave differently (e.g., return `null` or throw a `GrailNotFoundException`) for different tests. What would help is if you could create a mock implementation of `HolyGrailQuest` that lets you decide how it behaves. But even if you were to create a mock implementation, `KnightOfTheRoundTable` still retrieves its own `HolyGrailQuest`, meaning you would have to make a change to `KnightOfTheRoundTable` to retrieve the mock quest for testing purposes (and then change it back for production).

Decoupling with interfaces

The problem, in a word, is coupling. At this point, `KnightOfTheRoundTable` is statically coupled to `HolyGrailQuest`. They're handcuffed together in such a way that you can't have a `KnightOfTheRoundTable` without also having a `HolyGrailQuest`.

Coupling is a two-headed beast. On one hand, tightly coupled code is difficult to test, difficult to reuse, difficult to understand, and typically exhibits “whack-a-mole” bugs (i.e., fixing one bug results in the creation of one or more new bugs). On the other hand, completely uncoupled code doesn't do anything. In order to

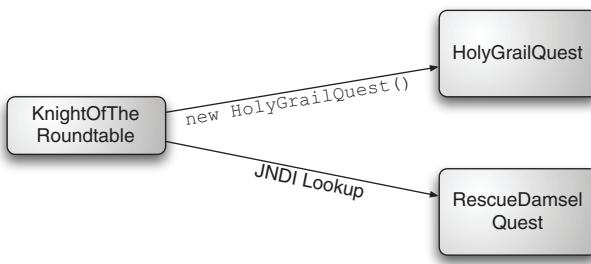


Figure 1.3
A knight is responsible for getting its own quest, through instantiation or some other means.

do anything useful, classes need to know about each other somehow. Coupling is necessary, but it should be managed carefully.

A common technique used to reduce coupling is to hide implementation details behind interfaces so that the actual implementation class can be swapped out without impacting the client class. For example, suppose you were to create a Quest interface:

```
package com.springinaction.chapter01.knight;
public interface Quest {
    abstract Object embark() throws QuestFailedException;
}
```

Then, you change `HolyGrailQuest` to implement this interface. Also, notice that `embark()` now returns an `Object` and throws a `QuestFailedException`.

```
package com.springinaction.chapter01.knight;
public class HolyGrailQuest implements Quest {
    public HolyGrailQuest() {}
    public Object embark() throws QuestFailedException {
        // Do whatever it means to embark on a quest
        return new HolyGrail();
    }
}
```

Also, the following method must change in `KnightOfTheRoundTable` to be compatible with these Quest types:

```
private Quest quest;
...
public Object embarkOnQuest() throws QuestFailedException {
    return quest.embark();
}
```

Likewise, you could also have `KnightOfTheRoundTable` implement the following Knight interface:

```
public interface Knight {
    Object embarkOnQuest() throws QuestFailedException;
}
```

Hiding your class's implementation behind interfaces is certainly a step in the right direction. But where many developers fall short is in how they retrieve a Quest instance. For example, consider this possible change to `KnightOfTheRoundTable`:

```
public class KnightOfTheRoundTable implements Knight {
    private String name;
    private Quest quest;
```

```
public KnightOfTheRoundTable(String name) {
    this.name = name;
    quest = new HolyGrailQuest();
}

public Object embarkOnQuest() throws QuestFailedException {
    return quest.embark();
}
```

Here the `KnightOfTheRoundTable` class embarks on a quest through the `Quest` interface. But the knight still retrieves a specific type of `Quest` (here a `HolyGrailQuest`). This isn't much better than before. A `KnightOfTheRoundTable` is stuck going only on quests for the Holy Grail and no other types of quest.

Giving and taking

The question you should be asking at this point is whether a knight should be responsible for obtaining a quest, or should a knight be given a quest to embark upon?

Consider the following change to `KnightOfTheRoundTable`:

```
public class KnightOfTheRoundTable implements Knight {
    private String name;
    private Quest quest;

    public KnightOfTheRoundTable(String name) {
        this.name = name;
    }

    public Object embarkOnQuest() throws QuestFailedException {
        return quest.embark();
    }

    public void setQuest(Quest quest) {
        this.quest = quest;
    }
}
```

Notice the difference? Compare figure 1.4 with figure 1.3 to see the difference in how a knight obtains its quest.

Now the knight is given a quest instead of retrieving one itself. `KnightOfTheRoundTable` is no longer responsible for retrieving its own quests. And because it only knows about a quest through the `Quest` interface, you could give a knight any implementation of `Quest` you want. In one configuration, you might give it a `HolyGrailQuest`. In a different configuration, maybe a different `Quest` implementation, such as `RescueDamselQuest`, will be given to the knight. Similarly, in a test case you would give it a mock implementation of `Quest`.

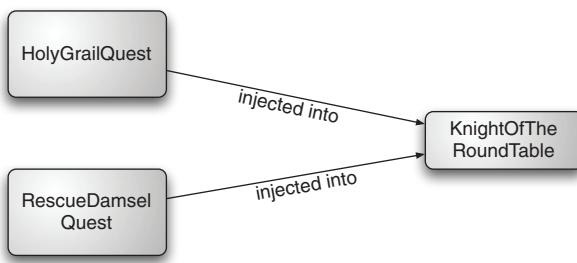


Figure 1.4
The knight is no longer responsible for getting its own quest. Instead, it is given (injected with) a quest through its `setQuest()` method.

In a nutshell, that is what DI is all about: the responsibility of coordinating collaboration between dependent objects is transferred away from the objects themselves.

Assigning a quest to a knight

Now that you've written your `KnightOfTheRoundTable` class to be given any arbitrary `Quest` object, how can you specify which `Quest` it should be given?

The act of creating associations between application components is referred to as wiring. In Spring, there are many ways to wire components together, but the most common approach is via XML. Listing 1.8 shows a simple Spring configuration file, `knight.xml`, that gives a quest (specifically, a `HolyGrailQuest`) to a `KnightOfTheRoundTable`.

Listing 1.8 Wiring a quest into a knight in the Spring configuration XML

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/
                           spring-beans-2.0.xsd">
    <!-- Defines a quest -->
    <bean id="quest"
          class="com.springinaction.chapter01.knight.HolyGrailQuest"/>
    <!-- Defines a knight -->
    <bean id="knight"
          class="com.springinaction.chapter01.knight.
                           KnightOfTheRoundTable">
        <constructor-arg value="Bedivere" />
        <property name="quest" ref="quest" />
    </bean>
</beans>
  
```

Defines a quest

Defines a knight

Sets the knight's name

Gives the knight a quest

This is just a simple approach to wiring beans. Don't worry too much about the details right now. In chapter 2 we'll explain more about what is going on here, as well as show you even more ways you can wire your beans in Spring.

Now that we've declared the relationship between a knight and a quest, we need to load up the XML file and kick off the application.

Seeing it work

In a Spring application, a BeanFactory loads the bean definitions and wires the beans together. Because the beans in the knight example are declared in an XML file, an XmlBeanFactory is the appropriate factory for this example. The `main()` method in listing 1.9 uses an XmlBeanFactory to load `knight.xml` and to get a reference to the `Knight` object.

Listing 1.9 Running the knight example

```
package com.springinaction.chapter01.knight;
import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.FileSystemResource;

public class KnightApp {                                Loads XML beans file
    public static void main(String[] args) throws Exception {
        BeanFactory factory =
            new XmlBeanFactory(new FileSystemResource("knight.xml"));

        Knight knight =                               Retrieves knight
            (Knight) factory.getBean("knight");      from factory

        knight.embarkOnQuest();                      Sends knight
    }                                              on its quest
}
```

Once the application has a reference to the `Knight` object, it simply calls the `embarkOnQuest()` method to kick off the knight's adventure. Notice that this class knows nothing about the quest the knight will take. Again, the only thing that knows which type of quest will be given to the knight is the `knight.xml` file.

It's been a lot of fun sending knights on quests using dependency injection, but now let's see how you can use DI in your real-world enterprise applications.

1.3.3 Dependency injection in enterprise applications

Suppose that you've been tasked with writing an online shopping application. Included in the application is an order service component to handle all functions related to placing an order. Figure 1.5 illustrates several ways that a web layer

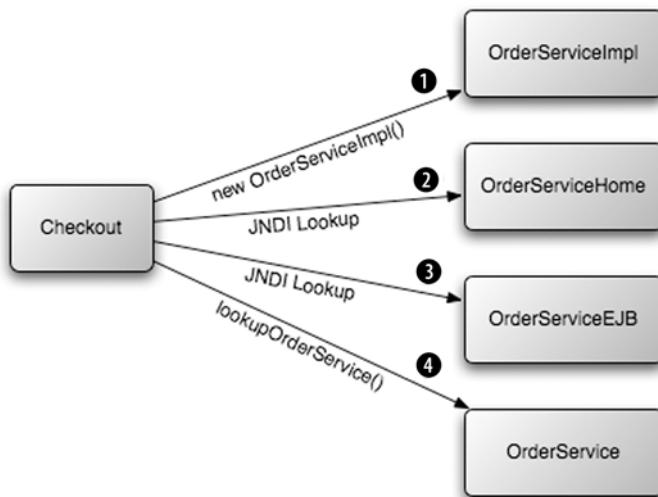


Figure 1.5 Conventional approaches to service lookup would lead to tight coupling between the checkout object and the order service.

Checkout component (which, perhaps, could be a WebWork action or a Tapestry page) might access the order service.

A simple, but naive, approach would be to directly instantiate the order service when it's needed ①. Aside from directly coupling the web layer to a specific service class, this approach will result in wasteful creation of the `OrderServiceImpl` class, when a shared, stateless singleton will suffice.

If the order service is implemented as a 2.x EJB, you would access the service by first retrieving the home interface through JNDI ②, which would then be used to access an implementation of the EJB's service interface. In this case, the web layer is no longer coupled to a specific interface, but it is coupled to JNDI and to the EJB 2.x programming model.

As an EJB 3 bean, the order service could be looked up from JNDI directly ③ (without going through a home interface). Again, there's no coupling to a specific implementation class, but there is a dependence on JNDI.

With or without EJB, you might choose to hide the lookup details behind a service locator ④. This would address the coupling concerns seen with the other approaches, but now the web layer is coupled to the service locator.

The key issue with all of these approaches is that the web layer component is too involved in obtaining its own dependencies. It knows too much about where the order service comes from and how it's implemented.

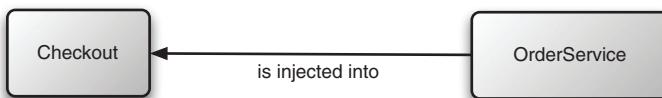


Figure 1.6 By injecting an `OrderService` into the `Checkout` component, `Checkout` is relieved from knowing how the service is implemented and where it is found.

If knowing too much about your dependencies leads to tightly coupled code, it stands to reason that knowing as little as possible about your dependencies leads to loosely coupled code. Consider figure 1.6, which shows how the `Checkout` component could be *given* an `OrderService` instead of asking for one.

Now let's see how this would be implemented using DI:

```
private OrderService orderService;  
  
public void doRequest(HttpServletRequest request) {  
    Order order = createOrder(request);  
    orderService.createOrder(order);  
}  
  
public void setOrderService(OrderService orderService) {  
    this.orderService = orderService;  
}
```

No lookup code! The reference to `OrderService` (which is an interface) is given to the class through the `setOrderService()` method. The web component does not know or care where the `OrderService` comes from. It could be injected by Spring or it could be manually injected by an explicit call to `setOrderService()`. It also has no idea as to how `OrderService` is implemented—it only knows about it through the `OrderService` interface. With DI, your application objects are freed from the burden of fetching their own dependencies and are able to focus on their tasks, trusting that their dependencies will be available when needed.

Dependency injection is a boon to loosely coupled code, making it possible to keep your application objects at arm's length from each other. However, we've only scratched the surface of the Spring container and DI. In chapters 2 and 3, you'll see more ways to wire objects in the Spring container.

Dependency injection is only one technique that Spring offers to POJOs in support of loose coupling. Aspect-oriented programming provides a different kind of decoupling power by separating application-spanning functionality (such as security and transactions) from the objects they affect. Let's take a quick look at Spring's support for AOP.

1.4 Applying aspect-oriented programming

Although DI makes it possible to tie software components together loosely, aspect-oriented programming enables you to capture functionality that is used throughout your application in reusable components.

1.4.1 Introducing AOP

Aspect-oriented programming is often defined as a programming technique that promotes separation of concerns within a software system. Systems are composed of several components, each responsible for a specific piece of functionality. Often, however, these components also carry additional responsibility beyond their core functionality. System services such as logging, transaction management, and security often find their way into components whose core responsibility is something else. These system services are commonly referred to as cross-cutting concerns because they tend to cut across multiple components in a system.

By spreading these concerns across multiple components, you introduce two levels of complexity to your code:

- The code that implements the systemwide concerns is duplicated across multiple components. This means that if you need to change how those concerns work, you'll need to visit multiple components. Even if you've abstracted the concern to a separate module so that the impact to your components is a single method call, that single method call is duplicated in multiple places.
- Your components are littered with code that isn't aligned with their core functionality. A method to add an entry to an address book should only be concerned with how to add the address and not with whether it is secure or transactional.

Figure 1.7 illustrates this complexity. The business objects on the left are too intimately involved with the system services. Not only does each object know that it is being logged, secured, and involved in a transactional context, but also each object is responsible for performing those services for itself.

AOP makes it possible to modularize these services and then apply them declaratively to the components that they should affect. This results in components that are more cohesive and that focus on their own specific concerns, completely ignorant of any system services that may be involved. In short, aspects ensure that POJOs remain plain.

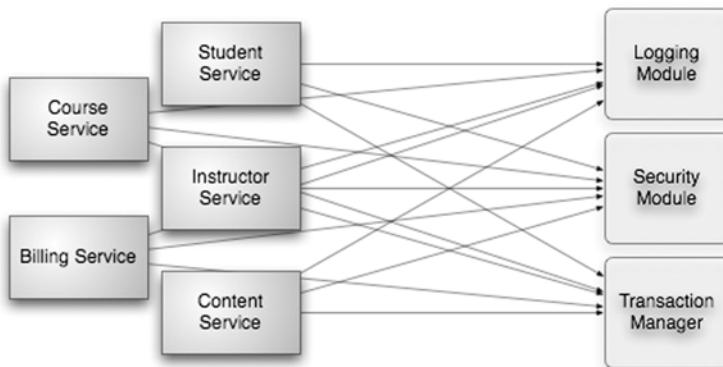


Figure 1.7 Calls to systemwide concerns such as logging and security are often scattered about in modules where those concerns are not their primary concern.

It may help to think of aspects as blankets that cover many components of an application, as illustrated in figure 1.8. At its core, an application consists of modules that implement the business functionality. With AOP, you can then cover your core application with layers of functionality. These layers can be applied declaratively throughout your application in a flexible manner without your core application even knowing they exist. This is a powerful concept, as it keeps the security, transaction, and logging concerns from littering the application's core business logic.

To demonstrate how aspects can be applied in Spring, let's revisit the knight example, adding a basic logging aspect.

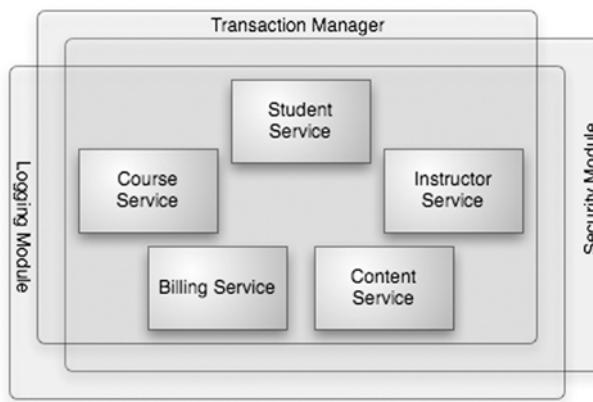


Figure 1.8
Using AOP, systemwide concerns blanket the components that they impact. This leaves the application components to focus on their specific business functionality.

1.4.2 AOP in action

Suppose that after showing your progress to marketing, they came back with an additional requirement. In this new requirement, a minstrel must accompany each knight, chronicling the actions and deeds of the knight in song.

Hmm...a minstrel who sings about a knight, eh? That doesn't sound too hard. Getting started, you create a `Minstrel` class, as shown in listing 1.10.

Listing 1.10 A Minstrel, a musically inclined logging component

```
package com.springinaction.chapter01.knight;

import org.apache.log4j.Logger;

public class Minstrel {
    private static final Logger SONG =
        Logger.getLogger(Minstrel.class);

    public void singBefore(Knight knight) {
        SONG.info("Fa la la; Sir " + knight.getName() +
            " is so brave!");
    }

    public void singAfter(Knight knight) {
        SONG.info("Tee-hee-he; Sir " + knight.getName() +
            " did embark on a quest!");
    }
}
```

**Sings
before
quest**

**Sings
after
quest**

In keeping with the dependency injection way of thinking, you alter `KnightOfTheRoundTable` to be given an instance of `Minstrel`:

```
public class KnightOfTheRoundTable implements Knight {
    ...
    private Minstrel minstrel;
    public void setMinstrel(Minstrel minstrel) {
        this.minstrel = minstrel;
    }
    ...
    public HolyGrail embarkOnQuest() throws QuestFailedException {
        minstrel.singBefore(this);
        HolyGrail grail = quest.embarke();
        minstrel.singAfter(this);
        return grail;
    }
}
```

That should do it! Oh wait... there's only one small problem. As it is, each knight must stop and tell the minstrel to sing a song before the knight can continue with his quest (as in figure 1.9). Then after the quest, the knight must remember to tell the minstrel to continue singing of his exploits. Having to remember to stop and tell a minstrel what to do can certainly impede a knight's quest-embarking.

Ideally, a minstrel would take more initiative and automatically sing songs without being explicitly told to do so. A knight shouldn't know (or really even care) that his deeds are being written into song. After all, you can't have your knight being late for quests because of a lazy minstrel.

In short, the services of a minstrel transcend the duties of a knight. Another way of stating this is to say that a minstrel's services (song writing) are orthogonal to a knight's duties (embarking on quests). Therefore, it makes sense to turn the minstrel into an aspect that adds his song-writing services to a knight. Then the minstrel's services would cover the functionality of the knight—all without the knight even knowing that the minstrel is there, as shown in figure 1.10.

As it turns out, it's rather easy to turn the `Minstrel` class in listing 1.10 into an aspect using Spring's AOP support. Let's see how.

Weaving the aspect

There are several ways to implement aspects in Spring, and we'll dig into all of them in chapter 4. But for the sake of this example, we'll use the new AOP namespace introduced in Spring 2.0. To get started, you'll want to be sure that you declare the namespace in the context definition XML:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/
                           spring-beans-2.0.xsd
                           http://www.springframework.org/schema/aop"
```

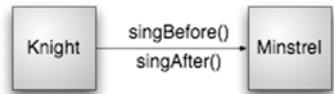


Figure 1.9 Without AOP, a knight must tell his minstrel to sing songs. This interferes with the knight's primary dragon-slaying and damsel-rescuing activities.



Figure 1.10 An aspect-oriented minstrel covers a knight, chronicling the knight's activities without the knight's knowledge of the minstrel.

```
http://www.springframework.org/schema/aop/spring-aop-2.0.xsd">
...
</beans>
```

With the namespace declared, we're ready to create the aspect. The bit of XML in listing 1.11 declares a minstrel as a bean in the Spring context and then creates an aspect that advises the knight bean.

Listing 1.11 Weaving MinstrelAdvice into a knight

```
<bean id="minstrel"
      class="com.springinaction.chapter01.knight.Minstrel"/> | Declares
                                                               | minstrel bean

<aop:config>
  <aop:aspect ref="minstrel" > <-- Creates minstrel
    <aop:pointcut > aspect
      id="questPointcut"
      expression="execution(* *.embarkOnQuest(..))
                   and target(bean)" /> | Creates pointcut
                                         | to wrap
                                         | embarkOnQuest()

    <aop:before > <-- Weaves in
      method="singBefore"
      pointcut-ref="questPointcut"
      arg-names="bean" /> | minstrel
                           | before

    <aop:after-returning > <-- Weaves in
      method="singAfter"
      pointcut-ref="questPointcut"
      arg-names="bean" /> | minstrel
                           | after

  </aop:aspect>
</aop:config>
```

There's a lot going on in listing 1.11, so let's break it down one bit at a time:

- The first thing we find is a `<bean>` declaration, creating a `minstrel` bean in Spring. This is the `Minstrel` class from listing 1.10. `Minstrel` doesn't have any dependencies, so there's no need to inject it with anything.
- Next up is the `<aop:config>` element. This element indicates that we're about to do some AOP stuff. Most of Spring's AOP configuration elements must be contained in `<aop:config>`.
- Within `<aop:config>` we have an `<aop:aspect>` element. This element indicates that we're declaring an aspect. The functionality of the aspect is defined in the bean that is referred to by the `ref` attribute. In this case, the `minstrel` bean, which is a `Minstrel`, will provide the functionality of the aspect.

- An aspect is made up of pointcuts (places where the aspect functionality will be applied) and advice (how to apply the functionality). The `<aop:pointcut>` element defines a pointcut that is triggered by the execution of an `embarkOnQuest()` method. (If you're familiar with AspectJ, you may recognize the pointcut as being expressed in AspectJ syntax.)
- Finally, we have two bits of AOP advice. The `<aop:before>` element declares that the `singBefore()` method of `Minstrel` should be called before the pointcut, while the `<aop:after>` element declares that the `singAfter()` method of `Minstrel` should be called after the pointcut. The pointcut in both cases is a reference to `questPointcut`, which is the execution of `embarkOnQuest()`.

That's all there is to it! We've just turned `Minstrel` into a Spring aspect. Don't worry if this doesn't make complete sense yet—you'll see plenty more examples of Spring AOP in chapter 4 that should help clear this up. For now, there are two important points to take away from this example.

First, `Minstrel` is still a POJO—there's nothing about `Minstrel` that indicates that it is to be used as an aspect. Instead, `Minstrel` was turned into an aspect declaratively in the Spring context.

Second, and perhaps more important, the knight no longer needs to tell the minstrel to sing about his exploits. As an aspect, the minstrel will take care of that automatically. In fact, the knight doesn't even need to know of the minstrel's existence. Consequently, the `KnightOfTheRoundTable` class can revert back to a simpler form as before:

```
public class KnightOfTheRoundTable implements Knight {  
    private String name;  
    private Quest quest;  
  
    public KnightOfTheRoundTable(String name) {  
        this.name = name;  
    }  
  
    public HolyGrail embarkOnQuest() throws QuestFailedException {  
        return quest.embarc();  
    }  
  
    public void setQuest(Quest quest) {  
        this.quest = quest;  
    }  
}
```

Using AOP to chronicle a knight's activities has been a lot of fun. But Spring's AOP can be used for even more practical things than composing ageless sonnets about

knights. As you'll see later, Spring employs AOP to provide enterprise services such as declarative transactions (chapter 6) and security (chapter 7).

1.5 **Summary**

You should now have a pretty good idea of what Spring brings to the table. Spring aims to make enterprise Java development easier and to promote loosely coupled code. Vital to this is dependency injection and AOP.

In this chapter, we got a small taste of dependency injection in Spring. DI is a way of associating application objects such that the objects don't need to know where their dependencies come from or how they're implemented. Rather than acquiring dependencies on their own, dependent objects are given the objects that they depend on. Because dependent objects often only know about their injected objects through interfaces, coupling is kept very low.

In addition to dependency injection, we also saw a glimpse of Spring's AOP support. AOP enables you to centralize logic that would normally be scattered throughout an application in one place—an aspect. When Spring wires your beans together, these aspects can be woven in at runtime, effectively giving the beans new behavior.

Dependency injection and AOP are central to everything in Spring. Thus you must understand how to use these principal functions of Spring to be able to use the rest of the framework. In this chapter, we've just scratched the surface of Spring's DI and AOP features. Over the next few chapters, we'll dig deeper into DI and AOP. Without further ado, let's move on to chapter 2 to learn how to wire objects together in Spring using dependency injection.

Basic bean wiring



This chapter covers

- Introducing the Spring container
- Declaring beans
- Injecting constructors and setters
- Wiring beans
- Controlling bean creation and destruction

Have you ever stuck around after a movie long enough to watch the credits? It's incredible how many different people it takes to pull together a major motion picture. There are the obvious participants: the actors, the scriptwriters, the directors, and the producers. Then there are the not-so-obvious: the musicians, the special effects crew, and the art directors. And that's not to mention the key grip, the sound mixer, the costumers, the make-up artists, the stunt coordinators, the publicists, the first assistant to the cameraperson, the second assistant to the cameraperson, the set designers, the gaffer, and (perhaps most importantly) the caterers.

Now imagine what your favorite movie would have been like had none of these people talked to one another. Let's say that they all showed up at the studio and started doing their own thing without any coordination of any kind. If the director keeps to himself and doesn't say "roll 'em," then the cameraperson won't start shooting. It probably wouldn't matter anyway, because the lead actress would still be in her trailer and the lighting wouldn't work because the gaffer would not have been hired. Maybe you've seen a movie where it looks like this is what happened. But most movies (the good ones anyway) are the product of thousands of people working together toward the common goal of making a blockbuster movie.

In this respect, a great piece of software isn't much different. Any nontrivial application is made up of several objects that must work together to meet some business goal. These objects must be aware of one another and communicate with one another to get their job done. In an online shopping application, for instance, an order manager component may need to work with a product manager component and a credit card authorization component. All of these will likely need to work with a data access component to read from and write to a database.

But as we saw in chapter 1, the traditional approach to creating associations between application objects (via construction or lookup) leads to complicated code that is difficult to reuse and unit test. In the best case, these objects do more work than they should. In the worst case, they are highly coupled to one another, making them hard to reuse and hard to test.

In Spring, objects are not responsible for finding or creating the other objects that they need to do their job. Instead, they are given references to the objects that they collaborate with by the container. An order manager component, for example, may need a credit card authorizer—but it doesn't need to create the credit card authorizer. It just needs to show up empty-handed and it will be given a credit card authorizer to work with.

The act of creating these associations between application objects is the essence of dependency injection (DI) and is commonly referred to as *wiring*. In this chapter we'll explore the basics of bean wiring using Spring. As DI is the most

elemental thing Spring does, these are techniques you'll use almost every time you develop Spring-based applications.

2.1 Containing your beans

In a Spring-based application, your application objects will live within the Spring container. As illustrated in figure 2.1, the container will create the objects, wire them together, configure them, and manage their complete lifecycle from cradle to grave (or new to `finalize()` as the case may be).

In section 2.2, we'll see how to configure Spring to know what objects it should create, configure, and wire together. First, however, it's important to get to know the container where your objects will be hanging out. Understanding the container helps you grasp how your objects will be managed.

The container is at the core of the Spring Framework. Spring's container uses dependency injection (DI) to manage the components that make up an application. This includes creating associations between collaborating components. As such, these objects are cleaner and easier to understand, support reuse, and are easy to unit-test.

There is no single Spring container. Spring comes with several container implementations that can be categorized into two distinct types. Bean factories (defined by the `org.springframework.beans.factory.BeanFactory` interface) are the simplest of containers, providing basic support for DI. Application contexts (defined by the `org.springframework.context.ApplicationContext` interface) build on the notion of a bean factory by providing application framework services, such as the ability to resolve textual messages from a properties file and the ability to publish application events to interested event listeners.

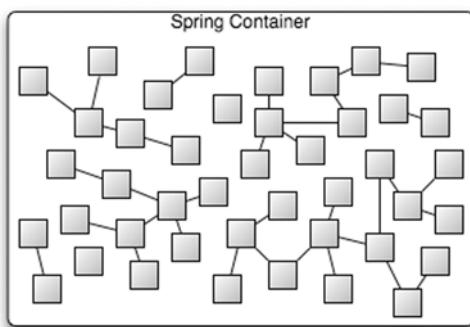


Figure 2.1
In a Spring application, objects are created, wired together, and live within the Spring container.

NOTE Although Spring uses the words “bean” and “JavaBean” liberally when referring to application components, this does not mean that a Spring component must follow the JavaBeans specification to the letter. A Spring component can be any type of POJO (plain-old Java object). In this book, I assume the loose definition of JavaBean, which is synonymous with POJO.

Let’s start our exploration of Spring containers with the most basic of the Spring containers: the `BeanFactory`.

2.1.1 **Introducing the BeanFactory**

As its name implies, a bean factory is an implementation of the Factory design pattern. That is, it is a class whose responsibility is to create and dispense beans. However, unlike many implementations of the Factory pattern, which often dole out a single type of object, a bean factory is a general-purpose factory, creating and dispensing many types of beans.

There’s more to a bean factory than simply instantiation and delivery of application objects. Because a bean factory knows about many objects within an application, it is able to create associations between collaborating objects as they are instantiated. This removes the burden of configuration from the bean itself and the bean’s client. As a result, when a bean factory hands out objects, those objects are fully configured, are aware of their collaborating objects, and are ready to use. What’s more, a bean factory also takes part in the lifecycle of a bean, making calls to custom initialization and destruction methods, if those methods are defined.

There are several implementations of `BeanFactory` in Spring. But the one that is most commonly used is `org.springframework.beans.factory.xml.XmlBeanFactory`, which loads its beans based on the definitions contained in an XML file.

To create an `XmlBeanFactory`, you must pass an instance of `org.springframework.core.io.Resource` to the constructor. The `Resource` object will provide the XML to the factory. Spring comes with a handful of `Resource` implementations as described in table 2.1.

For example, the following code snippet uses a `FileSystemResource` to create an `XmlBeanFactory` whose bean definitions are read from an XML file in the file system:

```
BeanFactory factory =
    new XmlBeanFactory(new FileSystemResource("c:/beans.xml"));
```

This simple line of code tells the bean factory to read the bean definitions from the XML file. But the bean factory doesn’t instantiate the beans just yet. Beans are

Table 2.1 `XmlBeanFactory`s can be created using one of several Resource implementations to allow Spring configuration details to come from a variety of sources.

Resource implementation	Purpose
<code>org.springframework.core.io.ByteArrayResource</code>	Defines a resource whose content is given by an array of bytes
<code>org.springframework.core.io.ClassPathResource</code>	Defines a resource that is to be retrieved from the classpath
<code>org.springframework.core.io.DescriptiveResource</code>	Defines a resource that holds a resource description but no actual readable resource
<code>org.springframework.core.io.FileSystemResource</code>	Defines a resource that is to be retrieved from the file system
<code>org.springframework.core.io.InputStreamResource</code>	Defines a resource that is to be retrieved from an input stream
<code>org.springframework.web.portlet.context.PortletContextResource</code>	Defines a resource that is available in a portlet context
<code>org.springframework.web.context.support.ServletContextResource</code>	Defines a resource that is available in a servlet context
<code>org.springframework.core.io.UrlResource</code>	Defines a resource that is to be retrieved from a given URL

“lazily” loaded into bean factories, meaning that while the bean factory will immediately load the bean definitions (the description of beans and their properties), the beans themselves will not be instantiated until they are needed.

To retrieve a bean from a BeanFactory, simply call the `getBean()` method, passing the ID of the bean you want to retrieve:

```
MyBean myBean = (MyBean) factory.getBean("myBean");
```

When `getBean()` is called, the factory will instantiate the bean and set the bean’s properties using DI. Thus begins the life of a bean within the Spring container. We’ll examine the lifecycle of a bean in section 2.1.3, but first let’s look at the other Spring container: the application context.

2.1.2 Working with an application context

A bean factory is fine for simple applications, but to take advantage of the full power of the Spring Framework, you’ll probably want to load your application beans using Spring’s more advanced container: the application context.

On the surface, an `ApplicationContext` is much the same as a `BeanFactory`. Both load bean definitions, wire beans together, and dispense beans upon request. But an `ApplicationContext` offers much more:

- Application contexts provide a means for resolving text messages, including support for internationalization (I18N) of those messages.
- Application contexts provide a generic way to load file resources, such as images.
- Application contexts can publish events to beans that are registered as listeners.

Because of the additional functionality it provides, an `ApplicationContext` is preferred over a `BeanFactory` in nearly all applications. The only times you might consider using a `BeanFactory` are in circumstances where resources are scarce, such as a mobile device. We will be using an `ApplicationContext` throughout this book.

Among the many implementations of `ApplicationContext` are three that are commonly used:

- `ClassPathXmlApplicationContext`—Loads a context definition from an XML file located in the classpath, treating context definition files as class-path resources.
- `FileSystemXmlApplicationContext`—Loads a context definition from an XML file in the file system.
- `XmlWebApplicationContext`—Loads context definitions from an XML file contained within a web application.

We'll talk more about `XmlWebApplicationContext` in chapter 13 when we discuss web-based Spring applications. For now, let's simply load the application context from the file system using `FileSystemXmlApplicationContext` or from the classpath using `ClassPathXmlApplicationContext`.

Loading an application context from the file system or from the classpath is similar to how you load beans into a bean factory. For example, here's how you'd load a `FileSystemXmlApplicationContext`:

```
ApplicationContext context =
    new FileSystemXmlApplicationContext("c:/foo.xml");
```

Similarly, you can load an application context from within the application's classpath using `ClassPathXmlApplicationContext`:

```
ApplicationContext context =  
    new ClassPathXmlApplicationContext("foo.xml");
```

The difference between these uses of `FileSystemXmlApplicationContext` and `ClassPathXmlApplicationContext` is that `FileSystemXmlApplicationContext` will look for `foo.xml` in a specific location within the file system, whereas `ClassPathXmlApplicationContext` will look for `foo.xml` anywhere in the classpath (including JAR files).

In either case, you can retrieve a bean from an `ApplicationContext` just as you would from a `BeanFactory`: by using the `getBean()` method. This is no surprise because the `ApplicationContext` interface extends the `BeanFactory` interface.

Aside from the additional functionality offered by application contexts, another big difference between an application context and a bean factory is how singleton beans are loaded. A bean factory lazily loads all beans, deferring bean creation until the `getBean()` method is called. An application context is a bit smarter and preloads all singleton beans upon context startup. By preloading singleton beans, you ensure that they will be ready to use when needed—your application won't have to wait for them to be created.

Now that you know the basics of how to create a Spring container, let's take a closer look at the lifecycle of a bean in the bean container.

2.1.3 A bean's life

In a traditional Java application, the lifecycle of a bean is quite simple. Java's `new` keyword is used to instantiate the bean (or perhaps it is deserialized) and it's ready to use. Once the bean is no longer in use, it is eligible for garbage collection and eventually goes to the big bit bucket in the sky. In contrast, the lifecycle of a bean within a Spring container is a bit more elaborate. It is important to understand the lifecycle of a Spring bean, because you may want to take advantage of some of the opportunities that Spring offers to customize how a bean is created.

Figure 2.2 shows the startup lifecycle of a typical bean as it is loaded into a `BeanFactory` container.

As you can see, a bean factory performs several setup steps before a bean is ready to use. Table 2.2 explains each of these steps in more detail.

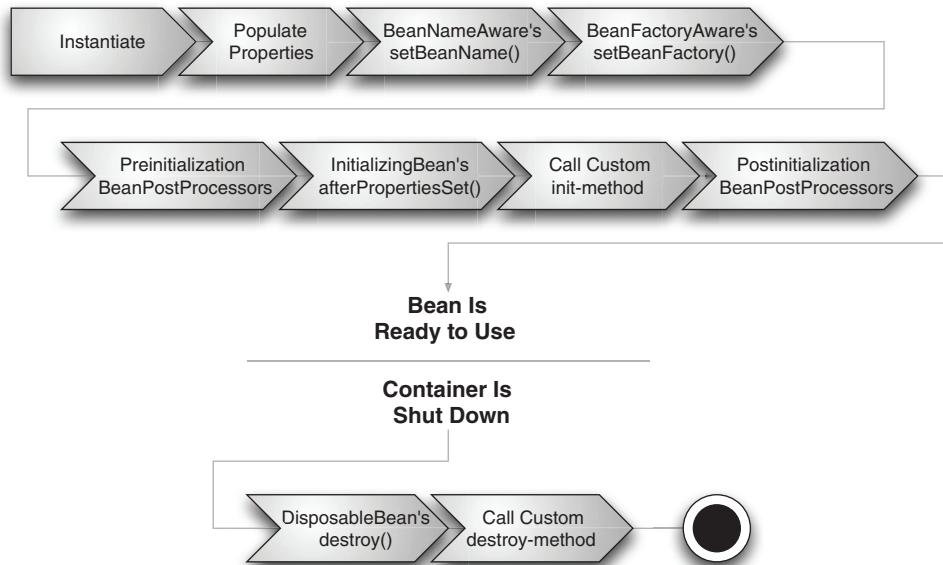


Figure 2.2 A bean goes through several steps between creation and destruction in the Spring container. Each step is an opportunity to customize how the bean is managed in Spring.

Table 2.2 The steps taken in the life of a bean

Step	Description
1. Instantiate.	Spring instantiates the bean.
2. Populate properties.	Spring injects the bean's properties.
3. Set bean name.	If the bean implements BeanNameAware , Spring passes the bean's ID to setBeanName() .
4. Set bean factory.	If the bean implements BeanFactoryAware , Spring passes the bean factory to setBeanFactory() .
5. Postprocess (before initialization).	If there are any BeanPostProcessors , Spring calls their postProcessBeforeInitialization() method.
6. Initialize beans.	If the bean implements InitializingBean , its afterPropertiesSet() method will be called. If the bean has a custom init method declared, the specified initialization method will be called.

Table 2.2 The steps taken in the life of a bean (continued)

Step	Description
7. Postprocess (after initialization).	If there are any BeanPostProcessors, Spring calls their <code>postProcessAfterInitialization()</code> method.
8. Bean is ready to use.	At this point the bean is ready to be used by the application and will remain in the bean factory until it is no longer needed.
9. Destroy bean.	If the bean implements <code>DisposableBean</code> , its <code>destroy()</code> method will be called. If the bean has a custom <code>destroy</code> -method declared, the specified method will be called.

The lifecycle of a bean within a Spring application context differs only slightly from that of a bean within a bean factory, as shown in figure 2.3.

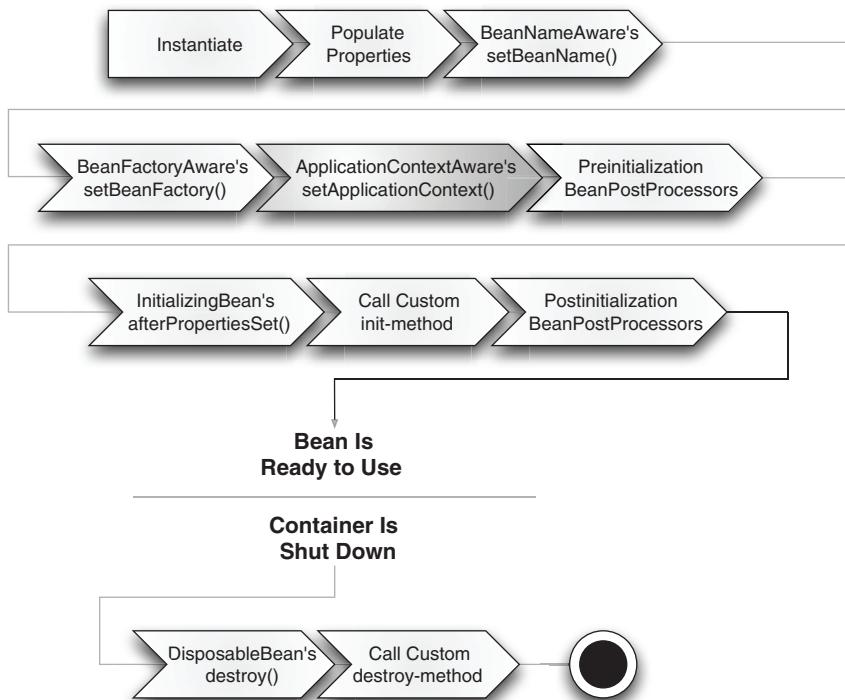


Figure 2.3 The lifecycle of a bean in a Spring application context extends the lifecycle of a factory-contained bean by adding a step to make the bean application context aware.

The only difference here is that if the bean implements the `ApplicationContextAware` interface, the `setApplicationContext()` method is called.

Now you know how to create and load a Spring container. However, an empty container isn't much good by itself; it doesn't really contain anything unless you put something in it. To achieve the benefits of Spring DI, we must wire our application objects into the Spring container. Let's have a look at how to configure beans in Spring using XML.

2.2 **Creating beans**

At this point, we'd like to welcome you to the first (and very likely the last) annual JavaBean talent competition. We've searched the nation (actually, just our IDE's workspace) for the best JavaBeans to perform and in the next few chapters, we'll set up the competition and our judges will weigh in. Spring programmers, this is your *Spring Idol*.

In our competition, we're going to need some performers, which are defined by the `Performer` interface:

```
public interface Performer {  
    void perform() throws PerformanceException;  
}
```

In the *Spring Idol* talent competition, you'll meet several contestants, all of which implement the `Performer` interface. To get started, let's meet our first contestant, who will help us illustrate how Spring supports constructor injection.

2.2.1 **Declaring a simple bean**

Unlike some similarly named talent competitions that you may have heard of, *Spring Idol* doesn't cater to only singers. In fact, many of the performers can't carry a tune at all. For example, one of the performers is a `Juggler`, as defined in listing 2.1.

Listing 2.1 A juggling bean

```
package com.springinaction.springidol;  
  
public class Juggler implements Performer {  
    private int beanBags = 3;  
  
    public Juggler() {}  
  
    public Juggler(int beanBags) {  
        this.beanBags = beanBags;  
    }
```

```
public void perform() throws PerformanceException {  
    System.out.println("JUGGLING " + beanBags + " BEANBAGS");  
}  
}
```

With the `Juggler` class defined, please welcome our first performer, Duke, to the stage. Here's how Duke is declared in the Spring configuration file (`spring-idol.xml`):

```
<bean id="duke" class="com.springinaction.springidol.Juggler" />
```

The `<bean>` element is the most basic configuration unit in Spring. It tells Spring to create an object for us. Here we've declared Duke as a Spring-managed bean using what is nearly the simplest `<bean>` declaration possible. The `id` attribute gives the bean a name by which it will be referred to in the Spring container. This bean will be known as `duke`. Meanwhile, the `class` attribute tells Spring what type the bean will be. As you can see, Duke is a `Juggler`.

When the Spring container loads its beans, it will instantiate the `duke` bean using the default constructor. In essence, `duke` will be created using the following Java code:¹

```
new com.springinaction.springidol.Juggler();
```

To give Duke a try, you can load the Spring application context using the following code:

```
ApplicationContext ctx = new ClassPathXmlApplicationContext(  
    "com/springinaction/springidol/spring-idol.xml");  
  
Performer performer = (Performer) ctx.getBean("duke");  
performer.perform();
```

Although this isn't the real competition, the previous code gives Duke a chance to practice. When run, this code prints the following:

```
JUGGLING 3 BEANBAGS
```

By default, Duke juggles only three beanbags at once. But juggling three beanbags isn't all that impressive—anybody can do that. If Duke is to have any hope of winning the talent competition, he's going to need to juggle many more beanbags at once. Let's see how to configure Duke to be a champion juggler.

¹ Emphasis on “In essence.” Actually, Spring creates its beans using reflection.

2.2.2 Injecting through constructors

To really impress the judges, Duke has decided to break the world record by juggling as many as 15 beanbags at once.²

As you can see in listing 2.1, the `Juggler` class can be constructed in two different ways:

- Using the default constructor
- Using a constructor that takes an `int` argument that indicates the number of beanbags that the `Juggler` will attempt to keep in the air

Although the declaration of the `duke` bean in section 2.2.1 is valid, it uses the `Juggler`'s default constructor, which limits Duke to juggling only three balls at once. To make Duke a world-record juggler, we'll need to use the other constructor. The following XML redeclares Duke as a 15-ball juggler:

```
<bean id="duke" class="com.springinaction.springidol.Juggler">
    <constructor-arg value="15" />
</bean>
```

The `<constructor-arg>` element is used to give Spring additional information to use when constructing a bean. If no `<constructor-arg>`s are given, as in section 2.2.1, the default constructor is used. But here we've given a `<constructor-arg>` with a `value` attribute set to 15, so the `Juggler`'s other constructor will be used instead.

Now when Duke performs, the following is printed:

JUGGLING 15 BEANBAGS

Juggling 15 beanbags at once is mighty impressive. But there's something we didn't tell you about Duke. Not only is Duke a good juggler, but he is also skilled at reciting poetry. Juggling while reciting poetry takes a lot of mental discipline. If Duke can juggle while reciting a Shakespearean sonnet then he should be able to establish himself as the clear winner of the competition. (We told you this wouldn't be like those other talent shows!)

² Juggling trivia: Who holds the actual world record for juggling beanbags depends on how many beanbags are juggled and for how long. Bruce Sarafian holds several records, including juggling 12 beanbags for 12 catches. Another record-holding juggler is Anthony Gatto who juggled 7 balls for 10 minutes and 12 seconds in 2005. Another juggler, Peter Bone, claims to have juggled as many as 13 beanbags for 13 catches—but there is no video evidence of the feat.

Injecting object references with constructors

Because Duke is more than just an average juggler—he's a poetic juggler—we need to define a new type of juggler for him to be. PoeticJuggler (listing 2.2) is a class more descriptive of Duke's talent.

Listing 2.2 A juggler who waxes poetic

```
package com.springinaction.springidol;

public class PoeticJuggler extends Juggler {
    private Poem poem;

    public PoeticJuggler(Poem poem) {
        super ();
        this.poem = poem;
    }

    public PoeticJuggler(int beanBags, Poem poem) {
        super(beanBags);
        this.poem = poem;
    }

    public void perform() throws PerformanceException {
        super.perform();
        System.out.println("WHILE RECITING...");
        poem.recite();
    }
}
```

This new type of juggler does everything a regular juggler does, but it also has a reference to a poem to be recited. Speaking of the poem, here's an interface that generically defines what a poem looks like:

```
public interface Poem {
    void recite();
}
```

One of Duke's favorite Shakespearean sonnets is "When in disgrace with fortune and men's eyes." Sonnet29 (listing 2.3) is an implementation of the Poem interface that defines this sonnet.

Listing 2.3 A class that represents a great work of the Bard

```
package com.springinaction.springidol;

public class Sonnet29 implements Poem {
    private static String[] LINES = {
        "When, in disgrace with fortune and men's eyes,",
        "I all alone beweep my outcast state",
        "And trouble deaf heaven with my bootless cries",
    }
```

```

    "And look upon myself and curse my fate,",
    "Wishing me like to one more rich in hope,",
    "Featured like him, like him with friends possess'd,",
    "Desiring this man's art and that man's scope,",
    "With what I most enjoy contented least;",
    "Yet in these thoughts myself almost despising,",
    "Haply I think on thee, and then my state,",
    "Like to the lark at break of day arising",
    "From sullen earth, sings hymns at heaven's gate;",
    "For thy sweet love remember'd such wealth brings",
    "That then I scorn to change my state with kings."
};

public Sonnet29() {}

public void recite() {
    for (int i = 0; i < LINES.length; i++) {
        System.out.println(LINES[i]);
    }
}
}

```

Sonnet29 can be declared as a Spring <bean> with the following XML:

```
<bean id="sonnet29"
      class="com.springinaction.springidol.Sonnet29" />
```

With the poem chosen, all we need to do is give it to Duke. Now that Duke is a PoeticJuggler, his <bean> declaration will need to change slightly:

```
<bean id="duke" class="com.springinaction.springidol.PoeticJuggler">
    <constructor-arg value="15" />
    <constructor-arg ref="sonnet29" />
</bean>
```

As you can see from listing 2.2, there is no default constructor. The only way to construct a PoeticJuggler is to use a constructor that takes arguments. In this example, we're using the constructor that takes an int and a Poem as arguments. The duke bean declaration configures the number of beanbags as 15 through the int argument using <constructor-arg>'s value attribute.

But we can't use value to set the second constructor argument because a Poem is not a simple type. Instead, the ref attribute is used to indicate that the value passed to the constructor should be a reference to the bean whose ID is sonnet29. Although the Spring container does much more than just construct beans, you may imagine that when Spring encounters the sonnet29 and duke <bean>s, it performs some logic that is essentially the same as the following lines of Java:

```
Poem sonnet29 = new Sonnet29();
Performer duke = new PoeticJuggler(15, sonnet29);
```

Constructor or setter injection: how do you choose?

There are certain things that most people can agree upon: the fact that the sky is blue, that Michael Jordan is the greatest player to touch a basketball, and that *Star Trek V* should have never happened. And then there are those things that should never be discussed in polite company, such as politics, religion, and the eternal “tastes great/less filling” debates.

Likewise, the choice between constructor injection and setter injection stirs up as much discourse as the arguments surrounding creamy versus crunchy peanut butter. Both have their merits and their weaknesses. Which should you choose?

Those on the constructor-injection side of the debate will tell you that:

- Constructor injection enforces a strong dependency contract. In short, a bean cannot be instantiated without being given all of its dependencies. It is perfectly valid and ready to use upon instantiation. Of course, this assumes that the bean’s constructor has all of the bean’s dependencies in its parameter list.
- Because all of the bean’s dependencies are set through its constructor, there’s no need for superfluous setter methods. This helps keep the lines of code at a minimum.
- By only allowing properties to be set through the constructor, you are, effectively, making those properties immutable, preventing accidental change in the course of application flow.

However, the setter injection-minded will be quick to respond with:

- If a bean has several dependencies, the constructor’s parameter list can be quite lengthy.
- If there are several ways to construct a valid object, it can be hard to come up with unique constructors, since constructor signatures vary only by the number and type of parameters.
- If a constructor takes two or more parameters of the same type, it may be difficult to determine what each parameter’s purpose is.
- Constructor injection does not lend itself readily to inheritance. A bean’s constructor will have to pass parameters to `super()` in order to set private properties in the parent object.

Fortunately, Spring doesn’t take sides in this debate and will let you choose the injection model that suits you best. In fact, you can even mix-and-match constructor and setter injection in the same application... or even in the same bean. Personally, I tend to favor setter injection in most cases, but will occasionally use constructor injection should the mood strike me.

Now when Duke performs, he not only juggles but will also recite Shakespeare, resulting in the following being printed to the standard output stream:

```
JUGGLING 15 BEANBAGS
WHILE RECITING...
When, in disgrace with fortune and men's eyes,
I all alone beweep my outcast state
And trouble deaf heaven with my bootless cries
And look upon myself and curse my fate,
Wishing me like to one more rich in hope,
Featured like him, like him with friends possess'd,
Desiring this man's art and that man's scope,
With what I most enjoy contented least;
Yet in these thoughts myself almost despising,
Haply I think on thee, and then my state,
Like to the lark at break of day arising
From sullen earth, sings hymns at heaven's gate;
For thy sweet love remember'd such wealth brings
That then I scorn to change my state with kings.
```

Constructor injection is a surefire way to guarantee that a bean is fully configured before it is used. But it doesn't lend itself to complex configuration. Fortunately, Spring doesn't restrict you to always configuring your beans with constructors and offers setter injection as another choice. Next up, let's have a look at how to configure beans using their properties' setter methods.

2.3 Injecting into bean properties

Typically, a JavaBean's properties are private and will have a pair of accessor methods in the form of `setXXX()` and `getXXX()`. Spring can take advantage of a property's setter method to configure the property's value through setter injection.

To demonstrate Spring's other form of DI, let's welcome our next performer to the stage. Kenny is a talented instrumentalist, as defined by the `Instrumentalist` class in listing 2.4.

Listing 2.4 Defining a performer who is talented with musical instruments

```
package com.springinaction.springidol;

public class Instrumentalist implements Performer {
    public Instrumentalist() {}
    public void perform() throws PerformanceException {
        System.out.print("Playing " + song + " : ");
        instrument.play();
    }
    private String song;
```

```
public void setSong(String song) {  
    this.song = song;  
}  
  
private Instrument instrument;  
public void setInstrument(Instrument instrument) {  
    this.instrument = instrument;  
}  
}
```

↳ Injects song and instrument

From listing 2.4, we can see that an `Instrumentalist` has two properties: `song` and `instrument`. The `song` property holds the name of the song that the instrumentalist will play and is used in the `perform()` method. The `instrument` property holds a reference to an `Instrument` that the instrumentalist will play. An `Instrument` is defined by the following interface:

```
public interface Instrument {  
    void play();  
}
```

Because the `Instrumentalist` class has a default constructor, Kenny could be declared as a `<bean>` in Spring with the following XML:

```
<bean id="kenny"  
      class="com.springinaction.springidol.Instrumentalist" />
```

Although Spring will have no problem instantiating kenny as an `Instrumentalist`, Kenny will have a hard time performing without a song or an instrument. Let's look at how to give Kenny his song and `instrument` by using setter injection.

2.3.1 Injecting simple values

Bean properties can be configured in Spring using the `<property>` element. `<property>` is similar to `<constructor-arg>` in many ways, except that instead of injecting values through a constructor argument, `<property>` injects by calling a property's setter method.

To illustrate, let's give Kenny a song to perform using setter injection. The following XML shows an updated declaration of the `kenny` bean:

```
<bean id="kenny"  
      class="com.springinaction.springidol.Instrumentalist">  
    <property name="song" value="Jingle Bells" />  
  </bean>
```

Once the `Instrumentalist` has been instantiated, Spring will use property setter methods to inject values into the properties specified by `<property>` elements.

The `<property>` element in this XML instructs Spring to call `setSong()` to set a value of "Jingle Bells" for the `song` property. This is essentially the same as the following Java code:

```
Instrumentalist kenny = new Instrumentalist();
kenny.setSong("Jingle Bells");
```

The key difference between setting the `song` property through Spring and setting it through Java is that by setting it in Spring, the `Instrumentalist` bean is decoupled from its configuration. That is, `Instrumentalist` isn't hard-coded with a specific song and is more flexible to perform any song given to it.

In the case of the `song` property, the `value` attribute of the `<property>` element is used to inject a `String` value into a property. But `<property>` isn't limited to injecting `String` values. The `value` attribute can also specify numeric (`int`, `float`, `java.lang.Double`, etc.) values as well as `boolean` values.

For example, let's pretend that the `Instrumentalist` class has an `age` property of type `int` to indicate the age of the instrumentalist. We could set Kenny's age using the following XML:

```
<bean id="kenny"
      class="com.springinaction.springidol.Instrumentalist">
    <property name="song" value="Jingle Bells" />
    <property name="age" value="37" />
</bean>
```

Notice that the `value` attribute is used exactly the same when setting a numeric value as it is when setting a `String` value. Spring will determine the correct type for the value based on the property's type. Since the `age` property is an `int`, Spring knows to convert 37 to an `int` value before calling `setAge()`.

Using `<property>` to configure simple properties of a bean is great, but there's more to DI than just wiring hard-coded values. The real value of DI is found in wiring an application's collaborating objects together so that they don't have to wire themselves together. To that aim, let's see how to give Kenny an instrument that he can play.

2.3.2 Referencing other beans

Kenny's a very talented instrumentalist and can play virtually any instrument given to him. As long as it implements the `Instrument` interface, he can make music with it. Naturally, however, Kenny does have a favorite instrument. His instrument of choice is the saxophone, which is defined by the `Saxophone` class in listing 2.5.

Listing 2.5 A saxophone implementation of Instrument

```
package com.springinaction.springidol;

public class Saxophone implements Instrument {
    public Saxophone() {}

    public void play() {
        System.out.println("TOOT TOOT TOOT");
    }
}
```

Before we can give Kenny a saxophone to play, we must declare it as a <bean> in Spring. The following XML should do:

```
<bean id="saxophone"
      class="com.springinaction.springidol.Saxophone" />
```

Notice that the Saxophone class has no properties that need to be set. Consequently, there is no need for <property> declarations in the saxophone bean.

With the saxophone declared, we're ready to give it to Kenny to play. The following modification to the kenny bean uses setter injection to set the instrument property:

```
<bean id="kenny"
      class="com.springinaction.springidol.Instrumentalist">
    <property name="song" value="Jingle Bells" />
    <property name="instrument" ref="saxophone" />
</bean>
```

Now the kenny bean has been injected with all of its properties and Kenny is ready to perform. As with Duke, we can prompt Kenny to perform by executing the following Java code (perhaps in a `main()` method):

```
ApplicationContext ctx = new ClassPathXmlApplicationContext(
    "com/springinaction/springidol/spring-idol.xml");

Performer performer = (Performer) ctx.getBean("kenny");
performer.perform();
```

This isn't the exact code that will run the *Spring Idol* competition, but it does give Kenny a chance to practice. When it is run, the following will be printed:

```
Playing Jingle Bells : TOOT TOOT TOOT
```

At the same time, it illustrates an important concept. If you compare this code with the code that instructed Duke to perform, you'll find that it isn't much different. In fact, the only difference is the name of the bean retrieved from Spring.

The code is the same, even though one causes a juggler to perform and the other causes an instrumentalist to perform.

This isn't a feature of Spring as much as it's a benefit of coding to interfaces. By referring to a performer through the `Performer` interface, we're able to blindly cause any type of performer to perform, whether it's a poetic juggler or a saxophonist. Spring encourages the use of interfaces for this reason. And, as you're about to see, interfaces work hand in hand with DI to provide loose coupling.

As mentioned, Kenny can play virtually any instrument that is given to him as long as it implements the `Instrument` interface. Although he favors the saxophone, we could also ask Kenny to play piano. For example, consider the `Piano` class defined in listing 2.6.

Listing 2.6 A piano implementation of Instrument

```
package com.springinaction.springidol;

public class Piano implements Instrument {
    public Piano() {}

    public void play() {
        System.out.println("PLINK PLINK PLINK");
    }
}
```

The `Piano` class can be declared as a `<bean>` in Spring using the following XML:

```
<bean id="piano"
      class="com.springinaction.springidol.Piano" />
```

Now that a piano is available, changing Kenny's instrument is as simple as changing the `kenny` bean declaration as follows:

```
<bean id="kenny"
      class="com.springinaction.springidol.Instrumentalist">
    <property name="song" value="Jingle Bells" />
    <property name="instrument" ref="piano" />
</bean>
```

With this change, Kenny will play a piano instead of a saxophone. However, because the `Instrumentalist` class only knows about its `instrument` property through the `Instrument` interface, nothing about the `Instrumentalist` class needed to change to support a new implementation of `Instrument`. Although an `Instrumentalist` can play either a `Saxophone` or `Piano`, it is decoupled from both. If Kenny decides to take up the hammered dulcimer, the only change

required will be to create a HammeredDulcimer class and to change the instrument property on the kenny bean declaration.

Injecting inner beans

We've seen that Kenny is able to play saxophone, piano, or any instrument that implements the Instrument interface. But what's also true is that the saxophone and piano beans could also be shared with any other bean by injecting them into an Instrument property. So, not only can Kenny play any Instrument, any Instrumentalist can play the saxophone bean. In fact, it's quite common for beans to be shared among other beans in an application.

The problem, however, is that Kenny's a bit concerned with the hygienic implications of sharing his saxophone with others. He'd rather keep his saxophone to himself. To help Kenny avoid germs, we'll use a handy Spring technique known as *inner beans*.

As a Java developer, you're probably already familiar with the concept of inner classes—that is, classes that are defined within the scope of other classes. Similarly, inner beans are beans that are defined within the scope of another bean. To illustrate, consider this new configuration of the kenny bean where his saxophone is declared as an inner bean:

```
<bean id="kenny"
      class="com.springinaction.springidol.Instrumentalist">
    <property name="song" value="Jingle Bells" />
    <property name="instrument">
      <bean class="org.springinaction.springidol.Saxophone" />
    </property>
  </bean>
```

As you can see, an inner bean is defined by declaring a `<bean>` element directly as a child of the `<property>` element to which it will be injected. In this case, a Saxophone will be created and wired into Kenny's instrument property.

Inner beans aren't limited to setter injection. You may also wire inner beans into constructor arguments, as shown in this new declaration of the duke bean:

```
<bean id="duke" class="com.springinaction.springidol.PoeticJuggler">
  <constructor-arg value="15" />
  <constructor-arg>
    <bean class="com.springinaction.springidol.Sonnet29" />
  </constructor-arg>
</bean>
```

Here, a Sonnet29 instance will be created as an inner bean and sent as an argument to the PoeticJuggler's constructor.

Notice that the inner beans do not have an `id` attribute set. While it's perfectly legal to declare an ID for an inner bean, it's not necessary because you'll never refer to the inner bean by name. This highlights the main drawback of using inner beans: they can't be reused. Inner beans are only useful for injection once and can't be referred to by other beans.

You may also find that using inner-bean definitions has a negative impact on the readability of the XML in the Spring context files.

Kenny's talent extends to virtually any instrument. Nevertheless, he does have one limitation: he can play only one instrument at a time. Next to take the stage in the *Spring Idol* competition is Hank, a performer who can simultaneously play multiple instruments.

2.3.3 **Wiring collections**

Up to now, you've seen how to use Spring to configure both simple property values (using the `value` attribute) and properties with references to other beans (using the `ref` attribute). But `value` and `ref` are only useful when your bean's properties are singular. How can Spring help you when your bean has properties that are plural—what if a property is a collection of values?

Spring offers four types of collection configuration elements that come in handy when configuring collections of values. Table 2.3 lists these elements and what they're good for.

The `<list>` and `<set>` elements are useful when configuring properties that are either arrays or some implementation of `java.util.Collection`. As you'll soon see, the actual implementation of the collection used to define the property has little correlation to the choice of `<list>` or `<set>`. Both elements can be used almost interchangeably with properties of any type of `java.util.Collection`.

Table 2.3 Just as Java has several kinds of collections, Spring allows for injecting several kinds of collections.

Collection element	Useful for...
<code><list></code>	Wiring a list of values, allowing duplicates.
<code><set></code>	Wiring a set of values, ensuring no duplicates
<code><map></code>	Wiring a collection of name-value pairs where name and value can be of any type
<code><props></code>	Wiring a collection of name-value pairs where the name and value are both Strings

As for `<map>` and `<props>`, these two elements correspond to collections that are `java.util.Map` and `java.util.Properties`, respectively. These types of collections are useful when you need a collection that is made up of a collection of key-value pairs. The key difference between the two is that when using `<props>`, both the keys and values are `Strings`, while `<map>` allows keys and values of any type.

To illustrate collection wiring in Spring, please welcome Hank to the *Spring Idol* stage. Hank's special talent is that he is a one-man band. Like Kenny, Hank's talent is playing several instruments, but Hank can play several instruments at the same time. Hank is defined by the `OneManBand` class in listing 2.7.

Listing 2.7 A performer that is a one-man-band

```
package com.springinaction.springidol;
import java.util.Collection;

public class OneManBand implements Performer {
    public OneManBand() {}

    public void perform() throws PerformanceException {
        for(Instrument instrument : instruments) {
            instrument.play();
        }
    }

    private Collection<Instrument> instruments;
    public void setInstruments(Collection<Instrument> instruments) {
        this.instruments = instruments;
    }
}
```

Injects Instrument collection

As you can see, a `OneManBand` iterates over a collection of instruments when it performs. What's most important here is that the collection of instruments is injected through the `setInstruments()` method. Let's see how Spring can provide Hank with his collection of instruments.

Lists and arrays

To give Hank a collection of instruments to perform with, let's use the `<list>` configuration element:

```
<bean id="hank" class="com.springinaction.springidol.OneManBand">
    <property name="instruments">
        <list>
            <ref bean="guitar" />
            <ref bean="cymbal" />
            <ref bean="harmonica" />
        </list>
    </property>

```

```
</list>
</property>
</bean>
```

The `<list>` element contains one or more values. Here `<ref>` elements are used to define the values as references to other beans in the Spring context, configuring Hank to play a guitar, cymbal, and harmonica. However, it's also possible to use other value-setting Spring elements as the members of a `<list>`, including `<value>`, `<bean>`, and `<null/>`. In fact, a `<list>` may contain another `<list>` as a member for multidimensional lists.

In listing 2.7, OneManBand's `instruments` property is a `java.util.Collection` using Java 5 generics to constrain the collection to `Instrument` values. But `<list>` may be used with properties that are of any implementation of `java.util.Collection` or an array. In other words, the `<list>` element we just used would still work, even if the `instruments` property were to be declared as

```
java.util.List<Instrument> instruments;
```

or even if it were to be declared as

```
Instrument[] instruments;
```

Sets

As mentioned, Spring's `<list>` element is more about how the collection is handled by Spring than the actual type of the argument. Anywhere you can use `<list>`, you may also use `<set>`. But `<set>` has a useful side effect that `<list>` does not have: `<set>` ensures that each of its members is unique. To illustrate, here's a new declaration of the hank bean using `<set>` instead of `<list>` to configure the `instruments` property:

```
<bean id="hank" class="com.springinaction.springidol.OneManBand">
  <property name="instruments">
    <set>
      <ref bean="guitar" />
      <ref bean="cymbal" />
      <ref bean="harmonica" />
      <ref bean="harmonica" />
    </set>
  </property>
</bean>
```

Although Hank can play multiple instruments at once, he can only realistically play one harmonica at a time (he only has one mouth, after all). In this example, Hank has been given two references to harmonica. But because `<set>` is used to configure the `instruments` property, the extra harmonica is effectively ignored.

Just like `<list>`, the `<set>` element can be used to configure properties whose type is an array or an implementation of `java.util.Collection`. It may seem odd to configure a `java.util.List` property using `<set>`, but it's certainly possible. In doing so, you'll be guaranteed that all members of the `List` will be unique.

Maps

When a `OneManBand` performs, each instrument's sound is printed as the `perform()` method iterates over the collection of instruments. But let's suppose that we also want to see which instrument is producing each sound. To accommodate this, consider the changes to the `OneManBand` class in listing 2.8.

Listing 2.8 Changing `OneManBand`'s instrument collection to a Map

```
package com.springinaction.springidol;
import java.util.Map;

public class OneManBand implements Performer {
    public OneManBand() {}

    public void perform() throws PerformanceException {
        for (String key : instruments.keySet()) {
            System.out.print(key + " : ");
            Instrument instrument = instruments.get(key);
            instrument.play();
        }
    }

    private Map<String,Instrument> instruments;
    public void setInstruments(Map<String,Instrument> instruments) {
        this.instruments = instruments;
    }
}
```

Prints instrument's key

Injects instruments as Map

In the new version of `OneManBand`, the `instruments` property is a `java.util.Map` where each member has a `String` as its key and an `Instrument` as its value. Because a `Map`'s members are made up of key-value pairs, a simple `<list>` or `<set>` configuration element will not suffice when wiring the property.

Instead, the following declaration of the `hank` bean uses the `<map>` element to configure the `instruments` property:

```
<bean id="hank" class="com.springinaction.springidol.OneManBand">
    <property name="instruments">
        <map>
            <entry key="GUITAR" value-ref="guitar" />
            <entry key="CYMBAL" value-ref="cymbal" />
            <entry key="HARMONICA" value-ref="harmonica" />
        </map>
    </property>

```

```
</map>
</property>
</bean>
```

The `<map>` element declares a value of type `java.util.Map`. Each `<entry>` element defines a member of the `Map`. In the previous example, the `key` attribute specifies the key of the entry while the `value-ref` attribute defines the value of the entry as a reference to another bean within the Spring context.

Although our example uses the `key` attribute to specify a `String` key and `value-ref` to specify a reference value, the `<entry>` element actually has two attributes each for specifying the key and value of the entry. Table 2.4 lists those attributes.

Table 2.4 An `<entry>` in a `<map>` is made up of a key and a value, either of which can be a primitive value or a reference to another bean. These attributes help specify the keys and values of an `<entry>`.

Attribute	Purpose
<code>key</code>	Specifies the key of the map entry as a <code>String</code>
<code>key-ref</code>	Specifies the key of the map entry as a reference to a bean in the Spring context
<code>value</code>	Specifies the value of the map entry as a <code>String</code>
<code>value-ref</code>	Specifies the value of the map entry as a reference to a bean in the Spring context

`<map>` is only one way to inject key-value pairs into bean properties when either of the objects is not a `String`. Let's see how to use Spring's `<props>` element to configure `String`-to-`String` mappings.

Properties

When declaring a `Map` of values for `OneManBand`'s `instrument` property, it was necessary to specify the value of each entry using `value-ref`. That's because each entry is ultimately another bean in the Spring context.

But if you find yourself configuring a `Map` whose entries have both `String` keys and `String` values, you may want to consider using `java.util.Properties` instead of a `Map`. The `Properties` class serves roughly the same purpose as `Map`, but limits the keys and values to `Strings`.

To illustrate, imagine that instead of being wired with a collection of references to `Instrument` beans, `OneManBand` is wired with a collection of `Strings` that are the sounds made by the instruments. The new `OneManBand` class is in listing 2.9.

Listing 2.9 Wiring a OneManBand's instruments as a Properties collection

```
package com.springinaction.springidol;
import java.util.Iterator;
import java.util.Properties;

public class OneManBand implements Performer {
    public OneManBand() {}

    public void perform() throws PerformanceException {
        for (Iterator iter = instruments.keySet().iterator();
             iter.hasNext();)
            {
                String key = (String) iter.next();
                System.out.println(key + " : " +
                    instruments.getProperty(key));
            }
    }

    private Properties instruments;
    public void setInstruments(Properties instruments) {
        this.instruments = instruments;
    }
}
```

Injects
instruments
as Properties

To wire the instrument sounds into the `instruments` property, we use the `<props>` element in the following declaration of the `hank` bean:

```
<bean id="hank" class="com.springinaction.springidol.OneManBand">
    <property name="instruments">
        <props>
            <prop key="GUITAR">STRUM STRUM STRUM</prop>
            <prop key="CYMBAL">CRASH CRASH CRASH</prop>
            <prop key="HARMONICA">HUM HUM HUM</prop>
        </props>
    </property>
</bean>
```

The `<props>` element constructs a `java.util.Properties` value where each member is defined by a `<prop>` element. Each `<prop>` element has a `key` attribute that defines the key of each `Properties` member, while the value is defined by the contents of the `<prop>` element. In our example, the element whose `key` is “`GUITAR`” has a value of “`STRUM STRUM STRUM`”.

This may very well be the most difficult Spring configuration element to talk about. That’s because the term “property” is highly overloaded. It’s important to keep the following straight:

- <property> is the element used to inject a value into a property of a bean class.
- <props> is the element used to define a collection value of type `java.util.Properties`.
- <prop> is the element used to define a member value of a <props> collection.

Now you've seen several ways of wiring values into a bean's properties. Now let's do something a little different and see how to wire nothing into a bean property.

2.3.4 Wiring nothing (`null`)

In almost every situation, you will use DI to wire a value or an object reference into a bean's properties. But what if you want to ensure that a property is `null`?

You're probably rolling your eyes and thinking, "What's this guy talking about? Why would I ever want to wire `null` into a property? Aren't all properties `null` until they're set? What's the point?"

While it's often true that properties start out `null` and will remain that way until set, some beans may themselves set a property to a non-`null` default value. What if, for whatever twisted reasons you have, you want to force that property to be `null`? If that's the case, it's not sufficient to just assume that the property will be `null`—you must explicitly wire `null` into the property.

To set a property to `null`, you simply use the <`null`/> element. For example:

```
<property name="someNonNullProperty"><null/></property>
```

Another reason for explicitly wiring `null` into a property is to override an autowired property value. What's auto-wiring, you ask? Keep reading—we're going to explore autowiring in the next section.

2.4 Autowiring

So far you've seen how to wire all of your bean's properties using either the <`constructor-arg`> or the <`property`> element. In a large application, however, all of this explicit wiring can result in a lot of XML. Rather than explicitly wiring all of your bean's properties, you can have Spring automatically figure out how to wire beans together by setting the `autowire` property on each <`bean`> that you want Spring to autowire.

2.4.1 The four types of autowiring

Spring provides four flavors of autowiring:

- `byName`—Attempts to find a bean in the container whose name (or ID) is the same as the name of the property being wired. If a matching bean is not found, the property will remain unwired.
- `byType`—Attempts to find a single bean in the container whose type matches the type of the property being wired. If no matching bean is found, the property will not be wired. If more than one bean matches, an `org.springframework.beans.factory.UnsatisfiedDependencyException` will be thrown.
- `constructor`—Tries to match up one or more beans in the container with the parameters of one of the constructors of the bean being wired. In the event of ambiguous beans or ambiguous constructors, an `org.springframework.beans.factory.UnsatisfiedDependencyException` will be thrown.
- `autodetect`—Attempts to autowire by constructor first and then using `byType`. Ambiguity is handled the same way as with `constructor` and `byType` wiring.

Each of these options has its pros and cons. Let's start by looking at how you can have Spring autowire a bean's properties by their names.

Autowiring by name

In Spring, everything is given a name. Bean properties are given names. And the beans that are wired into those properties are given names. If the name of a property happens to match the name of the bean that is to be wired into that property, that could serve as a hint to Spring that the bean should automatically be wired into the property.

For example, let's revisit the kenny bean from section 2.3.2:

```
<bean id="kenny"
      class="com.springinaction.springidol.Instrumentalist">
    <property name="song" value="Jingle Bells" />
    <property name="instrument" ref="saxophone" />
</bean>
```

Here we've explicitly configured Kenny's instrument property using `<property>`. Just for the moment, let's pretend that we declared the Saxophone as a `<bean>` with an `id` of `instrument`:

```
<bean id="instrument"
      class="com.springinaction.springidol.Saxophone" />
```

If this were the case, the `id` of the `Saxophone` bean would be the same as the name of the `instrument` property. Spring can take advantage of this to automatically configure Kenny's instrument by setting the `autowire` property as follows:

```
<bean id="kenny"
      class="com.springinaction.springidol.Instrumentalist"
      autowire="byName">
    <property name="song" value="Jingle Bells" />
</bean>
```

`byName` autowiring sets a convention where a property will automatically be wired with a bean of the same name. In setting the `autowire` property to `byName`, you are telling Spring to consider all properties of `kenny` and look for beans declared with the same names as the properties. In this case, the `instrument` property is eligible for autowiring through setter injection. As illustrated in figure 2.4, if there is a bean in the context whose `id` is `instrument`, it will be autowired into the `instrument` property.

The limitation of using `byName` autowiring is that it assumes that you'll have a bean whose name is the same as the property of another bean that you'll be injecting into. In our example, it would require creating a bean whose name is `instrument`. If multiple `Instrumentalist` beans are configured to be autowired by name then all of them will be playing the same instrument. This may not be a problem in all circumstances, but it is a limitation to keep in mind.

Autowiring by type

Autowiring using `byType` works in a similar way to `byName`, except that instead of considering a property's name, the property's type is examined. When attempting to autowire a property by type, Spring will look for beans whose type is assignable to the property's type.

For example, suppose that the `kenny` bean's `autowire` property is set to `byType` instead of `byName`. The container will search itself for a bean whose type is

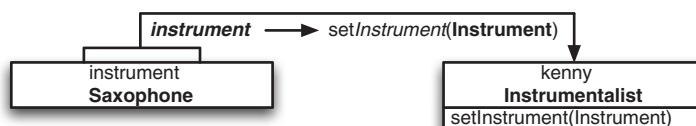


Figure 2.4 When autowiring by name, a bean's name is matched against properties that have the same name.

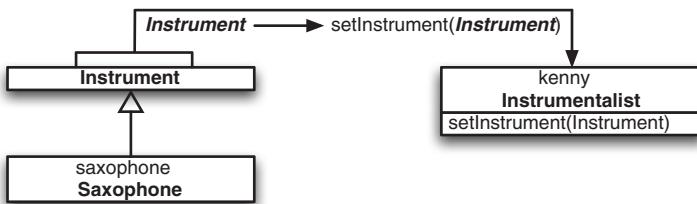


Figure 2.5 Autowire by type matches beans to properties of the same type.

Instrument and wire that bean into the instrument property. As shown in figure 2.5, the saxophone bean will be automatically wired into Kenny's instrument property because both the instrument property and the saxophone bean are of type Instrument.

But there is a limitation to autowiring by type. What happens if Spring finds more than one bean whose type is assignable to the autowired property? In such a case, Spring isn't going to guess which bean to autowire and will instead throw an exception. Consequently, you are allowed to have only one bean configured that matches the autowired property. In the *Spring Idol* competition, there are likely to be several beans whose types are subclasses of Instrument. Therefore, autowiring by type will not be helpful in our example.

As with the limitation to byName, this may or may not be a problem for your application, but you should be aware of Spring's behavior when ambiguities occur in autowiring.

Using constructor autowiring

If your bean is configured using constructor injection, you may choose to put away the <constructor-arg> elements and let Spring automatically choose constructor arguments from beans in the Spring context.

For example, consider the following redeclaration of the duke bean:

```
<bean id="duke"
      class="com.springinaction.springidol.PoeticJuggler"
      autowire="constructor" />
```

In this new declaration of duke, the <constructor-arg> elements are gone and the autowire attribute has been set to constructor. This tells Spring to look at PoeticJuggler's constructors and try to find beans in the Spring configuration to satisfy the arguments of one of the constructors. We've already declared the sonnet29 bean, which is a Poem and matches the constructor argument of one of



Figure 2.6 When autowired by constructor, the duke PoeticJuggler is instantiated with the constructor that takes a Poem argument.

PoeticJuggler's constructors. Therefore, Spring will use that constructor, passing in the sonnet29 bean, when constructing the duke bean, as expressed in figure 2.6.

Autowiring by constructor shares the same limitations as `byType`. That is, Spring will not attempt to guess which bean to autowire when it finds multiple beans that match a constructor's arguments. Furthermore, if a constructor has multiple constructors, any of which can be satisfied by autowiring, Spring will not attempt to guess which constructor to use.

Autodetect autowiring

If you want to autowire your beans, but you can't decide which type of autowiring to use, have no fear. By setting the `autowire` attribute to `autodetect`, you can let the container make the decision for you. For example:

```
<bean id="duke"
      class="com.springinaction.springidol.PoeticJuggler"
      autowire="autodetect" />
```

When a bean has been configured to autowire by `autodetect`, Spring will attempt to autowire by constructor first. If a suitable constructor-bean match can't be found then Spring will attempt to autowire by type.

Autowiring by default

By default, beans will not be autowired unless you set the `autowire` attribute. However, you can set a default autowiring for all beans within the Spring context by setting `default-autowire` on the root `<beans>` element:

```
<beans default-autowire="byName">
  ...
</beans>
```

Set this way, all beans will be autowired using `byName` unless specified otherwise.

2.4.2 Mixing auto with explicit wiring

Just because you choose to autowire a bean, that doesn't mean you can't explicitly wire some properties. You can still use the `<property>` element on any property as if you hadn't set autowire.

For example, to explicitly wire Kenny's instrument property even though he is set to autowire by type, use this code:

```
<bean id="kenny"
      class="com.springinaction.springidol.Instrumentalist"
      autowire="byType">
    <property name="song" value="Jingle Bells" />
    <property name="instrument" ref="saxophone" />
</bean>
```

As illustrated here, mixing automatic and explicit wiring is also a great way to deal with ambiguous autowiring that may occur when autowiring using `byType`. There may be several beans in the Spring context that implement `Instrument`. To keep Spring from throwing an exception due to the ambiguity of several Instruments to choose from, we can explicitly wire the `instrument` property, effectively overriding autowiring.

We mentioned earlier that you could use `<null/>` to force an autowired property to be null. This is just a special case of mixing autowiring with explicit wiring. For example, if you wanted to force Kenny's instrument to be null, you'd use the following configuration:

```
<bean id="kenny"
      class="com.springinaction.springidol.Instrumentalist"
      autowire="byType">
    <property name="song" value="Jingle Bells" />
    <property name="instrument"><null/></property>
</bean>
```

This is just for illustration's sake, of course. Wiring null into `instrument` will result in a `NullPointerException` being thrown when the `perform()` method is invoked.

One final note on mixed wiring: When using constructor autowiring, you must let Spring wire all of the constructor arguments—you cannot mix `<constructor-arg>` elements with constructor autowiring.

2.4.3 To autowire or not to autowire

Although autowiring seems to be a powerful way to cut down on the amount of manual configuration required when writing the bean wiring file, it may lead to some problems. As you've already seen, autowiring by type or constructor imposes

a restriction where you may only have one bean of a given type. And autowiring by name forces you to name your beans to match the properties that they'll be autowired into. However, there are other shortcomings associated with autowiring.

For example, suppose that an `Instrumentalist` bean is set to be autowired using `byName`. As a result, its `instrument` property will automatically be set to the bean in the container whose name is `instrument`. Let's say that you decide that you want to refactor the `instrument` property, renaming it as `musicalInstrument`. After refactoring, the container will try to autowire by looking for a bean named `musicalInstrument`. Unless you have changed the bean XML file, it won't find a bean by that name and will leave the property unwired. When the `Instrumentalist` bean tries to use the `musicalInstrument` property, you'll get a `NullPointerException`.

Worse still, what if there is a bean named `musicalInstrument` (not likely, but it could happen) but it isn't the bean you want wired to the `musicalInstrument` property? Depending on the type of the `musicalInstrument` bean, Spring may quietly wire in the unwanted bean, resulting in unexpected application behavior.

But the most serious shortcoming of autowiring is that it lacks clarity. When you autowire a property, your Spring configuration XML no longer contains the explicit details on how your beans are being wired. When you read the configuration, you are left to perform the autowiring yourself to figure out what's going on.

Similarly, Spring documentation tools such as Spring IDE and Spring BeanDoc won't have enough information to properly document autowired properties. This could result in documentation that implies that some beans aren't wired, even when they are.

Autowiring is a powerful feature. Nevertheless, as you may have heard, with great power comes great responsibility. If you choose to autowire, do so with caution.

Because autowiring hides so much of what is going on and because we want our examples to be abundantly clear, most of the examples in this book will not use autowiring. We'll leave it up to you whether you will autowire in your own applications.

2.5 **Controlling bean creation**

So far, you've seen a few of the most basic ways to configure beans in Spring and wire them into properties of other beans. In all cases, we've created beans in the most basic way: we've assumed that Spring will create one instance of the bean,

using a public constructor on the bean's class, and will perform no special initialization or destruction logic.

But before we wrap up this chapter, you should also know that Spring provides a handful of options for creating beans that go beyond what we've seen so far. You can do the following:

- Control how many instances of a specific bean are created, whether it is one instance for the entire application, one instance per user request, or a brand-new instance each time the bean is used.
- Create beans from static factory methods instead of public constructors.
- Initialize a bean after it is created and clean up just before it is destroyed.

Let's start by seeing how we can control how often a bean is created by using bean scoping.

2.5.1 Bean scoping

By default, all Spring beans are singletons. That is, when the container dispenses a bean (either through wiring or as the result of a call to the container's `getBean()` method) it will always hand out the exact same instance of the bean. But there may be times when you need a unique instance of a bean each time it is asked for. How can you override the default singleton nature of Spring?

When declaring a `<bean>` in Spring, you have the option of declaring a scope for that bean. To force Spring to produce a new bean instance each time one is needed, you should declare the bean's scope attribute to be `prototype`. For example, consider the following declaration of the saxophone bean:

```
<bean id="saxophone"
      class="com.springinaction.springidol.Saxophone"
      scope="prototype" />
```

Now when Spring wires the `saxophone` bean into an `Instrumentalist`'s `instrument` property, each `Instrumentalist` will get their own instance of `Saxophone` instead of sharing the same instance. This is another way to calm Kenny's concerns about hygiene and sharing his saxophone—other `Instrumentalists` can have their own saxophones and keep their spit off his.

In addition to `prototype`, Spring offers a handful of other scoping options out of the box, as listed in table 2.5.

For the most part, you'll probably want to leave scoping to the default singleton, but `prototype` scope may be useful in situations where you want to use

Table 2.5 Spring's bean scopes let you declare the scope under which beans are created without hard-coding the scoping rules in the bean class itself.

Scope	What it does
singleton	Scopes the bean definition to a single instance per Spring container (default).
prototype	Allows a bean to be instantiated any number of times (once per use).
request	Scopes a bean definition to an HTTP request. Only valid when used with a web-capable Spring context (such as with Spring MVC).
session	Scopes a bean definition to an HTTP session. Only valid when used with a web-capable Spring context (such as with Spring MVC).
global-session	Scopes a bean definition to a global HTTP session. Only valid when used in a portlet context.

Spring as a factory for new instances of domain objects. If domain objects are configured as prototype beans, you are able to easily configure them in Spring, just like any other bean. But Spring is guaranteed to always dispense a unique instance each time a prototype bean is asked for.

NOTE Scoping is new to Spring 2.0. Prior to Spring 2.0, you would set a <bean>'s singleton attribute to false to make it a prototype bean. The binary nature of the singleton attribute was quite limiting and didn't allow for more interesting bean scopes, so the scope attribute was added. This is one area where backward compatibility is somewhat broken between Spring 1.x and Spring 2.0. If you're using the Spring 2.0 DTD or XML schema when defining your context, you must use the scope attribute. But if you are still using the Spring 1.x DTD, you must use the singleton attribute.

The astute reader will recognize that Spring's notion of singletons is limited to the scope of the Spring context. Unlike true singletons, which guarantee only a single instance of a class per classloader, Spring's singleton beans only guarantee a single instance of the bean definition per the application context—there's nothing stopping you from instantiating that same class in a more conventional way or even defining several <bean> declarations that instantiate the same class.

2.5.2 ***Creating beans from factory methods***

Most of the time, the beans that you configure in the Spring application context will be created by invoking one of the class's constructors. Sure, you'll probably

write all of your classes with public constructors, but what if you’re using a third-party API that exposes some of its types through a static factory method? Does this mean that the class can’t be configured as a bean in Spring?

To illustrate, consider the case of configuring a singleton³ class as a bean in Spring. Singleton classes generally ensure that only one instance is created by only allowing creation through a static factory method. The `Stage` class in listing 2.10 is a basic example of a singleton class.

Listing 2.10 The Stage singleton class

```
package com.springinaction.springidol;

public class Stage {
    private Stage() {}

    private static class StageSingletonHolder {
        static Stage instance = new Stage();
    }

    public static Stage getInstance() {
        return StageSingletonHolder.instance;    ↴ Returns
    }                                         instance
}
```

↳ Lazily loads instance

In the *Spring Idol* competition, we want to ensure that there’s only one stage for the performers to show their stuff. `Stage` has been implemented as a singleton to ensure that there’s absolutely no way to create more than one instance of `Stage`.

But notice that `Stage` doesn’t have a public constructor. Instead, the static `getInstance()` method returns the same instance every time it’s called. (For thread safety, `getInstance()` employs a technique known as “initialization on demand holder” to create the singleton instance.⁴) How can we configure `Stage` as a bean in Spring without a public constructor?

Fortunately, the `<bean>` element has a `factory-method` attribute that lets you specify a static method to be invoked instead of the constructor to create an instance of a class. To configure `Stage` as a bean in the Spring context, we simply use `factory-method` as follows:

³ I’m talking about Gang of Four Singleton pattern here, not the Spring notion of singleton bean definitions.

⁴ For information on the “initialization on demand holder” idiom, see http://en.wikipedia.org/wiki/Initialization_on_demand_holder_idiom.

```
<bean id="theStage"
      class="com.springinaction.springidol.Stage"
      factory-method="getInstance" />
```

Here I've shown you how to use `factory-method` to configure a singleton as a bean in Spring, but it's perfectly suitable for any occasion where you need to wire an object produced by a static method. You'll see more of `factory-method` in chapter 4 when we use it to get references to AspectJ aspects so that they can be injected with dependencies.

But there's more bean-wiring ground to cover first. Some beans need setup performed after they're created and teardown work when they're done. Wrapping up this chapter, let's see how to hook into the bean's lifecycle to perform bean initialization and cleanup.

2.5.3 **Initializing and destroying beans**

When a bean is instantiated, it may be necessary to perform some initialization to get it into a usable state. Likewise, when the bean is no longer needed and is removed from the container, some cleanup may be in order. To accommodate setup and teardown of beans, Spring provides hooks into the bean lifecycle.

To define setup and teardown for a bean, simply declare the `<bean>` with `init-method` and/or `destroy-method` parameters. The `init-method` attribute specifies a method that is to be called on the bean immediately upon instantiation. Similarly, `destroy-method` specifies a method that is called just before a bean is removed from the container.

To illustrate, consider the `Instrumentalist` class. Anyone who's talented at playing musical instruments will tell you that the first thing they do before performing their art is to tune their instrument. So, let's add the following method to `Instrumentalist`:

```
public void tuneInstrument() {
    instrument.tune();
}
```

Likewise, after a performance, it's a good idea to clean the instrument:

```
public void cleanInstrument() {
    instrument.clean();
}
```

Now all we need is a way to ensure that the `tuneInstrument()` method is called when an `Instrumentalist` is created and `cleanInstrument()` is called when it is destroyed. For that, let's use the `init-method` and `destroy-method` attributes when declaring the `kenny` bean:

```
<bean id="kenny"
      class="com.springinaction.springidol.Instrumentalist"
      init-method="tuneInstrument"
      destroy-method="cleanInstrument">
    <property name="song" value="Jingle Bells" />
    <property name="instrument" ref="saxophone" />
</bean>
```

When declared this way, the `tuneInstrument()` method will be called soon after the `kenny` bean is instantiated, allowing it the opportunity to ensure a properly tuned instrument. And, just before the bean is removed from the container and discarded, `cleanInstrument()` will be called so that Kenny can take steps to preserve his instrument.

Defaulting init-method and destroy-method

If many of the beans in a context definition file will have initialization or destroy methods with the same name, you don't have to declare `init-method` or `destroy-method` on each individual bean. Instead you can take advantage of the `default-init-method` and `default-destroy-method` attributes on the `<beans>` element:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-2.0.xsd"
       default-init-method="tuneInstrument"
       default-destroy-method="cleanInstrument">
  ...
</beans>
```

The `default-init-method` attribute sets an initialization method across all beans in a given context definition. Likewise, `default-destroy-method` sets a common destroy method for all beans in the context definition. In this case, we're asking Spring to initialize all beans in the context definition file by calling `tuneInstrument()` and to tear them down with `cleanInstrument()` (if those methods exist—otherwise nothing happens).

InitializingBean and DisposableBean

As an option to `init-method` and `destroy-method`, we could also rewrite the `Instrumentalist` class to implement two special Spring interfaces: `InitializingBean` and `DisposableBean`. The Spring container treats beans that implement these interfaces in a special way by allowing them to hook into the bean lifecycle. Listing 2.11 shows an updated `Instrumentalist` class that implements both of these interfaces.

Listing 2.11 A version of Instrumentalist that implements Spring's lifecycle hook interfaces

```

package com.springinaction.springidol;
import org.springframework.beans.factory.InitializingBean;
import org.springframework.beans.factory.DisposableBean;

public class Instrumentalist implements Performer,
    InitializingBean, DisposableBean {
    public Instrumentalist() {}
    public void perform() throws PerformanceException {
        System.out.print("Playing " + song + " : ");
        instrument.play();
    }

    private String song;
    public void setSong(String song) {
        this.song = song;
    }

    private Instrument instrument;
    public void setInstrument(Instrument instrument) {
        this.instrument = instrument;
    }

    public void afterPropertiesSet() throws Exception {
        instrument.tune(); | Defines initialization method
    }

    public void destroy() throws Exception {
        instrument.clean(); | Defines teardown method
    }
}

```

The `InitializingBean` interface mandates that the class implements `afterPropertiesSet()`. This method will be called once all specified properties for the bean have been set. This makes it possible for the bean to perform initialization that can't be performed until the bean's properties have been completely set. In the `Instrumentalist` class, `afterPropertiesSet()` causes the `Instrumentalist` to tune its instrument.

Similarly, `DisposableBean` requires that a `destroy()` method be implemented. The `destroy()` method will be called on the other end of the bean's lifecycle, when the bean is disposed of by the container. In the case of `Instrumentalist`, the `destroy()` method ensures that the instrument is cleaned at the end of the bean's life.

The chief benefit of using these lifecycle interfaces is that the Spring container is able to automatically detect beans that implement them without any external

configuration. However, the disadvantage of implementing these interfaces is that you couple your application's beans to Spring's API. For this one reason alone, I recommend that you rely on the `init-method` and `destroy-method` attribute to initialize and destroy your beans. The only scenario where you might favor Spring's lifecycle interfaces is if you are developing a framework bean that is to be used specifically within Spring's container.

2.6 **Summary**

At the core of the Spring Framework is the Spring container. Spring comes with several implementations of its container, but they all fall into one of two categories. A `BeanFactory` is the simplest form of container, providing basic DI and bean-wiring services. But when more advanced framework services are needed, Spring's `ApplicationContext` is the container to use.

In this chapter, you've seen how to wire beans together within the Spring container. Wiring is typically performed within a Spring container using an XML file. This XML file contains configuration information for all of the components of an application, along with information that helps the container perform DI to associate beans with other beans that they depend on.

You've also seen how to instruct Spring to automatically wire beans together by using reflection and making some guesses about which beans should be associated with each other.

Everything you learned in this chapter is the basis for what is to come. It is the day-to-day stuff that you'll use when developing Spring-based applications. You'll continue working with Spring's bean definition XML file as we build the samples throughout the remainder of the book.

Whereas the techniques in this chapter are common to virtually every Spring application, the tricks in the next chapter are less commonplace. Coming up next, we'll explore some of the more advanced Spring configuration practices that, while not day-to-day, are very powerful and useful when you need them.

Advanced bean wiring

This chapter covers

- Parent/child bean creation
- Custom property editors
- Postprocessing beans
- Dynamically scripted beans

Most people have at least one drawer, cabinet, or closet somewhere in their house where miscellaneous odds and ends are kept. Although it's often called a junk drawer, many useful things end up in there. Things like measuring tape, binder clips, pens, pencils, thumbtacks, a handful of spare batteries, and endless supplies of twist ties tend to find a home in these places. You usually don't have a day-to-day use for those items, but you know that the next time that there's a power outage, you'll be digging in that drawer to find batteries to put in your flashlight.

In chapter 2, I showed you the day-to-day basics of Spring bean wiring. There's no doubt you'll have plenty of opportunities to use those techniques in your Spring-based applications. You'll keep them nearby and use them frequently.

In this chapter, however, we're going to dig a little into the Spring container's junk drawer. While the topics covered in this chapter are handy when you need them, they won't see nearly as much use as what we discussed in chapter 2. It's not often you'll need to replace a method in a bean or create a bean that knows its own name. And not every project needs to inject a Ruby class into a Spring-based Java application. But when you find yourself needing to do these kinds of things... well, you're just going to need them.

Since this chapter covers a few of Spring's more unusual features, you may want to skip past this chapter and move on to Spring's aspect-oriented features in chapter 4. This chapter will still be here waiting for you when you need it. But if you stick around, you'll see how to take your beans to the extreme, starting with how to create and extend abstract beans.

3.1 Declaring parent and child beans

One of the essentials of object-oriented programming is the ability to create a class that extends another class. The new class inherits many of the properties and methods of the parent class, but is able to introduce new properties and methods or even override the parent's properties and methods. If you're reading this book, you're probably a Java programmer and subclassing is old news to you. But did you know that your Spring beans can "sub-bean" other Spring beans?

A `<bean>` declaration in Spring is typically defined with a `class` attribute to indicate the type of the bean and zero or more `<property>` elements to inject values into the bean's properties. Everything about the `<bean>` is declared in one place.

But creating several individual `<bean>` declarations can make your Spring configuration unwieldy and brittle. For the same reasons you'd create hierarchies of classes in Java—to collect common functionality and properties in parent classes

that can be inherited by child classes—you’ll find it useful to create `<bean>`s that extend and inherit from other `<bean>` definitions. And it’s a great way to cut down on the amount of redundant XML in the Spring context definition files.

To accommodate sub-beaning, the `<bean>` element provides two special attributes:

- `parent`—Indicates the `id` of a `<bean>` that will be the parent of the `<bean>` with the `parent` attribute. The `parent` attribute is to `<bean>` what extends is to a Java class.
- `abstract`—If set to true, indicates that the `<bean>` declaration is abstract. That is, it should never be instantiated by Spring.

To illustrate Spring’s sub-beaning capabilities, let’s revisit the *Spring Idol* competition.

3.1.1 Abstracting a base bean type

As you’ll recall from chapter 2, Kenny was a contestant who entered the competition as an `Instrumentalist`. Specifically, Kenny’s specialty is the saxophone. Thus Kenny was declared as a bean in Spring using the following XML:

```
<bean id="kenny"
      class="com.springinaction.springidol.Instrumentalist">
    <property name="song" value="Jingle Bells" />
    <property name="instrument" ref="saxophone" />
</bean>
```

In the time that has passed since you read chapter 2, a new contestant has entered the contest. Coincidentally, David is also a saxophonist. What’s more, he will be playing the same song as Kenny. David is declared in Spring as follows:

```
<bean id="david"
      class="com.springinaction.springidol.Instrumentalist">
    <property name="song" value="Jingle Bells" />
    <property name="instrument" ref="saxophone" />
</bean>
```

Now we have two `<bean>`s declared in Spring with virtually the same XML. In fact, as illustrated in figure 3.1, the only difference between these two `<bean>`s are their `ids`. This may not seem like a big problem now, but imagine what might happen if we had 50 more saxophonists enter the contest, all wanting to perform “Jingle Bells.”

An obvious solution to the problem would be to disallow any further contestants from entering the competition. But we’ve already set a precedent by letting

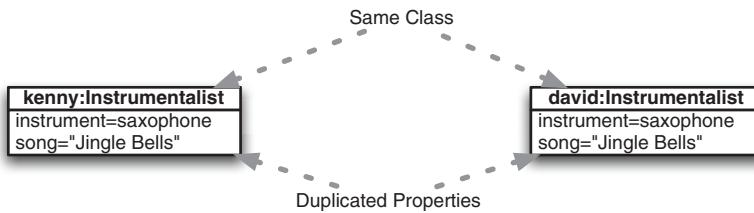


Figure 3.1 The two beans are the same type and have properties that are injected with the same values.

David in. And besides, we need an example for sub-beaning. So, we'll leave the contest open a little longer.

The other solution is to create a bean that will be the parent bean for all of our saxophonist contestants. The `baseSaxophonist` bean should do the trick:

```
<bean id="baseSaxophonist"
      class="com.springinaction.springidol.Instrumentalist"
      abstract="true">
    <property name="instrument" ref="saxophone" />
    <property name="song" value="Jingle Bells" />
</bean>
```

The `baseSaxophonist` bean isn't much different from the `kenny` and `david` beans from earlier. But notice that the `abstract` attribute has been set to true. This tells Spring to not try to instantiate this bean... even if it is explicitly requested from the container. In many ways this is the same as an abstract class in Java, which can't be instantiated.

Although it can't be instantiated, the `baseSaxophonist` bean is still very useful, because it contains the common properties that are shared between Kenny and David. Thus, we are now able to declare the `kenny` and `david` beans as follows:

```
<bean id="kenny" parent="baseSaxophonist" />
<bean id="david" parent="baseSaxophonist" />
```

The `parent` attribute indicates that both the `kenny` and `david` beans will inherit their definition from the `baseSaxophonist` bean. Notice that there's no `class` attribute. That's because `kenny` and `david` will inherit the parent bean's class as well as its properties. The redundant XML is now gone and these `<bean>` elements are much simpler and more concise, as shown in figure 3.2.

It is appropriate at this point to mention that the parent bean doesn't have to be abstract. It's entirely possible to create sub-beans that extend a concrete bean.

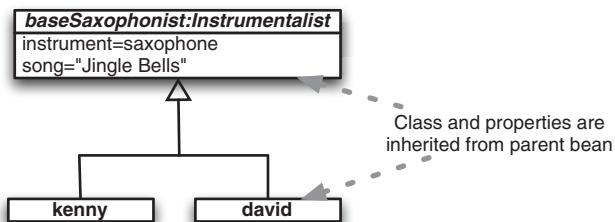


Figure 3.2 The `kenny` and `david` beans share some common configuration. By creating a parent bean with the common properties and inheriting from it, redundant configuration can be eliminated.

But in this example, we know that there's never any reason for Spring to instantiate the `baseSaxophonist` bean, so we declared it to be abstract.

Overriding inherited properties

Now let's suppose that another saxophonist were to enter the contest (I told you it would happen). But this saxophonist will be performing "Mary Had a Little Lamb" instead of "Jingle Bells." Does this mean that we can't sub-bean `baseSaxophonist` when declaring the new contestant?

Not at all. We can still sub-bean `baseSaxophonist`. But instead of accepting all of the inherited property values, we'll override the `song` property. The following XML defines the new saxophonist:

```

<bean id="frank" parent="baseSaxophonist">
    <property name="song" value="Mary had a little lamb" />
</bean>

```

The `frank` bean will still inherit the `class` and `<property>`s of the `baseSaxophonist` bean. But, as shown in figure 3.3, `frank` overrides the `song` property so that he can perform a song of a girl and her wooly pet.

In many ways, parent and child `<bean>` declarations mirror the capabilities of parent and child class definitions in Java. But hang on... I'm about to show you something that Spring inheritance can do that can't be done in Java.

3.1.2 Abstracting common properties

In the *Spring Idol* talent competition, we may have several musically inclined contestants. As you've seen already, we have several instrumentalists who perform songs on their instruments. But there may also be vocalists who use their voices to sing their songs.

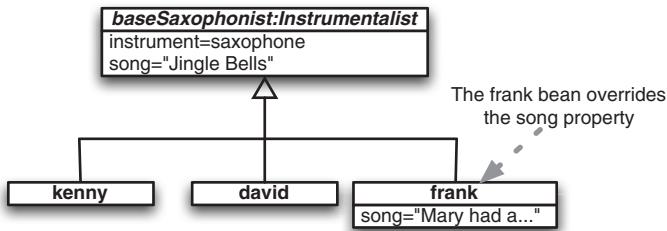


Figure 3.3 The `frank` bean inherits from the `baseSaxophonist` bean, but overrides the `song` property.

Suppose that the *Spring Idol* talent competition has two performers, one vocalist and one guitarist, who will be performing the same song. Configured as separate beans, these performers may appear like this in the Spring configuration file:

```

<bean id="taylor"
      class="com.springinaction.springidol.Vocalist">
    <property name="song" value="Somewhere Over the Rainbow" />
</bean>

<bean id="stevie"
      class="com.springinaction.springidol.Instrumentalist">
    <property name="instrument" ref="guitar" />
    <property name="song" value="Somewhere Over the Rainbow" />
</bean>

```

As before, these two beans share some common configuration. Specifically, they both have the same value for their `song` attribute. But unlike the example in the previous section, the two beans do not share a common class. So, we can't create a common parent bean for them. Or can we?

In Java, children classes share a common base type that is determined by their parent bean. This means that there's no way in Java for children classes to extend a common type to inherit properties and/or methods but not to inherit the common parent.

In Spring, however, sub-beans do not have to share a common parent type. Two `<bean>`s with completely different values in their `class` attributes can still share a common set of properties that are inherited from a parent bean.

Consider the following abstract parent `<bean>` declaration:

```

<bean id="basePerformer" abstract="true">
  <property name="song" value="Somewhere Over the Rainbow" />
</bean>

```

This `basePerformer` bean declares the common `song` property that our two performers will share. But notice that it doesn't have a `class` attribute set. That's okay, because each child bean will determine its own type with its own `class` attribute. The new declarations of `taylor` and `stevie` are as follows:

```
<bean id="taylor"
      class="com.springinaction.springidol.Vocalist"
      parent="basePerformer" />

<bean id="stevie"
      class="com.springinaction.springidol.Instrumentalist"
      parent="basePerformer">
    <property name="instrument" ref="guitar" />
</bean>
```

Notice that these beans use both the `class` attribute and the `parent` attribute alongside each other. This makes it possible for two or more otherwise disparate beans to share and inherit property values from a common base `<bean>` declaration. As illustrated in figure 3.4, the `song` property value is inherited from the `basePerformer` bean, even though each child bean has a completely different and unrelated type.

Sub-beaning is commonly used to reduce the amount of XML required to declare aspects and transactions. In chapter 6 (section 6.4.2), you'll see how sub-beaning can greatly reduce the amount of redundant XML when declaring transactions with `TransactionProxyFactoryBean`.

But for now, it's time to continue rummaging through the junk drawer to see what other useful tools we might find. We've already seen setter and constructor injection in chapter 2. Next up, let's see how Spring provides another kind of injection that allows you to declaratively inject methods into your beans, effectively altering a bean's functionality without having to change its underlying Java code.

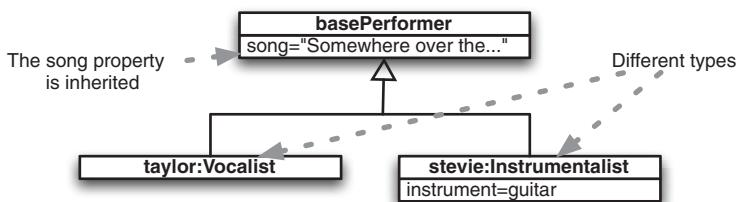


Figure 3.4 Parent beans don't have to specify a type. They can also be used to hold common configuration that can be inherited by beans of any type.

3.2 Applying method injection

In chapter 2, you saw two basic forms of dependency injection (DI). Constructor injection lets you configure your beans by passing values to the beans' constructors. Meanwhile, setter injection lets you configure your beans by passing values through the setter methods. Before you're finished with this book, you'll see hundreds of examples of setter injection and maybe a few more examples of constructor injection.

But in this section, I'd like to show you an unusual form of DI called method injection. With setter and constructor injection, you were injecting values into a bean's properties. But as illustrated in figure 3.5, method injection is quite different, because it lets you inject entire method definitions into a bean.

Some languages, such as Ruby, allow you to add new methods to any class at runtime, without changing the class's definition. For example, if you'd like to add a new method to Ruby's `String` class that will print the string's length, it's a simple matter of defining the new method:

```
class String
  def print_length
    puts "This string is #{self.length} characters long"
  end
end
```

With the method defined, you'll be able to invoke it on any `String` you create. For example:

```
message = "Hello"
message.print_length
```

will print “This string is 5 characters long” to the standard output.

But that's Ruby—the Java language isn't quite as flexible. Nevertheless, Spring offers Java programmers method injection to allow runtime injection of methods into Java classes. It's not quite as elegant as Ruby's language constructs for method injection, but it's a step in that direction.



Figure 3.5 Method injection is a type of injection where a class's methods are replaced by an alternate implementation.

Spring supports two forms of method injection:

- *Method replacement*—Enables existing methods (abstract or concrete) to be replaced at runtime with new implementations.
- *Getter injection*—Enables existing methods (abstract or concrete) to be replaced at runtime with a new implementation that returns a specific bean from the Spring context.

To get started with method injection, let's have a look at how Spring supports general-purpose method replacement.

3.2.1 Basic method replacement

Do you like magic shows? Magicians use sleight of hand and distraction to make the seemingly impossible appear right before our eyes. One of our favorite magic tricks is when the magician puts his assistant in a box, spins the box around, chants some magic words, then... PRESTO! The box opens to reveal that a tiger has replaced the assistant.

It just so happens that the Harry, a promising prestidigitator, has entered the *Spring Idol* talent competition and will be performing our favorite illusion. Allow me to introduce you to Harry, first by showing you the Magician class in listing 3.1.

Listing 3.1 A magician and his magic box

```
package com.springinaction.springidol;

public class Magician implements Performer {
    public Magician() {}

    public void perform() throws PerformanceException {
        System.out.println(magicWords);
        System.out.println("The magic box contains...");
        System.out.println(magicBox.getContents());      ↪ Inspects contents
                                                       of magic box
    }

    // injected
    private MagicBox magicBox;
    public void setMagicBox(MagicBox magicBox) {
        this.magicBox = magicBox;
    }                                              ↪ Injects magic box

    private String magicWords;
    public void setMagicWords(String magicWords) {
        this.magicWords = magicWords;
    }
}
```

As you can see, the Magician class has two properties that will be set by Spring DI. The Magician needs some magic words to make the illusion work, so those will be set with the `magicWords` property. But most importantly, he'll be given his magic box through the `magicBox` property. Speaking of the magic box, you'll find it implemented in listing 3.2.

Listing 3.2 The magic box implementation contains a beautiful assistant... or does it?

```
package com.springinaction.springidol;

public class MagicBoxImpl implements MagicBox {
    public MagicBoxImpl() {}

    public String getContents() {
        return "A beautiful assistant";    ↪ Contains a
    }                                ↪ beautiful assistant
}
```

The key thing about the `MagicBoxImpl` class that you should draw your attention to is the `getContents()` method. You'll notice that it's hard-coded to always return "A beautiful assistant"—but as you'll soon see, things aren't always as they appear.

But before I show you the trick, here's how Harry and his magic box are wired up in the Spring application context:

```
<bean id="harry"
      class="com.springinaction.springidol.Magician">
    <property name="magicBox" ref="magicBox" />
    <property name="magicWords" value="Bippity boppity boo" />
</bean>

<bean id="magicBox"
      class="com.springinaction.springidol.MagicBoxImpl" />
```

If you've ever seen this trick before, you'll know that the magician always teases the audience by closing the box, then opening it again to show that the assistant is still in there. Harry's act is no different. Let's kick off the magic trick using the following code snippet:

```
ApplicationContext ctx = ... // load Spring context
Performer magician = (Performer) ctx.getBean("harry");
magician.perform();
```

When this code snippet is run to retrieve the `harry` bean from the application context and when the `perform()` method is invoked, you'll see that everything is as it is supposed to be:

```
Bippity boppity boo
The magic box contains...
A beautiful assistant
```

This output should be no surprise to you. After all, the `MagicBoxImpl` is hard-coded to return “A beautiful assistant” when `getContents()` is invoked. But as I said, this was just a teaser. Harry hasn’t performed the actual magic trick yet. But now it’s time for the illusion to begin, so let’s tweak the configuration XML to appear as follows:

```
<bean id="magicBox"
      class="com.springinaction.springidol.MagicBoxImpl">
    <replaced-method
      name="getContents"
      replacer="tigerReplacer" />
  </bean>

  <bean id="tigerReplacer"
        class="com.springinaction.springidol.TigerReplacer" />
```

Now the `magicBox` bean has a `<replaced-method>` element (see figure 3.6). As its name implies, this element is used to replace a method with a new method implementation. In this case, the `name` attribute specifies the `getContents()` method to be replaced. And the `replacer` attribute refers to a bean called `tigerReplacer` to implement the replacement.

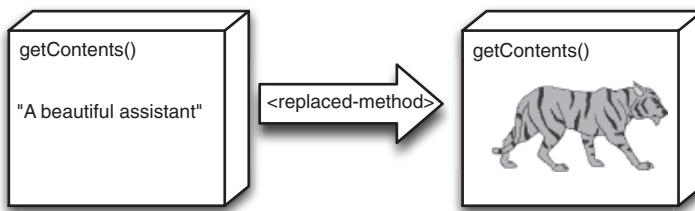


Figure 3.6 Using Spring’s `<replaced-method>` injection, it’s easy to replace the `getContents()` method that returns “A beautiful assistant” with an implementation that produces a tiger.

Here’s where the real sleight-of-hand takes place. The `tigerReplacer` bean is a `TigerReplacer`, as defined in listing 3.3.

Listing 3.3 TigerReplacer, which replaces a method implementation

```
package com.springinaction.springidol;
import java.lang.reflect.Method;
import org.springframework.beans.factory.support.MethodReplacer;
```

```

public class TigerReplacer implements MethodReplacer {
    public Object reimplement(Object target, Method method,
        Object[] args) throws Throwable {
        return "A ferocious tiger";   ← Puts tiger
    }                                in box
}

```

Replaces
method

TigerReplacer implements Spring’s MethodReplacer interface. MethodReplacer only requires that the reimplement() method be implemented. reimplement() takes three arguments: the target object that the method will be replaced on, the method that will be replaced, and the arguments passed to the method. In our case, we won’t be using those parameters, but they’re available if you need them.

The body of the reimplement() method effectively becomes the new implementation for the magic box’s getContents() method. For our example, the only thing we want to do is return “A ferocious tiger.” Effectively, the contents of the box are replaced with a tiger, as shown in figure 3.6.

Now when we invoke the magician’s perform() method, the following is printed to the console:

```

Bippity boppity boo
The magic box contains...
A ferocious tiger

```

Ta-da! The beautiful assistant has been replaced by a ferocious tiger—without changing the actual MagicBoxImpl code. The magic trick has been successfully performed with the help of Spring’s <replaced-method>.

It’s worth noting here that although MagicBoxImpl has a concrete implementation of the getContents() method, it would’ve also been possible for getContents() to have been written as an abstract method. In fact, method injection is a trick that is useful when the actual implementation of the method isn’t known until deployment time. At that time, the actual method replacer class can be provided in a JAR file placed in the application’s classpath.

General replacement of methods is certainly a neat trick. But there’s a more specific form of method injection that enables runtime binding of beans to a getter method. Let’s see how to perform getter injection on Spring beans.

3.2.2 Using getter injection

Getter injection is a special case of method injection where a method (which is typically abstract) is declared to return a bean of a certain type, but the actual bean to be returned is configured in the Spring context.

To illustrate, consider the new method-injected form of the `Instrumentalist` class in listing 3.4.

Listing 3.4 A method-injected `Instrumentalist`

```
package com.springinaction.springidol;

public abstract class Instrumentalist implements Performer {
    public Instrumentalist() {}

    public void perform() throws PerformanceException {
        System.out.print("Playing " + song + " : ");
        getInstrument().play();    ↪ Uses injected
    }                                getInstrument() method

    private String song;
    public void setSong(String song) {
        this.song = song;
    }

    public abstract Instrument getInstrument();    ↪ Injects
}                                getInstrument()
```

Unlike the original `Instrumentalist`, this class isn't given its instrument through setter injection. Instead, there's an abstract `getInstrument()` method that will return the performer's instrument. But if `getInstrument()` is abstract then the big question is how the method gets implemented.

One possible approach is to use general-purpose method replacement as described in the last section. But that would require writing a class that implements `MethodReplacer`, when all we really need to do is override the `getInstrument()` method to return a specific bean.

For getter-style injection, Spring offers the `<lookup-method>` configuration element. Like `<replaced-method>`, `<lookup-method>` replaces a method with a new implementation at runtime. But `<lookup-method>` is a getter-injection shortcut for `<replaced-method>` where you can specify which bean in the Spring context to return from the replaced method. `<lookup-method>` will take care of the rest. There's no need to write a `MethodReplacer` class.

The following XML demonstrates how to use `<lookup-method>` to replace the `getInstrument()` method with one that returns a reference to the `guitar` bean:

```
<bean id="stevie"
      class="com.springinaction.springidol.Instrumentalist">
    <lookup-method name="getInstrument" bean="guitar" />
    <property name="song" value="Greensleeves" />
</bean>
```

As with `<replaced-method>`, the name attribute of `<lookup-method>` indicates the method that is to be replaced. Here we're replacing the `getInstrument()` method. The `bean` attribute refers to a bean that should be returned when `getInstrument()` is called. In this case, we're wiring in the bean whose `id` is `guitar`. As a result, the `getInstrument()` method is effectively overridden as follows:

```
public Instrument getInstrument() {  
    ApplicationContext ctx = ...;  
  
    return (Instrument) ctx.getBean("guitar");  
}
```

On its own, getter injection is just a reversal of setter injection. However, it makes a difference when the referenced bean is prototype scoped:

```
<bean id="guitar" class="com.springinaction.springidol.Guitar"  
      scope="prototype" />
```

Even though it's prototype scoped, the `guitar` method would only be injected once into a property if we were using setter injection. By injecting it into the `getInstrument()` method through getter injection, however, we ensure that every call to `getInstrument()` will return a different guitar. This can come in handy if a guitarist breaks a string in the middle of a performance and needs a freshly stringed instrument.

You should be aware that although we used `<lookup-method>` to perform getter injection on the `getInstrument()` method, there's nothing about `<lookup-method>` that requires that the replaced method actually be a getter method (i.e., one whose name starts with `get`). Any non-void method is a candidate for replacement with `<lookup-method>`.

It's important to note that while method injection allows you to change a method's implementation, it's not capable of replacing a method's signature. The parameters and return type must remain the same. For `<lookup-method>`, this means that the `bean` attribute must refer to a bean that is assignable to the type being returned from the method (`Instrument` in the previous examples).

3.3 **Injecting non-Spring beans**

As you learned in chapter 2, one of Spring's main jobs is to configure instances of beans. But up to now there has always been one implied caveat: Spring can only configure beans that it also instantiates. This may not seem too odd at first glance, but it presents some interesting problems. Not all objects in an application are created by Spring. Consider the following scenarios:

- Custom JSP tags are instantiated by the web container that the application is running within. If a JSP tag needs to collaborate with some other object, it must explicitly create the object itself.
- Domain objects are typically instantiated at runtime by an ORM tool such as Hibernate or iBATIS. In a rich domain model, domain objects have both state and behavior. But if you can't inject service objects into a domain object then your domain object must somehow obtain its own instance of the service object or the behavior logic must be fully contained in the domain object.

And there may be other very valid reasons why you may want Spring to configure objects that it doesn't create. To illustrate, suppose that we were to explicitly instantiate an `Instrumentalist` from the *Spring Idol* example:

```
Instrumentalist pianist = new Instrumentalist();
pianist.perform();
```

Because `Instrumentalist` is a POJO, there's no reason why we can't instantiate it directly. But when the `perform()` method is invoked, a `NullPointerException` will be thrown. That's because although we are allowed to instantiate an `Instrumentalist` ourselves, the `instrument` property will be null.

Naturally, we could manually configure the properties of the `Instrumentalist`. For example:

```
Piano piano = new Piano();
pianist.setInstrument(piano);
pianist.setSong("Chopsticks");
```

Although manually injecting the properties of the `Instrumentalist` will work, it doesn't take advantage of Spring's ability to separate configuration from code. Moreover, if an ORM were to instantiate `Instrumentalist`, we'd never get a chance to configure its properties.

Fortunately, Spring 2.0 introduced the ability to declaratively configure beans that are instantiated outside of Spring. The idea is that these beans are Spring configured but not Spring created.

Consider the `Instrumentalist` bean that was explicitly created earlier. Ideally, we would like to configure the `pianist` external to our code and let Spring inject it with an instrument and a song to play. The following XML shows how we might do this:

```
<bean id="pianist"
      class="com.springinaction.springidol.Instrumentalist"
      abstract="true">
```

```
<property name="song" value="Chopsticks" />
<property name="instrument">
    <bean class="com.springinaction.springidol.Piano" />
</property>
</bean>
```

For the most part, there should be nothing out of the ordinary about this `<bean>` declaration. It declares the `pianist` bean as being an `Instrumentalist`. And it wires values into the `song` and `instrument` properties. It's just your run-of-the-mill Spring `<bean>`, except for one small thing: its `abstract` attribute is set to true.

As you'll recall from section 3.1, setting `abstract` to true tells Spring that you don't want the container to instantiate the bean. It's often used to declare a parent bean that will be extended by a sub-bean. But in this case we're just indicating to Spring that the `pianist` bean shouldn't be instantiated—it will be created outside of Spring.

Actually, the `pianist` bean only serves as a blueprint for Spring to follow when it configures an `Instrumentalist` that is created outside of Spring. With this blueprint defined, we need some way to associate it with the `Instrumentalist` class. To do that, we'll annotate the `Instrumentalist` class with `@Configurable`:

```
package com.springinaction.springidol;
import org.springframework.beans.factory.annotation.Configurable;

@Configurable("pianist")
public class Instrumentalist implements Performer {
    ...
}
```

The `@Configurable` annotation does two things:

- First, it indicates that `Instrumentalist` instances can be configured by Spring, even if they're created outside of Spring.
- It also associates the `Instrumentalist` class with a bean whose `id` is `pianist`. When Spring tries to configure an instance of `Instrumentalist`, it will look for a `pianist` bean to use as a template.

So, just how does Spring know to configure beans with `@Configurable` annotations? Well, there's one last thing to add to the Spring configuration:

```
<aop:spring-configured />
```

The `<aop:spring-configured>` configuration element is one of the many new configuration elements introduced in Spring 2.0. Its presence indicates to Spring that there are some beans that it will configure, even though they will be created elsewhere.

Under the hood, `<aop:spring-configured>` sets up an AspectJ aspect with a pointcut that is triggered when any bean annotated with `@Configurable` is instantiated. Once the bean has been created, the aspect cuts in and injects values into the new instance based on the `<bean>` template in the Spring configuration.

Because the Spring-configured aspect is an AspectJ aspect, this means that your application will need to run within an AspectJ-enabled JVM. The best way to AspectJ-enable a Java 5 JVM is to start it with the following JVM argument:

```
-javaagent:/path/to/aspectjweaver.jar
```

This effectively tells the JVM to perform load-time weaving of any AspectJ aspects. To do this, it will need to know where AspectJ's weaver classes are located. You'll need to replace

```
/path/to/aspectjweaver.jar
```

with the actual path to the `aspectjweaver.jar` file on your system (unless, of course, your `aspectjweaver.jar` file is located in the `/path/to` directory).

In this section we explicitly instantiated an `Instrumentalist` as a simple demonstration of Spring's ability to configure beans that it didn't create. Nevertheless, as I mentioned before, the configured bean more likely would've been instantiated by an ORM or some third-party library if this had been a real-world application.

Now let's see how Spring's property editors make simple work of injecting complex values based on String representations.

3.4 Registering custom property editors

As you go through this book, you'll see several examples where a complex property is set with a simple `String` value. For example, in chapter 9, you'll see how to wire web services in Spring using `JaxRpcPortProxyFactoryBean`. One of the properties of `JaxRpcPortProxyFactoryBean` that you'll need to set is `wsdlDocumentUrl`. This property is of the type `java.net.URL`. But instead of creating a `java.net.URL` bean and wiring it into this property, you can configure it using a `String` like this:

```
<property name="wsdlDocumentUrl"
  value="http://www.xmethods.net/sd/BabelFishService.wsdl" />
```

Under the covers, Spring automagically converts the `String` value to a `URL` object. Actually, the magic behind this trick isn't something Spring provides, but rather comes from a little-known feature of the original JavaBeans API. The

`java.beans.PropertyEditor` interface provides a means to customize how String values are mapped to non-String types. A convenience implementation of this interface—`java.beans.PropertyEditorSupport`—has two methods of interest to us:

- `getAsText()` returns the String representation of a property's value.
- `setAsText(String value)` sets a bean property value from the String value passed in.

If an attempt is made to set a non-String property to a String value, the `setAsText()` method is called to perform the conversion. Likewise, the `getAsText()` method is called to return a textual representation of the property's value.

Spring comes with several custom editors based on `PropertyEditorSupport`, including `org.springframework.beans.propertyeditors.URLEditor`, which is the custom editor used to convert Strings to and from `java.net.URL` objects. Spring's selection of custom editors appears in table 3.1.

In addition to the custom editors in table 3.1, you can write your own custom editor by extending the `PropertyEditorSupport` class. For example, suppose that your application has a `Contact` bean that conveniently carries contact

Table 3.1 Spring comes with several custom property editors that automatically turn injected String values into more complex types.

Property editor	What it does
<code>ClassEditor</code>	Sets a <code>java.lang.Class</code> property from a String whose value contains a fully qualified class name
<code>CustomDateEditor</code>	Sets a <code>java.util.Date</code> property from a String using a custom <code>java.text.DateFormat</code> object
<code>FileEditor</code>	Sets a <code>java.io.File</code> property from a String value containing the file's path
<code>LocaleEditor</code>	Sets a <code>java.util.Locale</code> property from a String value that contains a textual representation of the local (i.e., <code>en_US</code>)
<code>StringArrayPropertyEditor</code>	Converts a comma-delimited String to a String array property
<code>StringTrimmerEditor</code>	Automatically trims String properties with an option to convert empty String values to null
<code>URLEditor</code>	Sets a <code>java.net.URL</code> property from a String containing a URL specification

information about the people in your organization. Among other things, the Contact bean has a phoneNumber property that holds the contact phone number:

```
public Contact {
    private PhoneNumber phoneNumber;

    public void setPhoneNumber(PhoneNumber phoneNumber) {
        this.phoneNumber = phoneNumber;
    }
}
```

The phoneNumber property is of type PhoneNumber and is defined as follows:

```
public PhoneNumber {
    private String areaCode;
    private String prefix;
    private String number;

    public PhoneNumber() { }

    public PhoneNumber(String areaCode, String prefix,
                      String number) {
        this.areaCode = areaCode;
        this.prefix = prefix;
        this.number = number;
    }
    ...
}
```

Using basic wiring techniques learned in chapter 2, you could wire a PhoneNumber object into the Contact bean's phoneNumber property as follows:

```
<beans>
    <bean id="infoPhone"
          class="com.springinaction.chapter03.propeditor.PhoneNumber">
        <constructor-arg value="888" />
        <constructor-arg value="555" />
        <constructor-arg value="1212" />
    </bean>
    <bean id="contact"
          class="com.springinaction.chapter03.propeditor.Contact">
        <property name="phoneNumber" ref="infoPhone" />
    </bean>
</beans>
```

Notice that you had to define a separate infoPhone bean to configure the PhoneNumber object and then wire it into the phoneNumber property of the contact bean.

Instead, let's suppose you were to write a custom PhoneEditor like this:

```
public class PhoneEditor
    extends java.beans.PropertyEditorSupport {
```

```
public void setAsText(String textView) {
    String stripped = stripNonNumeric(textValue);

    String areaCode = stripped.substring(0,3);
    String prefix = stripped.substring(3,6);
    String number = stripped.substring(6);
    PhoneNumber phone = new PhoneNumber(areaCode, prefix, number);
    setValue(phone);
}

private String stripNonNumeric(String original) {
    StringBuffer allNumeric = new StringBuffer();

    for(int i=0; i<original.length(); i++) {
        char c = original.charAt(i);
        if(Character.isDigit(c)) {
            allNumeric.append(c);
        }
    }

    return allNumeric.toString();
}
}
```

Now the only thing left is to get Spring to recognize your custom property editor when wiring bean properties. For that, you'll need to use Spring's `CustomEditorConfigurer`. `CustomEditorConfigurer` is a `BeanFactoryPostProcessor` that loads custom editors into the `BeanFactory` by calling the `registerCustomEditor()` method. (Optionally, you can call the `registerCustomEditor()` method in your own code after you have an instance of the bean factory.)

By adding the following piece of XML to the bean configuration file, you'll tell Spring to register the `PhoneEditor` as a custom editor:

```
<bean
    class="org.springframework.beans.factory.config.
        ↪CustomEditorConfigurer">
    <property name="customEditors">
        <map>
            <entry key="com.springinaction.chapter03.propeditor.
                ↪PhoneNumber">
                <bean id="phoneEditor"
                    class="com.springinaction.chapter03.propeditor.
                        ↪PhoneEditor">
                    </bean>
            </entry>
        </map>
    </property>
</bean>
```

Now you'll be able to configure the Contact object's phoneNumber property using a simple String value and without creating a separate infoPhone bean:

```
<bean id="contact"
      class="com.springinaction.chapter03.propeditor.Contact">
    <property name="phoneNumber" value="888-555-1212" />
</bean>
```

Note that many of the custom editors that come with Spring (such as URLEditor and LocaleEditor) are already registered with the bean factory upon container startup. You do not need to register them yourself using CustomEditorConfigurer.

Property editors are just one way to customize how Spring creates and injects beans. There are other types of beans that the Spring container gives special consideration. Next up, let's see how to create some special beans that let you customize how the Spring container wires up beans.

3.5 **Working with Spring's special beans**

Most beans configured in a Spring container are treated equally. Spring configures them, wires them together, and makes them available for use within an application. Nothing special.

But some beans have a higher purpose. By implementing certain interfaces, you can cause Spring to treat beans as special—as part of the Spring Framework itself. By taking advantage of these special beans, you can configure beans that

- Become involved in the bean's creation and the bean factory's lifecycles by postprocessing bean configuration
- Load configuration information from external property files
- Load textual messages from property files, including internationalized messages
- Listen for and respond to application events that are published by other beans and by the Spring container itself
- Are aware of their identity within the Spring container

In some cases, these special beans already have useful implementations that come packaged with Spring. In other cases, you'll probably want to implement the interfaces yourself.

Let's start the exploration of Spring's special beans by looking at Spring's special beans that perform postprocessing of other beans after the beans have been wired together.

3.5.1 Postprocessing beans

In chapter 2, you learned how to define beans within the Spring container and how to wire them together. For the most part, you have no reason to expect beans to be wired in any way different than how you define them in the bean definition XML file. The XML file is perceived as the source of truth regarding how your application's objects are configured.

But as you saw in figures 2.2 and 2.3, Spring offers two opportunities for you to cut into a bean's lifecycle and review or alter its configuration. This is called *post-processing*. From the name, you probably deduced that this processing is done after some event has occurred. The event this postprocessing follows is the instantiation and configuration of a bean. The BeanPostProcessor interface gives you two opportunities to alter a bean after it has been created and wired:

```
public interface BeanPostProcessor {  
    Object postProcessBeforeInitialization(  
        Object bean, String name) throws BeansException;  
  
    Object postProcessAfterInitialization(  
        Object bean, String name) throws BeansException;  
}
```

The postProcessBeforeInitialization() method is called immediately prior to bean initialization (the call to afterPropertiesSet() and the bean's custom init-method). Likewise, the postProcessAfterInitialization() method is called immediately after initialization.

Writing a bean postprocessor

For example, suppose that you wanted to alter all String properties of your application beans to translate them into Elmer Fudd-speak. The Fuddifier class in listing 3.5 is a BeanPostProcessor that does just that.

Listing 3.5 Listing 3.5 Fuddifying String properties using a BeanPostProcessor

```
public class Fuddifier implements BeanPostProcessor {  
    public Object postProcessAfterInitialization(  
        Object bean, String name) throws BeansException {  
        Field[] fields = bean.getClass().getDeclaredFields();  
  
        try {  
            for(int i=0; i < fields.length; i++) {  
                if(fields[i].getType().equals(  
                    java.lang.String.class)) {  
                    fields[i].setAccessible(true);  
                    String original = (String) fields[i].get(bean);  
                    fields[i].set(bean, fuddify(original));  
                }  
            }  
        } catch (Exception e) {  
            throw new BeansException("Error fuddifying bean " +  
                bean + ": " + e.getMessage());  
        }  
    }  
}
```

“Fuddifies” all
String properties
of beans

```

        }
    } catch (IllegalAccessException e) {
        e.printStackTrace();
    }

    return bean;
}

private String fuddify(String orig) {
    if(orig == null) return orig;
    return orig.replaceAll("(r|l)", "w")
        .replaceAll("(R|L)", "W");
}

public Object postProcessBeforeInitialization(
    Object bean, String name) throws BeansException {
    return bean;
}
}

```

“Fuddifies” all String properties of beans

Does nothing before initialization

The `postProcessAfterInitialization()` method cycles through all of the bean’s properties, looking for those that are of type `java.lang.String`. For each `String` property, it passes it off to the `fuddify()` method, which translates the `String` into Fudd-speak. Finally, the property is changed to the “Fuddified” text. (You’ll also notice a call to each property’s `setAccessible()` method to get around the private visibility of a property. I realize that this breaks encapsulation, but how else could I pull this off?)

The `postProcessBeforeInitialization()` method is left purposefully unexciting; it simply returns the bean unaltered. Actually, the “Fuddification” process could have occurred just as well in this method.

Now that we have a Fuddifying BeanPostProcessor, let’s look at how to tell the container to apply it to all beans.

Registering bean postprocessors

If your application is running within a bean factory, you’ll need to programmatically register each BeanPostProcessor using the factory’s `addBeanPostProcessor()` method:

```

BeanPostProcessor fuddifier = new Fuddifier();
factory.addBeanPostProcessor(fuddifier);

```

More likely, however, you’ll be using an application context. For an application context, you’ll only need to register the postprocessor as a bean within the context:

```
<bean  
    class="com.springinaction.chapter03.postprocessor.Fuddifier"/>
```

The container will recognize the fuddifier bean as a BeanPostProcessor and call its postprocessing methods before and after each bean is initialized.

As a result of the fuddifier bean, all String properties of all beans will be Fuddified. For example, suppose you had the following bean defined in XML:

```
<bean id="bugs" class="com.springinaction.chapter03.postprocessor.Rabbit">  
    <property name="description" value="That rascally rabbit!" />  
</bean>
```

When the fuddifier postprocessor is finished, the description property will hold “That wascawwy wabbit!”

Spring's own bean postprocessors

The Spring Framework itself uses several implementations of BeanPostProcessor under the covers. For example, ApplicationContextAwareProcessor is a BeanPostProcessor that sets the application context on beans that implement the ApplicationContextAware interface (see section 3.5.6). You do not need to register ApplicationContextAwareProcessor yourself. It is preregistered by the application context itself.

In the next chapter, you'll learn of another implementation of BeanPostProcessor. You'll also learn how to automatically apply aspects to application beans using DefaultAdvisorAutoProxyCreator, which is a BeanPostProcessor that creates AOP proxies based on all candidate advisors in the container.

3.5.2 Postprocessing the bean factory

Whereas a BeanPostProcessor performs postprocessing on a bean after it has been loaded, a BeanFactoryPostProcessor performs postprocessing on the entire Spring container. The BeanFactoryPostProcessor interface is defined as follows:

```
public interface BeanFactoryPostProcessor {  
    void postProcessBeanFactory(  
        ConfigurableListableBeanFactory beanFactory)  
        throws BeansException;  
}
```

The postProcessBeanFactory() method is called by the Spring container after all bean definitions have been loaded but before any beans are instantiated (including BeanPostProcessor beans).

For example, the following `BeanFactoryPostProcessor` implementation gives a completely new meaning to the term “bean counter”:

```
public class BeanCounter implements BeanFactoryPostProcessor {
    private Logger LOGGER = Logger.getLogger(BeanCounter.class);
    public void postProcessBeanFactory(
        ConfigurableListableBeanFactory factory)
        throws BeansException {
        LOGGER.debug("BEAN COUNT: " +
            factory.getBeanDefinitionCount());
    }
}
```

`BeanCounter` is a `BeanFactoryPostProcessor` that simply logs the number of bean definitions that have been loaded into the bean factory. If you’re using an application context container, registering a `BeanFactoryPostProcessor` is as simple as declaring it as a regular bean:

```
<bean id="beanCounter"
    class="com.springinaction.chapter03.postprocessor.
        BeanCounter"/>
```

When the container sees that `beanCounter` is a `BeanFactoryPostProcessor`, it will automatically register it as a bean factory postprocessor. You cannot use `BeanFactoryPostProcessors` with basic bean factory containers—this feature is only available with application context containers.

`BeanCounter` is a naive use of `BeanFactoryPostProcessor`. To find more meaningful examples of `BeanFactoryPostProcessor`, we have to look no further than the Spring Framework itself. You’ve already seen `CustomerEditorConfigurer` (see section 3.4), which is an implementation of `BeanFactoryPostProcessor` used to register custom `PropertyEditors` in Spring.

Another very useful `BeanFactoryPostProcessor` implementation is `PropertyPlaceholderConfigurer`. `PropertyPlaceholderConfigurer` loads properties from one or more external property files and uses those properties to fill in placeholder variables in the bean wiring XML file. Speaking of `PropertyPlaceholderConfigurer`, that’s what we’ll look at next.

3.5.3 **Externalizing configuration properties**

For the most part, it is possible to configure your entire application in a single bean-wiring file. But sometimes you may find it beneficial to extract certain pieces of that configuration into a separate property file. For example, a configuration concern that is common to many applications is configuring a data source. In

Spring, you could configure a data source with the following XML in the bean-wiring file:

```
<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="url" value="jdbc:hsqldb:Training" />
    <property name="driverClassName" value="org.hsqldb.jdbcDriver" />
    <property name="username" value="appUser" />
    <property name="password" value="password" />
</bean>
```

Configuring the data source directly in the bean-wiring file may not be appropriate. The database specifics are a deployment detail. Conversely, the purpose of the bean-wiring file is mainly oriented toward defining how components within your application are put together. That's not to say that you cannot configure your application components within the bean-wiring file. In fact, when the configuration is application specific (as opposed to deployment specific), it makes perfect sense to configure components in the bean-wiring file. But deployment details should be separated.

Fortunately, externalizing properties in Spring is easy if you are using an ApplicationContext as your Spring container. You use Spring's PropertyPlaceholderConfigurer to tell Spring to load certain configuration from an external property file. To enable this feature, configure the following bean in your bean-wiring file:

```
<bean id="propertyConfigurer"
    class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
    <property name="location" value="jdbc.properties" />
</bean>
```

The location property tells Spring where to find the property file. In this case, the jdbc.properties file contains the following JDBC information:

```
database.url=jdbc:hsqldb:Training
database.driver=org.hsqldb.jdbcDriver
database.user=appUser
database.password=password
```

The location property allows you to work with a single property file. If you want to break down your configuration into multiple property files, use PropertyPlaceholderConfigurer's locations property to set a List of property files:

```
<bean id="propertyConfigurer"
    class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
    <property name="locations" value="classpath:jdbc.properties" />
</bean>
```

```

    ↗ PropertyPlaceholderConfigurer">
<property name="locations">
  <list>
    <value>jdbc.properties</value>
    <value>security.properties</value>
    <value>application.properties</value>
  </list>
</property>
</bean>

```

Now you are able to replace the hard-coded configuration in the bean-wiring file with placeholder variables. Syntactically, the placeholder variables take the form \${variable}, resembling both Ant properties and the JavaServer Pages (JSP) expression language. After plugging in the placeholder variables, the new dataSource bean declaration looks like this:

```

<bean id="dataSource"
      class="org.springframework.jdbc.datasource.
        ↗ DriverManagerDataSource">
<property name="url"
          value="${database.url}" />
<property name="driverClassName"
          value="${database.driver}" />
<property name="username"
          value="${database.user}" />
<property name="password"
          value="${database.password}" />
</bean>

```

When Spring creates the dataSource bean, the `PropertyPlaceholderConfigurer` will step in and replace the placeholder variables with the values from the property file, as shown in figure 3.7.

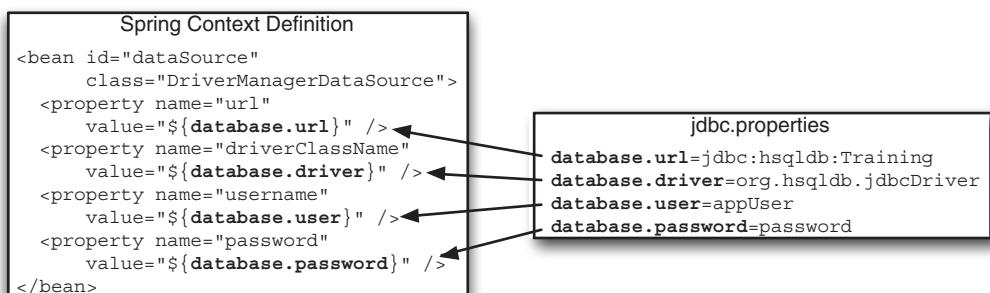


Figure 3.7 `PropertyPlaceholderConfigurer` enables externalizing configuration values into property files and then loading those values into placeholder variables in the Spring context definition.

`PropertyPlaceholderConfigurer` is useful for externalizing part of a Spring configuration into property files. But Java uses property files for more than just configuration; they're also commonly used to store text messages and for internationalization. Let's see how Spring message sources can be used to resolve text messages from a property file.

3.5.4 Resolving text messages

Often you may not want to hard-code certain text that will be displayed to the user of your application. This may be because the text is subject to change, or perhaps your application will be internationalized and you will display text in the user's native language.

Java's support for parameterization and internationalization (I18N) of messages enables you to define one or more property files that contain the text that is to be displayed in your application. There should always be a default message file along with optional language-specific message files. For example, if the name of your application's message bundle is "trainingtext," you may have the following set of message property files:

- *trainingtext.properties*—Default messages when a locale cannot be determined or when a locale-specific properties file is not available
- *trainingtext_en_US.properties*—Text for English-speaking users in the United States
- *trainingtext_es_MX.properties*—Text for Spanish-speaking users in Mexico
- *trainingtext_de_DE.properties*—Text for German-speaking users in Germany

For example, both the default and English property files may contain entries such as

```
course=class  
student=student  
computer=computer
```

Meanwhile, the Spanish message file would look like this:

```
course=clase  
student=estudiante  
computer=computadora
```

Spring's `ApplicationContext` supports parameterized messages by making them available to the container through the `MessageSource` interface:

```
public interface MessageSource {  
    String getMessage(
```

```

        MessageSourceResolvable resolvable, Locale locale)
        throws NoSuchMessageException;
    String getMessage(
        String code, Object[] args, Locale locale)
        throws NoSuchMessageException;
    String getMessage(
        String code, Object[] args, String defaultMessage,
        Locale locale);
}

```

Spring comes with a ready-to-use implementation of `MessageSource`. `ResourceBundleMessageSource` simply uses Java's own `java.util.ResourceBundle` to resolve messages. To use `ResourceBundleMessageSource`, add the following to the bean-wiring file:

```

<bean id="messageSource"
      class="org.springframework.context.support.ResourceBundleMessageSource">
    <property name="basename">
      <value>trainingtext</value>
    </property>
</bean>

```

It is very important that this bean be named `messageSource` because the `ApplicationContext` will look for a bean specifically by that name when setting up its internal message source. You'll never need to inject the `messageSource` bean into your application beans, but will instead access messages via `ApplicationContext`'s own `getMessage()` methods. For example, to retrieve the message whose name is `computer`, use this code:

```

Locale locale = ... ; //determine locale
String text =
    context.getMessage("computer", new Object[0], locale);

```

You'll likely be using parameterized messages in the context of a web application, displaying the text on a web page. In that case, you'll want to use Spring's `<spring:message>` JSP tag to retrieve messages and will not need to directly access the `ApplicationContext`:

```
<spring:message code="computer"/>
```

But if you need your beans, not a JSP, to retrieve the messages, how can you write them to access the `ApplicationContext`? Well, you're going to have to wait a bit for that. Or you can skip ahead to section 3.5.6, where I discuss making your beans aware of their container.

Right now, however, let's move on to examine the events that occur during an application context's lifecycle and how to handle these events to perform special

processing. You'll also see how to publish our own events to trigger behavior between otherwise decoupled beans.

3.5.5 Decoupling with application events

Dependency injection is the primary way that Spring promotes loose coupling among application objects—but it isn't the only way. Another way for objects to interact with one another is to publish and listen for application events. Using events, an event publisher object can communicate with other objects without even knowing which objects are listening. Likewise, an event listener can react to events without knowing which object published the events.

This event-driven interaction is analogous to a radio station and its audience of listeners, as illustrated in figure 3.8. The radios are not wired directly to the radio station, and the radio station has no idea what radios are tuning in. Nevertheless, the station is still able to communicate with its listeners in a completely decoupled fashion.

In Spring, any bean in the container can be either an event listener, an event publisher, or both. Let's see how to create beans that participate in events, starting with beans that publish application events.

Publishing events

Imagine that in a college's online enrollment system, you want to alert one or more application objects any time that a student signs up for a course and, as a result, the course is full. Maybe you want to fire off a routine to automatically schedule another course to handle the overflow.

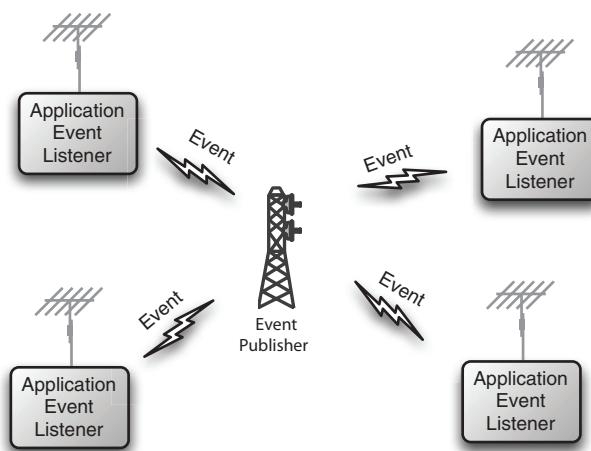


Figure 3.8
An event publisher is like a radio station that broadcasts application events to its listeners. The publisher and its listeners are completely decoupled from each other.

First, define a custom event, such as the following `CourseFullEvent`:

```
public class CourseFullEvent extends ApplicationEvent {  
    private Course course;  
  
    public CourseFullEvent(Object source, Course course) {  
        super(source);  
        this.course = course;  
    }  
  
    public Course getCourse() {  
        return course;  
    }  
}
```

Next, you'll need to publish the event. The `ApplicationContext` interface has a `publishEvent()` method that enables you to publish `ApplicationEvents`. Any `ApplicationListener` that is registered in the application context will receive the event in a call to its `onApplicationEvent()` method:

```
ApplicationContext context = ...;  
Course course = ...;  
context.publishEvent(new CourseFullEvent(this, course));
```

Unfortunately, in order to publish events, your beans will need to have access to the `ApplicationContext`. This means that beans will have to be made aware of the container that they're running in. You'll see how to make beans aware of their container in section 3.5.6.

But first, if an event publisher publishes an event and nobody listens to it, did the event really happen? I'll leave the philosophical questions for you to ponder later. For now, let's make sure that no events go unheard by creating beans that listen for events.

Listening for events

In addition to events that are published by other beans, the Spring container itself publishes a handful of events during the course of an application's lifetime. These events are all subclasses of the abstract class `org.springframework.context.ApplicationEvent`. Here are three such application events:

- `ContextClosedEvent`—Published when the application context is closed
- `ContextRefreshedEvent`—Published when the application context is initialized or refreshed
- `RequestHandledEvent`—Published within a web application context when a request is handled

For the most part, these events are published rather... uh... well, uneventfully. Most beans will never know or care that they were published. But what if you want to be notified of application events?

If you want a bean to respond to application events, whether published by another bean or by the container, all you need to do is implement the `org.springframework.context.ApplicationListener` interface. This interface forces your bean to implement the `onApplicationEvent()` method, which is responsible for reacting to the application event:

```
public class RefreshListener implements ApplicationListener {  
    public void onApplicationEvent(ApplicationEvent event) {  
        ...  
    }  
}
```

The only thing you need to do to tell Spring about an application event listener is to simply register it as a bean within the context:

```
<bean id="refreshListener"  
      class="com.springinaction.foo.RefreshListener"/>
```

When the container loads the bean within the application context, it will notice that it implements `ApplicationListener` and will remember to call its `onApplicationEvent()` method when an event is published.

One thing to keep in mind is that application events are handled synchronously. So, you want to take care that any events handled in this fashion are handled quickly. Otherwise, your application's performance could be negatively impacted.

As mentioned before, a bean must be aware of the application container to be able to publish events. Even if an object isn't interested in publishing events, you may want to develop a bean that knows about the application context that it lives in. Next up, let's see how to create beans that are injected with references to their container.

3.5.6 Making beans aware

Have you seen *The Matrix*? In the movie, humans have been unwittingly enslaved by machines, living their everyday lives in a virtual world while their life essence is being farmed to power the machines. Thomas Anderson, the main character, is given a choice between taking a red pill and learning the truth of his existence or taking a blue pill and continuing his life ignorant of the truth. He chooses the red pill, becoming aware of his real-world identity and the truth about the virtual world.

For the most part, beans running in the Spring container are like the humans in *The Matrix*. For these beans, ignorance is bliss. They don't know (or even need to know) their names or even that they are running within a Spring container. This is usually a good thing because if a bean is aware of the container, it becomes coupled with Spring and may not be able to exist outside the container.

But sometimes, beans need to know more. Sometimes they need to know who they are and where they are running. Sometimes they need to take the red pill.

The red pill, in the case of Spring beans, comes in the form of the `BeanNameAware`, `BeanFactoryAware`, and `ApplicationContextAware` interfaces. By implementing these three interfaces, beans can be made aware of their name, their `BeanFactory`, and their `ApplicationContext`, respectively.

Be warned, however, that by implementing these interfaces, a bean becomes coupled with Spring. And, depending on how your bean uses this knowledge, you may not be able to use it outside Spring.

Knowing who you are

The Spring container tells a bean what its name is through the `BeanNameAware` interface. This interface has a single `setBeanName()` method that takes a `String` containing the bean's name, which is set through either the `id` or the `name` attribute of `<bean>` in the bean-wiring file:

```
public interface BeanNameAware {  
    void setBeanName(String name);  
}
```

It may be useful for a bean to know its name for bookkeeping purposes. For example, if a bean can have more than one instance within the application context, it may be beneficial for that bean to identify itself by both name and type when logging its actions.

Within the Spring Framework itself, `BeanNameAware` is used several times. One notable use is with beans that perform scheduling. `CronTriggerBean`, for example, implements `BeanNameAware` to set the name of its Quartz `CronTrigger` job. The following code snippet from `CronTriggerBean` illustrates this:

```
package org.springframework.scheduling.quartz;  
public class CronTriggerBean extends CronTrigger  
    implements ..., BeanNameAware, ... {  
    ...  
    private String beanName;  
    ...  
    public void setBeanName(String beanName) {  
        this.beanName = beanName;  
    }  
}
```

```
...
    public void afterPropertiesSet() ... {
        if (getName() == null){
            setBeanName(this.beanName);
        }
    ...
}
```

You don't need to do anything special for a Spring container to call `setBeanName()` on a `BeanNameAware` class. When the bean is loaded, the container will see that the bean implements `BeanNameAware` and will automatically call `setBeanName()`, passing the name of the bean as defined by either the `id` or the `name` attribute of the `<bean>` element in the bean-wiring XML file.

Here `CronTriggerBean` extends `CronTrigger`. After the Spring context has set all properties on the bean, the bean name is sent to `setBeanName()` (defined in `CronTriggerBean`), which is used to set the name of the scheduled job.

This example illustrated how to use `BeanNameAware` by showing how it is used in Spring's own scheduling support. I'll talk more about scheduling in chapter 12. For now, let's see how to make a bean aware of the Spring container that it lives within.

Knowing where you live

As you've seen in this section, sometimes it's helpful for a bean to be able to access the application context. Perhaps your bean needs access to parameterized text messages in a message source. Or maybe it needs to be able to publish application events for application event listeners to respond to. Whatever the case, your bean should be aware of the container in which it lives.

Spring's `ApplicationContextAware` and `BeanFactoryAware` interfaces enable a bean to be aware of its container. These interfaces declare a `setApplicationContext()` method and a `setBeanFactory()` method, respectively. The Spring container will detect whether any of your beans implement either of these interfaces and provide the `BeanFactory` or `ApplicationContext`.

With access to the `ApplicationContext`, the bean can actively interact with the container. This can be useful for programmatically retrieving dependencies from the container (when they're not injected) or publishing application events. Going back to our event-publishing example earlier, we would finish that example like this:

```
public class StudentServiceImpl
    implements StudentService, ApplicationContextAware {
```

```
private ApplicationContext context;
public void setApplicationContext(ApplicationContext context) {
    this.context = context;
}
public void enrollStudentInCourse(Course course, Student student)
    throws CourseException {
    ...
    context.publishEvent(new CourseFullEvent(this, course));
    ...
}
...
}
```

Being aware of the application container is both a blessing and a curse for a bean. On the one hand, access to the application context affords the bean a lot of power. On the other hand, being aware of the container couples the bean to Spring and is something that should be avoided if possible.

So far we've been assuming that the beans in the Spring container are all implemented as Java classes. That's a reasonable assumption, but it doesn't necessarily have to be the case. Let's see how to add dynamic behavior to an application by wiring beans that are implemented using a scripting language.

3.6 Scripting beans

When writing the Java code that will become the beans in your Spring application, you eventually run that code through a compiler that compiles it into the bytecode that the JVM will execute. Moreover, you will likely package that compiled code into a JAR, WAR, or EAR file for deployment. But what if, after the application is deployed, you want to change the behavior of the code?

You see, the problem with statically compiled code is that... well... it's static. Once it's compiled into a class file and packaged in a deployment archive, it's difficult to change without recompiling, repackaging, and redeploying the entire application. In many circumstances, that is acceptable (and perhaps even desired). Even so, statically compiled code makes it difficult to respond quickly to dynamic business needs.

For example, suppose that you've developed an e-commerce application. Within that application is code that calculates the subtotal and total purchase based on the items in the cart and any applicable shipping charges and sales tax. But what if you need the ability to waive the sales tax for one day?

If the tax calculation portion of your application is statically compiled to the totaling code, the only option you'll have is to deploy a tax-free version of your

application at midnight, then redeploy the original version at the next midnight. You had better start brewing the coffee, because you're going to be up late two nights in a row.

With Spring, it's possible to write scripted code in either Ruby, Groovy, or BeanShell and wire it into a Spring application context as if it were any other Java-based bean, as illustrated in figure 3.9. In the next few subsections, I'll demonstrate how to wire Ruby, Groovy, and BeanShell scripts as Spring-managed beans.

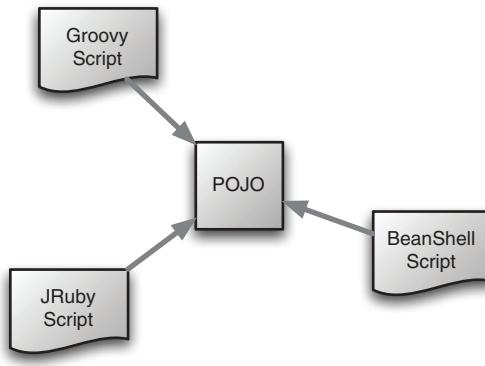


Figure 3.9
Spring isn't limited to only injecting POJOs into POJOs. You can also dynamically modify your application by injecting scripted beans into POJOs.

If you're fan of Calypso music then you're in for a treat—I'm going to demonstrate scripted beans to the tune of one of the genre's most famous songs. Please follow along as I demonstrate dynamically scripted beans by putting a scripted lime in a Java coconut.

3.6.1 Putting the lime in the coconut

To illustrate how to script beans in Spring, let's inject a scripted implementation of a Lime interface into a Java Coconut object. To start, let's look at the Coconut class as defined in listing 3.6.

Listing 3.6 A Java in a coconut shell

```
package com.springinaction.scripting;

public class Coconut {
    public Coconut() {}

    public void drinkThemBothUp() {
        System.out.println("You put the lime in the coconut...");
        System.out.println("and drink 'em both up...");
        System.out.println("You put the lime in the coconut...");
    }
}
```

```

lime.drink();    <-- Invokes the Lime's
}               drink() method

// injected
private Lime lime;
public void setLime(Lime lime) {
    this.lime = lime;
}
}

```

Injects the Lime

The Coconut class has one simple method, called `drinkThemBothUp()`. When this method is invoked, certain lyrics from Mr. Nilsson's song are printed to the `System.out`. The last line of the method invokes a `drink()` method on an injected `Lime` instance to print the final lyric. The injected `Lime` is any object that implements the following interface:

```

package com.springinaction.scripting;

public interface Lime {
    void drink();
}

```

When wired up in Spring, the `Coconut` class is injected with a reference to a `Lime` object using the following XML:

```

<bean id="coconut" class="com.springinaction.scripting.Coconut">
    <property name="lime" ref="lime" />
</bean>

```

At this point, I've shown you nothing special about the `Coconut` class or the `Lime` interface that hints to scripted beans. For the most part, the code presented up to this point resembles the basic JavaBean and DI examples from chapter 2.

The only thing missing is the exact implementation of the `Lime` interface and its declaration in the Spring context. The fact is that any Java-based implementation of the `Lime` interface will do. But I promised you a scripted `Lime` and so a scripted `Lime` is what I'll deliver next.

3.6.2 Scripting a bean

When scripting the `Lime` interface, we can choose to implement it as either a Ruby, Groovy, or BeanShell script. But regardless of which scripting language is chosen, we first need to do some setup in the Spring context definition file. Consider the following `<beans>` declaration to see what needs to be done:

```

<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

```

```
xmlns:lang="http://www.springframework.org/schema/lang"
xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
    http://www.springframework.org/schema/lang
    http://www.springframework.org/schema/lang/
    ↗spring-lang-2.0.xsd">
...
</beans>
```

Spring 2.0 comes with several new configuration elements, each defined in an XML namespace and schema. We'll see a lot more of Spring's custom configuration namespaces throughout this book. But for now, suffice it to say that the highlighted portions of this `<beans>` declaration tell Spring that we're going to use certain configuration elements from the `lang` namespace.

Now that the namespace has been declared in the Spring context file, we're ready to begin scripting our `Lime`. Let's start with a Ruby-colored `Lime`.

Scripting the Lime in Ruby

In recent years, Ruby has caught the attention of many Java developers, so it'd be no surprise if you'd like to write your scripted beans using this very popular scripting language. The following Ruby script implements the `Lime` interface and its `drink()` method:

```
class Lime
  def drink
    puts "Called the doctor woke him up!"
  end
end
Lime.new
```

A very important thing to note here is that the last line of the script instantiates a new `Lime` object. This line is crucial—without it, there will be no instance of the `Lime` created that can be wired into other Spring objects.

Wiring the Ruby `Lime` in Spring is a simple matter of using the `<lang:jruby>` configuration element as follows:

```
<lang:jruby id="lime"
  script-source="classpath:com/springinaction/scripting/Lime.rb"
  script-interfaces="com.springinaction.scripting.Lime" />
```

`<lang:jruby>` requires two attributes to be set. The first, `script-source`, tells Spring where it can locate the script file. Here, the script file is `Lime.rb` and can be found in the classpath in the same package as the rest of the example code. Meanwhile, the `script-interfaces` attribute tells Spring what Java interface that the script will be implementing. Here it has been set to our `Lime` interface.

Scripting a Groovy Lime

Despite Ruby's growing popularity, many developers will be targeting the Java platform for some time to come. Groovy is a language that mixes some of the best features of Ruby and other scripting languages into a familiar Java syntax—effectively giving a best-of-both-worlds option for Java developers.

For those of you who favor Groovy scripting, here's an implementation of the Lime interface as a Groovy class:

```
class Lime implements com.springinaction.scripting.Lime {  
    void drink() {  
        print "Called the doctor woke him up!"  
    }  
}
```

Wiring a Groovy script as a Spring bean is as simple as using the `<lang:groovy>` configuration element. The following `<lang:groovy>` configuration loads the Groovy implementation of the Lime interface:

```
<lang:groovy id="lime"  
    script-source="classpath:com/springinaction/  
        scripting/Lime.groovy" />
```

As with the `<lang:jruby>` element, the `script-source` attribute specifies the location of the Groovy script file. Again, we're locating the script file in the class-path in the same package as the example code.

Unlike the `<lang:jruby>` element, however, `<lang:groovy>` doesn't require (or even support) a `script-interfaces` attribute. That's because there's enough information in the Groovy script itself to indicate what interfaces the script implements. Notice that the Groovy Lime class explicitly implements `com.springinaction.scripting.Lime`.

Writing the Lime in BeanShell

Another scripting language supported in Spring is BeanShell. Unlike Ruby and Groovy, which both provide their own syntax, BeanShell is a scripting language that mimics Java's own syntax. This makes it an appealing option if you want to script portions of your application but do not want to have to learn another language.

Completing our tour of scripting languages that can be wired in Spring, here's a BeanShell implementation of the Lime interface:

```
void drink() {  
    System.out.println("Called the doctor woke him up!");  
}
```

Probably the first thing you noticed about this BeanShell implementation of Lime is that there isn't a class definition—only a drink() method is defined. In BeanShell scripts, you only define the methods required by the interface, but no class.

Wiring the BeanShell lime is quite similar to wiring the Ruby lime, except that you use the <lang:bsh> element as follows:

```
<lang:bsh id="lime"
  script-source="classpath:com/springinaction/scripting/Lime.bsh"
  script-interfaces="com.springinaction.scripting.Lime" />
```

As with all of the scripting elements, script-source indicates the location of the script file. And, as with the <lang:jruby> element, script-interfaces specifies the interface being defined in the script.

Now you've seen how to configure a scripted bean in Spring and how to wire it into a property of a POJO. But what if you want the injection to work the other way? Let's see how to inject a POJO into a scripted bean.

3.6.3 Injecting properties of scripted beans

To illustrate how to inject properties of a scripted bean, let's flip our lime-coconut example on its head. This time, the coconut will be a scripted bean and the lime will be a Java-based POJO. First up, here's the Lime class in Java:

```
package com.springinaction.scripting;

public class LimeImpl implements Lime {
    public LimeImpl() {}

    public void drink() {
        System.out.println("Called the doctor woke him up!");
    }
}
```

LimeImpl is just a simple Java class that implements the Lime interface. And here it is configured as a bean in Spring:

```
<bean id="lime" class="com.springinaction.scripting.LimeImpl" />
```

Nothing special so far. Now let's write the Coconut class as a script in Groovy:

```
class Coconut implements com.springinaction.scripting.ICoconut {
    public void drinkThemBothUp() {
        println "You put the lime in the coconut..."
        println "and drink 'em both up..."
        println "You put the lime in the coconut..."
        lime.drink()
    }
}
```

```
com.springinaction.scripting.Lime lime;  
}
```

As with the Java version of Coconut, a few lyrics are printed and then the `drink()` method is called on the `lime` property to finish the verse. Here the `lime` property is defined as being some implementation of the `Lime` interface.

Now all that's left is to configure the scripted Coconut bean and inject it with the `lime` bean:

```
<lang:groovy id="coconut"  
    script-source="classpath:com/springinaction/scripting/  
        ➔ Coconut.groovy">  
    <lang:property name="lime" ref="lime" />  
</lang:groovy>
```

Here the scripted `Coconut` has been declared similar to how we declared the scripted `Lime` in previous sections. But along with the `<lang:groovy>` element is a `<lang:property>` element to help us with dependency injection.

The `<lang:property>` element is available for use with all of the scripted bean elements. It is virtually identical in use to the `<property>` element that you learned about in chapter 2, except that its purpose is to inject values into the properties of scripted beans instead of into properties of POJO beans.

In this case, the `<lang:property>` element is the `lime` property of the `coconut` bean with a reference to the `lime` bean—which in this case is a JavaBean. You may find it interesting, however, that you can also wire scripted beans into the properties of other scripted beans. In fact, it's quite possible to wire a BeanShell-scripted bean into a Groovy-scripted bean, which is then wired into a Ruby-scripted bean. Following that thought to an extreme, it's theoretically possible to develop an entire Spring application using scripted languages!

3.6.4 Refreshing scripted beans

One of the key benefits of scripting certain code instead of writing it in statically compiled Java is that it can be changed on the fly without a recompile or redeployment of the application. If the `Lime` implementation were to be written in Java and you decided to change the lyric that it prints, you'd have to recompile the implementation class and then redeploy the application. But with a scripting language, you can change the implementation at any time and have the change applied almost immediately.

When I say “almost immediately,” that really depends on how often you'd like Spring to check for changes to the script. All of the scripting configuration ele-

ments have a `refresh-check-delay` attribute that allows you to define how often (in milliseconds) a script is refreshed by Spring.

By default, `refresh-check-delay` is set to `-1`, meaning that refreshing is disabled. But suppose that you'd like the Lime script refreshed every 5 seconds. The following `<lang:jruby>` configuration will do just that:

```
<lang:jruby id="lime"
    script-source="classpath:com/springinaction/scripting/Lime.rb"
    script-interfaces="com.springinaction.scripting.Lime"
    refresh-check-delay="5000"/>
```

It should be pointed out that although this example is for `<lang:jruby>`, the `refresh-check-delay` attribute works equally well with `<lang:groovy>` and `<lang:bsh>`.

3.6.5 Writing scripted beans inline

Typically you'll define your scripted beans in external scripting files and refer to them using the `script-source` attribute of the scripting configuration elements. However, in some cases, it may be more convenient to write the scripting code directly in the Spring configuration file.

To accommodate this, all of the scripting configuration elements support a `<lang:inline-script>` element as a child element. For example, the following XML defines a BeanShell-scripted Lime directly in the Spring configuration:

```
<lang:bsh id="lime"
    script-interfaces="com.springinaction.scripting.Lime">
<lang:inline-script><! [CDATA[
    void drink() {
        System.out.println("Called the doctor woke him up!");
    }
]]>
</lang:inline-script>
</lang:bsh>
```

Instead of using `script-source`, this lime bean has the BeanShell code written as the content of a `<lang:inline-script>` element.

Take note of the use of `<! [CDATA[...]]>` when writing the inline script. The script code may contain characters or text that may be misinterpreted as XML. The `<! [CDATA[...]]>` construct prevents scripted code from being parsed by the XML parser. In this example the script contains nothing that would confuse the XML parser. Nevertheless it's a good idea to use `<! [CDATA[...]]>` anyway, just in case the scripted code changes.

In this section, you were exposed to several Spring configuration elements that step beyond the `<bean>` and `<property>` elements that you were introduced to in chapter 2. These are just a few of the new configuration elements that were introduced in Spring 2.0. As you progress through this book, you'll be introduced to even more configuration elements.

3.7 **Summary**

Spring's primary function is to wire application objects together. While chapter 2 showed how to do the basic day-to-day wiring, this chapter showed the more odd-ball wiring techniques.

To reduce the amount of repeated XML that defines similar beans, Spring offers the ability to declare abstract beans that describe common properties and then "sub-bean" the abstract bean to create the actual bean definitions.

Perhaps one of the oddest features of Spring is the ability to alter a bean's functionality through method injection. Using method injection, you can swap out a bean's existing functionality for a replaced method definition. Alternatively, using setter injection, you can replace a getter method with a Spring-defined getter method that returns a specific bean reference.

Not all objects in an application are created or managed by Spring. To enable DI for those objects that Spring doesn't create, Spring provides a means to declare objects as "Spring configured." Spring-configured beans are intercepted by Spring, after they are created, and configured based on a Spring bean template.

Spring takes advantage of property editors so that even complex objects such as URLs and arrays can be configured using String values. In this chapter you saw how to create custom property editors to simplify configuration of complex properties.

Sometimes it is necessary for a bean to interact with the Spring container. For those circumstances, Spring provides several interfaces that enable a bean to be postprocessed, receive properties from an external configuration file, handle application events, and even know their own name.

Finally, for the fans of dynamically scripted languages, Spring lets you write beans in scripted languages such as Ruby, Groovy, and BeanShell. This feature supports dynamic behavior in an application by making it possible to hot-swap bean definitions written in one of three different scripting languages.

Now you know how to wire together objects in the Spring container and have seen how DI helps to loosen the coupling between application objects. But DI is only one of the ways that Spring supports loose coupling. In the next chapter, I'll look at how Spring's AOP features help break out common functionality from the application objects that it affects.

Advising beans



This chapter covers

- Basics of aspect-oriented programming
- Creating aspects from POJOs
- Automatically proxying beans
- Using @AspectJ annotations
- Injecting dependencies into AspectJ aspects

As I'm writing this chapter, Texas (where I reside) is going through several days of record-high temperatures. It's really hot. In weather like this, air-conditioning is a must. But the downside of air-conditioning is that it uses electricity and electricity costs money. And there's very little we can do to avoid paying for a cool and comfortable home. That's because every home has a meter that measures every single kilowatt, and once a month someone comes by to read that meter so that the electric company accurately knows how much to bill us.

Now imagine what would happen if the meter went away and nobody came by to measure our electricity usage. Suppose that it were up to each homeowner to contact the electric company and report their electricity usage. Although it's possible that some obsessive homeowners would keep careful record of their lights, televisions, and air-conditioning, most wouldn't bother. Most would estimate their usage and others wouldn't bother reporting it at all. It's too much trouble to monitor electrical usage and the temptation to not pay is too great.

Electricity on the honor system might be great for consumers, but it would be less than ideal for the electric companies. That's why we all have electric meters on our homes and why a meter-reader drops by once per month to report the consumption to the electric company.

Some functions of software systems are like the electric meters on our homes. The functions need to be applied at multiple points within the application, but it's undesirable to explicitly call them at every point.

Monitoring electricity consumption is an important function, but it isn't foremost in most homeowners' minds. Mowing the lawn, vacuuming the carpet, and cleaning the bathroom are the kinds of things that homeowners are actively involved in. Monitoring the amount of electricity used by their house is a passive event from the homeowner's point of view.

In software, several activities are common to most applications. Logging, security, and transaction management are important things to do, but should they be activities that your application objects are actively participating in? Or would it be better for your application objects to focus on the business domain problems they're designed for and leave certain aspects to be handled by someone else?

In software development, functions that span multiple points of an application are called *cross-cutting concerns*. Typically, these cross-cutting concerns are conceptually separate from (but often embedded directly within) the application's business logic. Separating these cross-cutting concerns from the business logic is where aspect-oriented programming (AOP) goes to work.

In chapter 2, you learned how to use dependency injection (DI) to manage and configure your application objects. Whereas DI helps you decouple your

application objects from each other, AOP helps you decouple cross-cutting concerns from the objects that they affect.

Logging is a common example of the application of aspects. But it isn't the only thing aspects are good for. Throughout this book, you'll see several practical applications of aspects, including declarative transactions, security, and caching.

This chapter explores Spring's support for aspects, including the exciting new AOP features added in Spring 2.0. In addition, you'll see how AspectJ—another popular AOP implementation—can complement Spring's AOP framework. But first, before we get too carried away with transactions, security, and caching, let's see how aspects are implemented in Spring, starting with a primer on a few of AOP's fundamentals.

4.1 Introducing AOP

As stated earlier, aspects help to modularize cross-cutting concerns. In short, a cross-cutting concern can be described as any functionality that affects multiple points of an application. Security, for example, is a cross-cutting concern in that many methods in an application can have security rules applied to them. Figure 4.1 gives a visual depiction of cross-cutting concerns.

Figure 4.1 represents a typical application that is broken down into modules. Each module's main concern is to provide services for its particular domain. However, each of these modules also requires similar ancillary functionalities, such as security and transaction management.

A common object-oriented technique for reusing common functionality is to apply inheritance or delegation. But inheritance can lead to a brittle object hierarchy if the same base class is used throughout an application, and delegation can be cumbersome because complicated calls to the delegate object may be required.

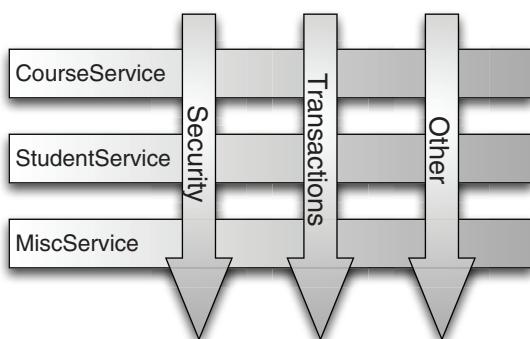


Figure 4.1
Aspects modularize cross-cutting concerns, applying logic that spans multiple application objects.

Aspects offer an alternative to inheritance and delegation that can be cleaner in many circumstances. With AOP, you still define the common functionality in one place, but you can declaratively define how and where this functionality is applied without having to modify the class to which you are applying the new feature. Cross-cutting concerns can now be modularized into special objects called *aspects*. This has two benefits. First, the logic for each concern is now in one place, as opposed to being scattered all over the code base. Second, our service modules are now cleaner since they only contain code for their primary concern (or core functionality) and secondary concerns have been moved to aspects.

4.1.1 Defining AOP terminology

Like most technologies, AOP has formed its own jargon. Aspects are often described in terms of advice, pointcuts, and joinpoints. Figure 4.2 illustrates how these concepts are tied together.

Unfortunately, many of the terms used to describe AOP features are not intuitive. Nevertheless, they are now part of the AOP idiom, and in order to understand AOP, you must know these terms. In other words, before you walk the walk, you have to learn to talk the talk.

Advice

When a meter-reader shows up at your house, their purpose is to report the number of kilowatt-hours back to the electric company. Sure, they have a list of houses that they must visit and the information that they report is important. But the actual act of recording electricity usage is the meter-reader's main job.

Likewise, aspects have a purpose—a job that they are meant to do. In AOP terms, the job of an aspect is called *advice*.

Advice defines both the *what* and the *when* of an aspect. In addition to describing the job that an aspect will perform, advice addresses the question of when to

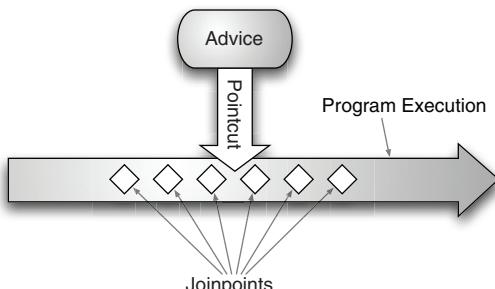


Figure 4.2
An aspect's functionality (advice) is woven into a program's execution at one or more joinpoints.

perform the job. Should it be applied before a method is invoked? After the method is invoked? Both before and after method invocation? Or should it only be applied if a method throws an exception?

Joinpoint

An electric company services several houses, perhaps even an entire city. Each house will have an electric meter that needs to be read and thus each house is a potential target for the meter-reader. The meter-reader could potentially read all kinds of devices, but to do his job, he needs to target electric meters that are attached to houses.

In the same way, your application may have thousands of opportunities for advice to be applied. These opportunities are known as joinpoints. A *joinpoint* is a point in the execution of the application where an aspect can be plugged in. This point could be a method being called, an exception being thrown, or even a field being modified. These are the points where your aspect's code can be inserted into the normal flow of your application to add new behavior.

Pointcut

It's not possible for any one meter-reader to visit all houses serviced by the electric company. Instead, they are assigned a subset of all of the houses to visit. Likewise, an aspect doesn't necessarily advise all joinpoints in an application. Pointcuts help narrow down the joinpoints advised by an aspect.

If advice defines the *what* and *when* of aspects then pointcuts define the *where*. A pointcut definition matches one or more joinpoints at which advice should be woven. Often you specify these pointcuts using explicit class and method names or through regular expressions that define matching class and method name patterns. Some AOP frameworks allow you to create dynamic pointcuts that determine whether to apply advice based on runtime decisions, such as the value of method parameters.

Aspect

When a meter-reader starts his day, he knows both what he is supposed to do (report electricity usage) and which houses to collect that information from. Thus he knows everything he needs to know to get his job done.

An *aspect* is the merger of advice and pointcuts. Taken together, advice and pointcuts define everything there is to know about an aspect—what it does and where and when it does it.

Introduction

An *introduction* allows you to add new methods or attributes to existing classes (kind of mind-blowing, huh?). For example, you could create an `Auditatable` advice class that keeps the state of when an object was last modified. This could be as simple as having one method, `setLastModified(Date)`, and an instance variable to hold this state. The new method and instance variable can then be introduced to existing classes without having to change them, giving them new behavior and state.

Target

A *target* is the object that is being advised. This can be either an object you write or a third-party object to which you want to add custom behavior. Without AOP, this object would have to contain its primary logic plus the logic for any cross-cutting concerns. With AOP, the target object is free to focus on its primary concern, oblivious to any advice being applied.

Proxy

A *proxy* is the object created after applying advice to the target object. As far as the client objects are concerned, the target object (pre-AOP) and the proxy object (post-AOP) are the same—as they should be. That is, the rest of your application will not have to change to support the proxy object.

Weaving

Weaving is the process of applying aspects to a target object to create a new, proxied object. The aspects are woven into the target object at the specified joinpoints. The weaving can take place at several points in the target object’s lifetime:

- *Compile time*—Aspects are woven in when the target class is compiled. This requires a special compiler. AspectJ’s weaving compiler weaves aspects this way.
- *Classload time*—Aspects are woven in when the target class is loaded into the JVM. This requires a special `ClassLoader` that enhances that target class’s bytecode before the class is introduced into the application. AspectJ 5’s load-time weaving (LTW) support weaves aspects in this way.
- *Runtime*—Aspects are woven in sometime during the execution of the application. Typically, an AOP container will dynamically generate a proxy object that will delegate to the target object while weaving in the aspects. This is how Spring AOP aspects are woven.

That's a lot of new terms to get to know. Revisiting figure 4.2, you can now understand that advice contains the cross-cutting behavior that needs to be applied to an application's objects. The joinpoints are all the points within the execution flow of the application that are candidates to have advice applied. The pointcut defines where (at what joinpoints) that advice is applied. The key concept you should take from this? Pointcuts define which joinpoints get advised.

Now that you're familiar with some basic AOP terminology, let's see how these core AOP concepts are implemented in Spring.

4.1.2 **Spring's AOP support**

Not all AOP frameworks are created equal. They may differ in how rich of a joinpoint model they offer. Some allow you to apply advice at the field modification level, while others only expose the joinpoints related to method invocations. They may also differ in how and when they weave the aspects. Whatever the case, the ability to create pointcuts that define the joinpoints at which aspects should be woven is what makes it an AOP framework.

Much has changed in the AOP framework landscape in the past few years. There has been some housecleaning among the AOP frameworks, resulting in some frameworks merging and others going extinct. In 2005, the AspectWerkz project merged with AspectJ, marking the last significant activity in the AOP world and leaving us with three dominant AOP frameworks:

- AspectJ (<http://eclipse.org/aspectj>)
- JBoss AOP (<http://labs.jboss.com/portal/jbossaop/index.html>)
- Spring AOP (<http://www.springframework.org>)

Since this is a Spring book, we will, of course, focus on Spring AOP. Even so, there's a lot of synergy between the Spring and AspectJ projects, and the AOP support in Spring 2.0 borrows a lot from the AspectJ project. In fact, the `<aop:spring-configured />` configuration element described in chapter 3 (see section 3.3) takes advantage of AspectJ's support for constructor pointcuts and load-time weaving.

Spring's support for AOP comes in four flavors:

- Classic Spring proxy-based AOP (available in all versions of Spring)
- `@AspectJ` annotation-driven aspects (only available in Spring 2.0)
- Pure-POJO aspects (only available in Spring 2.0)
- Injected AspectJ aspects (available in all versions of Spring)

The first three items are all variations on Spring's proxy-based AOP. Consequently, Spring's AOP support is limited to method interception. If your AOP needs exceed simple method interception (constructor or property interception, for example), you'll want to consider implementing aspects in AspectJ, perhaps taking advantage of Spring DI to inject Spring beans into AspectJ aspects.

I'll talk about AspectJ and how it fits into Spring a little later in this chapter (in sections 4.3.2 and 4.5). Because Spring's AOP support is proxy based, that will be the focus of most of this chapter. But before we get started, it's important to understand a few key points of Spring's AOP framework.

Spring advice is written in Java

All of the advice you create within Spring will be written in a standard Java class. That way, you will get the benefit of developing your aspects in the same integrated development environment (IDE) you would use for your normal Java development. What's more, the pointcuts that define where advice should be applied are typically written in XML in your Spring configuration file. This means both the aspect's code and configuration syntax will be familiar to Java developers.

Contrast this with AspectJ, which is implemented as a language extension to Java. There are benefits and drawbacks to this approach. By having an AOP-specific language, you get more power and fine-grained control, as well as a richer AOP toolset. However, you are required to learn a new tool and syntax to accomplish this.

Spring advises objects at runtime

In Spring, aspects are woven into Spring-managed beans at runtime by wrapping them with a proxy class. As illustrated in figure 4.3, the proxy class poses as the target bean, intercepting advised method calls and forwarding those calls to the target bean.

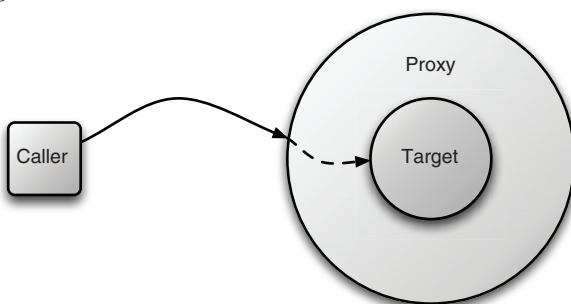


Figure 4.3
Spring aspects are implemented as proxies that wrap the target object. The proxy handles method calls, performs additional aspect logic, and then invokes the target method.

Between the time that the proxy intercepts the method call and the time it invokes the target bean's method, the proxy performs the aspect logic.

Spring does not create a proxied object until that proxied bean is needed by the application. If you are using an `ApplicationContext`, the proxied objects will be created when it loads all of the beans from the `BeanFactory`. Because Spring creates proxies at runtime, you do not need a special compiler to weave aspects in Spring's AOP.

Spring generates proxied classes in two ways. If your target object implements an interface(s) that exposes the required methods, Spring will use the JDK's `java.lang.reflect.Proxy` class. This class allows Spring to dynamically generate a new class that implements the necessary interfaces, weave in any advice, and proxy any calls to these interfaces to your target class.

If your target class does not implement an interface, Spring uses the CGLIB library to generate a subclass to your target class. When creating this subclass, Spring weaves in advice and delegates calls to the subclass to your target class. There are two important things to take note of when using this approach:

- Creating a proxy with interfaces is favored over proxying classes, since this leads to a more loosely coupled application. The ability to proxy classes is provided so that legacy or third-party classes that do not implement interfaces can still be advised. This approach should be taken as the exception, not the rule.
- Methods marked as `final` cannot be advised. Remember, Spring generates a subclass to your target class. Any method that needs to be advised is overwritten and advice is woven in. This is not possible with `final` methods.

Spring only supports method joinpoints

As mentioned earlier, multiple joinpoint models are available through various AOP implementations. Because it is based on dynamic proxies, Spring only supports method joinpoints. This is in contrast to some other AOP frameworks, such as AspectJ and JBoss, which provide field and constructor joinpoints in addition to method pointcuts. Spring's lack of field pointcuts prevents you from creating very fine-grained advice, such as intercepting updates to an object's field. And without constructor pointcuts, there's no way to apply advice when a bean is instantiated.

However, as Spring focuses on providing a framework for implementing J2EE services, method interception should suit most, if not all, of your needs. If you find yourself in need of more than method interception, you'll want to complement Spring AOP with AspectJ.

Now you have a general idea of what AOP does and how it is supported by Spring. It's time to get our hands dirty creating aspects in Spring.

4.2 Creating classic Spring aspects

In chapter 2, we demonstrated dependency injection by putting on a talent show called *Spring Idol*. In that example, we wired up several performers as <bean>s to show their stuff. It was all greatly amusing. But a show like that needs an audience or else there's little point in it.

Therefore, we're now going to provide an audience for the talent show. The Audience class in listing 4.1 defines the functions of an audience.

Listing 4.1 Defining an audience for the *Spring Idol* competition

```
package com.springinaction.springidol;

public class Audience {
    public Audience() {}

    public void takeSeats() {
        System.out.println("The audience is taking their seats.");
    }

    public void turnOffCellPhones() {
        System.out.println("The audience is turning off " +
            "their cellphones");
    }

    public void applaud() {
        System.out.println("CLAP CLAP CLAP CLAP CLAP");
    }

    public void demandRefund() {
        System.out.println("Boo! We want our money back!");
    }
}
```

Executes before performance

Executes after performance

Executes after bad performance

There's nothing particularly special about the Audience class. In fact, it's just a simple POJO. Nonetheless, this class will provide the basis for many of the examples in this chapter. It can be wired as Spring <bean> with the following XML:

```
<bean id="audience"
      class="com.springinaction.springidol.Audience" />
```

Taking a closer look at the Audience class, you can see that it defines four different things that an Audience can do:

- They can take their seats.
- They can courteously turn off their cell phones.
- They can give a round of applause.
- They can demand a refund.

Although these functions clearly define an Audience's behavior, it's not clear when each method will be called. What we'd like is for the Audience to take their seats and turn off their cell phones prior to the performance, to applaud when the performance is good, and to demand a refund when the performance goes bad.

One option would be for us to inject an Audience into each performer and to change the `perform()` method to call the Audience's methods. For example, consider an updated version of Instrumentalist in listing 4.2.

Listing 4.2 An instrumentalist that tells its audience how to respond

```
package com.springinaction.springidol;

public class Instrumentalist implements Performer {
    public Instrumentalist() {}

    public void perform() throws PerformanceException {
        audience.takeSeats();
        audience.turnOffCellPhones();

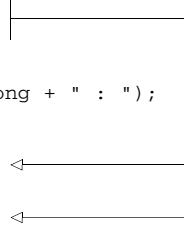
        try {
            System.out.print("Playing " + song + " : ");
            instrument.play();

            audience.applaud();
        } catch (Throwable e) {
            audience.demandRefund();
        }
    }

    private String song;
    public void setSong(String song) {
        this.song = song;
    }

    private Instrument instrument;
    public void setInstrument(Instrument instrument) {
        this.instrument = instrument;
    }

    private Audience audience;
    public void setAudience(Audience audience) {
        this.audience = audience;
    }
}
```



Manipulates Audience



Injects Audience

This would certainly work, but doesn't it seem odd that the `Instrumentalist` has to tell its `Audience` what to do? The performer's job is to give a performance, not to prompt its audience to respond to the performance. The audience's job is to respond to that performance on its own, without being prompted to do so.

Another caveat to this approach is that every performer will need to be injected with an `Audience` and will need to call the `Audience`'s methods. As a result, the various implementations of `Performer` are dependent and coupled to the `Audience` class. This would mean that a performer couldn't perform without an audience. So much for singing in the shower!

The most notable thing about injecting an audience into a performer is that the performer is completely responsible for asking the audience to applaud. In real life, this would be like a performer holding up an "Applaud!" sign. But when you think about it, the performer should focus on performing and not concern himself with whether or not the audience applauds. The audience should react to the performer's performance... not be prodded by the performer.

In other words, the audience is a cross-cutting concern relative to the performer. Since cross-cutting concerns are the province of aspects, perhaps we should define the audience as an aspect. And that's precisely what we're going to do. Let's start by defining some advice.

4.2.1 Creating advice

As mentioned earlier in this chapter, advice defines what an aspect does and when it does it. In Spring AOP, there are five types of advice, each defined by an interface (see table 4.1).

Notice that all of these interfaces are part of the Spring Framework, except for `MethodInterceptor`. When defining around advice, Spring takes advantage of a

Table 4.1 Spring AOP advice comes in five forms that let you choose when advice is executed relative to a joinpoint.

Advice type	Interface
Before	<code>org.springframework.aop.MethodBeforeAdvice</code>
After-returning	<code>org.springframework.aop.AfterReturningAdvice</code>
After-throwing	<code>org.springframework.aop.ThrowsAdvice</code>
Around	<code>org.aopalliance.intercept.MethodInterceptor</code>
Introduction	<code>org.springframework.aop.IntroductionInterceptor</code>

suitable interface that is already provided by the AOP Alliance, an open source project whose goal is to facilitate and standardize AOP. You can read more about the AOP Alliance on their website at <http://aopalliance.sourceforge.net>.

When you think about what an audience is expected to do and match it up against the advice types in table 4.1, it seems clear that taking their seats and turning off their cell phones is best performed as before advice. Likewise, applause is an after-returning advice. And after-throwing is an appropriate advice for demanding a refund.

`AudienceAdvice` (listing 4.3) is a class that implements three of the five advice interfaces from table 4.1 to define the advice applied by an audience. (We'll talk about the other advice types a little later in this chapter.)

Listing 4.3 Advice that defines how an audience's functionality is applied

```
package com.springinaction.springidol;
import java.lang.reflect.Method;
import org.springframework.aop.AfterReturningAdvice;
import org.springframework.aop.MethodBeforeAdvice;
import org.springframework.aop.ThrowsAdvice;

public class AudienceAdvice implements
    MethodBeforeAdvice,
    AfterReturningAdvice,
    ThrowsAdvice {  
    Implements three  
types of advice  
    public AudienceAdvice() {}  
  
    public void before(Method method, Object[] args, Object target)
        throws Throwable {
        audience.takeSeats();
        audience.turnOffCellPhones();
    }  
    Invokes before  
method  
  
    public void afterReturning(Object.returnValue, Method method,
        Object[] args, Object target) throws Throwable {
        audience.applaud();
    }  
    Executes after successful return  
  
    public void afterThrowing(Throwable throwable) {
        audience.demandRefund();
    }  
    Executes after  
exception thrown  
  
    private Audience audience;
    public void setAudience(Audience audience) {
        this.audience = audience;
    }
}
```

There's a lot going on in listing 4.3, but one thing to take notice of is that AudienceAdvice has an Audience as a dependency. Therefore, we'll need to declare the AudienceAdvice in Spring as follows:

```
<bean id="audienceAdvice"
      class="com.springinaction.springidol.AudienceAdvice">
    <property name="audience" ref="audience" />
</bean>
```

AudienceAdvice is a single class that implements three different types of AOP advice. Let's break it down one advice type at a time, starting with before advice.

Before advice

We want our audience to take their seats and turn off their cell phones prior to a performance. Therefore, AudienceAdvice provides before advice by implementing the MethodBeforeAdvice interface. This interface requires that a before() method be implemented:

```
public void before(Method method, Object[] args, Object target)
    throws Throwable {
    audience.takeSeats();
    audience.turnOffCellPhones();
}
```

The before() method takes three parameters. The first parameter is a java.lang.reflect.Method object that represents the method to which the advice is being applied. The second parameter is an array of Objects that are the arguments that were passed to the method when the method was called. The final parameter is the target of the method invocation (i.e., the object on which the method was called).

If you're familiar with Java's dynamic proxy support that was introduced in Java 1.3, these parameters may seem familiar. They are nearly the same parameters that are given to the invoke() method of java.lang.reflect.InvocationHandler.

These parameters are available to you if your advice needs them. In this case, however, they're ignored, as their values have no bearing on the functionality of the audience.

After returning advice

If a performance goes well (i.e., if no exceptions are thrown), we'd like the audience to graciously applaud. We know that a performance is successful if the perform() method returns. Therefore, AudienceAdvice implements After-

ReturningAdvice to applaud a good performance. AfterReturningAdvice requires that an `afterReturning()` method be implemented:

```
public void afterReturning(Object returnValue, Method method,
    Object[] args, Object target) throws Throwable {
    audience.applaud();
}
```

You'll notice that the parameters to the `afterReturning()` method aren't much different than the parameters to the `before()` method of MethodBeforeAdvice. The only difference is that an additional parameter has been added as the first parameter. This parameter holds the value that was returned from the invoked method.

Again, as with `before()`, the parameters are irrelevant to the audience and are thus ignored.

After throwing advice

People paid good money to be in the audience to see these performances. If anything goes wrong, they're going to want their money back. Therefore, if the `perform()` method fails for any reason—that is, if the method throws an exception—the audience will demand their money back. To accommodate after throwing advice, the AudienceAdvice class implements the ThrowsAdvice interface.

Unlike MethodBeforeAdvice and AfterReturningAdvice, however, ThrowsAdvice doesn't require that any method be implemented. ThrowsAdvice is only a marker interface that tells Spring that the advice may wish to handle a thrown exception.

An implementation of ThrowsAdvice may implement one or more `afterThrowing()` methods whose signatures take the following form:

```
public void afterThrowing([method], [args], [target], throwable);
```

All of the parameters of `afterThrowing()` are optional except for the one that is a `Throwable` type. It is this parameter that tells Spring which exceptions should be handled by the advice. For example, suppose we want to write a log entry every time that a `NullPointerException` is thrown. The following `afterThrowing()` method handles that task:

```
public void afterThrowing(Method method, Object[] args,
    Object target, NullPointerException e) {
    LOGGER.error("NPE thrown from " + method.getName());
}
```

In the case of AudienceAdvice, only one afterThrowing() method is defined:

```
public void afterThrowing(Throwable throwable) {  
    audience.demandRefund();  
}
```

This indicates that we want the audience to demand a refund if any Exception is thrown from the perform() method. Furthermore, since the invocation target, method, and arguments are unimportant to the audience, the parameters are left out of the method signature.

With the audience advice class defined, we're ready to associate the advice with a pointcut to create a complete aspect. But first, let's look at how the same advice could have been implemented as an around advice.

Around advice

Around advice is effectively before, after-returning, and after-throwing advice all rolled into one. In Spring, around advice is defined by the AOP Alliance's MethodInterceptor interface. In our example, the AudienceAdvice class could be rewritten as around advice, as shown in listing 4.4. This new AudienceAroundAdvice class is equivalent to AudienceAdvice, but is implemented as around advice.

Listing 4.4 Defining audience advice as around advice

```
package com.springinaction.springidol;  
import org.aopalliance.intercept.MethodInterceptor;  
import org.aopalliance.intercept.MethodInvocation;  
  
public class AudienceAroundAdvice implements MethodInterceptor {  
    public Object invoke(MethodInvocation invocation)  
        throws Throwable {  
  
        try {  
            audience.takeSeats(); | Implements MethodInterceptor  
            audience.turnOffCellPhones(); | Executes before  
                                         | method call  
            Object returnValue = invocation.proceed(); | Calls target  
                audience.applaud(); | Executes after  
                                         | successful return  
                return returnValue; |  
            } catch (PerformanceException throwable) {  
                audience.demandRefund(); | Executes after  
                                         | exception thrown  
                throw throwable;  
            }  
        }  
  
        // injected
```

```
private Audience audience;
public void setAudience(Audience audience) {
    this.audience = audience;
}
```

The `MethodInterceptor` interface requires that only an `invoke()` method be implemented. In the case of `AudienceAroundAdvice`, the `invoke()` method instructs the audience to take their seats and turn off their cell phones. Next it calls `proceed()` on the method invocation to cause the advised method to be invoked. If a `PerformanceException` is caught from calling `invocation.proceed()`, the audience will demand a refund. Otherwise, the audience will applaud.

The nice thing about writing around advice is that you can succinctly define before and after advice in one method. If you have advice that will be applied both before and after a method, you may find around advice preferable to implementing the individual interface for each advice type. It's less beneficial, however, if you only need before advice or after advice (but not both).

Around advice also offers you the opportunity to inspect and alter the value returned from the advised method. This makes it possible to write advice that performs some postprocessing on a method's return value before returning the value to the caller. `AfterReturningAdvice` only allows you to inspect the returned value—you can't change it.

But around advice has one minor gotcha: you must remember to call `proceed()`. Failure to call `proceed()` will result in the advice being applied but the target method never being executed. But then again, that may be what you want. Perhaps you'd like to prevent execution of a method under certain conditions. Compare that to `MethodBeforeAdvice`, where you can inspect the method and its parameters prior to invocation but you can't stop the method from being invoked (short of throwing an exception, breaking the execution chain).

At this point, we've seen several ways to create advice that defines both the *what* and the *when* of aspects. But if you take a close look at either `AudienceAdvice` or `AudienceAroundAdvice`, you won't find any clues as to what methods those advices will be applied to. That brings us to the topic of pointcuts to define the *where* characteristic of aspects.

4.2.2 Defining pointcuts and advisors

So far we have only discussed how to create AOP advice. This is not very useful if we cannot expressively define where this advice should be applied in our application.

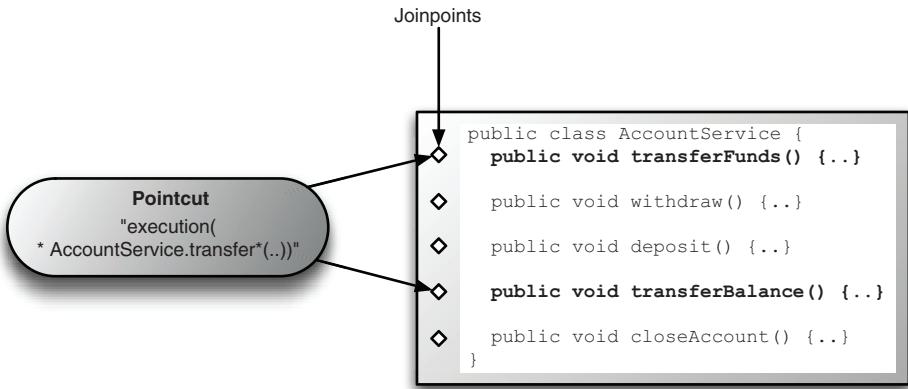


Figure 4.4 Pointcuts select one or more joinpoints where advice should be applied by an aspect. In this case, all methods that perform a transfer operation are singled out by the pointcut.

This is where pointcuts come in. Recall that joinpoints are the points within application code where aspect advice *could* be woven in. Pointcuts are a way of selecting a subset of all possible joinpoints where advice should be woven, as illustrated in figure 4.4.

Spring comes with several different types of pointcuts to choose from. Two of the most useful pointcuts are regular expression pointcuts and AspectJ expression pointcuts. Let's look at regular expression pointcuts first.

Declaring a regular expression pointcut

The main purpose of a pointcut is to choose which method(s) that advice will be applied to, usually by matching a method signature against some pattern. If you're a fan of regular expressions, you may want to use a regular expression pointcut to match the method signature.

Spring comes with two classes that implement regular expression pointcuts:

- `org.springframework.aop.support.Perl5RegexpMethodPointcut`—Useful when an application will be running in a pre-Java 1.4 environment. Requires Jakarta ORO.
- `org.springframework.aop.support.JdkRegexpMethodPointcut`—Best choice when running in Java 1.4 or higher. Does not require Jakarta ORO.

Since we'll be targeting a Java 1.5 runtime, we're going to define the pointcut using `JdkRegexpMethodPointcut` as follows:

```
<bean id="performancePointcut"
      class="org.springframework.aop.support.JdkRegexpMethodPointcut">
    <property name="pattern" value=".perform" />
</bean>
```

The pattern property is used to specify the pointcut pattern used in method matching. Here the pattern property has been set to a regular expression that should match any method called `perform()` on any class.

Once you have defined a pointcut, you'll need to associate it with advice. The following `<bean>` associates the regular expression pointcut we just defined with the audience advice defined in the previous section:

```
<bean id="audienceAdvisor"
      class="org.springframework.aop.support.DefaultPointcutAdvisor">
    <property name="advice" ref="audienceAdvice" />
    <property name="pointcut" ref="performancePointcut" />
</bean>
```

`DefaultPointcutAdvisor` is an advisor class that simply associates advice with a pointcut. Here the `advice` property has been set to reference the `audienceAdvice` bean from the previous section. Meanwhile, the `pointcut` property references the `performancePointcut` bean, which is our pointcut that matches the `perform()` method.

Combining a pointcut with an advisor

Although the `audienceAdvisor` bean completely defines an aspect by associating a pointcut with advice, there's a slightly terser way to define an advisor with a regular expression pointcut.

`RegexpMethodPointcutAdvisor` is a special advisor class that lets you define both a pointcut and an advisor in a single bean. To illustrate, consider the following `<bean>` declaration:

```
<bean id="audienceAdvisor"
      class="org.springframework.aop.support.
      ➔ RegexpMethodPointcutAdvisor">
    <property name="advice" ref="audienceAdvice" />
    <property name="pattern" value=".perform" />
</bean>
```

This single `<bean>` does the work of two beans. It is effectively equivalent to both the `performancePointcut` bean and the previously defined `audienceAdvisor` bean.

Defining AspectJ pointcuts

Although regular expressions work fine as a pointcut definition language, their purpose is for general-purpose text parsing—they weren’t created with pointcuts in mind. Contrast them with how pointcuts are defined in AspectJ and you’ll find that AspectJ’s pointcut language is a true pointcut expression language.

If you’d prefer to use AspectJ-style expressions when defining your Spring pointcuts, you’ll want to use `AspectJExpressionPointcut` instead of `JdkRegexpMethodPointcut`. The following `<bean>` declares the performance pointcut using an AspectJ pointcut expression:

```
<bean id="performancePointcut"
      class="org.springframework.aop.aspectj.
           ↗ AspectJExpressionPointcut">
    <property name="expression" value="execution(* Performer+.perform(..))" />
</bean>
```

The pointcut expression is defined as a value of the `expression` property. In this case, we’re indicating that the pointcut should trigger advice when any `perform()` method taking any arguments is executed on a `Performer`, returning any type. Figure 4.5 summarizes the AspectJ expression used.

To associate the AspectJ expression pointcut with the audience advice, you could use `DefaultPointcutAdvisor`, just as with regular expression pointcut. But just as with regular expression pointcuts, you can also simplify how pointcuts and advice are tied together by using a special advisor that lets you define the pointcut expression as a property of the advisor. For AspectJ pointcut expressions, the advisor class to use is `AspectJExpressionPointcutAdvisor`. The following `<bean>` applies the audience advice to the `perform()` method using an AspectJ pointcut expression:

```
<bean id="audienceAdvisor"
      class="org.springframework.aop.aspectj.
           ↗ AspectJExpressionPointcutAdvisor">
    <property name="advice" ref="audienceAdvice" />
    <property name="expression" value="execution(* *.perform(..))" />
</bean>
```

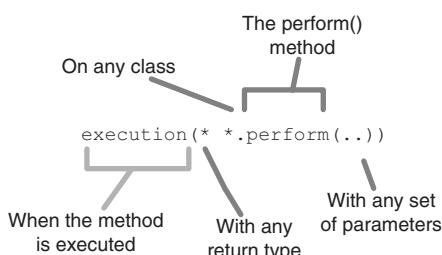


Figure 4.5
Spring AOP uses AspectJ-style pointcuts to select places to apply advice. This pointcut specifies that advice should be applied on any method named “perform” on any class with any number of arguments.

The advice property references the advice being applied—here it's the audienceAdvice bean from earlier. The expression property is where the AspectJ pointcut expression is set.

In Spring AOP, advisors completely define an aspect by associating advice with a pointcut. But aspects in Spring are proxied. Whether you use regular expression pointcuts or AspectJ pointcuts, you'll still need to proxy your target beans for the advisors to take effect. For that, you'll need to declare one or more `ProxyFactoryBeans`.

4.2.3 Using `ProxyFactoryBean`

As you may recall from chapter 2, one of the performers in the *Spring Idol* competition is the juggling poet named Duke. As a quick reminder, here's how Duke is declared as a `<bean>` in Spring:

```
<bean id="dukeTarget"
      class="com.springinaction.springidol.PoeticJuggler"
      autowire="constructor">
    <constructor-arg ref="sonnet29" />
</bean>
```

If you're paying close attention, you've probably noticed one small change that was made to this `<bean>` declaration. The `id` attribute has changed from `duke` to `dukeTarget`. We'll explain why this has been done in a moment. But for now we wanted to draw your attention to this new `id`.

For a bean to be advised by an advisor, it must be proxied. Spring's `ProxyFactoryBean` is a factory bean that produces a proxy that applies one or more interceptors (and advisors) to a bean. The following `<bean>` definition creates a proxy for the duke bean:

```
<bean id="duke"
      class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="target" ref="dukeTarget" />
    <property name="interceptorNames" value="audienceAdvisor" />
    <property name="proxyInterfaces"
              value="com.springinaction.springidol.Performer" />
</bean>
```

The most notable thing about this bean is that its `id` is `duke`. But hold on—won't that mean that when the Spring container is asked for a bean named `duke` it will be the proxy and not the `PoeticJuggler` that is returned? That's absolutely right. In fact, that's elemental to how Spring AOP works. As depicted in figure 4.6, when you invoke a method on an advised bean, you are actually invoking a method on

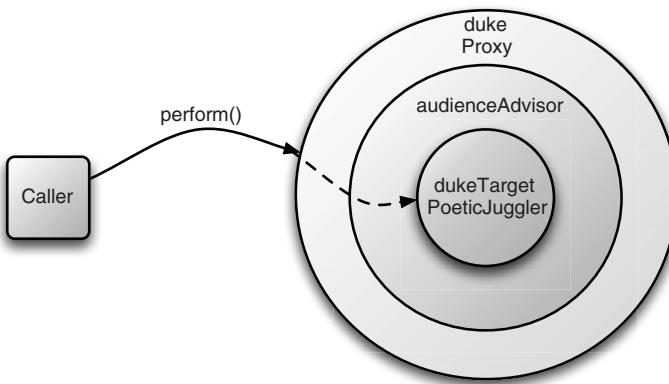


Figure 4.6 When the `perform()` method is called on the `PoeticJuggler`, the call is intercepted by the proxy and execution is given to the `audienceAdvisor` bean before the actual `perform()` method is executed.

the proxy. The proxy will use the pointcut to decide whether advice should be applied (or not), and then it invokes the advised bean itself.

Because the `ProxyFactoryBean` has been given the `id` of the advised bean (`duke`), the advised bean will need to be given a new `id`. That's why we renamed the actual `PoeticJuggler` bean as `dukeTarget`. And it's the `dukeTarget` bean that is referenced by the `target` property of `ProxyFactoryBean`. Put simply, this property tells `ProxyFactoryBean` which bean it will be proxying.

The `interceptorNames` property tells `ProxyFactoryBean` which advisors to apply to the proxied bean. This property takes an array of `String`s, of which each member is the name of an interceptor/advisor bean in the Spring context. In our case, we only want to apply a single advisor, so we provide a single value of `audienceAdvisor` (don't worry; Spring will automatically turn that value into a single member array). However, we could have just as easily set that property explicitly as an array using the following XML:

```
<property name="interceptorNames">
  <list>
    <value>audienceAdvisor</value>
  </list>
</property>
```

The final property set on `ProxyFactoryBean` is `proxyInterfaces`. `ProxyFactoryBean` produces a Java dynamic proxy that advises the target bean, but you'll still need a way to invoke methods on that proxy. The `proxyInterfaces` property tells

ProxyFactoryBean which interface(s) the proxy should implement. As with the interceptorNames property, this property is an array property—actually, an array of `java.lang.Class`. But we specified the value as a single `String` value. Fortunately, Spring is smart enough (using the `ClassEditor` property editor from table 3.1) to translate that single `String` value into a single-member `Class` array.

Abstracting ProxyFactoryBean

So far we've only proxied Duke. This means that the audience will only attend Duke's performance. If we want the audience to take their seats, turn off their cell phones, and applaud for our other performers, then we'll need to proxy the other performers as well.

To that end, here's Stevie, proxied with the audience advisor:

```
<bean id="stevie"
      class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="target" ref="stevieTarget" />
    <property name="proxyInterfaces"
              value="com.springinaction.springidol.Performer" />
    <property name="interceptorNames" value="audienceAdvisor" />
</bean>
```

Now the audience will watch Stevie's performance as well as Duke's. But wait a minute—do we have to write all of this XML for each and every bean that we want to proxy? The `proxyInterfaces` property will be the same for all performers. And the `interceptorNames` property will be the same. It seems a bit too much to have to repeat this information for all of our performers when the only thing that will be different will be the `target` property.

It's often the case that an aspect will be applied to multiple beans in your application. In fact, that's why aspects are said to handle cross-cutting concerns—because an aspect's concern cuts across multiple objects. Although you could write the same `ProxyFactoryBean` declaration for all of the beans to be advised, there's a better way that cuts down on the amount of redundant XML.

The trick is to declare a single `ProxyFactoryBean` as an abstract bean, then reuse that declaration as a parent for each of the advised beans. For example, `audienceProxyBase` declares an abstract bean with the common `proxyInterfaces` and `interceptorNames` properties set:

```
<bean id="audienceProxyBase"
      class="org.springframework.aop.framework.ProxyFactoryBean"
      abstract="true">
    <property name="proxyInterfaces"
              value="com.springinaction.springidol.Performer" />
    <property name="interceptorNames" value="audienceAdvisor" />
</bean>
```

The `audienceProxyBase` bean has its `abstract` attribute set to true, indicating that it is an abstract bean and that Spring shouldn't try to instantiate it directly. Instead, this bean will serve as the basis for the other performer beans. Here are the new, terser, declarations of `stevie` and `duke`, which use the `parent` attribute to extend the `audienceProxyBase` bean:

```
<bean id="stevie" parent="audienceProxyBase">
    <property name="target" ref="stevieTarget" />
</bean>

<bean id="duke" parent="audienceProxyBase">
    <property name="target" ref="dukeTarget" />
</bean>
```

That's much more succinct, isn't it? In this form, only the variant `target` property is declared. The common properties are inherited from the parent bean.

Using abstract beans to define a parent for all of your advised beans is a great way to cut down on the amount of XML in your Spring configuration. However, there's still more that you can do to reduce the amount of XML required to proxy beans with advisors. Coming up next, you'll learn how to eliminate the need for `ProxyFactoryBean` and have Spring automatically proxy beans to be advised.

4.3 Autoproxying

One thing that may have struck you as odd from the previous section is that we had to rename our bean to `dukeTarget` and then give the `ProxyFactoryBean` an `id` of `duke`. This left us with a strange arrangement of beans: the bean that actually represents Duke is named `dukeTarget`, while the bean named `duke` is really a `ProxyFactoryBean` with the purpose of proxying the real Duke with an audience.

If you found that unclear, don't feel too bad. It's a confusing concept that baffles most programmers who are just getting their feet wet with Spring AOP.

In addition to confusion, `ProxyFactoryBean` also lends to the verbosity in the Spring configuration file. Even if you define an abstract `ProxyFactoryBean`, you will still need declare two beans for each bean that is advised: the target bean and the proxy bean. It would be so much nicer if we could simply declare the advisor once and let Spring automatically create proxies for beans whose methods match the advisor's pointcut.

Good news! Spring provides support for automatic proxying of beans. Autoproxying provides a more complete AOP implementation by letting an aspect's pointcut definition decide which beans need to be proxied, rather than requiring you to explicitly create proxies for specific beans.

Actually, there are two ways to autoproxy beans:

- *Basic autoproxying of beans based on advisor beans declared in the Spring context*—The advisor's pointcut expression is used to determine which beans and which methods will be proxied.
- *Autoproxying based on @AspectJ annotation-driven aspects*—The pointcut specified on the advice contained within the aspect will be used to choose which beans and methods will be proxied.

Using either of these autoproxying strategies can eliminate `ProxyFactoryBean` from your Spring context XML file. The former approach to autoproxying uses the advisors we've already created up to this point in Spring. Let's start by looking at this basic autoproxy mechanism.

4.3.1 Creating autoproxies for Spring aspects

If you take a look at the `audienceAdvisor` bean declared in section 4.2.2, you'll see that it has all of the information needed to advise our performer beans:

```
<bean id="audienceAdvisor"
      class="org.springframework.aop.aspectj.
          ↗ AspectJExpressionPointcutAdvisor">
    <property name="advice" ref="audienceAdvice" />
    <property name="expression" value="execution(* *.perform(..))" />
</bean>
```

The `advice` property tells it what advice to apply and the `expression` property tells it where to apply that advice. Despite that wealth of information, we still have to explicitly declare a `ProxyFactoryBean` for Spring to proxy our performers.

However, Spring comes with a handy implementation of `BeanPostProcessor` (see chapter 3) called `DefaultAdvisorAutoProxyCreator`, which automatically checks to see whether an advisor's pointcut matches a bean's methods and replaces that bean's definition with a proxy that applies the advice. In a nutshell, it automatically proxies beans with matching advisors.

To use `DefaultAdvisorAutoProxyCreator`, all you have to do is declare the following `<bean>` in your Spring context:

```
<bean class="org.springframework.aop.framework.autoproxy.
          ↗ DefaultAdvisorAutoProxyCreator" />
```

Notice that this bean doesn't have an `id`. That's because we'll never refer to it directly. Instead, the Spring container will recognize it as a `BeanPostProcessor` and put it to work creating proxies.

With `DefaultAdvisorAutoProxyCreator` declared, we no longer need to declare `ProxyFactoryBeans` in the Spring context. What's more, we no longer have to give our beans weird names that end with `target`. We can now give them appropriate names like `steve` or `duke`:

```
<bean id="duke"
      class="com.springinaction.springidol.PoeticJuggler"
      autowire="constructor">
    <constructor-arg ref="sonnet29" />
</bean>
```

In this way, we are able to keep both bean declarations and bean code free from the aspect-related details.

Spring's basic autoproxy facility is fine for working with simple advice or when in a pre-Java 5 environment. But if you're targeting Java 5, you may want to consider Spring's support for AspectJ's annotation-based aspects. Let's see how to create aspects in Spring that are annotation based.

4.3.2 Autoproxying @AspectJ aspects

A major new feature of AspectJ 5 is the ability to annotate POJO classes to be aspects. This new feature is commonly referred to as `@AspectJ`. Prior to AspectJ 5, writing AspectJ aspects involved learning a Java language extension. But AspectJ's new aspect annotations make it simple to turn any class into an aspect just by sprinkling a few annotations around.

Looking back at our `Audience` class, we see that `Audience` contained all of the functionality needed for an audience, but none of the details to make it an aspect. That left us having to create advice, pointcuts, and advisors—AOP plumbing—to define an audience aspect.

But with `@AspectJ` annotations, we can revisit our `Audience` class and turn it into an aspect without the need for any additional classes or bean declarations. Listing 4.5 shows the new `Audience` class, now annotated to be an aspect.

Listing 4.5 Annotating Audience to be an aspect

```
package com.springinaction.springidol;
import org.aspectj.lang.annotation.AfterReturning;
import org.aspectj.lang.annotation.AfterThrowing;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.aspectj.lang.annotation.Pointcut;

@Aspect           ◀— Declares aspect
public class Audience {
```

```

public Audience() {}

@Pointcut("execution(* *.perform(..))")
public void performance() {}           | Defines performance pointcut

@Before("performance()")
public void takeSeats() {
    System.out.println("The audience is taking their seats.");
}

@Before("performance()")
public void turnOffCellPhones() {
    System.out.println("The audience is turning off
        ➔ their cellphones");
}

@AfterReturning("performance()")
public void applaud() {
    System.out.println("CLAP CLAP CLAP CLAP CLAP");
}

@AfterThrowing("performance()")
public void demandRefund() {
    System.out.println("Boo! We want our money back!");
}
}

```

■

Executes before performance**Executes after performance****Executes after bad performance**

The new Audience class is now annotated with `@Aspect`. This annotation indicates that Audience is not just any old POJO but that it is an aspect.

The `@Pointcut` annotation is used to define a reusable pointcut within an `@AspectJ` aspect. The value given to the `@Pointcut` annotation is an AspectJ pointcut expression—here indicating that the pointcut should match the `perform()` method of any class. The name of the pointcut is derived from the name of the method to which the annotation is applied. Therefore, the name of this pointcut is `performance()`. The actual body of the `performance()` method is irrelevant and, in fact, should be empty. The method itself is just a marker, giving the `@Pointcut` annotation something to attach itself to.

Each of the audience's methods has been annotated with advice annotations. The `@Before` annotation has been applied to both `takeSeats()` and `turnOffCellPhones()` to indicate that these two methods are before advice. The `@AfterReturning` annotation indicates that the `applaud()` method is an after-returning advice method. And the `@AfterThrowing` annotation is placed on `demandRefund()` so that it will be called if any exceptions are thrown during the performance.

The name of the `performance()` pointcut is given as the value parameter to all of the advice annotations. This tells each advice method where it should be applied.

Notice that aside from the annotations and the no-op `performance()` method, the `Audience` class is functionally unchanged. This means that it's still a simple Java object and can be used as such. It can also still be wired in Spring as follows:

```
<bean id="audience"
      class="com.springinaction.springidol.Audience" />
```

Because the `Audience` class contains everything that's needed to define its own pointcuts and advice, there's no more need for a class that explicitly implements one of Spring's advice interfaces. There's also no further need to declare an advisor bean in Spring. Everything needed to use `Audience` as advice is now contained in the `Audience` class itself.

There's just one last thing to do to make Spring apply `Audience` as an aspect. You must declare an autoproxy bean in the Spring context that knows how to turn `@AspectJ`-annotated beans into proxy advice.

For that purpose, Spring comes with an autoproxy creator class called `AnnotationAwareAspectJAutoProxyCreator`. You could register an `AnnotationAwareAspectJAutoProxyCreator` as a `<bean>` in the Spring context, but that would require a lot of typing (believe me... I've typed it twice so far). Instead, to simplify that rather long name, Spring also provides a custom configuration element in the `aop` namespace that's much easier to remember:

```
<aop:aspectj-autoproxy />
```

`<aop:aspectj-autoproxy/>` will create an `AnnotationAwareAspectJAutoProxyCreator` in the Spring context and will automatically proxy beans whose methods match the pointcuts defined with `@Pointcut` annotations in `@AspectJ`-annotated beans.

To use the `<aop:aspectj-autoproxy>` configuration element, you'll need to remember to include the `aop` namespace in your Spring configuration file:

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/
                           spring-beans-2.0.xsd
                           http://www.springframework.org/schema/aop
                           http://www.springframework.org/schema/aop/spring-aop-2.0.xsd">
    ...
</beans>
```

You should be aware that `AnnotationAwareAspectJAutoProxyCreator` only uses `@AspectJ`'s annotations as a guide for creating proxy-based aspects. Under the covers, it's still Spring-style aspects. This is significant because it means that

although you are using @AspectJ's annotations, you are still limited to proxying method invocations.

You may also be interested to know that `AnnotationAwareAspectJAutoProxyCreator` also creates proxies based on classic Spring advisors. That is, it also does the same job that `DefaultAdvisorAutoProxyCreator` does. So, if you have any advisor beans declared in your Spring context, those will also automatically be used to advise proxied beans.

Annotating around advice

Just as with classic Spring advice, you are not limited to before and after advice types when using @AspectJ annotations. You may also choose to create around advice. For that, you must use the `@Around` annotation, as in the following example:

```
@Around("performance()")
public void watchPerformance (ProceedingJoinPoint joinpoint) {
    System.out.println("The audience is taking their seats.");
    System.out.println("The audience is turning off " +
        "their cellphones");

    try {
        joinpoint.proceed();

        System.out.println("CLAP CLAP CLAP CLAP CLAP");
    } catch (PerformanceException throwable) {
        System.out.println("Boo! We want our money back!");
    }
}
```

Here the `@Around` annotation indicates that the `watchPerformance()` method is to be applied as around advice to the `performance()` pointcut.

As you may recall from section 4.2.1, around advice methods must remember to explicitly invoke `proceed()` so that the proxied method will be invoked. But simply annotating a method with `@Around` isn't enough to provide a `proceed()` method to call. Therefore, methods that are to be around advice must take a `ProceedingJoinPoint` object as an argument and then call the `proceed()` method on that object.

Autoproxying of aspects sure makes configuring Spring aspects a lot simpler and makes the application of aspects transparent. But in its transparency, autoproxying obscures many details of the aspects. With autoproxying it is less apparent as to which beans are aspects and which beans are being proxied. In the next section, we'll see how some new features in Spring 2.0 achieve a middle ground where aspects are explicitly defined but without all of the XML verbosity of using `ProxyFactoryBean`.

4.4 Declaring pure-POJO aspects

The Spring development team recognized that using `ProxyFactoryBean` is somewhat clumsy. So, they set out to provide a better way of declaring aspects in Spring. The outcome of this effort is found in the new XML configuration elements that come with Spring 2.0.

You've already seen one of the new elements in the `aop` namespace—`<aop:aspectj-autoproxy>`. But Spring 2.0 comes with several more configuration elements in the `aop` namespace that make it simple to turn any class into an aspect. The new AOP configuration elements are summarized in table 4.2.

Revisiting our audience example one last time, you'll recall that the `Audience` class has all of the methods that define an audience's functionality. We only need to turn that `Audience` class into an aspect with pointcuts that tell it when to perform each of its actions. In the previous section we did that with `@AspectJ` annotations, but this time we'll do it using Spring's AOP configuration elements.

The great thing about Spring's AOP configuration elements is that they can be used to turn *any* class into an aspect. The original `Audience` class from listing 4.1, for instance, is just a plain Java class—no special interfaces or annotations. Using Spring's AOP configuration elements, as shown in listing 4.6, we can turn the `audience` bean into an aspect.

Table 4.2 Spring 2.0's AOP configuration elements simplify declaration of POJO-based aspects.

AOP configuration element	Purpose
<code><aop:advisor></code>	Defines an AOP advisor.
<code><aop:after></code>	Defines an AOP after advice (regardless of whether the advised method returns successfully).
<code><aop:after-returning></code>	Defines an AOP after-returning advice.
<code><aop:after-throwing></code>	Defines an AOP after-throwing advice.
<code><aop:around></code>	Defines an AOP around advice.
<code><aop:aspect></code>	Defines an aspect.
<code><aop:before></code>	Defines an AOP before advice.
<code><aop:config></code>	The top-level AOP element. Most <code><aop:></code> elements must be contained within <code><aop:config></code> .
<code><aop:pointcut></code>	Defines a pointcut.

Listing 4.6 Defining an audience aspect using Spring's AOP configuration elements

```

<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xsi:schemaLocation="http://www.springframework.org/
                           schema/beans
                           http://www.springframework.org/schema/beans/
                           spring-beans-2.0.xsd
                           http://www.springframework.org/schema/aop
                           http://www.springframework.org/schema/aop/
                           spring-aop-2.0.xsd">

    <bean id="audience"
          class="com.springinaction.springidol.Audience" />

    <aop:config>
        <aop:aspect ref="audience"> References audience  
bean as aspect

            <aop:before
                method="takeSeats"
                pointcut="execution(* *.perform(..))" /> Executes before  
performance

            <aop:before
                method="turnOffCellPhones"
                pointcut="execution(* *.perform(..))" />

            <aop:after-returning
                method="applaud"
                pointcut="execution(* *.perform(..))" /> Executes after  
performance

            <aop:after-throwing
                method="demandRefund"
                pointcut="execution(* *.perform(..))" /> Executes after bad  
performance

        </aop:aspect>
    </aop:config>
</beans>
```

The first thing to notice about the Spring AOP configuration elements is that most of them must be used within the context of the `<aop:config>` element. There are a few exceptions to this rule, but none of those exceptions appear in this section. When we encounter such an exception elsewhere in this book, I'll be sure to point it out.

Within `<aop:config>` you may declare one or more advisors, aspects, or pointcuts. In listing 4.6, we've declared a single aspect using the `<aop:aspect>` element. The `ref` attribute references the POJO bean that will be used to supply the functionality of the aspect—in this case, `Audience`. The bean that is

referenced by the `ref` attribute will supply the methods called by any advice in the aspect.

The aspect has four different bits of advice. The two `<aop:before>` elements define *method before advice* that will call the `takeSeats()` and `turnOffCellPhones()` methods (declared by the `method` attribute) of the `Audience` bean before any methods matching the pointcut are executed. The `<aop:after-returning>` element defines an *after-returning advice* to call the `applaud()` method after the pointcut. Meanwhile, the `<aop:after-throwing>` element defines an *after-throwing advice* to call the `demandRefund()` method if any exceptions are thrown. Figure 4.7 shows how the advice logic is woven into the business logic.

In all advice elements, the `pointcut` attribute defines the pointcut where the advice will be applied. The value given to the `pointcut` attribute is a pointcut defined in AspectJ's pointcut expression syntax.

You'll notice that the value of the `pointcut` attribute is the same for all of the advice elements. That's because all of the advice is being applied to the same pointcut. This, however, presents a DRY (don't repeat yourself) principle violation. If you decide later to change the pointcut, you must change it in four different places.

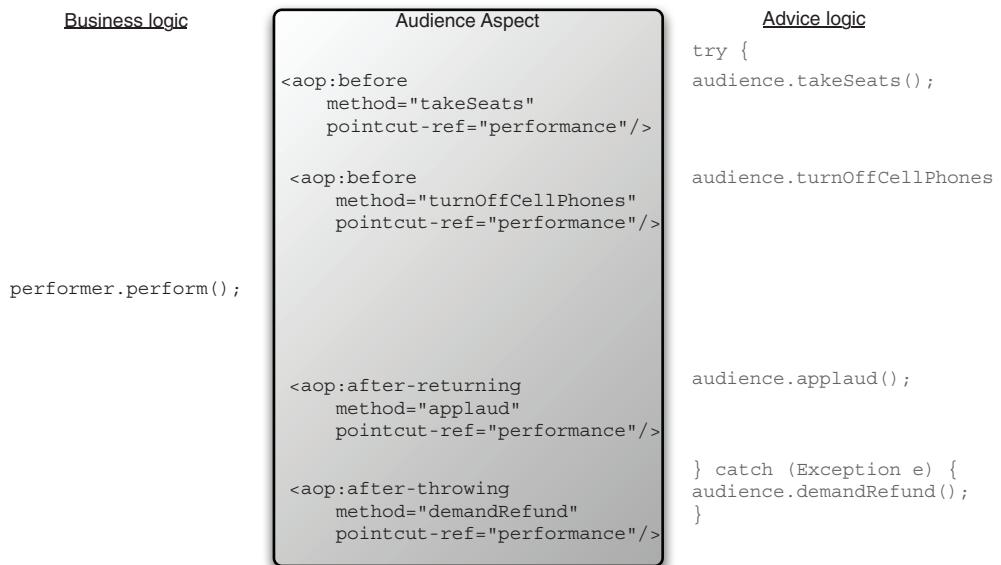


Figure 4.7 The `Audience aspect` includes four bits of advice that weave advice logic around methods that match the aspect's pointcut.

To avoid duplication of the pointcut definition, you may choose to define a named pointcut using the `<aop:pointcut>` element. The XML in listing 4.7 shows how the `<aop:pointcut>` element is used within the `<aop:aspect>` element to define a named pointcut that can be used by all of the advice elements.

Listing 4.7 Defining a named pointcut to eliminate redundant pointcut definitions

```

<aop:config>
  <aop:aspect ref="audience">
    <aop:pointcut
      id="performance"
      expression="execution(* *.perform(..))" /> Defines performance pointcut

    <aop:before
      method="takeSeats"
      pointcut-ref="performance" /> ↳ References pointcut
    <aop:before
      method="turnOffCellPhones"
      pointcut-ref="performance" /> ↳ References pointcut

    <aop:after-returning
      method="applaud"
      pointcut-ref="performance" /> ↳ References pointcut
    <aop:after-throwing
      method="demandRefund"
      pointcut-ref="performance" /> ↳ References pointcut
  </aop:aspect>
</aop:config>

```

Now the pointcut is defined in a single location and is referenced across multiple advice elements. The `<aop:pointcut>` element defines the pointcut to have an `id` of `performance`. Meanwhile, all of the advice elements have been changed to reference the named pointcut with the `pointcut-ref` attribute.

As used in listing 4.7, the `<aop:pointcut>` element defines a pointcut that can be referenced by all advices within the same `<aop:aspect>` element. But you can also define pointcuts that can be used across multiple aspects by placing the `<aop:pointcut>` elements within the scope of the `<aop:config>` element.

It's worth mentioning at this point that both the `<aop:aspect>` element and the `@AspectJ` annotations are effective ways to turn a POJO into an aspect. But `<aop:aspect>` has one distinct advantage over `@AspectJ` in that you do not need the source code of the class that is to provide the aspect's functionality. With `@AspectJ`, you must annotate the class and methods, which requires having the source code. But `<aop:aspect>` can reference any bean.

Spring AOP enables separation of cross-cutting concerns from an application's business logic. But as we've mentioned, Spring aspects are still proxy based and are limited to advising method invocations. If you need more than just method proxy support, you'll want to consider using AspectJ. In the next section, you'll see how AspectJ aspects can be used within a Spring application.

4.5 Injecting AspectJ aspects

Although Spring AOP is sufficient for many applications of aspects, it is a weak AOP solution when contrasted with AspectJ. AspectJ offers many types of pointcuts that are simply not possible with Spring AOP.

Constructor pointcuts, for example, are convenient when you need to apply advice upon the creation of an object. Unlike constructors in some other object-oriented languages, Java constructors are different from normal methods. This makes Spring's proxy-based AOP woefully inadequate for advising creation of an object.

For the most part, AspectJ aspects are independent of Spring. Although they can certainly be woven into any Java-based application, including Spring applications, there's little involvement on Spring's part in applying AspectJ aspects.

However, any well-designed and meaningful aspect will likely depend on other classes to assist in its work. If an aspect depends on one or more classes when executing its advice, you can instantiate those collaborating objects with the aspect itself. Or, better yet, you can use Spring's dependency injection to inject beans into AspectJ aspects.

To illustrate, let's create a new aspect for the *Spring Idol* competition. A talent competition needs a judge. So, let's create a judge aspect in AspectJ. JudgeAspect (listing 4.8) is such an aspect.

Listing 4.8 An AspectJ implementation of a talent competition judge

```
package com.springinaction.springidol;

public aspect JudgeAspect {
    public JudgeAspect() {}

    pointcut performance() : execution(* perform(...));

    after() returning() : performance() {
        System.out.println(criticismEngine.getCriticism());
    }

    // injected
    private CriticismEngine criticismEngine;
```

```
public void setCriticismEngine(CriticismEngine criticismEngine) {
    this.criticismEngine = criticismEngine;
}
}
```

The chief responsibility for JudgeAspect is to make commentary on a performance after the performance has completed. The `performance()` pointcut in listing 4.8 matches the `perform()` method. When it's married with the `after()` returning() advice, you get an aspect that reacts to the completion of a performance.

What makes listing 4.8 interesting is that the judge doesn't make simple commentary on its own. Instead, JudgeAspect collaborates with a CriticismEngine object, calling its `getCriticism()` method, to produce critical commentary after a performance. To avoid unnecessary coupling between JudgeAspect and the CriticismEngine, the JudgeAspect is given a reference to a CriticismEngine through setter injection. This relationship is illustrated in figure 4.8.

CriticismEngine itself is an interface that declares a simple `getCriticism()` method. An implementation of CriticismEngine is found in listing 4.9.

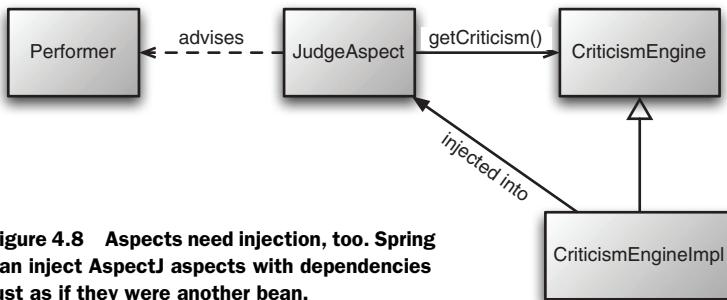


Figure 4.8 Aspects need injection, too. Spring can inject AspectJ aspects with dependencies just as if they were another bean.

Listing 4.9 An implementation of the CriticismEngine used by JudgeAspect

```
package com.springinaction.springidol;

public class CriticismEngineImpl implements CriticismEngine {
    public CriticismEngineImpl() {}

    public String getCriticism() {
        int i = (int) (Math.random() * criticismPool.length);
        return criticismPool[i];
    }

    // injected
}
```

```
private String[] criticismPool;
public void setCriticismPool(String[] criticismPool) {
    this.criticismPool = criticismPool;
}
}
```

CriticismEngineImpl implements the CriticismEngine interface by randomly choosing a critical comment from a pool of injected criticisms. This class can be declared as a Spring <bean> using the following XML:

```
<bean id="criticismEngine"
      class="com.springinaction.springidol.CriticismEngineImpl">
    <property name="criticisms">
      <list>
        <value>I'm not being rude, but that was appalling.</value>
        <value>You may be the least talented
          ↗ person in this show.</value>
        <value>Do everyone a favor and keep your day job.</value>
      </list>
    </property>
</bean>
```

So far, so good. We now have a CriticismEngine implementation to give to JudgeAspect. All that's left is to wire CriticismEngineImpl into JudgeAspect.

Before we show you how to do the injection, you should know that AspectJ aspects can be woven into your application without involving Spring at all. But if you want to use Spring's dependency injection to inject collaborators into an AspectJ aspect, you'll need to declare the aspect as a <bean> in Spring's configuration. The following <bean> declaration injects the criticismEngine bean into JudgeAspect:

```
<bean class="com.springinaction.springidol.JudgeAspect"
      factory-method="aspectOf">
    <property name="criticismEngine" ref="criticismEngine" />
</bean>
```

For the most part, this <bean> declaration isn't much different from any other <bean> you may find in Spring. But the big difference is the use of the factory-method attribute. Normally Spring beans are instantiated by the Spring container—but AspectJ aspects are created by the AspectJ runtime. By the time Spring gets a chance to inject the CriticismEngine into JudgeAspect, JudgeAspect has already been instantiated.

Since Spring isn't responsible for the creation of JudgeAspect, it isn't possible to simply declare JudgeAspect as a bean in Spring. Instead, we need a way

for Spring to get a handle to the `JudgeAspect` instance that has already been created by AspectJ so that we can inject it with a `CriticismEngine`. Conveniently, all AspectJ aspects provide a static `aspectOf()` method that returns the singleton instance of the aspect. So to get an instance of the aspect, you must use factory-method to invoke the `aspectOf()` method instead of trying to call `JudgeAspect`'s constructor.

In short, Spring doesn't use the `<bean>` declaration from earlier to create an instance of the `JudgeAspect`—it has already been created by the AspectJ runtime. Instead, Spring retrieves a reference to the aspect through the `aspectOf()` factory method and then performs dependency injection on it as prescribed by the `<bean>` element.

4.6 **Summary**

AOP is a powerful complement to object-oriented programming. With aspects, you can now group application behavior that was once spread throughout your applications into reusable modules. You can then declaratively or programmatically define exactly where and how this behavior is applied. This reduces code duplication and lets your classes focus on their main functionality.

Spring provides an AOP framework that lets you insert aspects around method executions. You have learned how you can weave advice before, after, and around a method invocation, as well as add custom behavior for handling exceptions.

You have several choices in how you can use aspects in your Spring applications. Wiring advice and pointcuts in Spring is much easier in Spring 2.0 with the addition of `@AspectJ` annotation support and a simplified configuration schema.

Finally, there are times when Spring AOP isn't powerful enough and you must turn to AspectJ for more powerful aspects. For those situations, we looked at how to use Spring to inject dependencies into AspectJ aspects.

At this point, we've covered the basics of the Spring Framework. You've seen how to configure the Spring container and how to apply aspects to Spring-managed objects. These core Spring techniques will be foundational throughout the rest of the book. In the coming chapters, we'll begin applying what we've learned as we develop enterprise capabilities into our applications. We'll start in the next chapter by looking at how to persist and retrieve data using Spring's JDBC and ORM abstractions.

Part 2

Enterprise Spring

In part 1, you learned about Spring’s core container and its support for dependency injection (DI) and aspect-oriented programming (AOP). With that foundation set, part 2 will show you how to apply DI and AOP to implement business layer functionality for your application.

Most applications ultimately persist business information in a relational database. Chapter 5, “Hitting the database,” will guide you in using Spring’s support for data persistence. You’ll be introduced to Spring’s JDBC support, which helps you remove much of the boilerplate code associated with JDBC. You’ll also see how Spring integrates with several popular object-relational mapping frameworks, such as Hibernate, JPA, and iBATIS.

Once you are persisting your data, you’ll want to ensure that its integrity is preserved. In chapter 6, “Managing transactions,” you’ll learn how Spring AOP can be used to declaratively apply transactional policies to your application objects using AOP. You’ll see that Spring affords EJB-like transaction support to POJOs.

As security is an important aspect of many applications, chapter 7, “Securing Spring,” will show you how to use the Spring Security (formerly known as Acegi Security) to protect the information your application contains.

In chapter 8, “Spring and POJO-based remote services,” you’ll learn how to expose your application objects as remote services. You’ll also learn how to transparently access remote services as though they are any other object in your application. Remoting technologies explored will include RMI, Hessian/Burlap, web services, and Spring’s own HTTP invoker.

Chapter 9, “Building contract-first web services in Spring,” approaches web services from a different angle by showing how to use the Spring Web Services framework to build contract-driven, document-centric web services.

Chapter 10, “Spring messaging,” explores a different approach to application integration by showing how Spring can be used with JMS to asynchronously send and receive messages between applications. You’ll also see how to develop message-driven POJOs and build asynchronous remote services using Lingo.

Chapter 11, “Spring and Enterprise JavaBeans,” covers the connection between Spring and EJBs. This includes how to wire EJBs in a Spring application and how to Spring-enable EJB session beans. We’ll also take a quick look at how to develop EJB 3-style beans in Spring.

Chapter 12, “Accessing enterprise services,” will wrap up the discussion of Spring in the business layer by showcasing some of Spring’s support for common enterprise services. In this chapter, you’ll learn how to use Spring to access objects in JNDI, send emails, and schedule tasks.



Hitting the database

This chapter covers

- Defining Spring's data access support
- Configuring database resources
- Using Spring's JDBC framework
- Integrating with Hibernate, JPA, and iBATIS

With the core of the Spring container now under your belt, it's time to put it to work in real applications. A perfect place to start is with a requirement of nearly any enterprise application: persisting data. Every one of us has probably dealt with database access in an application in the past. In doing so, you know that data access has many pitfalls. We have to initialize our data access framework, open connections, handle various exceptions, and close connections. If we get any of this wrong, we could potentially corrupt or delete valuable company data. In case you haven't experienced the consequences of mishandled data access, it is a *Bad Thing*.

Since we strive for *Good Things*, we turn to Spring. Spring comes with a family of data access frameworks that integrate with a variety of data access technologies. Whether you are persisting your data via direct JDBC, iBATIS, or an object-relational mapping (ORM) framework like Hibernate, Spring removes the tedium of data access from your persistence code. Instead, you can lean on Spring to handle the low-level data access work for you so that you can turn your attention to managing your application's data.

In this chapter, we're going to build the persistence layer of the RoadRantz application (see figure 5.1). In this layer, we are faced with some choices. We could use JDBC, Hibernate, the Java Persistence API (JPA), iBATIS, or any one of a number of persistence frameworks. Fortunately for us, Spring supports all of those persistence mechanisms.

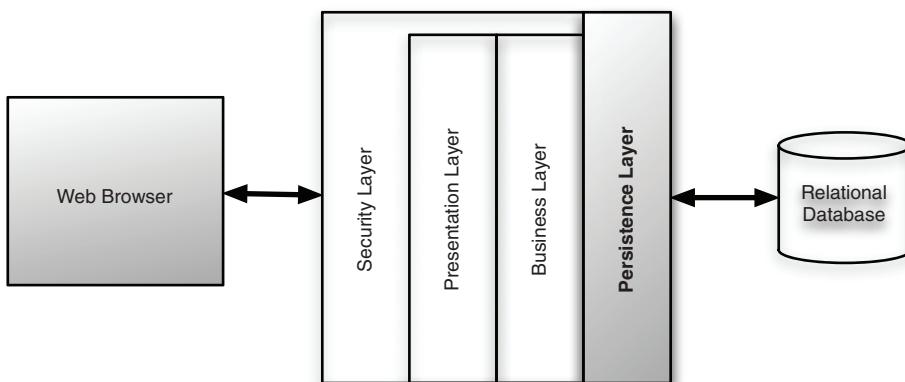


Figure 5.1 Like most applications, RoadRantz persists and restores data from a relational database. The persistence layer of the application is where all data access takes place.

As we build the persistence layer of the RoadRantz application, we'll see how Spring abstracts common data access functions, thus simplifying persistence code. You'll see how Spring makes working with JDBC, Hibernate, JPA, and iBATIS even easier. And before we wrap up our discussion of data access, we'll touch on how to use Spring support for declarative caching to beef up the performance of your application.

Regardless of which persistence technology you choose, simple JDBC or sophisticated JPA, there's a lot of common ground among all of Spring's data access frameworks. So, before we jump into Spring's support for data access, let's talk about the basics of Spring's DAO support.

5.1 Learning Spring's data access philosophy

From the previous chapters, you know that one of Spring's goals is to allow you to develop applications following the sound object-oriented (OO) principle of coding to interfaces. Spring's data access support is no exception.

DAO¹ stands for data access object, which perfectly describes a DAO's role in an application. DAOs exist to provide a means to read and write data to the database. They should expose this functionality through an interface by which the rest of the application will access them. Figure 5.2 shows the proper approach to designing your data access tier.

As you can see, the service objects are accessing the DAOs through interfaces. This has a couple of advantages. First, it makes your service objects easily testable since they are not coupled to a specific data access implementation. In fact, you

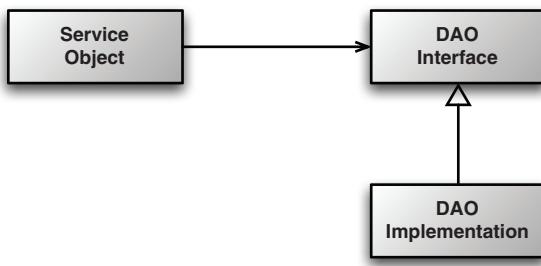


Figure 5.2
Service objects do not handle their own data access. Instead, they delegate data access to DAOs. The DAO's interface keeps it loosely coupled to the service object.

¹ Many developers, including Martin Fowler, refer to the persistence objects of an application as "repositories." While I fully appreciate the thinking that leads to the "repository" moniker, I believe that the word "repository" is already very overloaded, even without adding this additional meaning. So, please forgive me, but I'm going to buck the popular trend—I will continue referring to these objects as DAOs.

could create mock implementations of these data access interfaces. That would allow you to test your service object without ever having to connect to the database, which would significantly speed up your unit tests and rule out the chance of a test failure due to inconsistent data.

In addition, the data access tier is accessed in a persistence technology–agnostic manner. That is, the chosen persistence approach is isolated to the DAO while only the relevant data access methods are exposed through the interface. This makes for a flexible application design and allows the chosen persistence framework to be swapped out with minimal impact to the rest of the application. If the implementation details of the data access tier were to leak into other parts of the application, the entire application would become coupled with the data access tier, leading to a rigid application design.

NOTE If after reading the last couple of paragraphs, you feel that I have a strong bias toward hiding the persistence layer behind interfaces, then I'm happy that I was able to get that point across. The fact is, I believe that interfaces are key to writing loosely coupled code and that they should be used at all layers of an application, not just at the data access layer. That said, it's also important to note that while Spring encourages the use of interfaces, Spring does not require them—you're welcome to use Spring to wire a bean (DAO or otherwise) directly into a property of another bean without an interface between them.

One way Spring helps you insulate your data access tier from the rest of your application is by providing you with a consistent exception hierarchy that is used across all of its DAO frameworks.

5.1.1 **Getting to know Spring's data access exception hierarchy**

If you've ever written JDBC code (without Spring), you're probably keenly familiar with the fact that you can't do anything with JDBC without being forced to catch `java.sql.SQLException`. Some common problems that might cause an `SQLException` to be thrown include:

- The application is unable to connect to the database.
- The query being performed has errors in its syntax.
- The tables and/or columns referred to in the query do not exist.
- An attempt is made to insert or update values that violate a database constraint.

The big question surrounding `SQLException` is how it should be handled when it's caught. As it turns out, many of the problems that trigger an `SQLException` can't be remedied within a catch block. Most `SQLExceptions` that are thrown indicate a fatal condition. If the application can't connect to the database, that usually means that the application will be unable to continue. Likewise, if there are errors in the query, little can be done about it at runtime.

If there's nothing that can be done to recover from an `SQLException`, why are we forced to catch it?

Even if you have a plan for dealing with some `SQLExceptions`, you'll have to catch the `SQLException` and dig around in its properties for more information on the nature of the problem. That's because `SQLException` is treated as a "one size fits all" exception for problems related to data access. Rather than have a different exception type for each possible problem, `SQLException` is the exception that's thrown for all data access problems.

Some persistence frameworks offer a richer hierarchy of exceptions. Hibernate, for example, offers almost two dozen different exceptions, each targeting a specific data access problem. This makes it possible to write catch blocks for the exceptions that you want to deal with.

Even so, Hibernate's exceptions are specific to Hibernate. As stated before, we'd like to isolate the specifics of the persistence mechanism to the data access layer. If Hibernate-specific exceptions are being thrown then the fact that we're dealing with Hibernate will leak into the rest of the application. Either that, or you'll be forced to catch persistence platform exceptions and rethrow them as platform-agnostic exceptions.

On one hand, JDBC's exception hierarchy is too generic—in fact, it's not much of a hierarchy at all. On the other hand, Hibernate's exception hierarchy is proprietary to Hibernate. What we need is a hierarchy of data access exceptions that are descriptive but not directly associated with a specific persistence framework.

Spring's persistence platform agnostic exceptions

Spring JDBC provides a hierarchy of data access exceptions that solve both problems. In contrast to JDBC, Spring provides several data access exceptions, each descriptive of the problem that they're thrown for. Table 5.1 shows some of Spring's data access exceptions lined up against the exceptions offered by JDBC.

As you can see, Spring has an exception for virtually anything that could go wrong when reading or writing to a database. And the list of Spring's data access exceptions is vaster than shown in table 5.1. (I would've listed them all, but I didn't want JDBC to get an inferiority complex.)

Table 5.1 JDBC's exception hierarchy versus Spring's data access exceptions.

JDBC's exceptions	Spring's data access exceptions
BatchUpdateException	CannotAcquireLockException
DataTruncation	CannotSerializeTransactionException
SQLException	CleanupFailureDataAccessException
SQLWarning	ConcurrencyFailureException
	DataAccessException
	DataAccessResourceFailureException
	DataIntegrityViolationException
	DataRetrievalFailureException
	DeadlockLoserDataAccessException
	EmptyResultDataAccessException
	IncorrectResultSizeDataAccessException
	IncorrectUpdateSemanticsDataAccessException
	InvalidDataAccessApiUsageException
	InvalidDataAccessResourceUsageException
	OptimisticLockingFailureException
	PermissionDeniedDataAccessException
	PessimisticLockingFailureException
	TypeMismatchDataAccessException
	UncategorizedDataAccessException

Even though Spring's exception hierarchy is far more rich than JDBC's simple `SQLException`, it isn't associated with any particular persistence solution. This means that you can count on Spring to throw a consistent set of exceptions, regardless of which persistence provider you choose. This helps to keep your persistence choice confined to the data access layer.

Look, Ma! No catch blocks!

What isn't evident from table 5.1 is that all of those exceptions are rooted with `DataAccessException`. What makes `DataAccessException` special is that it is an unchecked exception. In other words, you don't have to catch any of the data access exceptions thrown from Spring (although you're perfectly welcome to if you'd like).

`DataAccessException` is just one example of Spring's across-the-board philosophy of checked versus unchecked exceptions. Spring takes the stance that many exceptions are the result of problems that can't be addressed in a `catch` block. Instead of forcing developers to write `catch` blocks (which are often left empty),

Spring promotes the use of unchecked exceptions. This leaves the decision of whether to catch an exception in the developer's hands.

To take advantage of Spring's data access exceptions, you must use one of Spring's supported data access templates. Let's look at how Spring templates can greatly simplify data access.

5.1.2 **Templating data access**

You have probably traveled by plane before. If so, you will surely agree that one of the most important parts of traveling is getting your luggage from point A to point B. There are many steps to this process. When you arrive at the terminal, your first stop will be at the counter to check your luggage. Next, security will scan it to ensure the safety of the flight. Then it takes a ride on the "luggage train" on its way to being placed on the plane. If you need to catch a connecting flight, your luggage needs to be moved as well. When you arrive at your final destination, the luggage has to be removed from the plane and placed on the carousel. Finally, you go down to the baggage claim area and pick it up.

Even though there are many steps to this process, you are only actively involved in a couple of those steps. The carrier itself is responsible for driving the process. You are only involved when you need to be; the rest is just "taken care of." This mirrors a powerful design pattern: the Template Method pattern.

A template method defines the skeleton of a process. In our example, the process is moving luggage from departure city to arrival city. The process itself is fixed; it never changes. The overall sequence of events for handling luggage occurs the same way every time: luggage is checked in, luggage is loaded on the plane, and so forth. Some steps of the process are fixed as well—that is, some steps happen the same every time. When the plane arrives at its destination, every piece of luggage is unloaded one at a time and placed on a carousel to be taken to baggage claim.

At certain points, however, the process delegates its work to a subclass to fill in some implementation-specific details. This is the variable part of the process. For example, the handling of luggage starts with a passenger checking in the luggage at the counter. This part of the process always has to happen at the beginning, so its sequence in the process is fixed. Because each passenger's luggage check-in is different, the implementation of this part of the process is determined by the passenger. In software terms, a template method delegates the implementation-specific portions of the process to an interface. Different implementations of this interface define specific implementations of this portion of the process.

This is the same pattern that Spring applies to data access. No matter what technology we are using, certain data access steps are required. For example, we always need to obtain a connection to our data store and clean up resources when we are done. These are the fixed steps in a data access process. But each data access method we write is slightly different. We query for different objects and update the data in different ways. These are the variable steps in the data access process.

Spring separates the fixed and variable parts of the data access process into two distinct classes: templates and callbacks. Templates manage the fixed part of the process while your custom data access code is handled in the callbacks. Figure 5.3 shows the responsibilities of both of these classes.

As you can see in figure 5.3, Spring's template classes handle the fixed parts of data access—controlling transactions, managing resources, and handling exceptions. Meanwhile, the specifics of data access as they pertain to your application—creating statements, binding parameters, and marshaling result sets—are handled in the callback implementation. In practice, this makes for an elegant framework because all you have to worry about is your data access logic.

Spring comes with several templates to choose from, depending on your persistence platform choice. If you're using straight JDBC then you'll want to use `JdbcTemplate`. But if you favor one of the object-relational mapping frameworks then perhaps `HibernateTemplate` or `JpaTemplate` is more suitable. Table 5.2 lists all of Spring's data access templates and their purpose.

As you'll see, using a data access template simply involves configuring it as a bean in the Spring context and then wiring it into your application DAO. Or you

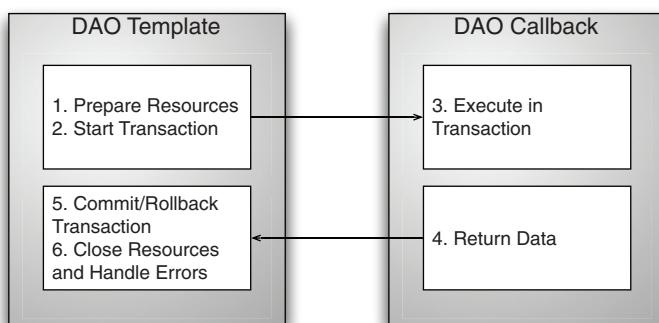


Figure 5.3 Spring's DAO template classes take responsibility for the common data access duties. For the application-specific tasks, it calls back into a custom DAO callback object.

Table 5.2 Spring comes with several data access templates, each suitable for a different persistence mechanism.

Template class (<code>org.springframework.*</code>)	Used to template...
<code>jca.cci.core.CciTemplate</code>	JCA CCI connections
<code>jdbc.core.JdbcTemplate</code>	JDBC connections
<code>jdbc.core.namedparam.NamedParameterJdbcTemplate</code>	JDBC connections with support for named parameters
<code>jdbc.core.simple.SimpleJdbcTemplate</code>	JDBC connections, simplified with Java 5 constructs
<code>orm.hibernate.HibernateTemplate</code>	Hibernate 2.x sessions
<code>orm.hibernate3.HibernateTemplate</code>	Hibernate 3.x sessions
<code>orm.ibatis.SqlMapClientTemplate</code>	iBATIS SqlMap clients
<code>orm.jdo.JdoTemplate</code>	Java Data Object implementations
<code>orm.jpa.JpaTemplate</code>	Java Persistence API entity managers
<code>orm.toplink.TopLinkTemplate</code>	Oracle's TopLink

can take advantage of Spring's DAO support classes to further simplify configuration of your application DAOs. Direct wiring of the templates is fine, but Spring also provides a set of convenient DAO base classes that can manage the template for you. Let's see how these template-based DAO classes work.

5.1.3 Using DAO support classes

The data access templates are not all there is to Spring's data access framework. Each template also provides convenience methods that simplify data access without the need to create an explicit callback implementation. Furthermore, on top of the template-callback design, Spring provides DAO support classes that are meant to be subclassed by your own DAO classes. Figure 5.4 illustrates the relationship between a template class, a DAO support class, and your own custom DAO implementation.

Later, as we examine Spring's individual data access support options, we'll see how the DAO support classes provide convenient access to the template class that they support. When writing your application DAO implementation, you can subclass a DAO support class and call a template retrieval method to have direct access to the underlying data access template. For example, if your application

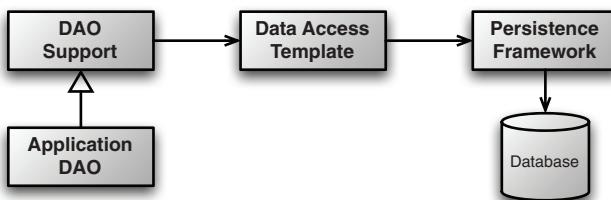


Figure 5.4
The relationship between an application DAO and Spring's DAO support and template classes

DAO subclasses `JdbcDaoSupport` then you only need to call `getJdbcTemplate()` to get a `JdbcTemplate` to work with.

Plus, if you need access to the underlying persistence platform, each of the DAO support classes provide access to whatever class it uses to communicate with the database. For instance, the `JdbcDaoSupport` class contains a `getConnection()` method for dealing directly with the JDBC connection.

Just as Spring provides several data access template implementations, it also provides several DAO support classes—one for each template. Table 5.3 lists the DAO support classes that come with Spring.

Table 5.3 Spring's DAO support classes provide convenient access to their corresponding data access template.

DAO support class (<code>org.springframework.*</code>)	Provides DAO support for...
<code>jca.cci.support.CciDaoSupport</code>	JCA CCI connections
<code>jdbc.core.support.JdbcDaoSupport</code>	JDBC connections
<code>jdbc.core.namedparam.NamedParameterJdbcDaoSupport</code>	JDBC connections with support for named parameters
<code>jdbc.core.simple.SimpleJdbcDaoSupport</code>	JDBC connections, simplified with Java 5 constructs
<code>orm.hibernate.support.HibernateDaoSupport</code>	Hibernate 2.x sessions
<code>orm.hibernate3.support.HibernateDaoSupport</code>	Hibernate 3.x sessions
<code>orm.ibatis.support.SqlMapClientDaoSupport</code>	iBATIS SqlMap clients
<code>orm.jdo.support.JdoDaoSupport</code>	Java Data Object implementations
<code>orm.jpa.support.JpaDaoSupport</code>	Java Persistence API entity managers
<code>orm.toplink.support.TopLinkDaoSupport</code>	Oracle's TopLink

Even though Spring provides support for several persistence frameworks, there simply isn't enough space to cover them all in this chapter. Therefore, I'm going to focus on what I believe are the most beneficial persistence options and the ones that you'll most likely be using.

We'll start with basic JDBC access (section 5.3), as it is the most basic way to read and write data from a database. Then we'll look at Hibernate and JPA (sections 5.4 and 5.5), two of the most popular POJO-based ORM solutions. Finally, I'll dig into Spring's support for iBATIS (in section 5.6), which is a persistence framework that provides the mapping support of an ORM solution with the complete query control of JDBC.

But first things first—most of Spring's persistence support options will depend on a data source. So, before we can get started with creating templates and DAOs, we need to configure Spring with a data source for the DAOs to access the database.

5.2 **Configuring a data source**

Regardless of which form of Spring DAO support you use, you'll likely need to configure a reference to a data source. Spring offers several options for configuring data source beans in your Spring application, including:

- Data sources that are defined by a JDBC driver
- Data sources that are looked up by JNDI
- Data sources that pool connections

For production-ready applications, I recommend using a data source that draws its connections from a connection pool. When possible, I prefer to retrieve the pooled data source from an application server via JNDI. With that preference in mind, let's start by looking at how to configure Spring to retrieve a data source from JNDI.

5.2.1 **Using JNDI data sources**

Spring applications will quite often be deployed to run within a JEE application server such as WebSphere, JBoss, or even a web container like Tomcat. These servers allow you to configure data sources to be retrieved via JNDI. The benefit of configuring data sources in this way is that they can be managed completely external to the application, leaving the application to simply ask for a data source when it's ready to access the database. Moreover, data sources managed in an

application server are often pooled for greater performance and can be hot-swapped by system administrators.

With Spring, we can configure a reference to a data source that is kept in JNDI and wire it into the classes that need it as if it were just another Spring bean. Spring's `JndiObjectFactoryBean` makes it possible to retrieve any object, including data sources, from JNDI and make it available as a Spring bean.

We'll discuss `JndiObjectFactoryBean` a bit more when we get to chapter 11. For now here's a `JndiObjectFactoryBean` that retrieves a data source from JNDI:

```
<bean id="dataSource"
      class="org.springframework.jndi.JndiObjectFactoryBean"
      scope="singleton">
    <property name="jndiName" value="/jdbc/RantzDatasource" />
    <property name="resourceRef" value="true" />
</bean>
```

The `jndiName` attribute is used to specify the name of the resource in JNDI. If only the `jndiName` property is set then the data source will be looked up as is. But if the application is running within a Java application server then you'll want to set the `resourceRef` to true.

When `resourceRef` is true, the value of `jndiName` will be prepended with `java:comp/env/` to retrieve the data source as a Java resource from the application server's JNDI directory. Consequently, the actual name used will be `java:comp/env/jdbc/RantzDatasource`.

JNDI data sources in Spring 2.0

If you're using Spring 2.0, the XML required for retrieving a data source from JNDI is greatly simplified using the `jee` namespace. You can use the configuration elements from the `jee` namespace by declaring your `<beans>` element as follows:

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:jee="http://www.springframework.org/schema/jee"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/
                           spring-beans-2.0.xsd
                           http://www.springframework.org/schema/jee
                           http://www.springframework.org/schema/jee/
                           spring-jee-2.0.xsd">
```

The `jee` namespace offers the `<jee:jndi-lookup>` element for retrieving objects from JNDI. The following XML is equivalent to the explicit declaration of `JndiObjectFactoryBean` shown earlier:

```
<jee:jndi-lookup id="dataSource"
    jndi-name="/jdbc/RantzDatasource"
    resource-ref="true" />
```

The `jndi-name` and `resource-ref` attributes map directly to the `jndiName` and `resourceRef` properties of `JndiObjectFactoryBean`.

5.2.2 Using a pooled data source

If you're unable to retrieve a data source from JNDI, the next best thing is to configure a pooled data source directly in Spring. Although Spring doesn't provide a pooled data source, there's a suitable one available in the Jakarta Commons Database Connection Pools (DBCP) project (<http://jakarta.apache.org/commons/dbcp>). To add DBCP to your application, either download it and place the JAR file in your Ant's build classpath or add the following `<dependency>` to the Maven 2 Project Object Model (POM):

```
<dependency>
    <groupId>commons-dbcp</groupId>
    <artifactId>commons-dbcp</artifactId>
    <version>1.2.1</version>
</dependency>
```

DBCP includes several data sources that provide pooling, but the `BasicDataSource` is one that's often used because it's quite simple to configure in Spring and because it resembles Spring's own `DriverManagerDataSource` (which we'll talk about next).

For the RoadRantz application, we'll configure a `BasicDataSource` bean as follows:

```
<bean id="dataSource"
    class="org.apache.commons.dbcp.BasicDataSource">
    <property name="driverClassName" value="org.hsqldb.jdbcDriver" />
    <property name="url"
        value="jdbc:hsqldb:hsqldb://localhost/roadrantz/roadrantz" />
    <property name="username" value="sa" />
    <property name="password" value="" />
    <property name="initialSize" value="5" />
    <property name="maxActive" value="10" />
</bean>
```

The first four properties are elemental to configuring a `BasicDataSource`. The `driverClassName` property specifies the fully qualified name of the JDBC driver class. Here we've configured it with the JDBC driver for the Hypersonic database. The `url` property is where we set the complete JDBC URL for the database. Finally,

the `username` and `password` properties are used to authenticate when you're connecting to the database.

Those four basic properties define connection information for `BasicDataSource`. In addition, several properties can be used to configure the data source pool itself. Table 5.4 lists a few of the most useful pool-configuration properties of `BasicDataSource`.

For our purposes, we've configured the pool to start with five connections. Should more connections be needed, `BasicDataSource` is allowed to create them, up to a maximum of 10 active connections.

Table 5.4 BasicDataSource's pool-configuration properties.

Pool-configuration property	What it specifies
<code>initialSize</code>	The number of connections created when the pool is started.
<code>maxActive</code>	The maximum number of connections that can be allocated from the pool at the same time. If zero, there is no limit.
<code>maxIdle</code>	The maximum number of connections that can be idle in the pool without extras being released. If zero, there is no limit.
<code>maxOpenPreparedStatements</code>	The maximum number of prepared statements that can be allocated from the statement pool at the same time. If zero, there is no limit.
<code>maxWait</code>	How long the pool will wait for a connection to be returned to the pool (when there are no available connections) before an exception is thrown. If -1, wait indefinitely.
<code>minEvictableIdleTimeMillis</code>	How long a connection can remain idle in the pool before it is eligible for eviction.
<code>minIdle</code>	The minimum number of connections that can remain idle in the pool without new connections being created.
<code>poolPreparedStatements</code>	Whether or not to pool prepared statements (boolean).

5.2.3 JDBC driver-based data source

The simplest data source you can configure in Spring is one that is defined through a JDBC driver. Spring offers two such data source classes to choose from (both in the `org.springframework.jdbc.datasource` package):

- `DriverManagerDataSource`—Returns a new connection every time that a connection is requested. Unlike DBCP's `BasicDataSource`, the connections provided by `DriverManagerDataSource` are not pooled.
- `SingleConnectionDataSource`—Returns the same connection every time that a connection is requested. Although `SingleConnectionDataSource` isn't exactly a pooled data source, you can think of it as a data source with a pool of exactly one connection.

Configuring either of these data sources is similar to how we configured DBCP's `BasicDataSource`:

```
<bean id="dataSource"
      class="org.springframework.jdbc.datasource.
        ↗  DriverManagerDataSource">
    <property name="driverClassName"
              value="org.hsqldb.jdbcDriver" />
    <property name="url"
              value="jdbc:hsqldb:hsq1://localhost/roadrantz/roadrantz" />
    <property name="username" value="sa" />
    <property name="password" value="" />
</bean>
```

The only difference is that since neither `DriverManagerDataSource` nor `SingleConnectionDataSource` provides a connection pool, there are no pool configuration properties to set.

Although `SingleConnectionDataSource` and `DriverManagerDataSource` are great for small applications and running in development, you should seriously consider the implications of using either in a production application. Because `SingleConnectionDataSource` has one and only one database connection to work with, it doesn't work well in a multithreaded application. At the same time, even though `DriverManagerDataSource` is capable of supporting multiple threads, it incurs a performance cost for creating a new connection each time a connection is requested. Because of these limitations, I strongly recommend using pooled data sources.

Now that we have established a connection to the database through a data source, we're ready to actually access the database. The most basic way to access a database is by using JDBC. So, let's begin our exploration of Spring's data access abstractions by looking at how Spring makes working with simple JDBC even simpler.

5.3 Using JDBC with Spring

There are many persistence technologies out there. Hibernate, iBATIS, and JPA are just a few. Despite this, a wealth of applications are writing Java objects to a database the old-fashioned way: they earn it. No, wait—that's how people make money. The tried-and-true method for persisting data is with good old JDBC.

And why not? JDBC does not require mastering another framework's query language. It is built on top of SQL, which is the data access language. Plus, you can more finely tune the performance of your data access when you use JDBC than practically any other technology. And JDBC allows you to take advantage of your database's proprietary features where other frameworks may discourage or flat-out prohibit this.

What's more, JDBC lets you work with data at a much lower level than the persistence frameworks, allowing you to access and manipulate individual columns in a database. This fine-grained approach to data access comes in handy in applications, such as reporting applications, where it doesn't make sense to organize the data into objects, just to then unwind it back into raw data.

But all is not sunny in the world of JDBC. With its power, flexibility, and other niceties also comes well, some not-so-niceties.

5.3.1 Tackling runaway JDBC code

While JDBC gives you an API that works closely with your database, you are responsible for handling everything related to accessing the database. This includes managing database resources and handling exceptions.

If you have ever written JDBC that inserts data into the database, the code in listing 5.1 shouldn't be all too alien to you.

Listing 5.1 Using JDBC to insert a row into a database

```
private static final String MOTORIST_INSERT =
    "insert into motorist (id, email, password, " +
    "firstName, lastName) " +
    "values (null, ?, ?, ?, ?);"

public void saveMotorist(Motorist motorist) {
    Connection conn = null;
    PreparedStatement stmt = null;
    try {
        conn = dataSource.getConnection();    ← Opens connection
        stmt = conn.prepareStatement(MOTORIST_INSERT);   ← Creates statement

        stmt.setString(1, motorist.getEmail());
        stmt.setString(2, motorist.getPassword());      | Binds
        stmt.setString(3, motorist.getFirstName());     | parameters
        stmt.setString(4, motorist.getLastName());
```

```
stmt.setString(3, motorist.getFirstName());
stmt.setString(4, motorist.getLastName());
stmt.execute();    ↪ Executes statement
} catch (SQLException e) { | Handles exceptions—
...                                somehow
} finally {
try {
    if(stmt != null) { stmt.close(); } | Cleans up
    if(conn != null) { conn.close(); } resources
} catch (SQLException e) {}
}
```

Holy runaway code, Batman! That's over 20 lines of code to insert a simple object into a database. As far as JDBC operations go, this is about as simple as it gets. So why does it take this many lines to do something so simple? Actually, it doesn't. Only a handful of lines actually do the insert. But JDBC requires that you properly manage connections and statements and somehow handle the `SQLException` that may be thrown.

Now have a look at listing 5.2, where we use traditional JDBC to update a row in the motorist table in the database.

Listing 5.2 Using JDBC to update a row in a database

```

        if(stmt != null) { stmt.close(); }
        if(conn != null) { conn.close(); }
    } catch (SQLException e) {}
}
}

```

At first glance, listing 5.2 may appear to be identical to listing 5.1. In fact, disregarding the SQL String and the line where the statement is created, they are identical. Again, that's a lot of code to do something as simple as update a single row in a database. What's more, that's a lot of repeated code. Ideally, we'd only have to write the lines that are specific to the task at hand. After all, those are the only lines that distinguish listing 5.2 from listing 5.1. The rest is just boilerplate code.

To round out our tour of traditional JDBC, let's see how we might retrieve data out of the database. As you can see in listing 5.3, that's not too pretty, either.

Listing 5.3 Using JDBC to query a row from a database

```

private static final String MOTORIST_QUERY =
    "select id, email, password, firstName, lastName " +
    " from motorist where id=?";

public Motorist getMotoristById(Integer id) {
    Connection conn = null;
    PreparedStatement stmt = null;
    ResultSet rs = null;
    try {
        conn = dataSource.getConnection();           ↗ Opens connection
        stmt = conn.prepareStatement(MOTORIST_QUERY); ↗ Creates statement
        stmt.setInt(1, id);                         ↗ Binds parameters
        rs = stmt.executeQuery();                   ↗ Executes query
        Motorist motorist = null;
        if(rs.next()) {
            motorist = new Motorist();
            motorist.setId(rs.getInt("id"));
            motorist.setEmail(rs.getString("email"));
            motorist.setPassword(rs.getString("password"));
            motorist.setFirstName(rs.getString("firstName"));
            motorist.setLastName(rs.getString("lastName"));
        }
        return motorist;
    } catch (SQLException e) {           ↗ Handles exceptions—
        ...                                somehow
    } finally {
        try {

```

Processes
results

```
    if(rs != null) { rs.close(); }
    if(stmt != null) { stmt.close(); }
    if(conn != null) { conn.close(); }
} catch (SQLException e) {}
}

return null;
}
```

Cleans up resources

That's just about as verbose as the insert and update examples—maybe more. It's like the Pareto principle² flipped on its head; 20 percent of the code is needed to actually query a row while 80 percent is just boilerplate code.

By now you should see that much of JDBC code is boilerplate code for creating connections and statements and exception handling. With our point made, we will end the torture here and not make you look at any more of this nasty, nasty code.

But the fact is that this boilerplate code is important. Cleaning up resources and handling errors is what makes data access robust. Without it, errors would go undetected and resources would be left open, leading to unpredictable code and resource leaks. So not only do we need this code, we also need to make sure that it is correct. This is all the more reason to let a framework deal with the boilerplate code so that we know that it written once and written right.

5.3.2 Working with JDBC templates

Spring's JDBC framework will clean up your JDBC code by shouldering the burden of resource management and exception handling. This leaves you free to write only the code necessary to move data to and from the database.

As we explained in section 5.1.1, Spring abstracts away the boilerplate data access code behind template classes. For JDBC, Spring comes with three template classes to choose from:

- `JdbcTemplate`—The most basic of Spring's JDBC templates, this class provides simple access to a database through JDBC and simple indexed-parameter queries.
- `NamedParameterJdbcTemplate`—This JDBC template class enables you to perform queries where values are bound to named parameters in SQL, rather than indexed parameters.

² http://en.wikipedia.org/wiki/Pareto%27s_principle

- `SimpleJdbcTemplate`—This version of the JDBC template takes advantage of Java 5 features such as autoboxing, generics, and variable parameter lists to simplify how a JDBC template is used.

Which one of these templates you choose is largely a matter of preference. That said, if you’re targeting an older Java runtime environment, `SimpleJdbcTemplate` won’t be available, as it depends on Java 5 features.

To help you decide which of these JDBC templates will be best for you, let’s look at them one by one, starting with `JdbcTemplate`.

Accessing data using `JdbcTemplate`

All a `JdbcTemplate` needs to do its work is a `DataSource`. This makes it easy enough to configure a `JdbcTemplate` bean in Spring with the following XML:

```
<bean id="jdbcTemplate"
      class="org.springframework.jdbc.core.JdbcTemplate">
    <property name="dataSource" ref="dataSource" />
</bean>
```

The actual `DataSource` being referred to by the `dataSource` property can be any implementation of `javax.sql.DataSource`, including those we created in section 5.2.

Now we can wire the `JdbcTemplate` into our DAO and use it to access the database. For example, suppose that the `RoadRantz` DAO is based on `JdbcTemplate`:

```
public class JdbcRantDao implements RantDao {
    ...
    private JdbcTemplate jdbcTemplate;
    public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }
}
```

We would then wire the `jdbcTemplate` property of `JdbcRantDao` as follows:

```
<bean id="rantDao"
      class="com.roaddrantz.dao.jdbc.JdbcRantDao">
    <property name="jdbcTemplate" ref="jdbcTemplate" />
</bean>
```

With a `JdbcTemplate` at our DAO’s disposal, we can greatly simplify the `saveMotorist()` method from listing 5.1. The new `JdbcTemplate`-based `saveMotorist()` method is shown in listing 5.4.

Listing 5.4 A JdbcTemplate-based saveMotorist() method

```

private static final String MOTORIST_INSERT =
    "insert into motorist (id, email, password, " +
    "firstName, lastName) " +
    "values (null, ?, ?, ?, ?)";

public void saveMotorist(Motorist motorist) {
    jdbcTemplate.update(MOTORIST_INSERT,
        new Object[] { motorist.getEmail(), motorist.getPassword(),
            motorist.getFirstName(), motorist.getLastName() });
}

```

Inserts
motorist
data

We think you'll agree that this version of `saveMotorist()` is significantly simpler. There's no more connection or statement creation code—and no more exception-handling code. There's nothing but pure data insertion code.

Actually, all of the boilerplate code is there. Just because you don't see it, that doesn't mean it's not there. It's cleverly hidden inside the `JdbcTemplate`. When the `update()` method is called, `JdbcTemplate` will get a connection, create a statement, and execute the insert SQL.

What you also don't see is how the `SQLException` is handled. Internally, `JdbcTemplate` will catch any `SQLExceptions` that are thrown. It will then translate the generic `SQLException` into one of the more specific data access exceptions from table 5.1 and rethrow it. Because Spring's data access exceptions are all runtime exceptions, we didn't have to catch it in the `saveMotorist()` method.

Reading data is also simplified with `JdbcTemplate`. Listing 5.5 shows a new version of `getMotoristById()` that uses `JdbcTemplate` callbacks to map a result set to domain objects.

Listing 5.5 Querying for a motorist using JdbcTemplate

```

private static final String MOTORIST_SELECT =
    "select id, email, password, firstName, lastName from motorist";
private static final String MOTORIST_BY_ID_SELECT =
    MOTORIST_SELECT + " where id=?";

public Motorist getMotoristById(long id) {
    List matches = jdbcTemplate.query(MOTORIST_BY_ID_SELECT,
        new Object[] { Long.valueOf(id) }, ←
        new RowMapper() { ←
            public Object mapRow(ResultSet rs, int rowNum)
                throws SQLException, DataAccessException {

```

Queries for
motorist

Binds query
parameter

```

        Motorist motorist = new Motorist();

        motorist.setId(rs.getInt(1));
        motorist.setEmail(rs.getString(2));
        motorist.setPassword(rs.getString(3));
        motorist.setFirstName(rs.getString(4));
        motorist.setLastName(rs.getString(5));
        return motorist;
    }
});

return matches.size() > 0 ? (Motorist) matches.get(0) : null;
}

```

Maps query results to Motorist object

This `getMotoristById()` method uses `JdbcTemplate`'s `query()` method to query a `Motorist` from the database. The `query()` method takes three parameters:

- A String containing the SQL to be used to select the data from the database
- An array of Object that contains values to be bound to indexed parameters of the query
- A `RowMapper` object that extracts values from a `ResultSet` and constructs a domain object (in this case a `Motorist`)

The real magic happens in the `RowMapper` object. For every row that results from the query, `JdbcTemplate` will call the `mapRow()` method of the `RowMapper`. Within `RowMapper`, we've written the code that creates a `Motorist` object and populates it with values from the `ResultSet`.

`getMotoristById()` is a bit lengthier than the `saveMotorist()` method. Even so, it's still focused on retrieving a `Motorist` object from the database. Unlike traditional JDBC, there's no resource management or exception-handling code.

Using named parameters

The `saveMotorist()` method in listing 5.4 used indexed parameters. This meant that we had to take notice of the order that the parameters in the query and list the values in the correct order when passing them to the `update()` method. If we were to ever change the SQL in such a way that the order of the parameters would change, we'd also need to change the order of the values.

Optionally, we could use named parameters. Named parameters let us give each parameter in the SQL an explicit name and to refer to the parameter by that name when binding values to the statement. For example, suppose that the `MOTORIST_INSERT` query were defined as follows:

```
private static final String MOTORIST_INSERT =
    "insert into motorist (id, email, password, " +
    "firstName, lastName) " +
    "values (null, :email, :password, :firstName, :lastName);
```

With named parameter queries, the order of the bound values isn't important. We can bind each value by name. If the query changes and the order of the parameters is no longer the same, we won't have to change the binding code.

Unfortunately, `JdbcTemplate` doesn't support named parameters. Instead, we'll need to use a special JDBC template called `NamedParameterJdbcTemplate`. It is configured in the Spring configuration XML similarly to `JdbcTemplate`:

```
<bean id="jdbcTemplate"
      class="org.springframework.jdbc.core.
          ↗ namedparam.NamedParameterJdbcTemplate">
    <property name="dataSource" ref="dataSource" />
</bean>
```

Because `NamedParameterJdbcTemplate` is a special JDBC template and because it isn't a subclass of `JdbcTemplate`, we'll need to change the `jdbcTemplate` property in our DAO to match the new template class:

```
public class JdbcRantDao implements RantDao {
    ...
    private NamedParameterJdbcTemplate jdbcTemplate;
    public void setJdbcTemplate(
        NamedParameterJdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }
}
```

Now we're ready to update our `saveMotorist()` method to use named parameters. Listing 5.6 shows the new named-parameter version of `saveMotorist()`.

Listing 5.6 Using named parameters with Spring JDBC templates

```
public void saveMotorist(Motorist motorist) {
    Map parameters = new HashMap();
    parameters.put("email", motorist.getEmail());
    parameters.put("password", motorist.getPassword());
    parameters.put("firstName", motorist.getFirstName());
    parameters.put("lastName", motorist.getLastName());

    jdbcTemplate.update(MOTORIST_INSERT, parameters); ↪
}
```

**Binds
parameter
values**

**Performs
insert**

The first thing you'll notice is that this version of `saveMotorist()` is a bit longer than the previous version. That's because named parameters are bound through a

`java.util.Map`. Nevertheless, every line is focused on the goal of inserting a `Motorist` object into the database. There's still no resource management or exception-handling code cluttering up the chief purpose of the method.

Simplifying JDBC in Java 5

Spring provides one more specialized JDBC template that you may want to consider using. If you take another look at listing 5.4, you'll see that the parameters passed to `update()` were passed in as an `Object` array. That's a typical strategy used to pass variable-length parameter lists to a method. With Java 5's new language constructs (known as `varargs`), it is possible to pass variable-length parameter lists without having to construct an array of `Object`.

Taking advantage of Java 5 `varargs` means that the `saveMotorist()` method can be further simplified as follows:

```
public void saveMotorist(Motorist motorist) {  
    jdbcTemplate.update(MOTORIST_INSERT,  
        motorist.getEmail(), motorist.getPassword(),  
        motorist.getFirstName(), motorist.getLastName());  
}
```

The `jdbcTemplate` in this new `saveMotorist()` method isn't a standard `JdbcTemplate` object. Instead, it's a special JDBC template, `SimpleJdbcTemplate`, that takes advantage of some of Java 5's syntax features. We configure the `SimpleJdbcTemplate` bean in Spring much like the regular `JdbcTemplate` bean:

```
<bean id="jdbcTemplate"  
    class="org.springframework.jdbc.core.simple.SimpleJdbcTemplate">  
    <property name="dataSource" ref="dataSource" />  
</bean>
```

In addition, we'll need to change the type of the `jdbcTemplate` property to be a `SimpleJdbcTemplate` instead of `JdbcTemplate`:

```
public class JdbcRantDao implements RantDao {  
    ...  
    private SimpleJdbcTemplate jdbcTemplate;  
    public void setJdbcTemplate(SimpleJdbcTemplate jdbcTemplate) {  
        this.jdbcTemplate = jdbcTemplate;  
    }  
}
```

Because `SimpleJdbcTemplate`'s power comes from its use of Java 5 features, it will only work when deployed to a Java 5 runtime.

SimpleJdbcTemplate does more than provide varargs support for binding parameter values. It also takes advantage of Java 5's support for autoboxing when mapping result sets.

Take another look at the getMotoristById() method in listing 5.5. Two things you should take note of are:

- The id parameter had to be converted to a wrapped type (`java.lang.Long`) to be passed in the array of `Object`.
- The return type of the RowMapper's `mapRow()` method is `java.lang.Object`. That's because the RowMapper is meant to be generic enough to support any type of object. The consequence is that the result must be returned as the most generic type of all: `Object`.

Now consider the new version of `getMotoristById()` in listing 5.7 that uses `SimpleJdbcTemplate` to do its work.

Listing 5.7 Using SimpleJdbcTemplate to retrieve a Motorist from the database

```
public Motorist getMotoristById(long id) {
    List<Motorist> matches = getSimpleJdbcTemplate().query(
        MOTORIST_BY_ID_SELECT,
        new ParameterizedRowMapper<Motorist>() {
            public Motorist mapRow(ResultSet rs, int rowNum)
                throws SQLException {
                Motorist motorist = new Motorist();
                motorist.setId(rs.getInt(1));
                motorist.setEmail(rs.getString(2));
                motorist.setPassword(rs.getString(3));
                motorist.setFirstName(rs.getString(4));
                motorist.setLastName(rs.getString(5));
                return motorist;
            }
        },
        id // Shows id isn't wrapped
    );
    return matches.size() > 0 ? matches.get(0) : null;
}
```

>Returns
Motorist

The differences between listing 5.5 and listing 5.7 are subtle. The first difference to note is that we're using `ParameterizedRowMapper` to map the results to an object. `ParameterizedRowMapper` takes advantage of Java 5 covariant return types

to specify a specific return type for the `mapRow()` method. In other words, this row mapper knows that it's dealing with a `Motorist` and not just an `Object`.

The other difference is that the `id` parameter no longer needs to be wrapped in a `Long` object and then wrapped again in an array of `Object`. That's because `SimpleJdbcTemplate` takes advantage of Java 5 autoboxing and varargs to automatically convert the `id` parameter to a boxed `Long`.

One more slight difference is that the order of the parameters passed to the `query()` method are different for `SimpleJdbcTemplate` than they are for `JdbcTemplate`. Because `query()` can take query parameters as varargs, the query parameters had to be moved to the end of the parameter list to avoid confusion. Otherwise, it'd be tricky for the `query()` method to know when the list of varargs ends and the row mapper argument begins.

As you've seen thus far, writing a JDBC-based DAO involves configuring a JDBC template bean, wiring it into your DAO class, and then using the template to access the database. This process involves configuring at least three beans in the Spring configuration file: a data source, a template, and a DAO. Let's see how to cut out one of those beans from the configuration XML by using Spring's JDBC DAO support.

5.3.3 Using Spring's DAO support classes for JDBC

For all of an application's JDBC-backed DAO classes, we'll need to be sure to add a `JdbcTemplate` property and setter method. And we'll need to be sure to wire the `JdbcTemplate` bean into the `JdbcTemplate` property of each DAO. That's not a big deal if the application only has one DAO, but if you have multiple DAOs, that's a lot of repeated code.

One solution would be for you to create a common parent class for all your DAO objects where the `JdbcTemplate` property resides. Then all of your DAO classes would extend that class and use the parent class's `JdbcTemplate` for its data access. Figure 5.5 shows the proposed relationship between an application DAO and the base DAO class.

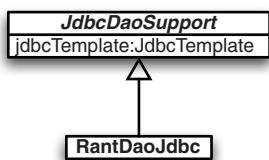


Figure 5.5
Spring's `JdbcDaoSupport` defines a placeholder for the `JdbcTemplate` object so that subclasses won't have to manage their own `JdbcTemplate`.

The idea of creating a base DAO class that holds the `JdbcTemplate` is such a good idea that Spring comes with just such a base class out of the box. Spring's `JdbcDaoSupport` is a base class for writing JDBC-backed DAO classes. To use `JdbcDaoSupport`, simply write your DAO class to extend it. For example, the JDBC-based DAO for the RoadRantz application might be written like this:

```
public class JdbcRantDao extends JdbcDaoSupport
    implements RantDao {
    ...
}
```

The `JdbcDaoSupport` provides convenient access to the `JdbcTemplate` through the `getJdbcTemplate()` method. For example, the `saveMotorist()` method may be written like this:

```
public void saveMotorist(Motorist motorist) {
    getJdbcTemplate().update(MOTORIST_INSERT,
        new Object[] { motorist.getEmail(), motorist.getPassword(),
            motorist.getFirstName(), motorist.getLastName() });
}
```

When configuring your DAO class in Spring, you could directly wire a `JdbcTemplate` bean into its `jdbcTemplate` property as follows:

```
<bean id="rantDao" class="com.roaddrantz.dao.jdbc.JdbcRantDao">
    <property name="jdbcTemplate" ref="jdbcTemplate" />
</bean>
```

This will work, but it isn't much different from how we configured the DAO that didn't extend `JdbcDaoSupport`. Alternatively, we can skip the middleman (or middle bean, as the case may be) and wire a data source directly into the `dataSource` property that `JdbcRantDao` inherits from `JdbcDaoSupport`:

```
<bean id="rantDao" class="com.roaddrantz.dao.jdbc.JdbcRantDao">
    <property name="dataSource" ref="dataSource" />
</bean>
```

When `JdbcRantDao` has its `dataSource` property configured, it will internally create a `JdbcTemplate` instance for you. This eliminates the need to explicitly declare a `JdbcTemplate` bean in Spring.

DAO support for named parameters

In section 5.2.2 we showed you some variations on the `JdbcTemplate` concept. One variation, `NamedParameterJdbcTemplate`, offers the option to use named parameter in queries instead of indexed parameters. If you'd like to use named

parameters with your queries, you can use Spring's `NamedParameterJdbcDaoSupport` as the parent class for your DAOs.

For example, if we want to use named parameter queries in the `RoadRant` application, we could write `JdbcRantDao` to extend `NamedParameterJdbcDaoSupport` as follows:

```
public class JdbcRantDao extends NamedParameterJdbcDaoSupport
    implements RantDao {
    ...
}
```

Just as with `JdbcDaoSupport`, `NamedParameterJdbcDaoSupport` provides convenient access to the template. However, instead of calling `getJdbcTemplate()`, we'll need to call `getNamedParameterJdbcTemplate()` to retrieve the JDBC template. Here's how the `saveMotorist()` method might look if written to use named-parameter queries:

```
public void saveMotorist(Motorist motorist) {
    Map parameters = new HashMap();
    parameters.put("email", motorist.getEmail());
    parameters.put("password", motorist.getPassword());
    parameters.put("firstName", motorist.getFirstName());
    parameters.put("lastName", motorist.getLastName());
    getNamedParameterJdbcTemplate().update(
        MOTORIST_INSERT, parameters);
}
```

`getNamedParameterJdbcTemplate()` returns a `NamedParameterJdbcDaoSupport` object that we use to perform the update. Just as when we used `NamedParameterJdbcDaoSupport`, the parameters are mapped into the query in a `java.util.Map`.

Simplified Java 5 DAOs

In section 5.2.2, we also showed you how to use a Java 5-savvy JDBC template called `SimpleJdbcTemplate`. If you'd like to exploit the advantages of Java 5's varargs and autoboxing in your DAOs then you may want to write your DAO class to extend `SimpleJdbcDaoSupport`:

```
public class JdbcRantDao extends SimpleJdbcDaoSupport
    implements RantDao {
    ...
}
```

To get access to the `SimpleJdbcTemplate` that is contained within `SimpleJdbcDaoSupport`, simply call the `getSimpleJdbcTemplate()` method. Here's the `saveMotorist()` method, updated to use `SimpleJdbcTemplate`:

```
public void saveMotorist(Motorist motorist) {  
    getSimpleJdbcTemplate().update(MOTORIST_INSERT,  
        motorist.getEmail(), motorist.getPassword(),  
        motorist.getFirstName(), motorist.getLastName());  
}
```

JDBC is the most basic way to access data in a relational database. Spring's JDBC templates save you the hassle of dealing with the boilerplate code that handles connection resources and exception handling, leaving you to focus on the actual work of querying and updating data.

Even though Spring takes much of the pain out of working with JDBC, it can still become somewhat cumbersome as applications grow larger and more complex. To help manage the persistence challenges of large applications, you may want to graduate to a persistence framework such as Hibernate.

5.4 Integrating Hibernate with Spring

When we were kids, riding a bike was fun, wasn't it? We would ride to school in the mornings. When school let out, we would cruise to our best friend's house. When it got late and our parents were yelling at us for staying out past dark, we would peddle home for the night. Gee, those days were fun.

Then we grew up and we needed more than a bike. Sometimes we have to travel quite a distance to work. Groceries have to be hauled, and ours kids need to get to soccer practice. And if you live in Texas, air-conditioning is a must! Our needs have simply outgrown our bike.

JDBC is the bike of the persistence world. It is great for what it does, and for some jobs it works just fine. But as our applications become more complex, so do our persistence requirements. We need to be able to map object properties to database columns and have our statements and queries created for us, freeing us from typing an endless string of question marks. We also need features that are more sophisticated:

- *Lazy loading*—As our object graphs become more complex, we sometimes don't want to fetch entire relationships immediately. To use a typical example, suppose we are selecting a collection of PurchaseOrder objects, and each of these objects contains a collection of LineItem objects. If we are only interested in PurchaseOrder attributes, it makes no sense to grab the LineItem data. This could be quite expensive. Lazy loading allows us to grab data only as it is needed.
- *Eager fetching*—This is the opposite of lazy loading. Eager fetching allows you to grab an entire object graph in one query. In the cases where we know

that we need a `PurchaseOrder` object and its associated `LineItems`, eager fetching lets us get this from the database in one operation, saving us from costly round-trips.

- *Cascading*—Sometimes changes to a database table should result in changes to other tables as well. Going back to our purchase order example, when an `Order` object is deleted, we also want to delete the associated `LineItems` from the database.

Several frameworks are available that provide these services. The general name for these services is object-relational mapping (ORM). Using an ORM tool for your persistence layer can save you literally thousands of lines of code and hours of development time. This lets you switch your focus from writing error-prone SQL code to addressing your application requirements.

Spring provides support for several ORM frameworks, including Hibernate, iBATIS, Apache OJB, Java Data Objects (JDO), Oracle’s TopLink, and the Java Persistence API (JPA).

As with Spring’s JDBC support, Spring’s support for ORM frameworks provides integration points to the frameworks as well as some additional services:

- Integrated support for Spring declarative transactions
- Transparent exception handling
- Thread-safe, lightweight template classes
- DAO support classes
- Resource management

We simply do not have enough space in this chapter to cover all of the ORM frameworks that are supported by Spring. That’s okay, because Spring’s support for one ORM solution is quite similar to the next. Once you get the hang of using one ORM framework with Spring, you’ll find it easy to switch to another one.

Let’s get started by looking at how Spring integrates with what is perhaps the most popular ORM framework in use—Hibernate. Later in this chapter, we’ll also look at how Spring integrates with JPA (in section 5.5) and iBATIS (section 5.6).

Hibernate is an open source persistence framework that has gained significant popularity in the developer community. It provides not only basic object-relational mapping but also all the other sophisticated features you would expect from a full-featured ORM tool, such as caching, lazy loading, eager fetching, and distributed caching.

In this section, we will focus on how Spring integrates with Hibernate, without dwelling too much on the intricate details of using Hibernate. If you need to learn more about working with Hibernate, we recommend either *Java Persistence with Hibernate* (Manning, 2006) or the Hibernate website at <http://www.hibernate.org>.

5.4.1 Choosing a version of Hibernate

At the time of this writing, Hibernate 3.2 is the latest version available. But it wasn't that long ago that Hibernate 2.x was the latest version available, and you may still encounter some applications that haven't moved up to Hibernate 3 yet. The choice between Hibernate 2 and Hibernate 3 is significant because the Hibernate API is quite different between the two versions.

While many functional improvements and features were introduced between Hibernate 2 and 3, one subtle change complicated Spring integration. You see, in version 2 the Hibernate API was packaged under the `net.sf.hibernate` package structure, but was repackaged under `org.hibernate` in version 3.

For the Spring development team, this change presented a dilemma. Because the Spring-Hibernate integration classes needed to import classes from either `net.sf.hibernate` or `org.hibernate`, they were presented with the choice of either:

- Dropping support for Hibernate 2, supporting only Hibernate 3 going forward

or:

- Splitting the Spring-Hibernate support code into two strains—one for Hibernate 2 and one for Hibernate 3

Recognizing the value of backward compatibility, the Spring team decided to split its Spring-Hibernate support into two. The Hibernate 2 support is found within the Spring distribution under the `org.springframework.orm.hibernate` package. Meanwhile, you can find the Hibernate 3 support under the `org.springframework.orm.hibernate3` package.

For the most part, the classes in the Hibernate 3 package mirror those in the Hibernate 2 package. For Hibernate 3, Spring also includes support for annotation-based mapping.

When possible, I recommend the choice of Hibernate 3 over Hibernate 2. The examples in this chapter will reflect that choice. But if your circumstance doesn't afford you the luxury of Hibernate 3, you'll find that Spring's Hibernate 2 integra-

tion doesn't differ from its support for Hibernate 3 much more than a package name (and that Hibernate 2 doesn't offer annotation-based mapping).

Regardless of which Hibernate version you choose, the first thing you'll need to do is to configure a Hibernate session factory bean in Spring.

5.4.2 Using Hibernate templates

The main interface for interacting with Hibernate is `org.hibernate.Session`. The `Session` interface provides basic data access functionality such as the ability to save, update, delete, and load objects from the database. It is through the Hibernate `Session` that an application's DAO will perform all of its persistence needs.

The standard way to get a reference to a Hibernate `Session` object is through an implementation of Hibernate's `SessionFactory` interface. Among other things, `SessionFactory` is responsible for opening, closing, and managing Hibernate Sessions.

Much as Spring's `JdbcTemplate` abstracted away the tedium of working with JDBC, Spring's `HibernateTemplate` provides an abstract layer over a Hibernate `Session`. `HibernateTemplate`'s main responsibility is to simplify the work of opening and closing Hibernate Sessions and to convert Hibernate-specific exceptions to one of the Spring ORM exceptions listed in table 5.1. (In the case of Hibernate 2, this means converting a checked `HibernateException` to an unchecked Spring exception.)

The following XML shows how to configure a `HibernateTemplate` in Spring:

```
<bean id="hibernateTemplate"
      class="org.springframework.orm.hibernate3.HibernateTemplate">
    <property name="sessionFactory" ref="sessionFactory" />
</bean>
```

The `sessionFactory` property takes a reference to an implementation of `org.hibernate.SessionFactory`. Here you have a few options, depending on how you use Hibernate to map your objects to database tables.

Using classic Hibernate mapping files

If you are using Hibernate's classic XML mapping files, you'll want to use Spring's `LocalSessionFactoryBean`. As shown in figure 5.6, `LocalSessionFactoryBean` is a Spring factory bean that produces a local Hibernate `SessionFactory` instance³ that draws its mapping metadata from one or more XML mapping files.

³ When you think about it, this class may be more appropriately named `LocalSessionFactoryBean`. But to avoid stuttering, the ancillary `Factory` was dropped.



Figure 5.6 Spring's `LocalSessionFactoryBean` is a factory bean that loads one or more Hibernate mapping XML files to produce a `Hibernate SessionFactory`.

The following chunk of XML shows how to configure a `LocalSessionFactoryBean` that loads the mapping files for the RoadRantz domain objects:

```
<bean id="sessionFactory"
      class="org.springframework.orm.hibernate3.
      ↗ LocalSessionFactoryBean">
    <property name="dataSource" ref="dataSource" />
    <property name="mappingResources">
      <list>
        <value>com/roaddrantz/domain/Rant.hbm.xml</value>
        <value>com/roaddrantz/domain/Motorist.hbm.xml</value>
        <value>com/roaddrantz/domain/Vehicle.hbm.xml</value>
      </list>
    </property>
    <property name="hibernateProperties">
      <props>
        <prop key="hibernate.dialect">${hibernate.dialect}</prop>
      </props>
    </property>
  </bean>
```

The `dataSource` property refers to any implementation of `javax.sql.DataSource`, including any of those declared in section 5.2. The `mappingResources` property takes a list of one or more paths to mapping files as resources in the classpath. Here we've specified three mapping files that describe the persistence of the RoadRantz application. Finally, the `hibernateProperties` property lets us provide any additional configuration pertinent to the Hibernate session. At minimum, we must specify the Hibernate dialect (that is, how Hibernate constructs its SQL for a particular database). Here we've left the dialect decision as a placeholder variable that will be replaced by `PropertyPlaceholderConfigurer` (see section 3.5.3).

Working with annotated domain objects

When targeting a Java 5 runtime, you may choose to use annotations to tag domain objects with persistence metadata. Hibernate 3 supports both JPA



Figure 5.7 Spring's `AnnotationSessionFactoryBean` produces a `Hibernate SessionFactory` by reading annotations on one or more domain classes.

annotations and Hibernate-specific annotations for describing how objects should be persisted. For annotation-based Hibernate, Spring's `AnnotationSessionFactoryBean` works much like `LocalSessionFactoryBean`, except that it creates a `SessionFactory` based on annotations in one or more domain classes (as shown in figure 5.7).

The XML required to configure an `AnnotationSessionFactoryBean` in Spring is similar to the XML for `LocalSessionFactoryBean`:

```

<bean id="sessionFactory"
      class="org.springframework.orm.hibernate3.
          ↗ annotation.AnnotationSessionFactoryBean">
    <property name="dataSource" ref="dataSource" />
    <property name="annotatedClasses">
        <list>
            <value>com.roaddrantz.domain.Rant</value>
            <value>com.roaddrantz.domain.Motorist</value>
            <value>com.roaddrantz.domain.Vehicle</value>
        </list>
    </property>
    <property name="hibernateProperties">
        <props>
            <prop key="hibernate.dialect">${hibernate.dialect}</prop>
        </props>
    </property>
</bean>
  
```

The `dataSource` and `hibernateProperties` properties serve the same purpose with `AnnotationSessionFactoryBean` as with `LocalSessionFactoryBean`. However, instead of configuring one or more mapping files, we must configure `AnnotationSessionFactoryBean` with one or more classes that are annotated for persistence with Hibernate. Here we've listed the domain objects in the RoadRantz application.

Accessing data through the Hibernate template

With the `HibernateTemplate` bean declared and wired with a session factory, we're ready to start using it to persist and retrieve objects from the database.

Listing 5.8 shows a portion of `HibernateRantDao` that is injected with a `HibernateTemplate`.

Listing 5.8 Creating a `HibernateTemplate`-based DAO

```
public class HibernateRantDao implements RantDao {
    public HibernateRantDao() {}

    ...

    private HibernateTemplate hibernateTemplate;
    public void setHibernateTemplate(HibernateTemplate template) {
        this.hibernateTemplate = template;
    }
}
```

Injects `HibernateTemplate`

`HibernateRantDao` accepts a `HibernateTemplate` reference via setter injection, so we'll need to configure it in Spring as follows:

```
<bean id="rantDao" class="com.roaddrantz.dao.hibernate.HibernateRantDao">
    <property name="hibernateTemplate" ref="hibernateTemplate" />
</bean>
```

As we flesh out the methods in `HibernateRantDao`, we can use the injected `HibernateTemplate` to access objects stored in the database. For example, here's the `saveVehicle()` method that is used to persist a `Vehicle` object to the database:

```
public void saveVehicle(Vehicle vehicle) {
    hibernateTemplate.saveOrUpdate(vehicle);
}
```

Here we're using the `saveOrUpdate()` method of `HibernateTemplate` to save a `Vehicle`. The `saveOrUpdate()` method inspects the object to see if its ID field is null. If it is, it must be a new object and thus it is inserted into the database. If it's not null, it is assumed that it is an existing object and its data is updated.

Here's the `findVehiclesByPlate()` method that uses `HibernateTemplate`'s `find()` method to retrieve a `Vehicle` by querying by the state and license plate number:

```
public Vehicle findVehicleByPlate(String state,
    String plateNumber) {
    List results = hibernateTemplate.find("from " + VEHICLE +
        " where state = ? and plateNumber = ?",
        new Object[] {state, plateNumber});

    return results.size() > 0 ? (Vehicle) results.get(0) : null;
}
```

And here's how you might use `HibernateTemplate`'s `load()` method to load a specific instance of a `Motorist` by the motorist's ID field:

```
public Motorist getMotoristById(Integer id) {  
    return (Motorist) hibernateTemplate.load(Motorist.class, id);  
}
```

These are just examples of three ways that you can use `HibernateTemplate`. `HibernateTemplate` offers several dozen methods that help you query and persist objects through Hibernate. If you're already familiar with the persistence methods available through Hibernate's `Session` interface, you'll be pleased to find most of those methods available with `HibernateTemplate`.

In listing 5.8 we explicitly injected a `HibernateTemplate` into `HibernateRantDao`. That's fine for some cases, but Spring also offers a DAO support class for Hibernate that provides a `HibernateTemplate` for you without explicitly wiring it. Let's rework `HibernateRantDao` to take advantage of Spring's DAO support for Hibernate.

5.4.3 **Building Hibernate-backed DAOs**

So far, the configuration of `HibernateRantDao` involves four beans. The data source is wired into the session factory bean (either `LocalSessionFactoryBean` or `AnnotationSessionFactoryBean`). The session factory bean is wired into the `HibernateTemplate`. Finally, the `HibernateTemplate` is wired into `HibernateRantDao`, where it is used to access the database.

To simplify things slightly, Spring offers `HibernateDaoSupport`, a convenience DAO support class, that enables you to wire a session factory bean directly into the DAO class. Under the covers, `HibernateDaoSupport` creates a `HibernateTemplate` that the DAO can use, as the UML in figure 5.8 illustrates.

Let's rework `HibernateRantDao` to use `HibernateDaoSupport`. The first step is to change `HibernateRantDao` to extend `HibernateDaoSupport`:

```
public class HibernateRantDao extends HibernateDaoSupport  
    implements RantDao {  
    ...  
}
```

`HibernateRantDao` no longer needs a `HibernateTemplate` property as it did in listing 5.8. Instead, you can use the `getHibernateTemplate()` method to get a `HibernateTemplate` that `HibernateDaoSupport` creates for you. So, the next step is to change all the data access methods in `HibernateRantDao` to use `getHibernateTemplate()`. For example, here's the `saveMotorist()` method updated for the new `HibernateDaoSupport`-based `HibernateRantDao`:

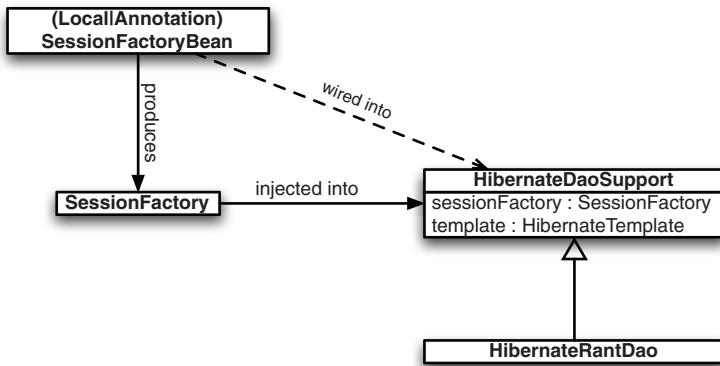


Figure 5.8 **HibernateDaoSupport** is a convenient superclass for a Hibernate-based DAO that provides a **HibernateTemplate** created from an injected **SessionFactory**.

```

public void saveMotorist(Motorist motorist) {
    getHibernateTemplate().saveOrUpdate(motorist);
}
  
```

The last thing that's left to do is rewire the **HibernateRantDao** in the Spring configuration. Since **HibernateRantDao** no longer needs a **HibernateTemplate** reference, we'll remove the **hibernateTemplate** bean. Instead of a **HibernateTemplate**, **HibernateRantDao**'s new parent, **HibernateDaoSupport**, does need a **Hibernate SessionFactory** so that it can produce a **HibernateTemplate** internally. So, we'll wire the **sessionFactory** bean into the **sessionFactory** property of **HibernateRantDao**:

```

<bean id="rantDao" class="com.roaddrantz.dao.hibernate.HibernateRantDao">
    <property name="sessionFactory" ref="sessionFactory" />
</bean>
  
```

At this point, we have created a Hibernate-based DAO for the RoadRantz application and have wired it up in Spring. Aside from transaction handling (which we'll cover in the next chapter), you know almost everything there is to know about using Hibernate within Spring.

But notice that **HibernateRantDao** extends a Spring-specific class. This may not be a problem for you, but there are some people who would think of this as an intrusion of Spring into their application code. For that reason, let's look at a way to take advantage of Hibernate 3's support for contextual sessions to remove Spring-specific dependencies from your DAOs.

5.4.4 Using Hibernate 3 contextual sessions

One of the responsibilities of `HibernateTemplate` is to manage Hibernate Sessions. This involves opening and closing sessions as well as ensuring one session per transaction. Without `HibernateTemplate`, you'd have no choice but to clutter your DAOs with boilerplate session management code.

The downside of `HibernateTemplate` is that it is somewhat intrusive. When we use Spring's `HibernateTemplate` in our DAO (whether directly or through `HibernateDaoSupport`), the `HibernateRantDao` class is coupled to the Spring API. Although this may not be of much concern to some developers, others may find Spring's intrusion into their DAO code undesirable.

There is another option, however. Contextual sessions, introduced in Hibernate 3, are a way in which Hibernate itself manages one Session per transaction. There's no need for `HibernateTemplate` to ensure this behavior. So, instead of wiring a `HibernateTemplate` into your DAO, you can wire a Hibernate SessionFactory instead, as shown in figure 5.9.

To illustrate, consider this new Spring-free version of `HibernateRantDao`:

```
public class HibernateRantDao implements RantDao {
    ...
    private SessionFactory sessionFactory;
    public void setSessionFactory(SessionFactory sessionFactory) {
        this.sessionFactory = sessionFactory;
    }
}
```

In this new `HibernateRantDao`, a `SessionFactory` reference is injected into the `sessionFactory` property. Since `SessionFactory` comes from the Hibernate API, `HibernateRantDao` no longer depends on the Spring Framework. Instead of using `HibernateTemplate` to perform persistence operations, you now ask the `SessionFactory` for the current session.

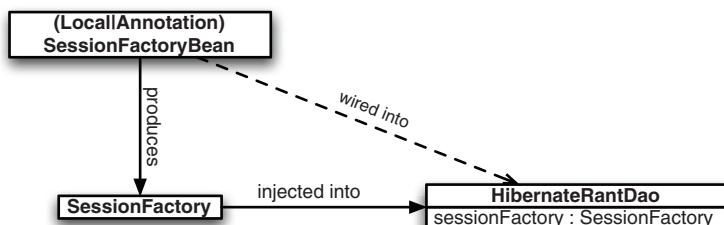


Figure 5.9 Taking advantage of Hibernate 3 contextual sessions, we can wire a `SessionFactory` (produced by a session factory bean) directly into a DAO, thus decoupling the DAO class from the Spring API.

For example, here's the `saveRant()` method updated to use Hibernate 3's contextual sessions:

```
public void saveRant(Rant rant) {  
    sessionFactory.getCurrentSession().saveOrUpdate(rant);  
}
```

When it comes to configuring the `HibernateRantDao` in Spring, it's no different than how we configured it for the `HibernateDaoSupport`-based version of `HibernateRantDao`. Both `HibernateDaoSupport` and our new pure-Hibernate version of `HibernateRantDao` require a Hibernate SessionFactory to be wired into their `sessionFactory` property. In either case, the `sessionFactory` bean (which is a `SessionFactory`-producing `LocalSessionFactoryBean` or `AnnotationSessionFactoryBean`) is suitable:

```
<bean id="rantDao" class="com.roaddrantz.dao.hibernate.HibernateRantDao">  
    <property name="sessionFactory" ref="sessionFactory" />  
</bean>
```

Now we have two options for creating Hibernate-based DAOs in our Spring applications: `HibernateTemplate` and contextual sessions. How do we choose? Here are some things to consider when making that choice:

- Certainly, if you're using Hibernate 2 then you have no other option than to use `HibernateTemplate`.
- The main benefit of Hibernate contextual sessions is that they decouple your DAO implementations from Spring.
- The primary drawback of contextual sessions is that they throw Hibernate-specific exceptions. Although `HibernateException` is a runtime exception, the exception hierarchy is specific to Hibernate and not as ORM-agnostic as Spring's persistence exception hierarchy. This may hinder migration to a different ORM solution.

Despite several attempts to come up with a standard persistence framework, including EJB entity beans and Java Data Objects (JDO), Hibernate has taken the position of the de facto persistence standard in the Java community. Even with Hibernate's unparalleled popularity, history may show that it ultimately sets the stage for a true persistence standard: the Java Persistence API (JPA).

The good news is that Spring's ORM abstraction APIs aren't limited to Hibernate. Spring also provides an abstraction API for JPA that mirrors that for Hibernate. Our survey of Spring's integration with persistence frameworks continues

in the next section with a discussion of how to use Spring with the Java Persistence API.

5.5 **Spring and the Java Persistence API**

From its beginning, the EJB specification has included the concept of entity beans. In EJB, entity beans are a type of EJB that describes business objects that are persisted in a relational database. Entity beans have undergone several tweaks over the years, including bean-managed persistence (BMP) entity beans and container-managed persistence (CMP) entity beans.

Entity beans both enjoyed the rise and suffered the fall of EJB's popularity. In recent years, developers have traded in their heavyweight EJBs for simpler POJO-based development. This presented a challenge to the Java Community Process to shape the new EJB specification around POJOs. The result is JSR-220—also known as EJB 3.

The portion of the EJB 3 specification that replaces entity beans is known as the Java Persistence API (JPA). JPA is a POJO-based persistence mechanism that draws ideas from both Hibernate and Java Data Objects (JDO) and mixes Java 5 annotations in for good measure.

With the Spring 2.0 release came the premiere of Spring integration with JPA. The irony is that many blame (or credit) Spring with the demise of EJB. But now that Spring provides support for JPA, many developers are recommending JPA for persistence in Spring-based applications. In fact, some say that Spring-JPA is the dream team for POJO development.

Spring's JPA support mirrors the template-based support Spring provides for the other persistence frameworks. Therefore, let's get started with Spring and JPA by looking at Spring's `JpaTemplate`.

5.5.1 **Using JPA templates**

Keeping consistent with Spring's support for other persistence solutions, the central element of Spring-JPA integration is a template class. `JpaTemplate`, specifically, is a template class that wraps a JPA `EntityManager`. The following XML configures a JPA template in Spring:

```
<bean id="jpaTemplate"
    class="org.springframework.orm.jpa.JpaTemplate">
    <property name="entityManagerFactory"
        ref="entityManagerFactory" />
</bean>
```

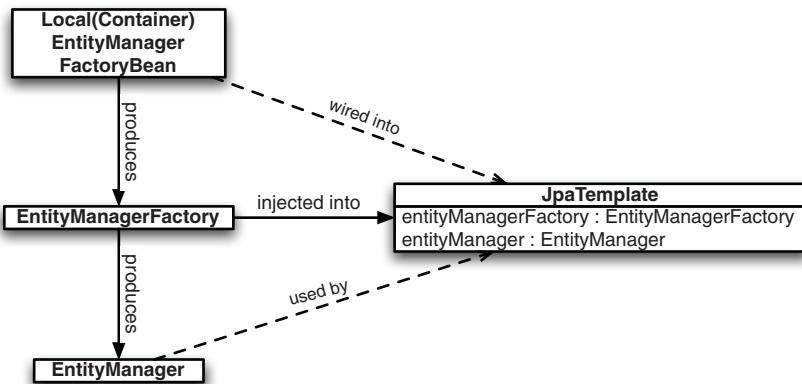


Figure 5.10 Spring’s JpaTemplate templates JPA data access, ensuring that EntityManagers are opened and closed as necessary, handling exceptions, and involving EntityManagers in Spring transactions.

The `entityManagerFactory` property of `JpaTemplate` must be wired with an implementation of JPA’s `javax.persistence.EntityManagerFactory` interface, as shown in figure 5.10. `JpaTemplate` will use the `EntityManagerFactory` to produce `EntityManager`s as needed. I’ll show you where the `entityManagerFactory` bean comes from in section 5.5.2.

Similar to Spring’s other persistence templates, `JpaTemplate` exposes many of the same data access methods provided by a native JPA `EntityManager`. But unlike a plain JPA, `JpaTemplate` ensures that `EntityManager`s are opened and closed as necessary, involves the `EntityManager`s in transactions, and handles exceptions.

To write a `JpaTemplate`-based DAO, add a `JpaTemplate` property to the DAO and provide a setter for injection. Here’s an excerpt from `JpaRantDao` showing the `JpaTemplate` property:

```

public class JpaRantDao implements RantDao {
    public JpaRantDao() {}

    ...

    // injected
    private JpaTemplate jpaTemplate;
    public void setJpaTemplate(JpaTemplate jpaTemplate) {
        this.jpaTemplate = jpaTemplate;
    }
}
  
```

When configuring JpaRantDao in Spring, we simply wire the JpaTemplate into the jpaTemplate property:

```
<bean id="rantDao"
      class="com.roaddrantz.dao.jpa.JpaRantDao">
    <property name="jpaTemplate" ref="jpaTemplate" />
</bean>
```

With the JpaTemplate injected into the DAO, we're now ready to use the template to access persisted objects.

Accessing data through the JPA template

As we mentioned, JpaTemplate provides many of the same persistence methods that are provided by JPA's EntityManager. This should make working with JpaTemplate second nature if you're already familiar with JPA. For example, the following implementation of the saveMotorist() method uses JpaTemplate's persist() method to save a Motorist object to the database:

```
public void saveMotorist(Motorist motorist) {
    jpaTemplate.persist(motorist);
}
```

In addition to the standard set of methods provided by EntityManager, JpaTemplate also provides some convenience methods for data access. For example, consider the following getRantsForDay() method that uses a native JPA EntityManager to find all of the Rant objects that were entered on a given day:

```
public List<Rant> getRantsForDay(Date day) {
    Query query = entityManager.createQuery(
        "select r from Rant r where r.date=?1");
    query.setParameter(1, day);
    return query.getResultList();
}
```

The first thing getRantsForDay() has to do is create a Query object. Then it sets the query parameters. In this case, there's only one query parameter, but you can imagine that a much more interesting example would involve one call to setParameter() for each parameter. Finally, the query is executed to retrieve the results.

Contrast that method with the following implementation of getRantsForDay():

```
public List<Rant> getRantsForDay(Date day) {
    return jpaTemplate.find(
        "select r from Rant r where r.date=?1", day);
}
```

In this version, `getRantsForDay()` takes advantage of a convenient `find()` method offered by `JpaTemplate`. `EntityManager` doesn't have such a simple `find()` method that takes a query and one or more parameters. Under the covers, `JpaTemplate`'s `find()` method creates and executes the `Query` for you, saving you a couple of lines of code.

The one unanswered question is where we get the `entityManagerFactory` bean that we wired into the `JpaTemplate`. Before we see what else Spring has to offer with regard to JPA integration, let's configure the `entityManagerFactory` bean.

5.5.2 Configuring an entity manager factory

In a nutshell, JPA-based applications use an implementation of `EntityManagerFactory` to get an instance of an `EntityManager`. The JPA specification defines two kinds of entity managers:

- *Application-managed*—entity managers are created when an application directly requests an entity manager from an entity manager factory. With application-managed entity managers, the application is responsible for opening or closing entity managers and involving the entity manager in transactions. This type of entity manager is most appropriate for use in stand-alone applications that do not run within a Java EE container.
- *Container-managed*—entity managers are created and managed by a Java EE container. The application does not interact with the entity manager factory at all. Instead, entity managers are obtained directly through injection or from JNDI. The container is responsible for configuring the entity manager factories. This type of entity manager is most appropriate for use by a Java EE container that wants to maintain some control over JPA configuration beyond what is specified in `persistence.xml`.

Both kinds of entity manager implement the same `EntityManager` interface. The key difference is not in the `EntityManager` itself, but rather in how the `EntityManager` is created and managed. Application-managed `EntityManagers` are created by an `EntityManagerFactory` obtained by calling the `createEntityManagerFactory()` method of the `PersistenceProvider`. Meanwhile, container-managed `EntityManagerFactories` are obtained through `PersistenceProvider`'s `createContainerEntityManagerFactory()` method.

So what does this all mean for Spring developers wanting to use JPA? Actually, not much. Regardless of which variety of `EntityManagerFactory` you want to use, Spring will take responsibility for managing `EntityManagers` for you. If using an

application-managed entity manager, Spring plays the role of an application and transparently deals with the `EntityManager` on your behalf. In the container-managed scenario, Spring plays the role of the container.

Each flavor of entity manager factory is produced by a corresponding Spring factory bean:

- `LocalEntityManagerFactoryBean` produces an application-managed `EntityManagerFactory`.
- `LocalContainerEntityManagerFactoryBean` produces a container-managed `EntityManagerFactory`.

It's important to point out that the choice made between an application-managed `EntityManagerFactory` and a container-managed `EntityManagerFactory` is completely transparent to a Spring-based application. Spring's `JpaTemplate` hides the intricate details of dealing with either form of `EntityManagerFactory`, leaving your data access code to focus on its true purpose: data access.

The only real difference between application-managed and container-managed entity manager factories, as far as Spring is concerned, is how each is configured within the Spring application context. Let's start by looking at how to configure the application-managed `LocalEntityManagerFactoryBean` in Spring. Then we'll see how to configure a container-managed `LocalContainerEntityManagerFactoryBean`.

Configuring application-managed JPA

Application-managed entity manager factories derive most of their configuration information from a configuration file called `persistence.xml`. This file must appear in the `META-INF` directory within the classpath.

The purpose of the `persistence.xml` file is to define one or more persistence units. A persistence unit is a grouping of one or more persistent classes that correspond to a single data source. In simple terms, `persistence.xml` enumerates one or more persistent classes along with any additional configuration such as data sources and XML-based mapping files. Here's a typical example of a `persistence.xml` file as it pertains to the RoadRantz application:

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
    version="1.0">
    <persistence-unit name="rantzPU">
        <class>com.roaddrantz.domain.Motorist</class>
        <class>com.roaddrantz.domain.Rant</class>
        <class>com.roaddrantz.domain.Vehicle</class>
        <properties>
```

```
<property name="toplink.jdbc.driver"
      value="org.hsqldb.jdbcDriver" />
<property name="toplink.jdbc.url" value=
      "jdbc:hsqldb:hsq://localhost/roadrantz/roadrantz" />
<property name="toplink.jdbc.user"
      value="sa" />
<property name="toplink.jdbc.password"
      value=" " />
</properties>
</persistence-unit>
</persistence>
```

Because so much configuration goes into a persistence.xml file, there's very little configuration that's required (or even possible) in Spring. The `<bean>` in listing 5.9 declares a LocalEntityManagerFactoryBean in Spring.

Listing 5.9 Configuring an application-managed EntityManagerFactory factory bean

```
<bean id="entityManagerFactory"
  class="org.springframework.orm.jpa.
    LocalEntityManagerFactoryBean">
  <property name="persistenceUnitName" value="rantzPU" />
</bean>
```

Selects persistence unit

The value given to the `persistenceUnitName` property refers to the persistence unit name as it appears in `persistence.xml`.

The reason why much of what goes into creating an application-managed `EntityManagerFactory` is contained in `persistence.xml` has everything to do with what it means to be application managed. In the application-managed scenario (not involving Spring), an application is entirely responsible for obtaining an `EntityManagerFactory` through the JPA implementation's `PersistenceProvider`. The application code would become incredibly bloated if it had to define the persistence unit every time it requested an `EntityManagerFactory`. By specifying it in `persistence.xml`, JPA can look in this well-known location for persistence unit definitions.

But with Spring's support for JPA, the `JpaTemplate` will be the one that interacts with the `PersistenceProvider`—not our application code. Therefore, it seems a bit silly to extract configuration information into `persistence.xml`. In fact, it prevents us from configuring the `EntityManagerFactory` in Spring (so that, for example, we can provide a Spring-configured data source).

For that reason, we should turn our attention to container-managed JPA.

Configuring container-managed JPA

Container-managed JPA takes a slightly different approach. When running within a container, an `EntityManagerFactory` can be produced using information provided by the container. This form of JPA is intended for use in JEE application servers (such as WebLogic or JBoss) where data source information will be configured through the application server's configuration.

Nevertheless, container-managed JPA is also possible with Spring. Instead of configuring data source details in `persistence.xml`, you can configure this information in the Spring application context. For example, listing 5.10 shows how to configure container-managed JPA in Spring using `LocalContainerEntityManagerFactoryBean`.

Listing 5.10 Configuring a container-managed EntityManagerFactory factory bean

```
<bean id="entityManagerFactory"
      class="org.springframework.orm.jpa.
          ↪ LocalContainerEntityManagerFactoryBean">

    <property name="dataSource" ref="dataSource" /> ← OpenConfigures data
                                                    sources connection

    <property name="jpaVendorAdapter">
        <bean class=
            "org.springframework.orm.jpa.vendor.
                ↪ TopLinkJpaVendorAdapter">
            <property name="showSql" value="true"/>
            <property name="generateDdl" value="true"/>
            <property name="database" value="HSQL"/>
        </bean>
    </property>

    <property name="loadTimeWeaver">
        <bean class="org.springframework.instrument.classloading.
                    ↪ SimpleLoadTimeWeaver" />
    </property>
</bean>
```

Configures JPA vendor-specifics

Specifies load-time weaver

Here we've configured the `dataSource` property with a Spring-configured data source. Any implementation of `javax.sql.DataSource` is appropriate, such as those that we configured in section 5.2. Although a data source may still be configured in `persistence.xml`, the data source specified through this property takes precedence.

The `jpaVendorAdapter` property can be used to provide specifics about the particular JPA implementation to use. In this case, we're using TopLink Essentials, so we've configured it with a `TopLinkJpaVendorAdapter`. Several properties are set on the vendor adapter, but the most important one is the `database` property, where we've specified the Hypersonic database as the database we'll be using. Other values supported for this property include those listed in table 5.5.

Certain dynamic persistence features require that the class of persistent objects be modified with instrumentation to support the feature. Objects whose properties are lazily loaded (that is, they will not be retrieved from the database until they are actually accessed) must have their class instrumented with code that knows to retrieve unloaded data upon access. Some frameworks use dynamic proxies to implement lazy loading. Others, such as JDO, perform class instrumentation at compile time.

JPA allows for load-time instrumentation of persistent classes so that a class is modified with dynamic persistence features as the class is loaded. The `loadTimeWeaver` property of `LocalContainerEntityManagerFactoryBean` lets us specify how the dynamic persistence features are woven into the persistent class. In this case, we've chosen Spring's `SimpleLoadTimeWeaver`.

Which entity manager factory bean you choose will depend primarily on how you will use it. For simple applications, `LocalEntityManagerFactoryBean` may be

Table 5.5 The TopLink vendor adapter supports several databases. You can specify which database to use by setting its `database` property.

Database platform	Value for <code>database</code> property
IBM DB2	DB2
Hypersonic	HSQSL
Informix	INFORMIX
MySQL	MYSQL
Oracle	ORACLE
PostgresQL	POSTGRESQL
Microsoft SQL Server	SQLSERVER
Sybase	SYBASE

sufficient. But because `LocalContainerEntityManagerFactoryBean` enables us to configure more of JPA in Spring, it is an attractive choice and likely the one that you'll choose for production use.

5.5.3 Building a JPA-backed DAO

Previously, we wired a reference to an entity manager factory bean into a `JpaTemplate` and then wired the `JpaTemplate` into our DAO. But Spring's `JpaDaoSupport` simplifies things a bit further by making it possible to wire the entity manager factory bean directly into our DAO class.

`JpaDaoSupport` provides the same convenience for JPA-backed DAOs as `JdbcDaoSupport` and `HibernateDaoSupport` provided for JDBC-backed and Hibernate-backed DAOs, respectively. As shown in figure 5.11, a JPA-backed DAO class extends `JpaDaoSupport` and is injected with an `EntityManagerFactory` (which may be produced by an `EntityManagerFactoryBean`). Under the covers, `JpaDaoSupport` creates a `JpaTemplate` and makes it available to the DAO for data access. To take advantage of Spring's JPA DAO support, we will write `JpaRantDao` to subclass `JpaDaoSupport`:

```
public class JpaRantDao extends JpaDaoSupport
    implements RantDao {
    ...
}
```

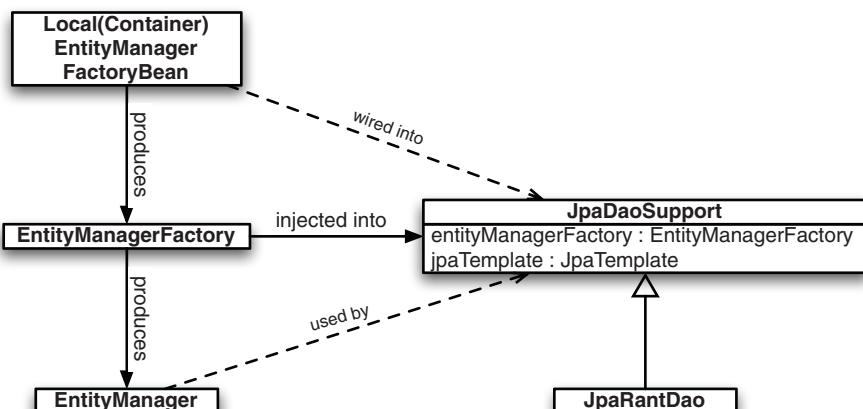


Figure 5.11 `JpaDaoSupport` is a convenient superclass for JPA-backed DAO classes. It is wired with an `EntityManagerFactory` (produced by an `EntityManager` factory bean) and make a `JpaTemplate` available for data access.

Now, instead of wiring `JpaRantDao` with a `JpaTemplate` reference, we'll wire it directly with the `entityManagerFactory` bean:

```
<bean id="rantDao" class="com.rodrantz.dao.jpa.JpaRantDao">
    <property name="entityManagerFactory"
        ref="entityManagerFactory" />
</bean>
```

Internally, `JpaDaoSupport` will use the entity manager factory wired into the `entityManagerFactory` property to create a `JpaTemplate`. As we flesh out the implementation of `JpaRantDao`, we can use the `JpaTemplate` by calling `getJpaTemplate()`. For example, the following reimplementations of `saveMotorist()` uses `JpaDaoSupport`'s `getJpaTemplate()` method to access the `JpaTemplate` and to persist a `Motorist` object:

```
public void saveMotorist(Motorist motorist) {
    getJpaTemplate().persist(motorist);
}
```

Both Hibernate and JPA are great solutions for object-relational mapping. Through ORM, the gory details of data access—SQL statements, database connections, and result sets—are hidden and we can deal with data persistence at the object level. However, although ORM hides data access specifics, it also hinders (or even prevents) fine-grained control of how persistence is handled.

At the other end of the spectrum is JDBC. With JDBC, you have complete control over data access. But with this control comes complete responsibility for the tedium of connection management and mapping result sets to objects.

Next up, let's have a look at how Spring integrates with iBATIS, a persistence framework that strikes a balance between the absolute control of JDBC and the transparent mapping of ORM.

5.6 Spring and iBATIS

Somewhere in between pure JDBC and ORM is where iBATIS resides. iBATIS is often classified among ORM solutions such as Hibernate and JPA, but I prefer to refer to it as an object-query mapping (OQM) solution. Although the iBATIS feature set overlaps that of ORM in many ways, iBATIS puts you in full control of the actual SQL being performed. iBATIS will still take responsibility for mapping query results to domain objects, but you are free to author the queries in any manner that suits you best.

Spring offers integration with iBATIS that mirrors that of its integration with JDBC and ORM frameworks. As with the other persistence frameworks described

in this chapter, we're going to keep our focus on how Spring integrates with iBATIS. If you'd like to learn more about iBATIS, I recommend you check out *iBATIS in Action* (Manning, 2007).

5.6.1 Configuring an iBATIS client template

At the center of the iBATIS API is the `com.ibatis.sqlmap.client.SqlMapClient` interface. `SqlMapClient` is roughly equivalent to Hibernate's `Session` or JPA's `EntityManager`. It is through this interface that all data access operations are performed.

Unfortunately, iBATIS shares many of the same problems as JDBC, Hibernate (pre-3.0), and JPA. Specifically, applications that use iBATIS for persistence are required to manage sessions. This session management code is typically nothing more than boilerplate code and distracts from the real goal of persisting objects to a database.

Furthermore, the persistence methods of `SqlMapClient` are written to throw `java.sql.SQLException` if there are any problems. As we've already discussed, `SQLException` is both a checked exception and too generic to react to in any useful way.

`SqlMapClientTemplate` is Spring's answer to the iBATIS session management and exception-handling problems. Much like the other templates that we've covered in this chapter, `SqlMapClientTemplate` wraps an `SqlMapClient` to transparently open and close sessions. It also will catch any `SQLExceptions` that are thrown and rethrow them as one of Spring's unchecked persistence exceptions in table 5.1.

Configuring an `SqlMapClientTemplate`

`SqlMapClientTemplate` can be configured in the Spring application context as follows:

```
<bean id="sqlMapClientTemplate"
    class="org.springframework.orm.ibatis.SqlMapClientTemplate">
    <property name="sqlMapClient" ref="sqlMapClient" />
</bean>
```

The `sqlMapClient` property must be wired with a reference to an iBATIS `SqlMapClient`. In Spring, the best way to get an `SqlMapClient` is through `SqlMapClientFactoryBean`:

```
<bean id="sqlMapClient"
    class="org.springframework.orm.ibatis.SqlMapClientFactoryBean">
    <property name="dataSource" ref="dataSource" />
    <property name="configLocation" value="sql-map-config.xml" />
</bean>
```

`SqlMapClientFactoryBean` is a Spring factory bean that produces an `SqlMapClient`. The `dataSource` property is wired with a reference to a `javax.sql.DataSource`. Any of the data sources described in section 5.2 will do.

Defining iBATIS SQL maps

As for the `configLocation` property, it should be configured with the path to an XML file that enumerates the locations of the iBATIS SQL maps. For the RoadRantz application, we've defined one SQL map file per domain object. Therefore, the `sql-map-config.xml` file will look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE sqlMapConfig PUBLIC "-//iBATIS.com//"
   ↗ DTD SQL Map Config 2.0//EN"
   "http://www.ibatis.com/dtd/sql-map-config-2.dtd">

<sqlMapConfig>
  <sqlMap resource="com/roaddrantz/domain/rant-sql.xml" />
  <sqlMap resource="com/roaddrantz/domain/motorist-sql.xml" />
  <sqlMap resource="com/roaddrantz/domain/vehicle-sql.xml" />
</sqlMapConfig>
```

The three SQL map files are loaded as resources from the classpath under the same package as the domain objects themselves. As an example of iBATIS SQL mapping, listing 5.11 shows an excerpt from `rant-sql.xml`.

Listing 5.11 An example of mapping SQL queries to Rant objects

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE sqlMap PUBLIC "-//iBATIS.com//DTD SQL Map 2.0//EN"
   "http://www.ibatis.com/dtd/sql-map-config-2.dtd">

<sqlMap> namespace="Rant"
...
  <resultMap id="rantResult"
    class="com.roaddrantz.domain.Rant">
    <result property="id" column="id" />
    <result property="rantText" column="rant_text" />
    <result property="postedDate" column="posted_date" />
    <result property="vehicle" column="vehicle_id"
      select="getVehicleById" />
  </resultMap>
...
  <select id="getRantsForDay"
    resultMap="rantResult"
    parameterClass="int">
    <! [CDATA[
      select id, posted_date, rant_text, vehicle_id
      from rant
    </CDATA>
  </select>

```

Defines
result
mapping

↓
Declares
getRantsForDay
query

```

        where posted_date = #VALUE#
    ]]>
</select>
...
</sqlMap>
```

↑ Declares
getRantsForDay
query

In listing 5.11, we've defined a query that loads a list of Rant objects based on data passed in as a parameter when the `getRantsForDay` query is performed. The query is associated with a `<resultMap>` entry that tells iBATIS to convert each row returned from the query into a Rant object. By the time our DAO sees the results, they will be in the form of a List of Rant objects.

Using the template in a DAO

Before we can use the `SqlMapClientTemplate` to perform data access operations, we must wire it into our DAO. The following excerpt from `IBatisRantDao` shows an implementation of `RantDao` that is injected with an `SqlMapClientTemplate`:

```

public class IBatisRantDao implements RantDao {
    ...
    // injected
    private SqlMapClientTemplate sqlMapClientTemplate;
    public void setSqlMapClientTemplate(
        SqlMapClientTemplate sqlMapClientTemplate) {
        this.sqlMapClientTemplate = sqlMapClientTemplate;
    }
}
```

Since `IBatisRantDao` depends on an `SqlMapClientTemplate`, we'll need to configure it as follows in the Spring configuration:

```

<bean id="rantDao"
      class="com.roaddrantz.dao.ibatis.IBatisRantDao">
    <property name="sqlMapClientTemplate"
              ref="sqlMapClientTemplate" />
</bean>
```

With the `SqlMapClientTemplate` injected into `IBatisRantDao`, we can now start fleshing out the persistence methods needed by the `RoadRantz` application. Here's what the `getRantsForDay()` method looks like when written to use the injected `SqlMapClientTemplate`:

```

public List<Rant> getRantsForDay(Date day) {
    return sqlMapClientTemplate.queryForList(
        "getRantsForDay", day);
}
```

As with the other persistence mechanisms, Spring also provides DAO support for iBATIS. Before we end our exploration of Spring-iBATIS integration, let's see how we can build the RoadRantz data access layer using iBATIS DAO support.

5.6.2 Building an iBATIS-backed DAO

The `SqlMapClientDaoSupport` class is a DAO support class for iBATIS. Much like the other DAO support classes offered by Spring, `SqlMapClientDaoSupport` is intended to be subclassed by a DAO implementation. As depicted in figure 5.12, `SqlMapClientDaoSupport` is a convenient superclass for iBATIS-backed DAOs that exposes an `SqlMapClientTemplate` object that can be used to execute iBATIS queries.

Rewriting the `IBatisRantDAO` class to use `SqlMapClientDaoSupport`, we have the following class definition.

```
public class IBatisRantDAO extends SqlMapClientDaoSupport
    implements RantDao {
    ...
}
```

`SqlMapClientDaoSupport` provides an `SqlMapClientTemplate` for your DAO to use through its `getSqlMapClientTemplate()` method. As an example of how to use `getSqlMapClientTemplate()`, here's the new `getRantsForDay()` method:

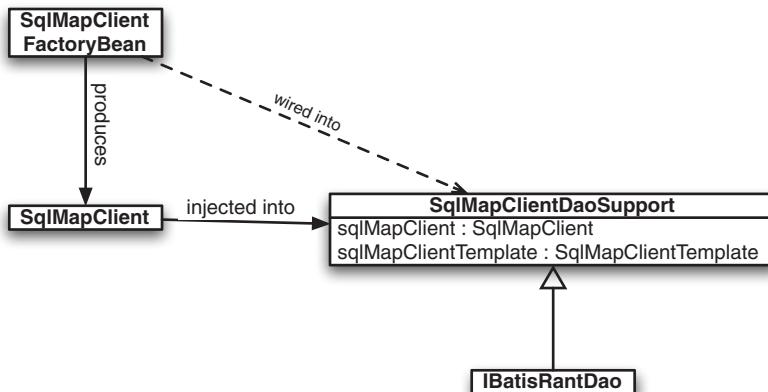


Figure 5.12 `SqlMapClientDaoSupport` is a convenient way to create iBATIS-backed DAO classes. `SqlMapClientDaoSupport` is injected with an `SqlMapClient` that it wraps with an `SqlMapClientTemplate` to hide iBATIS boilerplate code.

```
public List<Rant> getRantsForDay(Date day) {
    return getSqlMapClientTemplate().queryForList(
        "getRantsForDay", day);
}
```

The big difference between wiring an `SqlMapClientTemplate` directly into a DAO and subclassing `SqlMapClientDaoSupport` is that you can eliminate one of the beans in the Spring configuration. When a DAO subclasses `SqlMapClientDaoSupport`, you can bypass the `SqlMapClientTemplate` bean and wire an `SqlMapClient` (or an `SqlMapClientFactoryBean` that produces an `SqlMapClient`) directly into the DAO:

```
<bean id="rantDao" class="com.roaddrantz.dao.ibatis.IBatisRantDao">
    <property name="sqlMapClient" ref="sqlMapClient" />
</bean>
```

As with the other persistence frameworks that integrate with Spring, the decision to either use a DAO support class or wire a template directly into your DAO is mostly a matter of taste. Although `SqlMapClientDaoSupport` does slightly simplify configuration of an iBATIS-backed DAO, you may prefer to inject an `SqlMapClientTemplate` into an application's DAO—especially if your DAO class already subclasses another base class.

Thus far, you've seen several ways of reading and writing data to a database, and we've built the persistence layer of the RoadRantz application. Now that you know how to read data from a database, let's see how to avoid unnecessary database reads using Spring's support for data caching.

5.7 Caching

In many applications, data is read more frequently than it is written. In the RoadRantz application, for instance, more people will visit the site to view the rants for a particular day or vehicle than those who post rants. Although the list of rants will grow over time, it will not grow as often as it is viewed.

Moreover, the data presented by the RoadRantz application is not considered time sensitive. If a user were to browse the site and see a slightly outdated list of rants, it probably would not have any negative impact on them. Eventually, they could return to the site to see a newer list of rants and no harm would be done.

Nevertheless, every time that a list of rants is requested, the DAO will go back to the database and ask for the latest data (which, more often than not, is the same data as the last time it asked).

Database operations are often the number-one performance bottleneck in an application. Even the simplest queries against highly optimized data stores can add up to performance problems in a high-use application.

When you consider the infrequency of data changes along with the performance costs of querying a database, it seems silly to always query the database for the latest data. Instead, it seems to make sense to cache frequently accessed (but not frequently updated) data.

On the surface, caching sounds quite simple: after retrieving some information, store it away in a local (and more easily accessible) location so that it's handy the next time you need it. But implementing a caching solution by hand can be tricky. For example, have a look at `HibernateRantDao`'s `getRantsForDay()` method:

```
public List<Rant> getRantsForDay(Date day) {
    return getHibernateTemplate().find("from " + RANT +
        " where postedDate = ?", day);
}
```

The `getRantsForDay()` method is a perfect candidate for caching. There's no way to go back in time and add a rant for a day in the past. Unless the day being queried for is today, the list of rants returned for any given day will never change. Therefore, there's no point in always going back to the database for the list of rants that were posted last Tuesday. The database only needs to be queried once, and then we can remember it in case we're ever asked for it again.

Now let's modify `getRantsForDay()` to use some form of homegrown cache:

```
public List<Rant> getRantsForDay(Date day) {
    List<Rant> cachedResult =
        rantCache.lookup("getRantsForDay", day);
    if(cachedResult != null) {
        return cachedResult;
    }

    cachedResult = getHibernateTemplate().find("from " + RANT +
        " where postedDate = ?", day);

    rantCache.store("getRantsForDay", day, cachedResult);
    return cachedResult
}
```

This version of `getRantsForDay()` is much more awkward. The real purpose of `getRantsForDay()` is to look up the rants for a given day. But the bulk of the method is dealing with caching. Furthermore, it doesn't directly deal with some of the complexities of caching, such as cache expiration, flushing, or overflow.

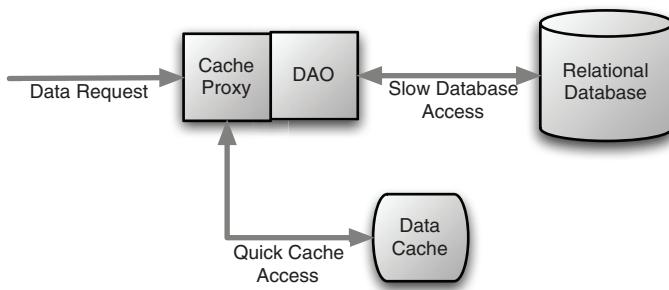


Figure 5.13 The Spring Modules caching module intercepts calls to a bean's methods, looking up data from a cache for quick data access and thus avoiding unnecessary slow queries to the database.

Fortunately, a more elegant caching solution is available for Spring applications. The Spring Modules project (<http://springmodules.dev.java.net>) provides caching via aspects. Rather than explicitly instrument methods to be cached, Spring Modules caching aspects apply advice to bean methods to transparently cache their results.

As illustrated in figure 5.13, Spring Modules support for caching involves a proxy that intercepts calls to one or more methods of Spring-managed beans. When a proxied method is called, Spring Modules Cache first consults a cache to see whether the method has already been called previously with the same arguments. If so, it will return the value in the cache and the actual method will not be invoked. Otherwise, the method is called and its return value is stored in the cache for the next time that the method is called.

In this section, we're going to cache-enable the DAO layer of the RoadRantz application using Spring Modules Cache. This will make the application perform better and give our hard-working database a well-earned break.

5.7.1 Configuring a caching solution

Although Spring Modules provides a proxy for intercepting methods and storing the results in a cache, it does not provide an actual caching solution. Instead, it relies on a third-party cache solution. Several caching solutions are supported, including:

- EHCACHE
- GigaSpaces

- JBoss Cache
- JCS
- OpenSymphony's OSCache
- Tangosol's Coherence

For our RoadRantz application, I've chosen EHCache. This decision was based primarily on my previous experience with EHCache and the fact that it is readily available in the Maven repository at www.ibiblio.org. However, regardless of which caching solution you choose, the configuration for Spring Modules Cache is quite similar for all caching solutions.

The first thing we'll need to do is create a new Spring configuration file to declare caching in. While we could have worked the Spring Modules Cache configuration into any of the Spring context configuration files loaded in the RoadRantz application, it's better to keep thing separate. So we'll create `roadrantz-cache.xml` to hold our caching configuration.

As with any Spring context configuration file, `roadrantz-cache.xml` is rooted with the `<beans>` element. However, to take advantage of Spring Modules' support for EHCache, we'll need to declare the `<beans>` element to recognize the `ehcache` namespace:

```

<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:ehcache="http://www.springmodules.org/schema/ehcache"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
                           http://www.springmodules.org/schema/ehcache
                           http://www.springmodules.org/schema/cache/
                           springmodules-ehcache.xsd">
...
</beans>

```

We're using EHCache for the RoadRantz application, but if you'd like to use one of the other supported caching providers, you'll need to swap out the namespace and schema declaration with the Spring Modules namespace and schema declaration appropriate for your choice. Table 5.6 lists each namespace along with its URI and schema URI.

Regardless of which caching provider you choose, you'll be given several Spring configuration elements for configuring declarative caching in Spring. Table 5.7 catalogs these elements.

Table 5.6 The namespaces and schemas for the various caching providers supported by Spring Modules.

Namespace	Namespace URI	Schema URI
ehcache	http://www.springmodules.org/schema/ehcache	http://www.springmodules.org/schema/cache/springmodules-ehcache.xsd
gigaspaces	http://www.springmodules.org/schema/gigaspaces	http://www.springmodules.org/schema/cache/springmodules-gigaspaces.xsd
jboss	http://www.springmodules.org/schema/jboss	http://www.springmodules.org/schema/cache/springmodules-jboss.xsd
jcs	http://www.springmodules.org/schema/jcs	http://www.springmodules.org/schema/cache/springmodules-jcs.xsd
oscache	http://www.springmodules.org/schema/oscache	http://www.springmodules.org/schema/cache/springmodules-oscache.xsd
tangosol	http://www.springmodules.org/schema/tangosol	http://www.springmodules.org/schema/cache/springmodules-tangosol.xsd

Table 5.7 Spring Modules' configuration elements.

Configuration element	What it's for
<namespace:annotations>	Declaring cached methods by tagging them with Java 5 annotations
<namespace:commons-attributes>	Declaring cached methods by tagging them with Jakarta Commons Attributes metadata
<namespace:config>	Configuring the EHCache cache provider in Spring XML
<namespace:proxy>	Declaring cached methods by declaring a proxy in Spring XML

Since we're using EHCache as the caching provider, we'll need to tell Spring where to find the EHCache configuration file.⁴ That's what the `<ehcache:config>` element is for:

```
<ehcache:config
    configLocation="classpath:ehcache.xml" />
```

Here we're setting the `configLocation` attribute to tell Spring to load EHCache's configuration from the root of the application's classpath.

Configuring EHCache

As for the `ehcache.xml` file itself, we've configured it as shown in listing 5.12.

Listing 5.12 Configuring EHCache in ehcache.xml

```
<ehcache>
    <defaultCache
        maxElementsInMemory="500"
        eternal="true"
        overflowToDisk="false"
        memoryStoreEvictionPolicy="LFU" />

    <cache name="rantzCache"
        maxElementsInMemory="500"
        eternal="true"
        overflowToDisk="false"
        memoryStoreEvictionPolicy="LFU" />
</ehcache>
```

Configures default cache

Configures rantzCache

To summarize the code, we've configured two caches for EHCache to manage. The `<defaultCache>` element is mandatory and describes the cache that will be used if no other suitable cache is found. The `<cache>` element defines other caches and may appear zero or more times in `ehcache.xml` (once for each cache it defines). Here we've defined `rantzCache` as the only nondefault cache.

The attributes specified on `<defaultCache>` and `<cache>` describe the behavior of the cache. Table 5.8 lists the attributes available when configuring a cache in EHCache.

⁴ At the time of this writing, the EHCache configuration (and the specific configuration for the other caching providers) is still specified in a provider-specific file external to Spring. But future versions may expose provider-specific configuration through the `<namespace:config>` element so that the external file is no longer necessary.

Table 5.8 Cache configuration attributes for EHCache.

Attribute	Used to specify...
diskExpiryThreadIntervalSeconds	How often (in seconds) the disk expiry thread is run—that is, how often the disk-persisted cache is cleansed of expired items. (Default: 120 seconds.)
diskPersistent	Whether or not the disk store persists between restarts of the VM. (Default: false.)
eternal	Whether or not elements are eternal. If they are eternal, the element never expires. (Required.)
maxElementsInMemory	The maximum number of elements that will be cached in memory. (Required.)
memoryStoreEvictionPolicy	How eviction will be enforced when <code>maxElementsInMemory</code> is reached. By default, the least recently used (LRU) policy is applied. Other options are first-in/first-out (FIFO) and less frequently used (LFU). (Default: LRU.)
name	The name of the cache. (Required for <code><cache></code> .)
overflowToDisk	Whether or not the cache is allowed to overflow to disk when the in-memory cache has reached the <code>maxElementsInMemory</code> limit. (Required.)
timeToIdleSeconds	The time (in seconds) between accesses before an element expires. A value of 0 indicates that the element can be idle forever. (Default: 0.)
timeToLiveSeconds	The time (in seconds) that an element is allowed to live in cache before it expires. A value of 0 indicates that the element can live in cache forever without expiring. (Default: 0.)

For the RoadRantz application, we've configured one default cache (because EHCache says that we have to) and another cache called `rantzCache` that will be the primary cache. We've configured both caches to allow for up to 500 elements to be kept in cache (with no expiration) and the least frequently used elements will be evicted. In addition, no disk overflow will be allowed.⁵

With EHCache configured in the Spring application context, we are now ready to declare which beans and methods should have their results cached. Let's start

⁵ These choices were made somewhat arbitrarily, but they are a good start. Naturally, the application's usage patterns should be measured and the cache settings adjusted accordingly.

by declaring a proxy that will cache the values returned from the methods of the RoadRantz DAO layer.

5.7.2 Proxying beans for caching

We've already identified the `getRantsForDay()` method of `HibernateRantDao` as a candidate for caching. Back in the Spring context definition, we'll use the `<ehcache:proxy>` element to wrap the `HibernateRantDao` with a proxy that will cache everything returned from `getRantsForDay()`:

```
<ehcache:proxy id="rantDao"
    refId="rantDaoTarget">
    <ehcache:caching
        methodName="getRantsForDay"
        cacheName="rantzCache" />
</ehcache:proxy>
```

The `<ehcache:caching>` element declares which method(s) will be intercepted and which cache their return values will be cached in. For our purposes, `methodName` has been set to intercept the `getRantsForDay()` method and to use the `rantzCache` cache.

You may declare as many `<ehcache:caching>` elements within `<ehcache:proxy>` as you need to describe caching for a bean's methods. You could use one `<ehcache:caching>` element for each cached method. Or you can also use wildcards to specify multiple methods with only one `<ehcache:caching>` element. The following `<ehcache:caching>` element, for example, will proxy all methods whose name starts with `get` to be cached:

```
<ehcache:caching
    methodName="get*"
    cacheName="rantzCache" />
```

Putting items into a cache is only half of the problem. After a while the cache will become littered with lots of data, some of which may no longer be relevant. Eventually, it may be desirable to clear out the cache (call it "Spring cleaning") and start over. Let's see how to flush the cache upon a method call.

Flushing the cache

Where the `<ehcache:caching>` element declares methods that populate the cache, `<ehcache:flushing>` declares methods that empty the cache. For example, let's suppose that you'd like to clear out the `rantzCache` cache whenever the `saveRant()` method is called. The following `<ehcache:flushing>` element will handle that for you:

```
<ehcache:flushing
    methodName="saveRant"
    cacheName="rantzCache" />
```

By default, the cache specified in the `cacheName` attribute will be flushed *after* the method specified with `methodName` is invoked. But you can change the timing of the flush by using the `when` attribute:

```
<ehcache:flushing
    methodName="saveRant"
    cacheName="rantzCache"
    when="before" />
```

By setting `when` to `before` we are asking for the cache to be flushed before the `saveRant()` method is invoked.

Declaring a proxied inner bean

Take note of `<ehcache:proxy>`'s `id` and `refId` attributes. The proxy produced by `<ehcache:proxy>` will be given an `id` of `rantDao`. However, that's the `id` of the real `HibernateRantDao` bean. Therefore, we'll need to rename the real bean to `rantDaoTarget`, which is referred to by the `refId` attribute. (This is consistent with how classic Spring AOP proxies and their targets are named. See section 4.2.3 for a reminder of how that works.)

If the `id/refId` arrangement seems awkward, then you also have the option of declaring the target bean as an inner bean of `<ehcache:proxy>`. For example, here's `<ehcache:proxy>` reconfigured with `HibernateRantDao` as an inner bean:

```
<ehcache:proxy id="rantDao">
    <bean class="com.roadrantz.dao.HibernateRantDao">
        <property name="sessionFactory"
            ref="sessionFactory" />
    </bean>
    <ehcache:caching
        methodName="getRantsForDay"
        cacheName="rantzCache" />
</ehcache:proxy>
```

Even using inner beans, you'll still need to declare one `<ehcache:proxy>` element for each bean to be proxied and one or more `<ehcache:caching>` element for the methods. For simple applications, this may be okay. But as the number of cache-proxied beans and methods goes up, it will mean more and more XML in your Spring configuration.

If the inner-bean approach still seems clumsy or if you will be proxying several beans to be cached, you may want to consider using Spring Modules' support for

declarative caching by annotation. Let's kiss `<ehcache:proxy>` goodbye and see how Spring Modules supports annotation-driven caching.

5.7.3 Annotation-driven caching

In addition to the XML-based caching configuration described in the previous section, Spring Modules supports declarative caching using code-level metadata. This support comes in two varieties:

- *Java 5 annotations*—This is the ideal solution if you're targeting the Java 5 platform.
- *Jakarta Commons Attributes*—If you're targeting pre-Java 5, you may choose Jakarta Commons Attributes.

For RoadRantz, we're targeting Java 5. Therefore, we'll be using Java 5 annotations to declare caching in the DAO layer. Spring Modules provides two annotations with regard to caching:

- `@Cacheable`—Declares that a method's return value should be cached
- `@CacheFlush`—Declares a method to be a trigger for flushing a cache

Using the `@Cacheable` annotation, we can declare the `getRantsForDay()` method to be cached like so:

```
@Cacheable(modelId="rantzCacheModel")
public List<Rant> getRantsForDay(Date day) {
    return getHibernateTemplate().find("from " + RANT +
        " where postedDate = ?", day);
}
```

The `modelId` attribute specifies a caching model that will be used to cache the values returned from `getRantsForDay()`. We'll talk more about how the caching model is defined in a moment. But first, let's use `@CacheFlush` to specify a flush action when the `saveRant()` method is called:

```
@CacheFlush(modelId="rantzFlushModel")
public void saveRant(Rant rant) {
    getHibernateTemplate().saveOrUpdate(rant);
}
```

The `modelId` attribute refers to the flushing model that will be cleared when the `saveRant()` method is invoked.

Speaking of caching and flushing models, you probably would like to know where those come from. The `<ehcache:annotations>` element is used to enable

Spring Modules' support for annotations. We'll configure it in `roaddrantz-cache.xml` as follows:

```
<ehcache:annotations>
    <ehcache:caching id="rantzCacheModel"
        cacheName="rantzCache" />
</ehcache:annotations>
```

Within the `<ehcache:annotations>` element, we must configure at least one `<ehcache:caching>` element. `<ehcache:caching>` defines a caching model. In simple terms, a caching model is little more than a reference to a named cache configured in `ehcache.xml`. Here we've associated the name `rantzCacheModel` with a cache named `rantzCache`. Consequently, any `@Cacheable` whose `modelId` is `rantzCacheModel` will target the cache named `rantzCache`.

A flushing model is quite similar to a caching model, except that it refers to the cache that will be flushed. We'll configure a flushing model called `rantzFlushModel` alongside the `rantzCacheModel` using the `<ehcache:flushing>` element:

```
<ehcache:annotations>
    <ehcache:caching id="rantzCacheModel"
        cacheName="rantzCache" />
    <ehcache:flushing id="rantzFlushModel"
        cacheName="rantzCache" />
</ehcache:annotations>
```

The one thing that sets cache models apart from flushing models is that a flushing model not only decides which cache to flush, but also *when* to flush it. By default, the cache is flushed after `@CacheFlush`-annotated methods are called. But you can change that by specifying a value for the `when` attribute of `<ehcache:flushing>`:

```
<ehcache:annotations>
    <ehcache:caching id="rantzCacheModel"
        cacheName="rantzCache" />
    <ehcache:flushing id="rantzFlushModel"
        cacheName="rantzCache"
        when="before" />
</ehcache:annotations>
```

By setting the `when` attribute to `before`, the cache will be flushed before a `@CacheFlush`-annotated method is invoked.

5.8 **Summary**

Data is the lifeblood of an application. Some of the data-centric among us may even contend that data *is* the application. With such significance being placed on

data, it's important that we develop the data access portion of our applications in a way that is robust, simple, and clear.

Spring's support for JDBC and ORM frameworks takes the drudgery out of data access by handling common boilerplate code that exists in all persistence mechanisms, leaving you to focus on the specifics of data access as they pertain to your application.

One way that Spring simplifies data access is by managing the lifecycle of database connections and ORM framework sessions, ensuring that they are opened and closed as necessary. In this way, management of persistence mechanisms is virtually transparent to your application code.

Also, Spring is able to catch framework-specific exceptions (some of which are checked exceptions) and convert them to one of a hierarchy of unchecked exceptions that are consistent among all persistence frameworks supported by Spring. This includes converting nebulous `SQLExceptions` thrown by JDBC and iBATIS into meaningful exceptions that describe the actual problem that led to the exception being thrown.

We've also seen how an add-on module from the Spring Modules project can provide declarative caching support for your data access layer, increasing performance when often-requested, but scarcely updated, data is retrieved from a database.

Transaction management is another aspect of data access that Spring can make simple and transparent. In the next chapter, we'll explore how to use Spring AOP for declarative transaction management.

Managing transactions



This chapter covers

- Integrating with transaction managers
- Managing transactions programmatically
- Using declarative transactions
- Describing transactions using annotations

Take a moment to recall your younger days. If you were like many children, you spent more than a few carefree moments on the playground swinging on the swings, traversing the monkey bars, getting dizzy while spinning on the merry-go-round, and going up and down on the teeter-totter.

The problem with the teeter-totter is that it is practically impossible to enjoy on your own. You see, to truly enjoy a teeter-totter, you need another person. You and a friend both have to agree to play on the teeter-totter. This agreement is an all-or-nothing proposition. Both of you will either teeter-totter or you will not. If either of you fails to take your respective seat on each end of the teeter-totter then there will be no teeter-tottering—there'll just be a sad little kid sitting motionless on the end of a slanted board.¹

In software, all-or-nothing operations are called *transactions*. Transactions allow you to group several operations into a single unit of work that either fully happens or fully doesn't happen. If everything goes well then the transaction is a success. But if anything goes wrong, the slate is wiped clean and it's as if nothing ever happened.

Probably the most common example of a real-world transaction is a money transfer. Imagine that you were to transfer \$100 from your savings account to your checking account. The transfer involves two operations: \$100 is deducted from the savings account and \$100 is added to the checking account. The money transfer must be performed completely or not at all. If the deduction from the savings account works but the deposit into the checking account fails, you'll be out \$100 (good for the bank, bad for you). On the other hand, if the deduction fails but the deposit succeeds, you'll be ahead \$100 (good for you, bad for the bank). It's best for both parties involved if the entire transfer is rolled back if either operation fails.

In chapter 5, we examined Spring's data access support and saw several ways to read from and write data to the database. When writing to a database, we must ensure that the integrity of the data is maintained by performing the updates within a transaction. Spring has rich support for transaction management, both programmatic and declarative. In this chapter, we'll see how to apply transactions to your application code so that when things go right they are made permanent. And when things go wrong... well, nobody needs to know. (Actually, almost nobody. You may still want to log the problem for the sake of auditing.)

¹ Since the first edition of this book, we have confirmed that this definitely qualifies as the most uses of the word “teeter-totter” in a technical book. That's just a bit of trivia to challenge your friends with.

6.1 Understanding transactions

To illustrate transactions, consider the purchase of a movie ticket. Purchasing a ticket typically involves the following actions:

- The number of available seats will be examined to verify that there are enough seats available for your purchase.
- The number of available seats is decremented by one for each ticket purchased.
- You provide payment for the ticket.
- The ticket is issued to you.

If everything goes well, you'll be enjoying a blockbuster movie and the theater will be a few dollars richer. But what if something goes wrong? For instance, what if you paid with a credit card that had reached its limit? Certainly, you would not receive a ticket and the theater wouldn't receive payment. But if the number of seats isn't reset to its value before the purchase, the movie may artificially run out of seats (and thus lose sales). Or consider what would happen if everything else works fine but the ticket issue fails. You'd be short a few dollars and be stuck at home watching reruns on cable TV.

To ensure that neither you nor the theater loses out, these actions should be wrapped in a transaction. As a transaction, they're all treated as a single action, guaranteeing that they'll either all fully succeed or they'll all be rolled back as if these steps never happened. Figure 6.1 illustrates how this transaction plays out.

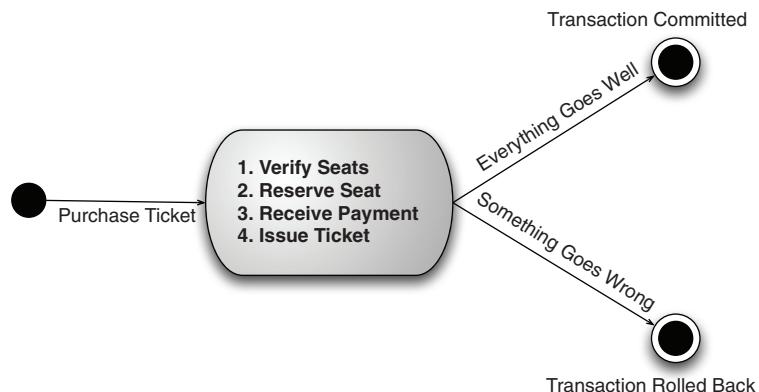


Figure 6.1 The steps involved when purchasing a movie ticket should be all or nothing. If every step is successful then the entire transaction is successful. Otherwise, the steps should be rolled back—as if they never happened.

Transactions play an important role in software, ensuring that data and resources are never left in an inconsistent state. Without them, there is potential for data to be corrupted or inconsistent with the business rules of the application.

Before we get too carried away with Spring's transaction support, it's important to understand the key ingredients of a transaction. Let's take a quick look at the four factors that guide transactions and how they work.

6.1.1 ***Explaining transactions in only four words***

In the grand tradition of software development, an acronym has been created to describe transactions: ACID. In short, ACID stands for:

- *Atomic*—Transactions are made up of one or more activities bundled together as a single unit of work. Atomicity ensures that all the operations in the transaction happen or that none of them happen. If all the activities succeed, the transaction is a success. If any of the activities fail, the entire transaction fails and is rolled back.
- *Consistent*—Once a transaction ends (whether successful or not), the system is left in a state that is consistent with the business that it models. The data should not be corrupted with respect to reality.
- *Isolated*—Transactions should allow multiple users to work with the same data, without each user's work getting tangled up with the others. Therefore, transactions should be isolated from each other, preventing concurrent reads and writes to the same data from occurring. (Note that isolation typically involves locking rows and/or tables in a database.)
- *Durable*—Once the transaction has completed, the results of the transaction should be made permanent so that they will survive any sort of system crash. This typically involves storing the results in a database or some other form of persistent storage.

In the movie ticket example, a transaction could ensure atomicity by undoing the result of all the steps if any step fails. Atomicity supports consistency by ensuring that the system's data is never left in an inconsistent, partially done state. Isolation also supports consistency by preventing another concurrent transaction from stealing seats out from under you while you are still in the process of purchasing them.

Finally, the effects are durable because they will have been committed to some persistent storage. In the event of a system crash or other catastrophic event, you shouldn't have to worry about results of the transaction being lost.

For a more detailed explanation of transactions, we suggest that you read Martin Fowler's *Patterns of Enterprise Application Architecture* (Addison-Wesley Professional, 2002). Specifically, chapter 5 discusses concurrency and transactions.

Now that you know the makings of a transaction, let's see the transaction capabilities available to a Spring application.

6.1.2 Understanding Spring's transaction management support

Spring, like EJB, provides support for both programmatic and declarative transaction management support. But Spring's transaction management capabilities exceed those of EJB.

Spring's support for programmatic transaction management differs greatly from that of EJB. Unlike EJB, which is coupled with a Java Transaction API (JTA) implementation, Spring employs a callback mechanism that abstracts away the actual transaction implementation from the transactional code. In fact, Spring's transaction management support doesn't even require a JTA implementation. If your application uses only a single persistent resource, Spring can use the transactional support offered by the persistence mechanism. This includes JDBC, Hibernate, Java Data Objects (JDO), and Apache's Object Relational Bridge (OJB). However, if your application has transaction requirements that span multiple resources, Spring can support distributed (XA) transactions using a third-party JTA implementation. We'll discuss Spring's support for programmatic transactions in section 6.3.

While programmatic transaction management affords you flexibility in precisely defining transaction boundaries in your code, declarative transactions help you decouple an operation from its transaction rules. Spring's support for declarative transactions is reminiscent of EJB's container-managed transactions (CMTs). Both allow you to define transaction boundaries declaratively. But Spring's declarative transactions go beyond CMTs by allowing you to declare additional attributes such as isolation level and timeouts. We'll begin working with Spring's declarative transaction support in section 6.4.

Choosing between programmatic and declarative transaction management is largely a decision of fine-grained control versus convenience. When you program transactions into your code, you gain precise control over transaction boundaries, beginning and ending them precisely where you want. Typically, you will not require the fine-grained control offered by programmatic transactions and will choose to declare your transactions in the context definition file.

Regardless of whether you choose to program transactions into your beans or to declare them as aspects, you'll be using a Spring transaction manager to

interface with a platform-specific transaction implementation. Let's see how Spring's transaction managers free you from dealing directly with platform-specific transaction implementations.

6.2 Choosing a transaction manager

Spring does not directly manage transactions. Instead, it comes with a selection of transaction managers that delegate responsibility for transaction management to a platform-specific transaction implementation provided by either JTA or the persistence mechanism. Spring's transaction managers are listed in table 6.1.

Table 6.1 Spring has transaction managers for every occasion.

Transaction manager (<code>org.springframework.*</code>)	Use it when...
<code>jca.cci.connection.</code> <code>CciLocalTransactionManager</code>	Using Spring's support for J2EE Connector Architecture (JCA) and the Common Client Interface (CCI).
<code>jdbc.datasource.</code> <code>DataSourceTransactionManager</code>	Working with Spring's JDBC abstraction support. Also useful when using iBATIS for persistence.
<code>jms.connection.JmsTransactionManager</code>	Using JMS 1.1+.
<code>jms.connection.</code> <code>JmsTransactionManager102</code>	Using JMS 1.0.2.
<code>orm.hibernate.</code> <code>HibernateTransactionManager</code>	Using Hibernate 2 for persistence.
<code>orm.hibernate3.</code> <code>HibernateTransactionManager</code>	Using Hibernate 3 for persistence.
<code>orm.jdo.JdoTransactionManager</code>	Using JDO for persistence.
<code>orm.jpa.JpaTransactionManager</code>	Using the Java Persistence API (JPA) for persistence.
<code>orm.toplink.TopLinkTransactionManager</code>	Using Oracle's TopLink for persistence.
<code>transaction.jta.JtaTransactionManager</code>	You need distributed transactions or when no other transaction manager fits the need.
<code>transaction.jta.</code> <code>OC4JJtaTransactionManager</code>	Using Oracle's OC4J JEE container.
<code>transaction.jta.</code> <code>WebLogicJtaTransactionManager</code>	You need distributed transactions and your application is running within WebLogic.

Each of these transaction managers acts as a façade to a platform-specific transaction implementation. (Figure 6.2 illustrates the relationship between transaction managers and the underlying platform implementations for a few of the transaction managers.) This makes it possible for you to work with a transaction in Spring with little regard to what the actual transaction implementation is.

To use a transaction manager, you'll need to declare it in your application context. In this section, you'll learn how to configure a few of Spring's most commonly used transaction managers, starting with `DataSourceTransactionManager`, which provides transaction support for plain JDBC and iBATIS.

6.2.1 JDBC transactions

If you're using straight JDBC for your application's persistence, `DataSourceTransactionManager` will handle transactional boundaries for you. To use `DataSourceTransactionManager`, wire it into your application's context definition using the following XML:

```
<bean id="transactionManager" class="org.springframework.jdbc.  
    ↪ dataSource.DataSourceTransactionManager">  
    <property name="dataSource" ref="dataSource"/>  
</bean>
```

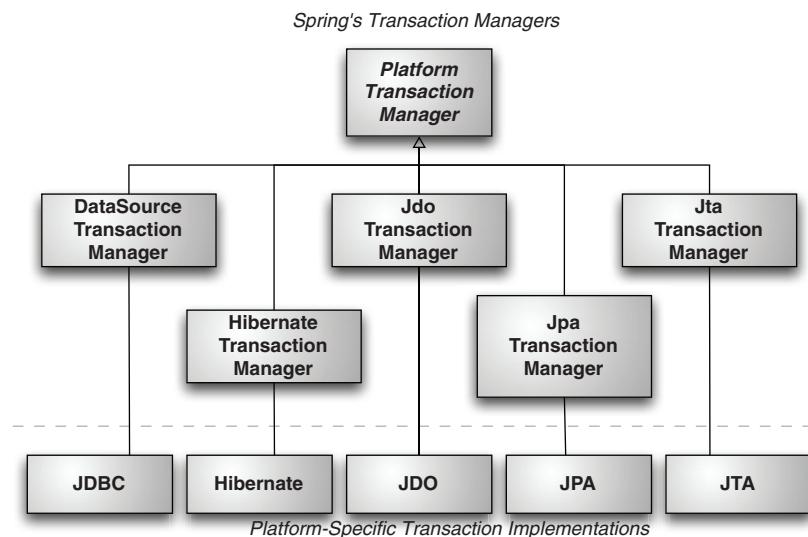


Figure 6.2 Spring's transaction managers delegate transaction-management responsibility to platform-specific transaction implementations.

Notice that the `dataSource` property is set with a reference to a bean named `dataSource`. Presumably, the `dataSource` bean is a `javax.sql.DataSource` bean defined elsewhere in your context definition file.

Behind the scenes, `DataSourceTransactionManager` manages transactions by making calls on the `java.sql.Connection` object retrieved from the `DataSource`. For instance, a successful transaction is committed by calling the `commit()` method on the connection. Likewise, a failed transaction is rolled back by calling the `rollback()` method.

6.2.2 **Hibernate transactions**

If your application's persistence is handled by Hibernate then you'll want to use `HibernateTransactionManager`. For Hibernate 2.x, it is a bean declared with the following XML:

```
<bean id="transactionManager" class="org.springframework.  
    ↪ orm.hibernate.HibernateTransactionManager">  
    <property name="sessionFactory" ref="sessionFactory"/>  
</bean>
```

On the other hand, if you're using Hibernate 3.x, you'll need to declare this version of the `HibernateTransactionManager` bean (pay careful attention to the package name):

```
<bean id="transactionManager" class="org.springframework.  
    ↪ orm.hibernate3.HibernateTransactionManager">  
    <property name="sessionFactory" ref="sessionFactory"/>  
</bean>
```

The `sessionFactory` property should be wired with a Hibernate `SessionFactory`, here cleverly named `sessionFactory`. See chapter 5 for details on setting up a Hibernate session factory.

`HibernateTransactionManager` delegates responsibility for transaction management to an `org.hibernate.Transaction` object that it retrieves from the Hibernate session. When a transaction successfully completes, `HibernateTransactionManager` will call the `commit()` method on the `Transaction` object. Similarly, when a transaction fails, the `rollback()` method will be called on the `Transaction` object.

6.2.3 **Java Persistence API transactions**

Hibernate has been Java's de facto persistence standard for a few years, but now the Java Persistence API (JPA) has entered the scene as the true standard for Java persistence. If you're ready to move up to JPA then you'll want to use Spring's

JpaTransactionManager to coordinate transactions. Here's how you might configure JpaTransactionManager in Spring:

```
<bean id="transactionManager"
      class="org.springframework.orm.jpa.JpaTransactionManager">
    <property name="entityManagerFactory"
              ref="entityManagerFactory" />
</bean>
```

JpaTransactionManager only needs to be wired with a JPA entity manager factory (any implementation of javax.persistence.EntityManagerFactory). JpaTransactionManager will collaborate with the JPA EntityManager produced by the factory to conduct transactions.

In addition to applying transactions to JPA operations, JpaTransactionManager also supports transactions on simple JDBC operations on the same Data-Source used by EntityManagerFactory. For this to work, JpaTransactionManager must also be wired with an implementation of JpaDialect. For example, suppose that you've configured TopLinkJpaDialect as follows:

```
<bean id="jpaDialect"
      class="org.springframework.orm.jpa.vendor.TopLinkJpaDialect" />
```

Then you must wire the jpaDialect bean into the JpaTransactionManager like this:

```
<bean id="transactionManager"
      class="org.springframework.orm.jpa.JpaTransactionManager">
    <property name="entityManagerFactory"
              ref="entityManagerFactory" />
    <property name="jpaDialect"
              ref="jpaDialect" />
</bean>
```

It's important to note that the JpaDialect implementation must support mixed JPA/JDBC access for this to work. All of Spring's vendor-specific implementations of JpaDialect (HibernateJpaDialect, OpenJpaDialect, and TopLinkJpaDialect) provide support for mixing JPA with JDBC. DefaultJpaDialect, however, does not.

6.2.4 Java Data Objects transactions

Perhaps JDBC and Hibernate aren't your style and you're not quite ready to move up to JPA. Suppose that instead you've decided to implement your application's persistence layer using Java Data Objects (JDOs). In that case, the transaction manager of choice will be JdoTransactionManager. It can be declared into your application's context like this:

```
<bean id="transactionManager"
      class="org.springframework.orm.jdo.JdoTransactionManager">
    <property name="persistenceManagerFactory"
      ref="persistenceManagerFactory" />
</bean>
```

With JdoTransactionManager, you need to wire in a javax.jdo.PersistenceManagerFactory instance to the persistenceManagerFactory property.

Under the covers, JdoTransactionManager works with the transaction object retrieved from the JDO persistence manager, calling commit() at the end of a successful transaction and rollback() if the transaction fails.

6.2.5 Java Transaction API transactions

If none of the aforementioned transaction managers meet your needs or if your transactions span multiple transaction sources (e.g., two or more different databases), you'll need to use JtaTransactionManager:

```
<bean id="transactionManager" class="org.springframework.
  ↗ transaction.jta.JtaTransactionManager">
  <property name="transactionManagerName"
    value="java:/TransactionManager" />
</bean>
```

JtaTransactionManager delegates transaction management responsibility to a JTA implementation. JTA specifies a standard API to coordinate transactions between an application and one or more data sources. The transactionManagerName property specifies a JTA transaction manager to be looked up via JNDI.

JtaTransactionManager works with javax.transaction.UserTransaction and javax.transaction.TransactionManager objects, delegating responsibility for transaction management to those objects. A successful transaction will be committed with a call to the UserTransaction.commit() method. Likewise, if the transaction fails, the UserTransaction's rollback() method will be called.

By now, I hope you've found a Spring transaction manager suitable for your application's needs and have wired it into your Spring configuration file. Now it's time to put that transaction manager to work. We'll start by employing the transaction manager to program transactions manually.

6.3 Programming transactions in Spring

There are two kinds of people: those who are control freaks and those who aren't. Control freaks like complete control over everything that happens and don't take anything for granted. If you're a developer and a control freak, you're probably

the kind of person who prefers the command line and would rather write your own getter and setter methods than to delegate that work to an IDE.

Control freaks also like to know exactly what is going on in their code. When it comes to transactions, they want full control over where a transaction starts, where it commits, and where it ends. Declarative transactions aren't precise enough for them.

This isn't a bad thing, though. The control freaks are at least partially right. As you'll see later in this chapter, you are limited to declaring transactions at the method level. If you need more fine-grained control over transactional boundaries, programmatic transactions are the only way to go.

We don't have to look hard to find a need for transactions in the RoadRantz application. Consider the `addRant()` method of `RantServiceImpl` (listing 6.1) as an example of a transactional method.

Listing 6.1 `addRant()`, which adds a Rant and associates the Rant with a Vehicle

```
public void addRant(Rant rant) {
    rant.setPostedDate(new Date());

    Vehicle rantVehicle = rant.getVehicle();
    Vehicle existingVehicle =
        rantDao.findVehicleByPlate(rantVehicle.getState(),
            rantVehicle.getPlateNumber());
```

**Checks for
existing
vehicle**

```
if(existingVehicle != null) {
    rant.setVehicle(existingVehicle);
```

**Associates
vehicle to rant**

```
} else {
    rantDao.saveVehicle(rantVehicle);
```

Saves new vehicle

```
}
```

```
rantDao.saveRant(rant);
```

Saves rant

}

There's a lot more going on in `addRant()` than just simply saving a `Rant` object:

- First, it's possible that the rant's vehicle already exists and, if so, the rant should be associated with the existing vehicle.
- If the rant's vehicle doesn't already exist, the vehicle needs to be saved.
- Finally, the rant itself must be saved.

If any of these actions go sour, all actions should be rolled back as if nothing happened. Otherwise, the database will be left in an inconsistent state. A vehicle

could be added to the database without any associated rants. In other words, `addRant()` should be transactional.

One approach to adding transactions is to programmatically add transactional boundaries directly within the `addRant()` method using Spring's `TransactionTemplate`. Like other template classes in Spring (such as `JdbcTemplate`, discussed in chapter 5), `TransactionTemplate` utilizes a callback mechanism. I've updated the `addRant()` method in listing 6.2 to show how to add a transactional context using a `TransactionTemplate`.

Listing 6.2 Programmatically adding transactions to `addRant()`

```
public void addRant(Rant rant) {  
    transactionTemplate.execute(  
        new TransactionCallback() {  
            public Object doInTransaction(TransactionStatus ts) {  
                try {  
                    rant.setPostedDate(new Date());  
  
                    Vehicle rantVehicle = rant.getVehicle();  
                    Vehicle existingVehicle =  
                        rantDao.findVehicleByPlate(rantVehicle.getState(),  
                            rantVehicle.getPlateNumber());  
  
                    if(existingVehicle != null) {  
                        rant.setVehicle(existingVehicle);  
                    } else {  
                        rantDao.saveVehicle(rantVehicle);  
                    }  
  
                    rantDao.saveRant(rant);  
                } catch (Exception e) {  
                    ts.setRollbackOnly(); ← Rolls back on  
exceptions  
                }  
                return null;  
            }  
        }  
    }  
}
```

To use the `TransactionTemplate`, you start by implementing the `TransactionCallback` interface. Because `TransactionCallback` has only one method to implement, it is often easiest to implement it as an anonymous inner class, as shown in listing 6.2. As for the code that needs to be transactional, place it within the `doInTransaction()` method.

Calling the `execute()` method on the `TransactionTemplate` instance will execute the code contained within the `TransactionCallback` instance. If your code encounters a problem, calling `setRollbackOnly()` on the `TransactionStatus`

object will roll back the transaction. Otherwise, if the `doInTransaction()` method returns successfully, the transaction will be committed.

Where does the `TransactionTemplate` instance come from? Good question. It should be injected into `RantServiceImpl`, as follows:

```
<bean id="rantService"
      class="com.roaddrantz.service.RantServiceImpl">
    ...
    <property name="transactionTemplate ">
      <bean class="org.springframework.transaction.support.
        TransactionTemplate">
        <property name="transactionManager"
          ref="transactionManager" />
      </bean>
    </property>
</bean>
```

Notice that the `TransactionTemplate` is injected with a `transactionManager`. Under the hood, `TransactionTemplate` uses an implementation of `PlatformTransactionManager` to handle the platform-specific details of transaction management. Here we've wired in a reference to a bean named `transactionManager`, which could be any of the transaction managers listed in table 6.1.

Programmatic transactions are good when you want complete control over transactional boundaries. But, as you can see from the code in listing 6.2, they are a bit intrusive. You had to alter the implementation of `addRant()`—using Spring-specific classes—to employ Spring's programmatic transaction support.

Usually your transactional needs won't require such precise control over transactional boundaries. That's why you'll typically choose to declare your transactions outside your application code (in the Spring configuration file, for instance). The rest of this chapter will cover Spring's declarative transaction management.

6.4 Declaring transactions

At one time not too long ago, declarative transaction management was a capability only available in EJB containers. But now Spring offers support for declarative transactions to POJOs. This is a significant feature of Spring because you now have an alternative to EJB for declaring atomic operations.

Spring's support for declarative transaction management is implemented through Spring's AOP framework. This is a natural fit because transactions are a system-level service above an application's primary functionality. You can think of a Spring transaction as an aspect that "wraps" a method with transactional boundaries.

Spring provides three ways to declare transactional boundaries in the Spring configuration. Historically, Spring has always supported declarative transactions by proxying beans using Spring AOP. But Spring 2.0 adds two new flavors of declarative transactions: simple XML-declared transactions and annotation-driven transactions.

We'll look at all of these approaches to declaring transactions later in this section, but first let's examine the attributes that define transactions.

6.4.1 Defining transaction attributes

In Spring, declarative transactions are defined with transaction attributes. A transaction attribute is a description of how transaction policies should be applied to a method. There are five facets of a transaction attribute, as illustrated in figure 6.3.

Although Spring provides several mechanisms for declaring transactions, all of them rely on these five parameters to govern how transaction policies are administered. Therefore, it's essential to understand these parameters in order to declare transaction policies in Spring.

Regardless of which declarative transaction mechanism you use, you'll have the opportunity to define these attributes. Let's examine each attribute to understand how it shapes a transaction.

Propagation behavior

The first facet of a transaction is propagation behavior. Propagation behavior defines the boundaries of the transaction with respect to the client and to the method being called. Spring defines seven distinct propagation behaviors, as described in table 6.2.

NOTE The propagation behaviors described in table 6.2 are defined as constants in the `org.springframework.transaction.TransactionDefinition` interface.

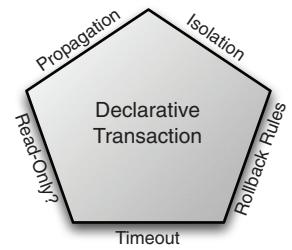


Figure 6.3 Declarative transactions are defined in terms of propagation behavior, isolation level, read-only hints, timeout, and rollback rules.

Table 6.2 Propagation rules define when a transaction is created or when an existing transaction can be used. Spring provides several propagation rules to choose from.

Propagation behavior	What it means
PROPAGATION_MANDATORY	Indicates that the method must run within a transaction. If no existing transaction is in progress, an exception will be thrown.
PROPAGATION_NESTED	Indicates that the method should be run within a nested transaction if an existing transaction is in progress. The nested transaction can be committed and rolled back individually from the enclosing transaction. If no enclosing transaction exists, behaves like PROPAGATION_REQUIRED. Vendor support for this propagation behavior is spotty at best. Consult the documentation for your resource manager to determine if nested transactions are supported.
PROPAGATION_NEVER	Indicates that the current method should not run within a transactional context. If there is an existing transaction in progress, an exception will be thrown.
PROPAGATION_NOT_SUPPORTED	Indicates that the method should not run within a transaction. If an existing transaction is in progress, it will be suspended for the duration of the method. If using JTATransactionManager, access to TransactionManager is required.
PROPAGATION_REQUIRED	Indicates that the current method must run within a transaction. If an existing transaction is in progress, the method will run within that transaction. Otherwise, a new transaction will be started.
PROPAGATION_REQUIRES_NEW	Indicates that the current method must run within its own transaction. A new transaction is started and if an existing transaction is in progress, it will be suspended for the duration of the method. If using JTATransactionManager, access to TransactionManager is required.
PROPAGATION_SUPPORTS	Indicates that the current method does not require a transactional context, but may run within a transaction if one is already in progress.

The propagation behaviors in table 6.2 may look familiar. That's because they mirror the propagation rules available in EJB's container-managed transactions (CMTs). For instance, Spring's PROPAGATION_REQUIRES_NEW is equivalent to CMT's RequiresNew. Spring adds an additional propagation behavior not available in CMT, PROPAGATION_NESTED, to support nested transactions.

Propagation rules answer the question of whether a new transaction should be started or suspended, or if a method should even be executed within a transactional context at all.

For example, if a method is declared to be transactional with `PROPAGATION_REQUIRE_NEW` behavior, it means that the transactional boundaries are the same as the method's own boundaries: a new transaction is started when the method begins and the transaction ends with the method returns or throws an exception. If the method has `PROPAGATION_REQUIRED` behavior, the transactional boundaries depend on whether a transaction is already under way.

Isolation levels

The second dimension of a declared transaction is the isolation level. An isolation level defines how much a transaction may be impacted by the activities of other concurrent transactions. Another way to look at a transaction's isolation level is to think of it as how selfish the transaction is with the transactional data.

In a typical application, multiple transactions run concurrently, often working with the same data to get their job done. Concurrency, while necessary, can lead to the following problems:

- *Dirty read*—Dirty reads occur when one transaction reads data that has been written but not yet committed by another transaction. If the changes are later rolled back, the data obtained by the first transaction will be invalid.
- *Nonrepeatable read*—Nonrepeatable reads happen when a transaction performs the same query two or more times and each time the data is different. This is usually due to another concurrent transaction updating the data between the queries.
- *Phantom reads*—Phantom reads are similar to nonrepeatable reads. These occur when a transaction (T1) reads several rows, and then a concurrent transaction (T2) inserts rows. Upon subsequent queries, the first transaction (T1) finds additional rows that were not there before.

In an ideal situation, transactions would be completely isolated from each other, thus avoiding these problems. However, perfect isolation can affect performance because it often involves locking rows (and sometimes complete tables) in the data store. Aggressive locking can hinder concurrency, requiring transactions to wait on each other to do their work.

Realizing that perfect isolation can impact performance and because not all applications will require perfect isolation, sometimes it is desirable to be flexible

Table 6.3 Isolation levels determine to what degree a transaction may be impacted by other transactions being performed in parallel.

Isolation level	What it means
ISOLATION_DEFAULT	Use the default isolation level of the underlying data store.
ISOLATION_READ_UNCOMMITTED	Allows you to read changes that have not yet been committed. May result in dirty reads, phantom reads, and nonrepeatable reads.
ISOLATION_READ_COMMITTED	Allows reads from concurrent transactions that have been committed. Dirty reads are prevented, but phantom and nonrepeatable reads may still occur.
ISOLATION_REPEATABLE_READ	Multiple reads of the same field will yield the same results, unless changed by the transaction itself. Dirty reads and nonrepeatable reads are prevented, but phantom reads may still occur.
ISOLATION_SERIALIZABLE	This fully ACID-compliant isolation level ensures that dirty reads, nonrepeatable reads, and phantom reads are all prevented. This is the slowest of all isolation levels because it is typically accomplished by doing full table locks on the tables involved in the transaction.

with regard to transaction isolation. Therefore, several levels of isolation are possible, as described in table 6.3.

NOTE The isolation levels described in table 6.3 are defined as constants in the org.springframework.transaction.TransactionDefinition interface.

ISOLATION_READ_UNCOMMITTED is the most efficient isolation level, but isolates the transaction the least, leaving the transaction open to dirty, nonrepeatable, and phantom reads. At the other extreme, ISOLATION_SERIALIZABLE prevents all forms of isolation problems but is the least efficient.

Be aware that not all data sources support all the isolation levels listed in table 6.3. Consult the documentation for your resource manager to determine what isolation levels are available.

Read-only

The third characteristic of a declared transaction is whether it is a read-only transaction. If a transaction performs only read operations against the underlying data store, the data store may be able to apply certain optimizations that take

advantage of the read-only nature of the transaction. By declaring a transaction as read-only, you give the underlying data store the opportunity to apply those optimizations as it sees fit.

Because read-only optimizations are applied by the underlying data store when a transaction begins, it only makes sense to declare a transaction as read-only on methods with propagation behaviors that may start a new transaction (`PROPAGATION_REQUIRED`, `PROPAGATION_REQUIRE_NEW`, and `PROPAGATION_NESTED`).

Furthermore, if you are using Hibernate as your persistence mechanism, declaring a transaction as read-only will result in Hibernate's flush mode being set to `FLUSH_NEVER`. This tells Hibernate to avoid unnecessary synchronization of objects with the database, thus delaying all updates until the end of the transaction.

Transaction timeout

For an application to perform well, its transactions can't carry on for a long time. Therefore, the next trait of a declared transaction is its timeout.

Suppose that your transaction becomes unexpectedly long-running. Because transactions may involve locks on the underlying data store, long-running transactions can tie up database resources unnecessarily. Instead of waiting it out, you can declare a transaction to automatically roll back after a certain number of seconds.

Because the timeout clock begins ticking when a transaction starts, it only makes sense to declare a transaction timeout on methods with propagation behaviors that may start a new transaction (`PROPAGATION_REQUIRED`, `PROPAGATION_REQUIRE_NEW`, and `PROPAGATION_NESTED`).

Rollback rules

The final facet of the transaction pentagon is a set of rules that define what exceptions prompt a rollback and which ones do not. By default, transactions are rolled back only on runtime exceptions and not on checked exceptions. (This behavior is consistent with rollback behavior in EJBs.)

However, you can declare that a transaction be rolled back on specific checked exceptions as well as runtime exceptions. Likewise, you can declare that a transaction not roll back on specified exceptions, even if those exceptions are runtime exceptions.

Now that you've got an overview of how transaction attributes shape the behavior of a transaction, let's see how to use these attributes when declaring transactions in Spring.

6.4.2 Proxying transactions

In pre-2.0 versions of Spring, declarative transaction management was accomplished by proxying your POJOs with Spring's `TransactionProxyFactoryBean`. `TransactionProxyFactoryBean` is a specialization of `ProxyFactoryBean` that knows how to proxy a POJO's methods by wrapping them with transactional boundaries. Listing 6.3 shows how you can declare a `TransactionProxyFactoryBean` that wraps the `RantServiceImpl` class.

Listing 6.3 Proxying the rant service for transactions

```
<bean id="rantService"
      class="org.springframework.transaction.interceptor.
           ➔ TransactionProxyFactoryBean">

    <property name="target"
              ref="rantServiceTarget" /> | Wires transaction target

    <property name="proxyInterfaces"
              value="com.rodrantz.service.RantService" /> | Specifies proxy interface

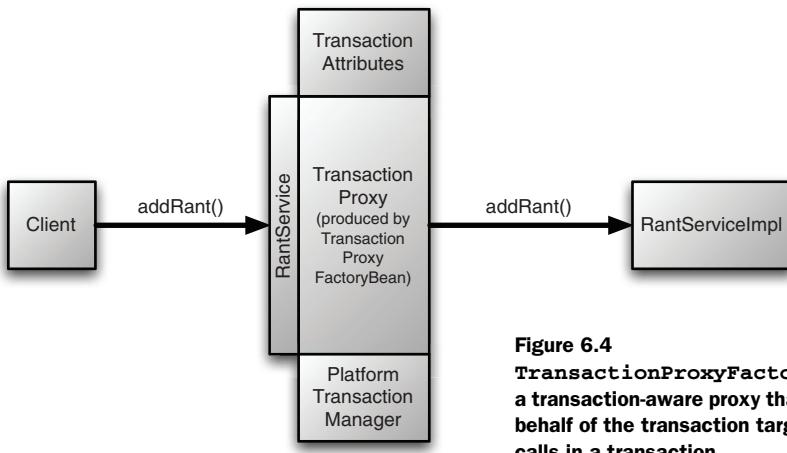
    <property name="transactionManager"
              ref="transactionManager" /> | Wires in transaction
                                            manager

    <property name="transactionAttributes">
        <props>
            <prop key="add*">PROPAGATION_REQUIRED</prop>
            <prop key="* ">PROPAGATION_SUPPORTS,readOnly</prop>
        </props>
    </property>
</bean>
```

**Configures
transaction
rules,
boundaries**

Notice that the bean's id is `rantService`. But wait—doesn't that conflict with the `RantServiceImpl` bean that we've already declared? As a matter of fact, it does, and here's why: the rant service has no idea that its methods are being called within the context of a transaction. If any object makes calls directly to the rant service, those calls will not be transactional. Instead, collaborating objects should invoke methods on the proxy that is produced by `TransactionProxyFactoryBean` (as shown in figure 6.4). The proxy will ensure that transactional rules are applied and then proxy the call to the real rant service. Therefore, rather than inject the rant service directly into those objects that use it, we'll inject the rant service proxy into those objects.

This means that the proxy produced by `TransactionProxyFactoryBean` must pretend to be a rant service. That's the purpose of the `proxyInterfaces` property.

**Figure 6.4**

TransactionProxyFactoryBean produces a transaction-aware proxy that receives calls on behalf of the transaction target, wrapping the calls in a transaction.

Here we're telling `TransactionProxyFactoryBean` to produce a proxy that implements the `RantService` interface.

So what becomes of the original `rantService` bean that we declared? Quite simply, it is renamed to `rantServiceTarget` and injected into the `TransactionProxyFactoryBean` to be proxied.

The `transactionManager` property supplies the appropriate transaction manager bean. This can be any of the transaction managers discussed in section 6.2. `TransactionProxyFactoryBean` will use the transaction manager to start, suspect, commit, and roll back transactions based on the transaction attributes defined in the `transactionAttributes` property of `TransactionProxyFactoryBean`.

Speaking of the `transactionAttributes` property, this property declares which methods are to be run within a transaction and what the transaction attributes are to be. This property is given a `<props>` collection where the key of each `<prop>` is a method name pattern and the value defines the transaction attributes for the method(s) selected.

The value of each `<prop>` given to the `transactionAttributes` property is a comma-separated value that takes the form shown in figure 6.5.

In the case of the rant service, we're declaring that all methods whose name starts with `add` (including `addRant()`) should be run within a transaction. All other methods support transactions (but do not necessarily require a transaction) and are read-only.

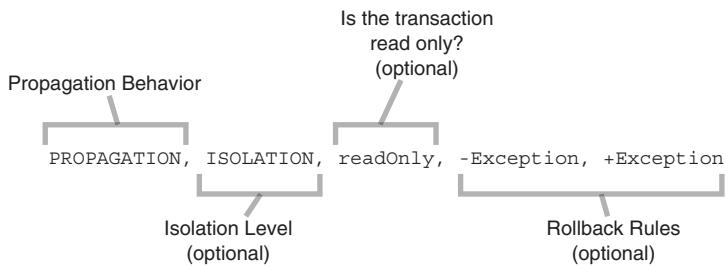


Figure 6.5 A transaction attribute definition is made up of a propagation behavior, an isolation level, a read-only flag, and rollback rules. The propagation behavior is the only required element.

Creating a transaction proxy template

It's one thing to proxy a single service bean using `TransactionProxyFactoryBean`. But what if you have multiple service beans in your application and they all must be transactional? The XML required to proxy a single service bean is verbose enough, but it gets really messy if you have to repeat it for more than one service bean.

Fortunately, you don't have to. Using Spring's ability to create abstract beans and then "sub-bean," you can define your transaction policies in one place and then apply them repeatedly to all of your service beans.

First, you must create an abstract declaration of `TransactionProxyFactoryBean`. The following declaration of `txProxyTemplate` does the trick:

```

<bean id="txProxyTemplate"
      class="org.springframework.transaction.interceptor.
           TransactionProxyFactoryBean"
      abstract="true">
    <property name="transactionManager"
              ref="transactionManager" />
    <property name="transactionAttributes">
      <props>
        <prop key="add*">PROPAGATION_REQUIRED</prop>
        <prop key="*">PROPAGATION_SUPPORTS,readOnly</prop>
      </props>
    </property>
  </bean>
  
```

You'll notice that this abstract declaration is virtually identical to the concrete declaration in listing 6.2. Missing, however, are the specifics of the bean that will be proxied. From this single abstract `TransactionProxyFactoryBean` declaration, we can now make any number of beans transactional by using `txProxyTemplate` as

the parent declaration of the bean. For example, the following XML extends txProxyTemplate for the rant service:

```
<bean id="rantService" parent="txProxyTemplate">
    <property name="target" ref="rantServiceTarget" />
    <property name="proxyInterfaces"
        value="com.roaddrantz.service.RantService" />
</bean>
```

This XML is much simpler and only specifies the target bean that is to be proxied with transactions and its interface. Proxying another bean with the same transactional policies involves creating another bean declaration whose parent is txProxyTemplate and targets the other bean.

Although TransactionProxyFactoryBean has been the workhorse of Spring's declarative transaction support since the very beginning, it is somewhat cumbersome to use. Recognizing that awkwardness of TransactionProxyFactoryBean, Spring 2.0 adds simplified support for declarative transaction. Let's switch gears and see what Spring 2.0 has to offer with regard to declarative transactions.

6.4.3 Declaring transactions in Spring 2.0

The problem with TransactionProxyFactoryBean is that using it results in extremely verbose Spring configuration files (transaction proxy templates notwithstanding). What's more, the practice of naming the target bean with a target suffix is somewhat peculiar and can be confusing.

The good news is that Spring 2.0 provides some new configuration elements especially for declaring transactions. These elements are in the tx namespace and can be used by adding the spring-tx-2.0.xsd schema to your Spring configuration XML file:

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/
            spring-beans-2.0.xsd
        http://www.springframework.org/schema/aop
        http://www.springframework.org/schema/aop/spring-aop-2.0.xsd
        http://www.springframework.org/schema/tx
        http://www.springframework.org/schema/tx/spring-tx-2.0.xsd">
```

Note that the aop namespace should also be included. This is important, because the new declarative transaction configuration elements rely on a few of Spring's new AOP configuration elements (as discussed in chapter 4).

The tx namespace provides a handful of new XML configuration elements, most notably the `<tx:advice>` element. The following XML snippet shows how `<tx:advice>` can be used to declare transactional policies similar to those we defined for the rant service in listing 6.3:

```
<tx:advice id="txAdvice">
  <tx:attributes>
    <tx:method name="add*" propagation="REQUIRED" />
    <tx:method name="*" propagation="SUPPORTS"
      read-only="true"/>
  </tx:attributes>
</tx:advice>
```

With `<tx:advice>`, the transaction attributes are defined in a `<tx:attributes>` element, which contains one or more `<tx:method>` elements. The `<tx:method>` element defines the transaction attributes for a given method (or methods) as defined by the `name` attribute (using wildcards).

`<tx:method>` has several attributes that help define the transaction policies for the method(s), as defined in table 6.4.

As defined in the `txAdvice` transaction advice, the transactional methods configured are divided into two categories: Those whose names begin with `add` and everything else. The `addRant()` method falls into the first category and is declared to require a transaction. The other methods are declared with `propagation = "supports"`—they'll run in a transaction if one already exists, but they don't need to run within a transaction.

Table 6.4 The six facets of the transaction pentagon (see figure 6.3) are specified in the attributes of the `<tx:method>` element.

Attribute	Purpose
<code>isolation</code>	Specifies the transaction isolation level
<code>no-rollback-for</code>	Specifies exceptions for which the transaction should continue and not be rolled back
<code>propagation</code>	Defines the transaction's propagation rule
<code>read-only</code>	Specifies that a transaction be read-only
<code>rollback-for</code>	Specifies checked exceptions for which a transaction should be rolled back and not committed
<code>timeout</code>	Defines a timeout for a long-running transaction

When declaring a transaction using `<tx:advice>`, you'll still need a transaction manager just like you did when using `TransactionProxyFactoryBean`. Choosing convention over configuration, `<tx:advice>` assumes that the transaction manager will be declared as a bean whose id is `transactionManager`. If you happen to give your transaction manager a different id (`txManager`, for instance), you'll need to specify the id of the transaction manager in the `transaction-manager` attribute:

```
<tx:advice id="txAdvice"
           transaction-manager="txManager">
...
</tx:advice>
```

On its own, `<tx:advice>` only defines an AOP advice for advising methods with transaction boundaries. But this is only transaction advice, not a complete transactional aspect. Nowhere in `<tx:advice>` did we indicate which beans should be advised—we need a pointcut for that. To completely define the transaction aspect, we must define an advisor. This is where the `aop` namespace gets involved. The following XML defines an advisor that uses the `txAdvice` advice to advise any beans that implement the `RantService` interface:

```
<aop:config>
  <aop:advisor
    pointcut="execution(* *..RantService.*(..))"
    advice-ref="txAdvice"/>
</aop:config>
```

The `pointcut` attribute uses an AspectJ pointcut expression to indicate that this advisor should advise all methods of the `RantService` interface. Which methods are actually run within a transaction and what the transactional attributes are for those methods is defined by the transaction advice, which is referenced with the `advice-ref` attribute to be the advice named `txAdvice`.

Although the `<tx:advice>` element goes a long way toward making declarative transactions more palatable for Spring developers, there's one more new feature of Spring 2.0 that makes it even nicer for those working in a Java 5 environment. Let's have a look at how Spring transactions can be annotation driven.

6.4.4 Defining annotation-driven transactions

The `<tx:advice>` configuration element greatly simplifies the XML required for declarative transactions in Spring. What if I told you that it could be simplified even further? What if I told you that, in fact, you only need to add a single line of XML to your Spring context in order to declare transactions?

In addition to the `<tx:advice>` element, the `tx` namespace provides the `<tx:annotation-driven>` element. Using `<tx:annotation-driven>` is often as simple as the following line of XML:

```
<tx:annotation-driven />
```

That's it! If you were expecting more, I apologize. I could make it slightly more interesting by specifying a specific transaction manager bean with the `transaction-manager` attribute (which defaults to `transactionManager`):

```
<tx:annotation-driven transaction-manager="txManager" />
```

Otherwise, there's not much more to it than that. That single line of XML packs a powerful punch that lets you define transaction rules where they make the most sense: on the methods that are to be transactional.

Annotations are one of the biggest and most debated new features of Java 5. Annotations let you define metadata directly in your code rather than in external configuration files. Although there's much discussion on the proper use of annotations, I think that annotations are a perfect fit for declaring transactions.

The `<tx:annotation-driven>` configuration element tells Spring to examine all beans in the application context and to look for beans that are annotated with `@Transactional`, either at the class level or at the method level. For every bean that is `@Transactional`, `<tx:annotation-driven>` will automatically advise it with transaction advice. The transaction attributes of the advice will be defined by parameters of the `@Transactional` annotation.

For example, listing 6.4 shows `RantServiceImpl`, updated to include the `@Transactional` annotations.

Listing 6.4 Annotating the rant service to be transactional

```
@Transactional(propagation=Propagation.SUPPORTS, readOnly=true)
public class RantServiceImpl implements RantService {
    ...
    @Transactional(propagation=Propagation.REQUIRED, readOnly=false)
    public void addRant(Rant rant) {
        ...
    }
    ...
}
```

At the class level, `RantServiceImpl` has been annotated with a `@Transactional` annotation that says that all methods will support transaction and be read-only. At the method level, the `addRant()` method has been annotated to indicate that this method requires a transactional context.

It may be interesting to note that the `@Transactional` annotation may also be applied to an interface. For example, listing 6.5 shows the `RantService` interface annotated with `@Transactional`.

Listing 6.5 Annotating the rant service to be transactional at the interface level

```
@Transactional(propagation=Propagation.SUPPORTS, readOnly=true)
public interface RantService {
    ...
    @Transactional(propagation=Propagation.REQUIRED, readOnly=false)
    void addRant(Rant rant);
    ...
}
```

By annotating `RantService` instead of `RantServiceImpl`, we're indicating that all implementations of `RantService` should be transactional.

6.5 Summary

Transactions are an important part of enterprise application development that leads to more robust software. They ensure an all-or-nothing behavior, preventing data from being inconsistent should the unexpected occur. They also support concurrency by preventing concurrent application threads from getting in each other's way as they work with the same data.

Spring supports both programmatic and declarative transaction management. In either case, Spring shields you from having to work directly with a specific transaction management implementation by abstracting the transaction management platform behind a common API.

Spring employs its own AOP framework to support declarative transaction management. Spring's declarative transaction support rivals that of EJB's CMT, enabling you to declare more than just propagation behavior on POJOs, including isolation levels, read-only optimizations, and rollback rules for specific exceptions.

This chapter showed you how to bring declarative transactions into the Java 5 programming model using annotations. With the introduction of Java 5 annotations, making a method transactional is simply a matter of tagging it with the appropriate transaction annotation.

As you've seen, Spring bestows the power of declarative transactions to POJOs. This is an exciting development—declarative transactions were previously only available to EJBs. But declarative transactions are only the beginning of what Spring has to offer to POJOs. In the next chapter, you'll see how Spring extends declarative security to POJOs.



Securing Spring

This chapter covers

- Introducing Spring Security
- Securing web applications using servlet filters
- Authentication against databases and LDAP
- Transparently securing method invocations

Have you ever noticed that most people in television sitcoms don't lock their doors? It happens all the time. On *Seinfeld*, Kramer frequently let himself into Jerry's apartment to help himself to the goodies in Jerry's refrigerator. On *Friends*, the various characters often entered one another's apartments without warning or hesitation. Even once, while in London, Ross burst into Chandler's hotel room, narrowly missing Chandler in a compromising situation with Ross's sister.

In the days of *Leave it to Beaver*, it wasn't so unusual for people to leave their doors unlocked. But it seems crazy that in a day when we're concerned with privacy and security we see television characters enabling unhindered access to their apartments and homes.

It's a sad reality that there are villainous individuals roaming around seeking to steal our money, riches, cars, and other valuables. And it should be no surprise that as information is probably the most valuable item we have, crooks are looking for ways to steal our data and identity by sneaking into unsecured applications.

As software developers, we must take steps to protect the information that resides in our applications. Whether it's an email account protected with a user-name/password pair or a brokerage account protected with a trading PIN, security is a crucial *aspect* of most applications.

It is no accident that I chose to describe application security with the word "aspect." Security is a concern that transcends an application's functionality. For the most part, an application should play no part in securing itself. Although you could write security functionality directly into your application's code (and that's not uncommon), it is better to keep security concerns separate from application concerns.

If you're thinking that it is starting to sound as if security is accomplished using aspect-oriented techniques, you're right. In this chapter we're going to explore ways to secure your applications with aspects. But we won't have to develop those aspects ourselves—we're going to look at Spring Security, a security framework based on Spring AOP and servlet filters.¹

7.1 ***Introducing Spring Security***

Spring Security is a security framework that provides declarative security for your Spring-based applications. Spring Security provides a comprehensive security

¹ I'm probably going to get a lot of emails about this, but I have to say it anyway: servlet filters are a primitive form of AOP, with URL patterns as a kind of pointcut expression language. There... I've said it... I feel better now.

solution, handling authentication and authorization, at both the web request level and at the method invocation level. Based on the Spring Framework, Spring Security takes full advantage of dependency injection (DI) and aspect-oriented techniques.

What's in a name?

Historically, Spring Security is also known as Acegi Security (or simply Acegi). Acegi has long been a subproject of Spring. But as I write this, plans are afoot to bring Acegi even closer under the Spring umbrella of projects. Part of that move involves dropping the Acegi name in favor of "Spring Security." This change is scheduled to take place in the 1.1.0 version of Acegi/Spring Security. Knowing that the change is imminent, I've decided to go ahead and start referring to it as Spring Security, although you'll still see the Acegi name thrown about a bit in this chapter.

When securing web applications, Spring Security uses servlet filters that intercept servlet requests to perform authentication and enforce security. And, as you'll find in section 7.4.1, Spring Security employs a unique mechanism for declaring servlet filters that enables you to inject them with their dependencies using Spring DI.

Spring Security can also enforce security at a lower level by securing method invocations. When securing methods, Spring Security uses Spring AOP to proxy objects, applying aspects that ensure that the user has proper authority to invoke the secured methods.

In any case, whether you only need security at the web request level or if you require lower-level method security, Spring Security employs five core components to enforce security, as shown in figure 7.1.

Before we get into the nitty-gritty of Spring Security, let's take a high-level view of Spring Security and the part that each of these components plays in securing applications.

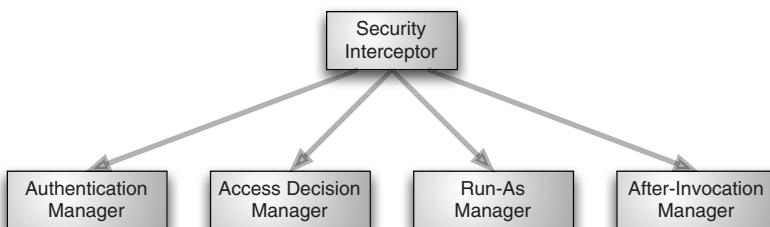


Figure 7.1 The fundamental elements of Spring Security.

Security interceptors

When you arrive home after a long day at work, you'll need to unlock the door to your home. To open the door, you must insert a key into the lock that trips the tumblers properly and releases the latch. If the cut of the key is incorrect, the tumblers won't be tripped and the latch won't be released. But if you have the right key, all of the tumblers will accept the key and the latch will be released, allowing you to open the door.

In Spring Security, the security interceptor can be thought of as a latch that prevents you from accessing a secured resource in your application. To flip the latch and get past the security interceptor, you must enter your "key" (typically a username and password) into the system. The key will then try to trip the security interceptor's "tumblers" in an attempt to grant you access to the secured resource.

The actual implementation of a security interceptor will depend on what resource is being secured. If you're securing a URL in a web application, the security interceptor will be implemented as a servlet filter. But if you're securing a method invocation, aspects will be used to enforce security. You'll see both forms of security interceptor later in this chapter.

A security interceptor does little more than intercept access to resources to enforce security. It does not actually apply security rules. Instead, it delegates that responsibility to the various managers that are pictured at the bottom of figure 7.1. Let's have a look at each of these managers, starting with the authentication manager.

Authentication managers

The first of the security interceptor's tumblers to be tripped is the *authentication manager*. The authentication manager is responsible for determining who you are. It does this by considering your *principal* (typically a username) and your *credentials* (typically a password).

Your principal defines who you are and your credentials are evidence that corroborates your identity. If your credentials are good enough to convince the authentication manager that your principal identifies you then Spring Security will know whom it is dealing with.

As with the rest of Spring Security (and Spring itself), the authentication manager is a pluggable interface-based component. This makes it possible to use Spring Security with virtually any authentication mechanism you can imagine. As you'll see later in this chapter, Spring Security comes with a handful of flexible authentication managers that cover the most common authentication strategies.

Access decisions managers

Once Spring Security has determined who you are, it must decide whether you are authorized to access the secured resource. An *access decision manager* is the second tumbler of the Spring Security lock to be tripped. The access decision manager performs authorization, deciding whether to let you in by considering your authentication information and the security attributes that have been associated with the secured resource.

For example, the security rules may dictate that only supervisors should be allowed access to a secured resource. If you have been granted supervisor privileges then the second and final tumbler, the access decision manager, will have been tripped and the security interceptor will move out of your way and let you gain access to the secured resource.

Just as with the authentication manager, the access decision manager is pluggable. Later in this chapter, we'll take a closer look at the access decision managers that come with Spring Security.

Run-as managers

If you've gotten past the authentication manager and the access decision manager then the security interceptor will be unlocked and the door is ready to open. But before you twist the knob and go in, there's one more thing that the security interceptor might do.

Even though you've passed authentication and been granted access to a resource, there may be more security restrictions behind the door. For example, you may be granted the rights to view a web page, but the objects that are used to create that page may have different security requirements than the web page. A run-as manager can be used to replace your authentication with an authentication that allows you access to the secured objects that are deeper in your application.

Note that not all applications have a need for identity substitution. Therefore, run-as managers are an optional security component and are not necessary in many applications secured by Spring Security.

After-invocation managers

Spring Security's after-invocation manager is a little different from the other security manager components. Whereas the other security manager components perform some form of security enforcement before a secured resource is accessed, the after-invocation manager enforces security after the secured resource is accessed.

After-invocation managers are kind of like the person who waits to examine a receipt at the exit of some discount and home electronics stores. They check to ensure that you have proper authority to remove the valuable items from the store. Instead of making sure that you are allowed to remove a big-screen television from a store, however, after-invocation managers make sure that you're allowed to view the data that is being returned from a secured resource.

If an after-invocation manager advises a service layer bean, it will be given the opportunity to review the value returned from the advised method. It can then make a decision as to whether the user is allowed to view the returned object. The after-invocation manager also has the option of altering the returned value to ensure that the user is only able to access certain properties of the returned object.

Like run-as managers, not all applications call for an after-invocation manager. You'll only need an after-invocation manager if your application's security scheme requires that access be restricted at the domain level on a per-instance basis.

Now that you've seen the big picture of Spring Security, we're ready to configure Spring Security for the RoadRantz application. For our purposes, we won't need a run-as manager or an after-invocation manager, so we'll defer those as advanced Spring Security topics. Meanwhile, let's get started by configuring an authentication manager.

7.2

Authenticating users

When applying security to an application, the first thing you need to do, before deciding whether to allow access, is figure out who the user is. In most applications, this means presenting a login screen to the user and asking them for the username and password.

How the user is prompted for their username and password will vary from application to application. For now, we'll assume that the user's login details have already been provided and we need Spring Security to authenticate the user. We'll look at different ways to prompt the user for their username and password a little later in this chapter.

In Spring Security, the authentication manager assumes the job of establishing a user's identity. An authentication manager is defined by the `org.acegisecurity.AuthenticationManager` interface:

```
public interface AuthenticationManager {  
    public Authentication authenticate(Authentication authentication)  
        throws AuthenticationException;
```

Third time's a charm

A-ha! There's the word "acegi" in the package name for AuthenticationManager. As mentioned earlier in this chapter, Spring Security has historically been known as Acegi Security. When Acegi is formally renamed to Spring Security, the packaging of its classes will also change. Actually, this will be the third base package name that Acegi/Spring Security has had. Acegi was originally packaged under net.sf.acegisecurity... then it was changed to org.acegisecurity. When version 1.1.0 is released, it will likely be repackaged under org.springframework.security. Nevertheless, as those changes haven't happened yet, the examples in this chapter show the org.acegisecurity packaging.

The `authenticate()` method will attempt to authenticate the user using the `org.acegisecurity.Authentication` object (which carries the principal and credentials). If successful, the `authenticate()` method returns a complete `Authentication` object, including information about the user's granted authorities (which will be considered by the authorization manager). If authentication fails, an `AuthenticationException` will be thrown.

As you can see, the `AuthenticationManager` interface is quite simple and you could easily implement your own `AuthenticationManager`. But Spring Security comes with `ProviderManager`, an implementation of `AuthenticationManager` that is suitable for most situations. So instead of rolling our own authentication manager, let's take a look at how to use `ProviderManager`.

7.2.1 Configuring a provider manager

`ProviderManager` is an authentication manager implementation that delegates responsibility for authentication to one or more authentication providers, as shown in figure 7.2.

The purpose of `ProviderManager` is to enable you to authenticate users against multiple identity management sources. Rather than relying on itself to perform authentication, `ProviderManager` steps one by one through a collection of authentication providers, until one of them successfully authenticates the user (or until it runs out of providers). This makes it possible for Spring Security to support multiple authentication mechanisms for a single application.

The following chunk of XML shows a typical configuration of `ProviderManager` in the Spring configuration file:

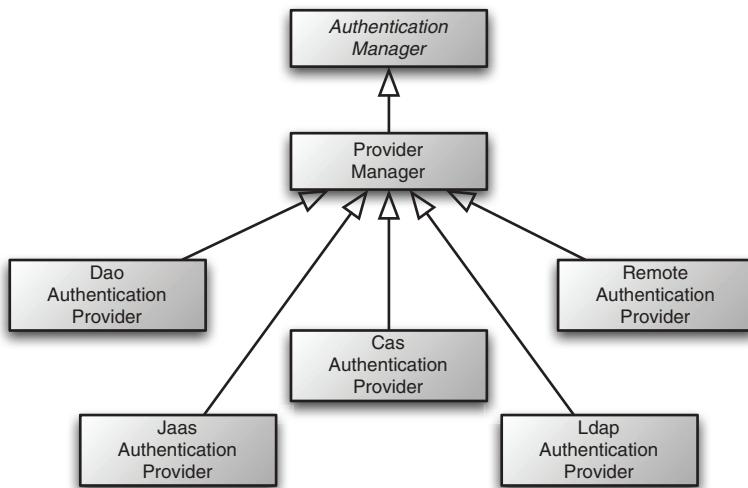


Figure 7.2 A ProviderManager delegates authentication responsibility to one or more authentication providers.

```

<bean id="authenticationManager"
      class="org.acegisecurity.providers.ProviderManager">
  <property name="providers">
    <list>
      <ref bean="daoAuthenticationProvider"/>
      <ref bean="ldapAuthenticationProvider"/>
    </list>
  </property>
</bean>
  
```

ProviderManager is given its list of authentication providers through its providers property. Typically, you'll only need one authentication provider, but in some cases, it may be useful to supply a list of several providers so that if authentication fails against one provider, another provider will be tried. Spring comes with several authentication providers, as listed in table 7.1.

Table 7.1 Spring Security comes with authentication providers for every occasion.

Authentication provider (<code>org.acegisecurity.*</code>)	Purpose
<code>adapters.AuthByAdapterProvider</code>	Authentication using container adapters. This makes it possible to authenticate against users created within the web container (e.g., Tomcat, JBoss, Jetty, Resin, etc.).

Table 7.1 Spring Security comes with authentication providers for every occasion. (continued)

Authentication provider (<code>org.acegisecurity.*</code>)	Purpose
<code>providers.anonymous.</code> <code>AnonymousAuthenticationProvider</code>	Authenticates a user as an anonymous user. Useful when a user token is needed, even when the user hasn't logged in yet.
<code>providers.cas.CasAuthenticationProvider</code>	Authentication against the JA-SIG Central Authentication Service (CAS). Useful when you need single sign-on capabilities.
<code>providers.dao.DaoAuthenticationProvider</code>	Retrieving user information, including username and password from a database.
<code>providers.dao.LdapAuthenticationProvider</code>	Authentication against a Lightweight Directory Access Protocol (LDAP) server.
<code>providers.jaas.JaasAuthenticationProvider</code>	Retrieving user information from a JAAS login configuration.
<code>providers.rememberme.</code> <code>RememberMeAuthenticationProvider</code>	Authenticates a user that was previously authenticated and remembered. Makes it possible to automatically log in a user without prompting for username and password.
<code>providers.rcp.</code> <code>RemoteAuthenticationProvider</code>	Authentication against a remote service.
<code>providers.TestingAuthenticationProvider</code>	Unit testing. Automatically considers a <code>TestingAuthenticationToken</code> as valid. Not for production use.
<code>providers.x509.</code> <code>X509AuthenticationProvider</code>	Authentication using an X.509 certificate. Useful for authenticating users that are, in fact, other applications (such as a web-service client).
<code>runas.RunAsImplAuthenticationProvider</code>	Authenticating a user who has had their identity substituted by a run-as manager.

As you can see in table 7.1, Spring Security provides an authentication provider to meet almost any need. But if you can't find an authentication provider that suits your application's security needs, you can always create your own authentication provider by implementing the `org.acegisecurity.providers.AuthenticationProvider` interface:

```
public interface AuthenticationProvider {
    Authentication authenticate(Authentication authentication)
        throws AuthenticationException;
    boolean supports(Class authentication);
}
```

You may have noticed that the `AuthenticationProvider` interface isn't much different from the `AuthenticationManager` interface shown a few pages back. They both share an `authenticate()` method that handles the authentication. In fact, you can think of authentication providers as subordinate authentication managers.

Space constraints do not allow me to go into the details of all 11 of Spring Security's authentication providers. However, I will focus on a couple of the most commonly used authentication providers, starting with `DaoAuthenticationProvider`, which supports simple database-oriented authentication.

7.2.2 **Authenticating against a database**

Many applications store user information, including the username and password, in a relational database. If that's how your application keeps user information, Spring Security's `DaoAuthenticationProvider` may be a good choice for your application.

A `DaoAuthenticationProvider` is a simple authentication provider that uses a Data Access Object (DAO) to retrieve user information (including the user's password) from a relational database.

With the username and password in hand, `DaoAuthenticationProvider` performs authentication by comparing the username and password retrieved from the database with the principal and credentials passed in an `Authentication` object from the authentication manager (see figure 7.3). If the username and password match up with the principal and credentials, the user will be authenticated and a fully populated `Authentication` object will be returned to the authentication manager. Otherwise, an `AuthenticationException` will be thrown and authentication will have failed.

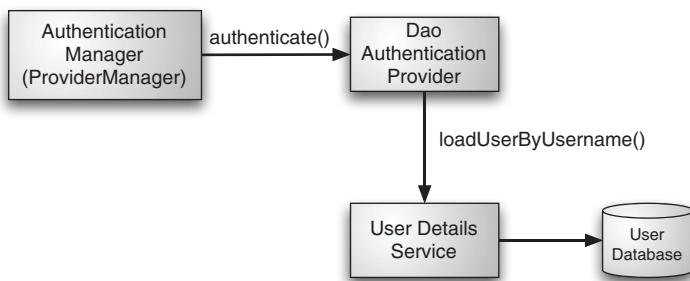


Figure 7.3 A `DaoAuthenticationManager` authenticates users on behalf of the authentication manager by pulling user information from a database.

Configuring a DaoAuthenticationProvider couldn't be simpler. The following XML excerpt shows how to declare a DaoAuthenticationProvider bean and wire it with a reference to its DAO:

```
<bean id="authenticationProvider"
      class="org.acegisecurity.providers.dao.DaoAuthenticationProvider">
    <property name="userDetailsService"
              ref="userDetailsService"/>
</bean>
```

The userDetailsService property is used to identify the bean that will be used to retrieve user information from the database. This property expects an instance of org.acegisecurity.userdetails.UserDetailsService. The question that remains is how the userDetailsService bean is configured.

The UserDetailsService interface requires that only one method be implemented:

```
public interface UserDetailsService {
    UserDetails loadUserByUsername(String username)
        throws UsernameNotFoundException, DataAccessException;
}
```

This method is fairly self-explanatory and you may already be thinking of several ways that you can implement this interface. But before you start writing your own implementation of UserDetailsService, you may be interested to know that Spring Security comes with two ready-made implementations of AuthenticationDao to choose from: InMemoryDaoImpl and JdbcDaoImpl. Let's see how these two classes work to look up user details, starting with InMemoryDaoImpl.

Using an in-memory DAO

Although it may seem natural to assume that an AuthenticationDao object will always query a relational database for user information, that doesn't necessarily have to be the case. If your application's authentication needs are trivial or for development-time convenience, it may be simpler to configure your user information directly in the Spring configuration file.

For that purpose, Spring Security comes with InMemoryDaoImpl, an implementation of UserDetailsService that draws its user information from its Spring configuration. Here's an example of how you may configure an InMemoryDaoImpl in the Spring configuration file:

```
<bean id="authenticationDao"
      class="org.acegisecurity.userdetails.memory.InMemoryDaoImpl">
    <property name="userMap">
      <value>
```

```

palmerd=4moreyears,disabled,ROLE_PRESIDENT
bauerj=ineedsleep,ROLE_FIELD_OPS
obrianc=nosmile,ROLE_SR_ANALYST,ROLE_OPS
myersn=traitor,disabled,ROLE_CENTRAL_OPS
</value>
</property>
</bean>
```

The `userMap` property is configured with an `org.acegisecurity.userdetails.memory.UserMap` object that defines a set of usernames, passwords, and privileges. Fortunately, you needn't concern yourself with constructing a `UserMap` instance when wiring `InMemoryDaoImpl` because there's a property editor that handles the conversion of a `String` to a `UserMap` object for you.

On each line of the `userMap`, `String` is a name-value pair where the name is the username and the value is a comma-separated list that starts with the user's password and is followed by one or more names that are the authorities to be granted to the user. Figure 7.4 breaks down the format of an entry in the user map.

In the declaration of the `authenticationDao` bean earlier, four users are defined: `parlmerd`, `bauerj`, `obrianc`, and `myersn`. Respectively, their passwords are `4moreyears`, `ineedsleep`, `nosmile`, and `traitor`. The authorities are granted as follows:

- `ROLE_PRESIDENT` authority has been given to the user whose username is `palmerd`.
- `ROLE_FIELD_OPS` has been given to `bauerj`.
- `ROLE_CENTRAL_OPS` has been given to `myersn`.
- The `obrianc` user has been granted two authorities: `ROLE_SR_ANALYST` and `ROLE_OPS`.

Take special note of the `palmerd` and `myersn` users. A special `disabled` flag immediately follows their passwords, indicating that they have been disabled (and thus can't authenticate).

Although `InMemoryDaoImpl` is convenient and simple, it has some obvious limitations. Primarily, security administration requires that you edit the Spring

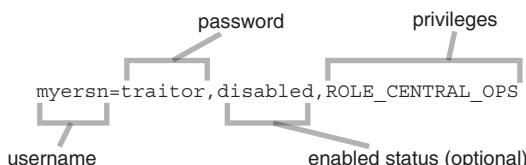


Figure 7.4
A Spring Security user map maps a username to a password, granted privileges, and optionally their status.

configuration file and redeploy your application. While this is acceptable (and maybe even helpful) in a development environment, it is probably too cumbersome for production use. Therefore, I strongly advise against using `InMemoryDaoImpl` in a production setting. Instead, you should consider using `JdbcDaoImpl`, which we'll look at next.

Declaring a JDBC DAO

`JdbcDaoImpl` is a simple, yet flexible, authentication DAO that retrieves user information from a relational database. In its simplest form, all it needs is a reference to a `javax.sql.DataSource`, and it can be declared in the Spring configuration file as follows:

```
<bean id="authenticationDao"
      class="org.acegisecurity.userdetails.jdbc.JdbcDaoImpl">
    <property name="dataSource" ref="dataSource"/>
</bean>
```

As configured here, `JdbcDaoImpl` makes some basic assumptions about how user information is stored in the database. Specifically, it assumes a `Users` table and an `Authorities` table, as illustrated in figure 7.5.

When `JdbcDaoImpl` looks up user information, it will query with the following SQL:

```
SELECT username, password, enabled
  FROM users
 WHERE username = ?
```

Likewise, when looking up a user's granted authorities, `JdbcDaoImpl` will use the following SQL:

```
SELECT username, authority
  FROM authorities
 WHERE username = ?
```

While the table structures assumed by `JdbcDaoImpl` are straightforward, they probably do not match the tables you have set up for your own application's security. For instance, in the RoadRantz application, the `Motorist` table holds registered users' usernames (in the `email` column) and password. Does this mean that we can't use `JdbcDaoImpl` to authenticate motorists in the RoadRantz application?

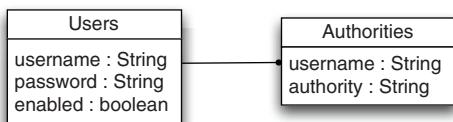


Figure 7.5
The database tables assumed by `JdbcDaoImpl`.

Not at all. But if we are to use `JdbcDaoImpl`, we must help it out a bit by telling it how to find the user information by setting the `usersByUsernameQuery` property. The following adjustment to the `authenticationDao` bean sets it up to query users from RoadRantz's Motorist table:

```
<bean id="authenticationDao"
      class="org.acegisecurity.userdetails.jdbc.JdbcDaoImpl">
    <property name="dataSource" ref="dataSource" />
    <property name="usersByUsernameQuery">
      <value>
        SELECT email as username, password, enabled
        FROM Motorist
        WHERE email=?
      </value>
    </property>
</bean>
```

Now `JdbcDaoImpl` knows to look in the `Motorist` table for authentication information. But we must also tell `JdbcDaoImpl` how to query the database for a user's granted authorities. For that we'll set the `authoritiesByUsernameQuery` property:

```
<bean id="authenticationDao"
      class="org.acegisecurity.userdetails.jdbc.JdbcDaoImpl">
    <property name="dataSource" ref="dataSource" />
    ...
    <property name="authoritiesByUsernameQuery">
      <value>
        SELECT email as username, privilege as authority
        FROM Motorist_Privileges mp, Motorist m
        WHERE mp.motorist_id = m.id
          AND m.email=?
      </value>
    </property>
</bean>
```

Here we've configured `JdbcDaoImpl` to retrieve the motorist's granted authorities from the `Motorist_Privileges` table. The query joins in the `Motorist` table because the `Motorist_Privileges` table only knows about a `Motorist` through a foreign key and `JdbcDaoImpl` expects the query to retrieve the authorities by username.

Working with encrypted passwords

When `DaoAuthenticationProvider` compares the user-provided password (at authentication) with the one retrieved from the database, it assumes that the password has been stored unencrypted. To beef up security, you may want to encrypt the password before storing it in the database. But if the password is stored encrypted in the database, the user-provided password must also be encrypted before the two passwords can be compared.

To accommodate encrypted passwords, DaoAuthenticationProvider can be wired with a password encoder. Spring Security comes with several password encoders to choose from, as described in table 7.2.

Table 7.2 Spring Security's password encoders.

Password encoder (<code>org.acegisecurity.providers.*</code>)	Purpose
<code>encoding.Md5PasswordEncoder</code>	Performs Message Digest (MD5) encoding on the password
<code>encoding.PlaintextPasswordEncoder</code>	Performs no encoding on the password, returning it unaltered
<code>encoding.ShaPasswordEncoder</code>	Performs Secure Hash Algorithm (SHA) encoding on the password
<code>ldap.authenticator.LdapShaPasswordEncoder</code>	Encodes the password using LDAP SHA and salted-SHA (SSHA) encodings

By default DaoAuthenticationProvider uses the `PlaintextPasswordEncoder`, which means that the password is left unencoded. But we can specify a different encoding by wiring DaoAuthenticationProvider's `passwordEncoder` property. For example, here's how to wire DaoAuthenticationProvider to use MD5 encoding:

```
<bean id="daoAuthenticationProvider"
    class="org.acegisecurity.providers.dao.
        ↗    DaoAuthenticationProvider">
    <property name="userDetailsService" ref="authenticationDao" />
    <property name="passwordEncoder">
        <bean class="org.acegisecurity.providers.encoding.
            ↗    Md5PasswordEncoder" />
    </property>
</bean>
```

You'll also need to set a *salt source* for the encoder. A salt source provides the *salt*, or encryption key, for the encoding. Spring Security provides two salt sources:

- `SystemWideSaltSource`—Provides the same salt for all users
- `ReflectionSaltSource`—Uses reflection on a specified property of the user's `User` object to generate the salt

`ReflectionSaltSource` is the more secure of the two salt sources because each user's password will likely be encoded using a different salt value. Even if a hacker were to figure out the salt used to encode one user's password, it's unlikely that

they'll be able to use the same salt to crack another user's password. To use a ReflectionSaltSource, wire it into DaoAuthenticationProvider's saltSource property like this:

```
<bean id="daoAuthenticationProvider"
    class="org.acegisecurity.providers.dao.
        ↗   DaoAuthenticationProvider">
    <property name="userDetailsService" ref="authenticationDao" />
    <property name="passwordEncoder">
        <bean class="org.acegisecurity.providers.encoding.
            ↗   Md5PasswordEncoder" />
    </property>
    <property name="saltSource">
        <bean class="org.acegisecurity.providers.dao.salt.
            ReflectionSaltSource">
            <property name="userPropertyToUse" value="userName" />
        </bean>
    </property>
</bean>
```

Here the user's `userName` property is used as the salt to encode the user's password. It's important that the salt be static and never change. Otherwise, it will be impossible to authenticate the user (unless the password is re-encoded after the change using the new salt).

Although `ReflectionSaltSource` is certainly more secure, `SystemWideSaltSource` is much simpler and is sufficient for most circumstances. `SystemWideSaltSource` uses a single salt value for encoding all users' passwords. To use a `SystemWideSaltSource`, wire the `saltSource` property like this:

```
<bean id="daoAuthenticationProvider"
    class="org.acegisecurity.providers.dao.
        ↗   DaoAuthenticationProvider">
    <property name="userDetailsService" ref="authenticationDao" />
    <property name="passwordEncoder">
        <bean class="org.acegisecurity.providers.encoding.
            ↗   Md5PasswordEncoder" />
    </property>
    <property name="saltSource">
        <bean class="org.acegisecurity.providers.dao.salt.
            ↗   SystemWideSaltSource">
            <property name="systemWideSalt" value="ABC123XYZ789" />
        </bean>
    </property>
</bean>
```

In this case, the same salt value, `ABC123XYZ789`, is used for encoding all passwords.

Caching user information

Every time that a request is made to a secured resource, the authentication manager is asked to retrieve the user's security information. But if retrieving the user's information involves performing a database query, querying for the same data every time may hinder application performance. Recognizing that a user's information will not frequently change, it may be better to cache the user data upon the first query and retrieve it from cache with every subsequent request.

To enable caching of user information, we must provide DaoAuthenticationProvider with an implementation of the org.acegisecurity.providers.dao.UserCache interface. This interface mandates the implementation of three methods:

```
public UserDetails getUserFromCache(String username);
public void putUserInCache(UserDetails user);
public void removeUserFromCache(String username);
```

The methods in the UserCache are self-explanatory, providing the ability to put, retrieve, or remove user details from the cache. It would be simple enough for you to write your own implementation of UserCache. However, Spring Security provides two convenient implementations that you should consider before developing your own:

- org.acegisecurity.providers.dao.cache.NullUserCache
- org.acegisecurity.providers.dao.cache.EhCacheBasedUserCache

NullUserCache does not actually perform any caching at all. Instead, it always returns null from its `getUserFromCache()` method, forcing DaoAuthenticationProvider to query for the user information. This is the default UserCache used by DaoAuthenticationProvider.

EhCacheBasedUserCache is a more useful cache implementation. As its name implies, it is based on EHCache. Using EHCache with DaoAuthenticationProvider is simple. Simply wire an EhCacheBasedUserCache bean into DaoAuthenticationProvider's `userCache` property:

```
<bean id="daoAuthenticationProvider"
      class="org.acegisecurity.providers.dao.
          ↗ DaoAuthenticationProvider">
    <property name="userDetailsService" ref="authenticationDao" />
    ...
    <property name="userCache">
      <bean class="org.acegisecurity.providers.dao.cache.
          ↗ EhCacheBasedUserCache">
        <property name="cache" ref="ehcache" />
      </bean>
    </property>
  </bean>
```

```

        </bean>
    </property>
</bean>
```

The cache property refers to an ehcache bean, which should be an EHCache Cache object. One way to get such a Cache object is to use the Spring Modules' cache module. For example, the following XML uses Spring Modules to configure EHCache:

```

<bean id="ehcache"
      class="org.springframework.cache.ehcache.EhCacheFactoryBean">
    <property name="cacheManager" ref="cacheManager" />
    <property name="cacheName" value="userCache" />
</bean>

<bean id="cacheManager"
      class="org.springframework.cache.ehcache.
           ↗ EhCacheManagerFactoryBean">
    <property name="configLocation" value="classpath:ehcache.xml" />
</bean>
```

As you may recall from chapter 5, Spring Modules' EhCacheFactoryBean is a Spring factory bean that produces an EHCache Cache object. The actual caching configuration is found in the ehcache.xml file, which will be retrieved from the classpath.

DaoAuthenticationProvider is great when your application's security information is kept in a relational database. Often, however, an application's security is architected to authenticate against an LDAP server. Let's see how to use Spring Security's LdapAuthenticationProvider, which is a more suitable choice when authentication must happen via LDAP.

7.2.3 **Authenticating against an LDAP repository**

Spring Security supports authentication against LDAP through LdapAuthenticationProvider, an authentication provider that knows how to check user credentials against an LDAP repository. The following `<bean>` illustrates a typical configuration for LdapAuthenticationProvider:

```

<bean id="ldapAuthProvider"
      class="org.acegisecurity.providers.ldap.
           ↗ LdapAuthenticationProvider">
    <constructor-arg ref="authenticator" />
    <constructor-arg ref="populator" />
</bean>
```

As you can see, there's not much exciting about the LdapAuthenticationProvider. There are no details on how to find the LDAP server or about the

repository's initial context. Instead, `LdapAuthenticationProvider` is wired with an authenticator and a populator through constructor injection. What are those beans and what are they used for?

In fact, although `LdapAuthenticationProvider` claims to know how to talk to an LDAP repository, it actually relies on two strategy objects to do the real work:

- The *authenticator* strategy handles the actual authentication (e.g., verification of user credentials) against the LDAP repository. The authenticator strategy can be any object that implements `org.acegisecurity.providers.ldap.LdapAuthenticator`.
- The *populator* strategy is responsible for retrieving a user's set of granted authorities from the LDAP repository. The populator strategy is any object that implements `org.acegisecurity.providers.ldap.LdapAuthoritiesPopulator`.

Because the authentication and authorities responsibilities are defined as strategies, separate from `LdapAuthenticationProvider`, you are able to wire in the strategy implementations that best fit your application's security needs.

So just how are the authenticator and populator beans defined? Let's start by looking at the authenticator bean, which defines the authentication strategy for `LdapAuthenticationProvider`.

Authenticating with LDAP binding

When it comes to authenticating against LDAP, two approaches are commonly taken:

- Binding to the LDAP server using the username and password of an LDAP user
- Retrieving a user's entry in LDAP and comparing the supplied password with a password attribute in the LDAP record

For bind authentication, Spring Security comes with an `LdapAuthenticator` implementation called `BindAuthenticator`. `BindAuthenticator` uses an LDAP bind operator to bind as a user to the LDAP server. This approach relies on the LDAP server to authenticate the user's credentials.

The following `<bean>` declares a `BindAuthenticator` in Spring:

```
<bean id="authenticator"
      class="org.acegisecurity.providers.ldap.authenticator.
            BindAuthenticator">
    <constructor-arg ref="initialDirContextFactory" />
```

```

<property name="userDnPatterns">
    <list>
        <value>uid={0},ou=motorists</value>
    </list>
</property>
</bean>

```

Here I've declared the `BindAuthenticator` to be injected through a constructor argument and through the `userDnPatterns` property. We'll come back to the constructor argument in a moment. First, let's consider the `userDnPatterns` property.

The `userDnPatterns` property is used to tell `BindAuthenticator` how to find a user in LDAP. It takes a list of one or more patterns that `BindAuthenticator` will use as the distinguished name (DN) to identify the user. In this case, we're only using a single DN pattern, as described in figure 7.6.

The `{0}` in the DN pattern is a pattern argument that serves as a placeholder for the username. For example, if the username is `cwagon`, the DN used to bind to LDAP will be `uid=cwagon,ou=motorists`.

Now back to the constructor argument. The main thing that a `BindAuthenticator` needs to know to be able to do its job is how to access the LDAP repository. Thus, it is constructed with a constructor argument wired to `initialDirContextFactory`, which is declared as follows:

```

<bean id="initialDirContextFactory"
    class="org.acegisecurity.ldap.
        ↗ DefaultInitialDirContextFactory">
    <constructor-arg
        value="ldap://ldap.roaddrantz.com:389/dc=roadrantz,dc=com" />
</bean>

```

`DefaultInitialDirContextFactory` captures all the information needed to connect to an LDAP server and produces a JNDI `DirContext` object. If you don't know much about JNDI or `DirContext`, don't worry about these details. Just keep in mind that `BindAuthenticator` uses `DefaultInitialDirContextFactory` to know how to get to the LDAP repository.

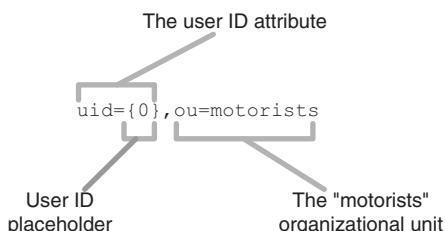


Figure 7.6
For our purposes, a user's distinguished name (DN) is broken into the user's ID (UID) and organizational unit (OU).

The constructor argument used to create `DefaultInitialDirContextFactory` is wired with the URL of the LDAP provider. In this case, I've wired it with a reference to the RoadRantz LDAP server² and established the initial context at `dc=roadrantz,dc=com`. The DN used to look up the user information will be relative to this initial context.

Authenticating by comparing passwords

As an alternative to bind authentication, Spring Security also supports authentication by password comparison with `PasswordComparisonAuthenticator`. `PasswordComparisonAuthenticator` works by comparing the supplied password with a password attribute (`userPassword`, by default) in the user record. Here's how it might be configured in Spring:

```
<bean id="authenticator"
      class="org.acegisecurity.providers.ldap.authenticator.
            ↗ PasswordComparisonAuthenticator">
    <constructor-arg ref="initialDirContextFactory" />
    <property name="userDnPatterns">
        <list>
            <value>uid={0},ou=motorists</value>
        </list>
    </property>
</bean>
```

Notice that with the exception of the class name, this `PasswordComparisonAuthenticator` declaration is identical to the `BindAuthenticator` declaration. That's because in their simplest forms, both are fundamentally the same. Both need an initial context factory to know how to get to the LDAP repository, and both need one or more DN patterns for locating user records.

But there are a few more properties you use to customize `PasswordComparisonAuthenticator`. For example, if the default `userPassword` attribute doesn't suit your needs, you can override it by wiring in a new value to the `passwordAttributeName` property. For example, declare `PasswordComparisonAuthenticator` as follows to compare the password against an attribute named `userCredentials`:

```
<bean id="authenticator"
      class="org.acegisecurity.providers.ldap.authenticator.
            ↗ PasswordComparisonAuthenticator">
    <constructor-arg ref="initialDirContextFactory" />
    <property name="userDnPatterns">
        <list>
```

² In case you're already thinking about it, don't try accessing the LDAP server at `ldap.roadrantz.com`. That address is just an example... there's no LDAP provider there.

```

        <value>uid={0},ou=motorists</value>
    </list>
</property>
<property name="passwordAttributeName" value="userCredentials" />
</bean>
```

Another customization you may choose is how the password is encoded in LDAP. By default, `PasswordComparisonAuthenticator` uses Spring Security's `LdapShaPasswordEncoder` to encode the password before comparison. `LdapShaPasswordEncoder` supports LDAP Secure Hash Algorithm (SHA) and SSHA (salted-SHA) encodings. But if these don't suit your needs, any implementation of `org.acegisecurity.providers.encoding.PasswordEncoder`, including those in table 7.2, can be wired into the `passwordEncoder` property.

For example, should you store the password in LDAP in plain text (not advised, but possible), declare `PasswordComparisonAuthenticator` like this:

```

<bean id="authenticator"
      class="org.acegisecurity.providers.ldap.authenticator.
            ↗ PasswordComparisonAuthenticator">
    <constructor-arg ref="initialDirContextFactory" />
    <property name="userDnPatterns">
        <list>
            <value>uid={0},ou=motorists</value>
        </list>
    </property>
    <property name="passwordEncoder">
        <bean class="org.acegisecurity.providers.encoding.
            ↗ PlaintextPasswordEncoder" />
    </property>
</bean>
```

Before we move on to the populator strategy bean, let's make one last tweak to the `initialDirContextFactory` bean.

Unlike `BindAuthenticator`, `PasswordComparisonAuthenticator` doesn't bind to LDAP using the user's DN. Some LDAP providers allow anonymous binding, in which case the `initialDirContextFactory` will work as is. However, for security reasons, most LDAP providers do not allow anonymous binding, so we'll need to provide a manager DN and password for `DefaultInitialDirContextFactory` to bind with:

```

<bean id="initialDirContextFactory"
      class="org.acegisecurity.ldap.
            ↗ DefaultInitialDirContextFactory">
    <constructor-arg
        value="ldap://ldap.roaddrantz.com:389/dc=roadrantz,dc=com"/>
    <property name="managerDn"
```

```
    value="cn=manager,dc=roadrantz,dc=com" />
<property name="managerPassword" value="letmein" />
</bean>
```

When DefaultInitialDirContextFactory accesses LDAP, it will bind as the manager and act on behalf of the user when comparing the user's password.

Now let's configure the populator strategy bean to complete the LDAP authentication picture.

Declaring the populator strategy bean

Authenticating the user is only the first step performed by LdapAuthenticationProvider. Once the user's identity is confirmed, LdapAuthenticationProvider must retrieve a list of the user's granted authorities to determine what rights the user has within the application.

As with authentication, LdapAuthenticatorProvider uses a strategy object to find a user's granted authorities from LDAP. Spring Security comes with one implementation of the LdapAuthoritiesPopulator interface: DefaultLdapAuthoritiesPopulator. Here's how DefaultLdapAuthoritiesPopulator is configured in Spring:

```
<bean id="populator"
  class="org.acegisecurity.providers.ldap.populator.
    ↗ DefaultLdapAuthoritiesPopulator">
<constructor-arg ref="initialDirContextFactory" />
<constructor-arg value="ou=groups" />
<property name="groupRoleAttribute" value="ou" />
</bean>
```

The first thing you'll notice is that DefaultLdapAuthoritiesPopulator is constructed with two constructor arguments, the first of which is a reference to our old friend, initialDirContextFactory. Just like the authenticator strategy bean, the populator strategy bean needs to know how to get to the LDAP repository to retrieve the user's granted authorities.

The second constructor argument helps DefaultLdapAuthoritiesPopulator find groups within the LDAP repository. Since an LDAP repository is hierarchical in nature, security groups could be found anywhere. This constructor argument specifies a base DN from which to search for groups. This base DN is relative to the initial context. Therefore, with the group base DN as ou=groups, we'll be searching for groups in ou=groups, dc=roadrantz, dc=com.

Finally, the groupRoleAttribute property specifies the name of the attribute that will contain role information (which effectively translates to a user's granted authorities). It defaults to cn, but for our example, we've set it to ou.

Configured this way, `DefaultLdapAuthoritiesPopulator` will retrieve all groups that the user is a member of—that is, all groups that have a `member` attribute with the user’s DN.

For example, suppose that you have an LDAP repository populated with the following LDIF:³

```
dn: ou=groups,dc=roadrantz,dc=com
objectClass: top
objectClass: organizationalUnit
ou: groups

dn: cn=motorists,ou=groups,dc=roadrantz,dc=com
objectClass: groupOfNames
objectClass: top
cn: motorists
description: Acegi Security Motorists
member: uid=craig,ou=people,dc=roadrantz,dc=com
member: uid=raymie,ou=people,dc=roadrantz,dc=com
ou: motorist

dn: cn=vips,ou=groups,dc=roadrantz,dc=com
objectClass: groupOfNames
objectClass: top
cn: vips
description: Acegi Security Motorists
member: uid=craig,ou=people,dc=roadrantz,dc=com
ou: vip
```

When the user named `craig` is authenticated, his granted authorities will include `ROLE_MOTORIST` and `ROLE_VIP`. But when `raymie` is authenticated, her granted authorities will only include `ROLE_MOTORIST`, because the `vips` group does not have her DN as a `member` attribute.

Note that the group name (which is in the `ou` attribute) is converted to uppercase and then prefixed with `ROLE_`. The case normalization is just a convenience that helps find a user’s authorities regardless of whether it’s lower or uppercase. You can turn off this behavior by setting the `convertToUpperCase` property to `false`.

The `ROLE_` prefix is provided for the sake of `RoleVoter`, which we’ll talk about in section 7.3.2. If you would rather use a different role prefix, you can configure `DefaultLdapAuthoritiesPopulator`’s `rolePrefix` property however you’d like.

For example, to turn off uppercase normalization and change the role prefix to `GROUP_`, configure `DefaultLdapAuthoritiesPopulator` like this:

³ LDAP aficionados know LDIF to be the LDAP Data Interchange Format. It’s the standard way of representing LDAP directory content.

```

<bean id="populator"
      class="org.acegisecurity.providers.ldap.populator.
           ↗ DefaultLdapAuthoritiesPopulator">
    <constructor-arg ref="initialDirContextFactory" />
    <constructor-arg value="ou=groups" />
    <property name="groupRoleAttribute" value="ou" />
    <property name="convertToUpperCase" value="false" />
    <property name="rolePrefix" value="GROUP_" />
  </bean>

```

One more tweak that you may want to make to `DefaultLdapAuthoritiesPopulator` is to change how it looks for members. Normally, it looks for groups whose member attribute has the user's DN. That's fine if your LDAP is set up to use the member attribute that way. But let's say that instead of `member`, your LDAP repository uses an `associate` attribute to track membership. In that case, you'll want to set the `groupSearchFilter` property like this:

```

<bean id="populator"
      class="org.acegisecurity.providers.ldap.populator.
           ↗ DefaultLdapAuthoritiesPopulator">
    <constructor-arg ref="initialDirContextFactory" />
    <constructor-arg value="ou=groups" />
    <property name="groupRoleAttribute" value="ou" />
    <property name="convertToUpperCase" value="false" />
    <property name="rolePrefix" value="GROUP_" />
    <property name="groupSearchFilter" value="(associate={0})" />
  </bean>

```

Notice that the `groupSearchFilter` property uses the `{0}` pattern argument to represent the user's DN.

Now we've wired in Spring Security's authentication processing beans to identify the user. Next let's see how Spring Security determines whether an authenticated user has the proper authority to access the secured resource.

7.3 Controlling access

Authentication is only the first step in Spring Security. Once Spring Security has figured out who the user is, it must decide whether to grant access to the resources that it secures. We've configured the authentication manager from figure 7.1. Now it's time to configure the access decision manager. An *access decision manager* is responsible for deciding whether the user has the proper privileges to access secured resources. Access decision managers are defined by the `org.acegisecurity.AccessDecisionManager` interface:

```

public interface AccessDecisionManager {
    public void decide(Authentication authentication, Object object,
        ConfigAttributeDefinition config)
        throws AccessDeniedException,
        InsufficientAuthenticationException;
    public boolean supports(ConfigAttribute attribute);
    public boolean supports(Class clazz);
}

```

The `supports()` methods consider the secured resource's class type and its configuration attributes (the access requirements of the secured resource) to determine whether the access decision manager is capable of making access decisions for the resource. The `decide()` method is where the ultimate decision is made. If it returns without throwing an `AccessDeniedException` or `InsufficientAuthenticationException`, access to the secured resource is granted. Otherwise, access is denied.

7.3.1 Voting access decisions

Spring Security's access decision managers are ultimately responsible for determining the access rights for an authenticated user. However, they do not arrive at their decision on their own. Instead, they poll one or more objects that vote on whether or not a user is granted access to a secured resource. Once all votes are in, the decision manager tallies the votes and arrives at its final decision.

Spring Security comes with three implementations of `AccessDecisionManager`, as listed in table 7.3. Each takes a different approach to tallying votes.

All of the access decision managers are configured the same in the Spring configuration file. For example, the following XML excerpt configures a `UnanimousBased` access decision manager:

Table 7.3 Spring Security's access decision managers help decide whether a user is granted access by tallying votes on whether to let the user in.

Access decision manager	How it decides to grant/deny access
<code>org.acegisecurity.vote.AffirmativeBased</code>	Allows access if at least one voter votes to grant access
<code>org.acegisecurity.vote.ConsensusBased</code>	Allows access if a consensus of voters vote to grant access
<code>org.acegisecurity.vote.UnanimousBased</code>	Allows access if all voters vote to grant access

```
<bean id="accessDecisionManager"
      class="org.acegisecurity.vote.UnanimousBased">
    <property name="decisionVoters">
      <list>
        <ref bean="roleVoter"/>
      </list>
    </property>
  </bean>
```

The `decisionVoters` property is where you provide the access decision manager with its list of voters. In this case, there's only one voter, which is a reference to a bean named `roleVoter`. Let's see how the `roleVoter` is configured.

7.3.2 Casting an access decision vote

Although access decision voters don't have the final say on whether access is granted to a secured resource (that job belongs to the access decision manager), they play an important part in the access decision process. An access decision voter's job is to consider the user's granted authorities alongside the authorities required by the configuration attributes of the secured resource. Based on this information, the access decision voter casts its vote for the access decision manager to use in making its decision.

An access decision voter is any object that implements the `org.acegisecurity.vote.AccessDecisionVoter` interface:

```
public interface AccessDecisionVoter {
    public static final int ACCESS_GRANTED = 1;
    public static final int ACCESS_ABSTAIN = 0;
    public static final int ACCESS_DENIED = -1;

    public boolean supports(ConfigAttribute attribute);
    public boolean supports(Class clazz);
    public int vote(Authentication authentication, Object object,
                   ConfigAttributeDefinition config);
}
```

As you can see, the `AccessDecisionVoter` interface is similar to that of `AccessDecisionManager`. The big difference is that instead of a `decide()` method that returns `void`, there is a `vote()` method that returns `int`. That's because an access decision voter doesn't decide whether to allow access—it only returns its vote as to whether or not to grant access.

When faced with the opportunity to place a vote, an access decision voter can vote one of three ways:

- ACCESS_GRANTED—The voter wishes to allow access to the secured resource.
- ACCESS_DENIED—The voter wishes to deny access to the secured resource.
- ACCESS_ABSTAIN—The voter is indifferent.

As with most Spring Security components, you are free to write your own implementation of `AccessDecisionVoter`. However, Spring Security comes with `RoleVoter`, a useful implementation that votes when the secured resources configuration attributes represent a role. More specifically, `RoleVoter` participates in a vote when the secured resource has a configuration attribute whose name starts with `ROLE_`.

The way that `RoleVoter` decides on its vote is by simply comparing all of the configuration attributes of the secured resource (that are prefixed with `ROLE_`) with all of the authorities granted to the authenticated user. If `RoleVoter` finds a match, it will cast an `ACCESS_GRANTED` vote. Otherwise, it will cast an `ACCESS_DENIED` vote.

The `RoleVoter` will only abstain from voting when the authorities required for access are not prefixed with `ROLE_`. For example, if the secured resource only requires non-role authorities (such as `CREATE_USER`), the `RoleVoter` will abstain from voting.

You can configure a `RoleVoter` with the following XML in the Spring configuration file:

```
<bean id="roleVoter"
      class="org.acegisecurity.vote.RoleVoter"/>
```

As stated earlier, `RoleVoter` only votes when the secured resource has configuration attributes that are prefixed with `ROLE_`. However, the `ROLE_` prefix is only a default. You may choose to override the default prefix by setting the `rolePrefix` property:

```
<bean id="roleVoter"
      class="org.acegisecurity.vote.RoleVoter">
    <property name="rolePrefix" value="GROUP_" />
</bean>
```

Here, the default prefix has been overridden to be `GROUP_`. Thus the `RoleVoter` will now only cast authorization votes on privileges that begin with `GROUP_`.

7.3.3 Handling voter abstinance

Knowing that any voter can vote to grant or deny access or abstain from voting, you may be wondering what would happen if all voters abstained from voting. Will the user be granted or denied access?

By default, all the access decision managers deny access to a resource if all the voters abstain. However, you can override this default behavior by setting the `allowIfAllAbstain` property on the access decision manager to true:

```
<bean id="accessDecisionManager"
      class="org.acegisecurity.vote.UnanimousBased">
    <property name="decisionVoters">
      <list>
        <ref bean="roleVoter"/>
      </list>
    </property>
    <property name="allowIfAllAbstain" value="true" />
  </bean>
```

By setting `allowIfAllAbstain` to true, you are establishing a policy of “silence is consent.” In other words, if all voters abstain from voting, access is granted as if they had voted to grant access.

Now that you’ve seen how Spring Security’s authentication and access control managers work, let’s put them to work. In the next section you’ll learn how to use Spring Security’s collection of servlet filters to secure a web application. Later, in section 7.6, we’ll dig deep into an application and see how to use Spring AOP to apply security at the method-invocation level.

7.4 Securing web applications

Spring Security’s support for web security is heavily based on servlet filters. These filters intercept an incoming request and apply some security processing before the request is handled by your application. Spring Security comes with a handful of filters that intercept servlet requests and pass them on to the authentication and access decision managers to enforce security. Depending on your needs, you may use several of the filters listed in table 7.4 to secure your application.

Even though table 7.4 lists 17 filters provided by Spring Security, most applications will suffice with only a handful of them. Specifically, when a request is submitted to a Spring-secured web application, it will pass through at least the following four filters (as illustrated in figure 7.7):

Table 7.4 Spring Security controls access to web applications through several servlet filters.

Filter (<code>org.acegisecurity.*</code>)	Purpose
<code>adapters.HttpRequestIntegrationFilter</code>	Populates the security context using information from the user principal provided by the web container.
<code>captcha.CaptchaValidationProcessingFilter</code>	Helps to identify a user as a human (as opposed to an automated process) using Captcha techniques. Captcha is a technique used to distinguish human users from automated/computer-driven users by challenging the user to identify something (typically an image) that is easily identified by a human, but difficult for a computer to make out.
<code>concurrent.ConcurrentSessionFilter</code>	Ensures that a user is not simultaneously logged in more than a set number of times.
<code>context.HttpSessionContextIntegrationFilter</code>	Populates the security context using information obtained from the HttpSession.
<code>intercept.web.FilterSecurityInterceptor</code>	Plays the role of security interceptor, deciding whether or not to allow access to a secured resource.
<code>providers.anonymous.AnonymousProcessingFilter</code>	Used to identify an unauthenticated user as an anonymous user.
<code>securechannel.ChannelProcessingFilter</code>	Ensures that a request is being sent over HTTP or HTTPS (as the need dictates).
<code>ui.basicauth.BasicProcessingFilter</code>	Attempts to authenticate a user by processing an HTTP Basic authentication.
<code>ui.cas.CasProcessingFilter</code>	Authenticates a user by processing a CAS (Central Authentication Service) ticket.
<code>ui.digestauth.DigestProcessingFilter</code>	Attempts to authenticate a user by processing an HTTP Digest authentication.
<code>ui.ExceptionTranslationFilter</code>	Handles any <code>AccessDeniedException</code> or <code>AuthenticationException</code> thrown by any of the other filters in the filter chain.
<code>ui.logout.LogoutFilter</code>	Used to log a user out of the application.
<code>ui.rememberme.RememberMeProcessingFilter</code>	Automatically authenticates a user who has asked to be “remembered” by the application.

Table 7.4 Spring Security controls access to web applications through several servlet filters. (continued)

Filter (<code>org.acegisecurity.*</code>)	Purpose
<code>ui.switchuser.SwitchUserProcessingFilter</code>	Used to switch out a user. Provides functionality similar to Unix's su.
<code>ui.webapp.AuthenticationProcessingFilter</code>	Accepts the user's principal and credentials and attempts to authenticate the user.
<code>ui.webapp.SiteminderAuthenticationProcessingFilter</code>	Authenticates a users by processing CA/Netegrity SiteMinder headers.
<code>ui.x509.X509ProcessingFilter</code>	Authenticates a user by processing an X.509 certificate submitted by a client web browser.
<code>wrapper.SecurityContextHolderAwareRequestFilter</code>	Populates the servlet request with a request wrapper.

- 1 Due to the stateless nature of HTTP, Spring Security needs a way to preserve a user's authentication between web requests. An *integration filter* is responsible for retrieving a previously stored authentication (most likely stored in the HTTP session) at the beginning of a request so that it will be ready for Spring Security's other filters to process.
- 2 Next, one of the *authentication-processing filters* will determine if the request is an authentication request. If so, the pertinent user information (typically a username/password pair) is retrieved from the request and passed on to the authentication manager to determine the user's identity. If this is not an authentication request, the filter performs no processing and the request flows on down the filter chain.
- 3 The next filter in line is the *exception translation filter*. The exception translation filter's sole purpose in life is to translate

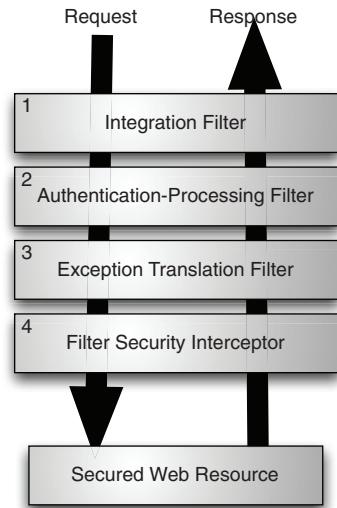


Figure 7.7 The flow of a request through Spring Security's core filters.

`AccessDeniedExceptions` and `AuthenticationExceptions` that may have been thrown into appropriate HTTP responses. If an `AuthenticationException` is detected, the request will be sent to an authentication entry point (e.g., login screen). If an `AccessDeniedException` is thrown, the default behavior will be to return an HTTP 403 error to the browser.

- 4 The last of the required filters is the *filter security interceptor*. This filter plays the part of the security interceptor (see section 7.1.1) for web applications. Its job is to examine the request and determine whether the user has the necessary privileges to access the secured resource. It doesn't work alone, though. It leans heavily on the authentication manager and the access decision manager to help it grant or restrict access to the resource.

If the user makes it past the filter security interceptor, the user will be granted access to the secured web resource. Otherwise, an `AccessDeniedException` will be thrown and the exception translation filter will handle it appropriately.

We'll explore each of these filters individually in more detail. But before you can start using them, you need to understand how Spring Security places a Spring-flavored twist on servlet filters.

7.4.1 **Proxying Spring Security's filters**

If you've ever used servlet filters, you know that for them to take effect, you must configure them in the web application's `web.xml` file, using the `<filter>` and `<filter-mapping>` elements. While this works, it doesn't lend itself to configuration using dependency injection.

For example, suppose you have the following filter declared in your `web.xml` file:

```
<filter>
    <filter-name>Foo</filter-name>
    <filter-class>FooFilter</filter-class>
</filter>
```

Now suppose that `FooFilter` needs a reference to a `Bar` bean to do its job. How can you inject an instance of `Bar` into `FooFilter`?

The short answer is that you can't. The `web.xml` file has no notion of dependency injection, nor is there a straightforward way of retrieving beans from the Spring application context and wiring them into a servlet filter. The only option you have is to use Spring's `WebApplicationContextUtils` to retrieve the `bar` bean from the Spring context. For example, you might place the following in the filter's code:

```
ApplicationContext ctx = WebApplicationContextUtils.  
    getWebApplicationContext(servletContext);  
Bar bar = (Bar) ctx.getBean("bar");
```

The problem with this approach, however, is that you must code Spring-specific code into your servlet filter. Furthermore, you end up hard-coding a reference to the name of the bar bean.

Fortunately, Spring Security provides a better way through `FilterToBeanProxy`.

Proxying servlet filters

`FilterToBeanProxy` is a special servlet filter that, by itself, doesn't do much. Instead, it delegates its work to a bean in the Spring application context, as illustrated in figure 7.8. The delegate bean implements the `javax.servlet.Filter` interface just like any other servlet filter, but is configured in the Spring configuration file instead of `web.xml`.



Figure 7.8 `FilterToBeanProxy` proxies filter handling to a delegate filter bean in the Spring application context.

By using `FilterToBeanProxy`, you are able to configure the actual filter in Spring, taking full advantage of Spring's support for dependency injection. The `web.xml` file only contains the `<filter>` declaration for `FilterToBeanProxy`. The actual `FooFilter` is configured in the Spring configuration file and uses setter injection to set the `bar` property with a reference to a `Bar` bean.

To use `FilterToBeanProxy`, you must set up a `<filter>` entry in the `web.xml` file. For example, if you are configuring a `FooFilter` using `FilterToBeanProxy`, you'd use the following code:

```
<filter>  
    <filter-name>Foo</filter-name>  
    <filter-class>org.acegisecurity.util.  
        ↗ FilterToBeanProxy</filter-class>  
    <init-param>  
        <param-name>targetClass</param-name>  
        <param-value>  
            com.roaddrantz.FooFilter
```

```

        </param-value>
    </init-param>
</filter>
```

Here the `targetClass` initialization parameter is set to the fully qualified class name of the delegate filter bean. When this `FilterToBeanProxy` is initialized, it will look for a bean in the Spring context whose type is `FooFilter`. `FilterToBeanProxy` will delegate its filtering to the `FooFilter` bean found in the Spring context:

```

<bean id="fooFilter"
      class="com.rodrantz.FooFilter">
    <property name="bar" ref="bar" />
</bean>
```

If a `FooFilter` bean isn't found, an exception will be thrown. If more than one matching bean is found, the first one found will be used.

Optionally, you can set the `targetBean` initialization parameter instead of `targetClass` to pick out a specific bean from the Spring context. For example, you might pick out the `fooFilter` bean by name by setting `targetBean` as follows:

```

<filter>
    <filter-name>Foo</filter-name>
    <filter-class>org.acegisecurity.
        util.FilterToBeanProxy</filter-class>
    <init-param>
        <param-name>targetBean</param-name>
        <param-value>fooFilter</param-value>
    </init-param>
</filter>
```

The `targetBean` initialization parameter enables you to be more specific about which bean to delegate filtering to, but requires that you match the delegate's name exactly between `web.xml` and the Spring configuration file. This creates extra work for you if you decide to rename the bean. For this reason, it's probably better to use `targetClass` instead of `targetBean`.

NOTE It may be interesting to know that there's nothing about `FilterToBeanProxy` that is specific to Spring Security or to securing web applications. You may find that `FilterToBeanProxy` is useful when configuring your own servlet filters. In fact, because it's so useful, a similar filter named `org.springframework.web.filter.DelegatingFilterProxy` was added to Spring in version 1.2.

Finally, you'll need to associate the filter to a URL pattern. The following `<filter-mapping>` ties the `Foo` instance of `FilterToBeanProxy` to a URL pattern of `/*` so that all requests are processed:

```
<filter-mapping>
  <filter-name>Foo</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

Regardless of whether you choose `targetClass` or `targetBean`, `FilterToBeanProxy` must be able to access the Spring application context. This means that the Spring context has to be loaded using Spring's `ContextLoaderListener` or `ContextLoaderServlet` (see chapter 13).

So now that you know how `FilterToBeanProxy` works, the burning question is: what does all of this have to do with Spring Security? I'm glad you asked.

As I mentioned earlier, Spring Security uses servlet filters in enforcing web security. Each of these filters must be injected with other beans from the Spring application context to do their job. For example, the `FilterSecurityInterceptor` needs to be injected with the `AuthenticationManager` and the `AccessDecisionManager` so that it can enforce security. Unfortunately, the servlet specification doesn't make it easy to do dependency injection on servlet filters. `FilterToBeanProxy` solves this problem by being the "front-man" for the real filters that are configured as beans in the Spring application context.

Proxying multiple filters

Now you're probably wondering, if `FilterToBeanProxy` handles requests by proxying to a Spring-configured bean, what is on the receiving end (on the Spring side)? That's an excellent question.

Actually, the bean that `FilterToBeanProxy` proxies to can be any implementation of `javax.servlet.Filter`. This could be any of Spring Security's filters, or it could be a filter of your own creation. But as I've already mentioned, Spring Security requires at least four and possibly a dozen or more filters to be configured. Does this mean that you have to configure a `FilterToBeanProxy` for each of Spring Security's filters?

Absolutely not. While it's certainly possible to add several `FilterToBeanProxy`s to `web.xml` (one for each of Spring Security's filters), that'd be way too much XML to write. To make life easier, Spring Security offers `FilterToBeanProxy`'s cohort, `FilterChainProxy`.

`FilterChainProxy` is an implementation of `javax.servlet.Filter` that can be configured to chain together several filters at once, as illustrated in figure 7.9.

`FilterChainProxy` intercepts the request from the client and sends it to `FilterChainProxy` for handling. `FilterChainProxy` then passes the request through one or more filters that are configured in the Spring application context. `FilterChainProxy` is configured like this in the Spring application context:

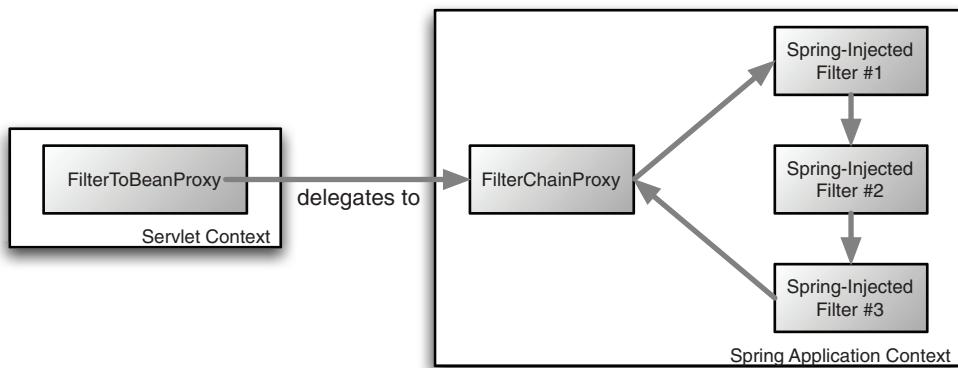


Figure 7.9 FilterChainProxy chains multiple filters together on behalf of FilterToBeanProxy.

```
<bean id="filterChainProxy"
    class="org.acegisecurity.util.FilterChainProxy">
    <property name="filterInvocationDefinitionSource">
        <value>
            CONVERT_URL_TO_LOWERCASE_BEFORE_COMPARISON
            PATTERN_TYPE_APACHE_ANT
            /**=filter1,filter2,filter3
        </value>
    </property>
</bean>
```

The `filterInvocationDefinitionSource` property takes a `String` that is parsed into a scheme that `FilterChainProxy` will use to chain filters together. In this example, the first line tells `FilterChainProxy` to normalize URL patterns to lowercase before comparing them. The next line says that Apache Ant-style paths are to be used when declaring URL patterns.

Finally, one or more URL-to-filter-chain mappings are provided. Here, the `/**` pattern (in Ant, this means all URLs will match) is mapped to three filters. The filter configured as the `filter1` bean will be the outermost filter and will receive the request first. The `filter2` bean is next. And the `filter3` bean will be the innermost bean and will be the last filter to receive the request before the actual secured resource is processed. When a response is returned, it flows in reverse order, from `filter3` to `filter1`.

Configuring proxies for Spring Security

Up to now, we've kept the configuration of the filter proxies mostly generic. But it's time to configure them for use in Spring Security. First up, let's configure a `FilterToBeanProxy` in `web.xml`:

```
<filter>
  <filter-name>Spring Security Filter Chain Proxy</filter-name>
  <filter-class>org.acegisecurity.util.
    ↪ FilterToBeanProxy</filter-class>
  <init-param>
    <param-name>targetClass</param-name>
    <param-value>org.acegisecurity.util.
      ↪ FilterChainProxy</param-value>
  </init-param>
</filter>
```

Here we've configured `FilterToBeanProxy` to proxy to any bean in the Spring context whose type is `FilterChainProxy`. This is perfect because, as you may have guessed, we're going to configure a `FilterChainProxy` in the Spring context.

But before we leave the `web.xml` file, we need to configure a filter mapping for the `FilterToBeanProxy`:

```
<filter-mapping>
  <filter-name>Spring Security Filter Chain Proxy</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

As with any servlet `<filter-mapping>`, the `<filter-name>` value must match the `<filter-name>` of the `<filter>` that is being mapped. As for the `<url-pattern>`, we recommend that you use `/*` so that all requests are piped through Spring Security and are potentially secured. Even if it isn't necessary to secure the entire application, filtering all requests through `/*` will keep the `web.xml` configuration simple. Later, when we configure `FilterSecurityInterceptor`, we can choose which parts of the application should be secured and which should not.

And that's all that's needed in the `web.xml` file! Even though Spring Security uses several filters to secure a web application, we only have to configure the one `FilterToBeanProxy` filter in `web.xml`. From here on out, we'll configure Spring Security in the Spring application context.

Speaking of the Spring application context, we're going to need a `FilterChainProxy` bean to handle the requests delegated from `FilterToBeanProxy`. For the RoadRantz application, let's start with the minimal Spring Security configuration by configuring a `FilterChainProxy` in the Spring application context (in `roadrantz-security.xml`) with the `<bean>` declaration shown in listing 7.1.

Listing 7.1 Configuring a FilterChainProxy for Spring Security

```
<bean id="filterChainProxy"
      class="org.acegisecurity.util.FilterChainProxy">
    <property name="filterInvocationDefinitionSource">
      <value>
        CONVERT_URL_TO_LOWERCASE_BEFORE_COMPARISON
        PATTERN_TYPE_APACHE_ANT
        /**=httpSessionIntegrationFilter,
         ↪ authenticationProcessingFilter,
         ↪ exceptionTranslationFilter,
         ↪ filterSecurityInterceptor
      </value>
    </property>
</bean>
```

Here, we've configured `FilterChainProxy` to chain together four of Spring Security's filters. We'll explore each of these filters in more detail in a moment. First, however, it's important to point out that if it weren't for `FilterChainProxy`, we'd have to configure four different `<filter>` entries and eight different `<filter-mapping>` entries in `web.xml`. But with `FilterChainProxy`, the `web.xml` configuration is simplified to a single `<filter>` and `<filter-mapping>` pair.

As the request comes in from the client and makes its way toward the secured resource, it passes through each of the filters. You can think of Spring Security's filters as different layers in the skin of an onion. Figure 7.10 shows how a request flows through each of Spring Security's filters on its way to a secured resource.

Aside from looking a bit like the opening segment of a Looney Tunes cartoon,⁴ figure 7.10 illustrates something very important about Spring Security. Although not all of Spring Security's filters are required for every application, it's crucial that the ones that are used be configured in a specific order in the `filterInvocationDefinitionSource` property. That's because some of the filters make assumptions about what security tasks have been performed before them. If the filters are configured out of order, they conflict with one another.

Each of the filters configured in the `filterInvocationDefinitionSource` property of `FilterChainProxy` refers to a `<bean>` configured in the Spring application context. Let's follow the path of the request through each of the filters, starting with the integration filter.

⁴ Admit it... you can't stop thinking about Porky Pig bursting out of the center of that picture and saying, "Th-th-d-th-d-th-d-d-that's all, folks!"

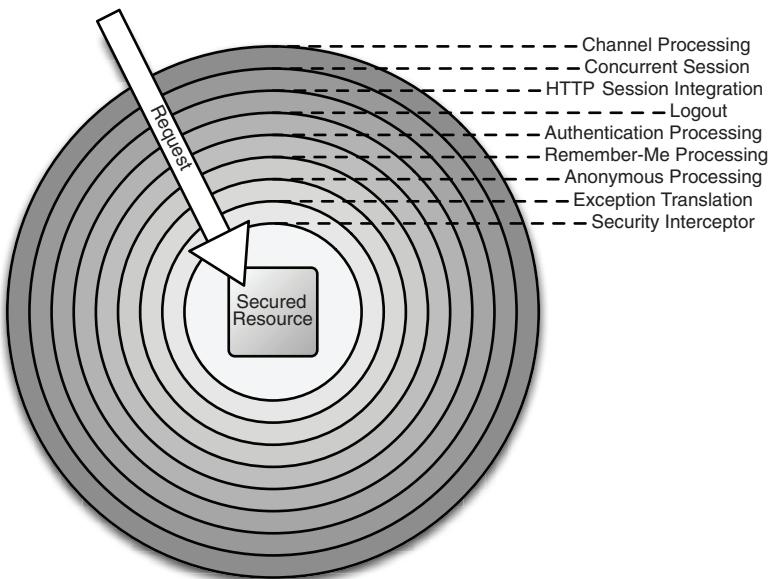


Figure 7.10 Spring Security's filters layer each other to apply security tasks.

7.4.2 Handling the security context

Have you seen the movie *Finding Nemo*? If so, you'll most certainly remember that one of the main characters was a blue tang (that's a fish, in case you didn't know) named Dory. Many of the funniest moments in the movie were a result of Dory's struggle with short-term memory loss. Throughout much of the movie, she would forget little things, such as where they were going or the name of Marlin, the clownfish who was looking for his son Nemo.

As it turns out, HTTP and Dory have a lot in common. You see, HTTP is a stateless protocol. That means that, like Dory, HTTP lives in the here-and-now and tends to forget things between requests. This poses a small problem for secure applications that are served over HTTP. Without something to help HTTP remember who you are, you'd have to log into an application with each request.

Fortunately, several solutions have been devised to help HTTP with its short-term memory loss. With Java-based web applications, sessions can be used to store data between requests. With each request, stored user information can be retrieved from the session, used to process the request, and then placed back into the session so that it's available for the next request.

The first Spring Security filter that a request must pass through is `HttpSessionContextIntegrationFilter`. This filter's main job is to try to remember an authenticated user between requests. It is configured in the Spring application context like this:

```
<bean id="httpSessionIntegrationFilter"
      class="org.acegisecurity.context.
      ↗ HttpSessionContextIntegrationFilter"/>
```

When a request first comes in, `HttpSessionContextIntegrationFilter` checks to see if it can find the user's authentication information in the session (stored there from a previous request). If so then `HttpSessionContextIntegrationFilter` makes the user information available for Spring Security to use in the course of the current request. At the end of the request, `HttpSessionContextIntegrationFilter` will deposit the user's authentication information back into the session so that it will be available for the next request.

If `HttpSessionContextIntegrationFilter` finds a user's authentication information in the session, there's no need for the user to log in again. But if the user's authentication can't be found, it probably means that they haven't logged in yet. To handle user login, we'll need to configure an authentication-processing filter, which is the next filter configured in `FilterChainProxy` and the next filter we'll discuss.

7.4.3 Prompting the user to log in

When securing web applications with Spring Security, authentication is performed using a tag-team made up of an authentication entry point and an authentication-processing filter. As illustrated in figure 7.11, an authentication entry point prompts the user for login information, which is then processed by the authentication-processing filter.

An authentication entry point starts the login process by prompting the user with a chance to provide their credentials. After the user submits the requested information, an authentication-processing filter attempts to authenticate the user.

Spring Security comes with five matched pairs of authentication entry points and authentication-processing filters, as described in table 7.5.



Figure 7.11 Authentication entry points and authentication-processing filters work together to authenticate a web user.

Table 7.5 Spring Security's authentication entry points prompt the user to log in. An authentication-processing filter processes the login request once the credentials are submitted.

Authentication entry point	Authentication-processing filter	Purpose
BasicProcessingFilterEntry Point	BasicProcessingFilter	Prompts the user to log in via a browser dialog using HTTP Basic authentication
AuthenticationProcessing FilterEntryPoint	AuthenticationProcessing Filter	Redirects the user to an HTML form-based login page
CasProcessingFilterEntry Point	CasProcessingFilter	Redirects the user to login page provided by JA-SIG's CAS single sign-on solution
DigestProcessingFilterEntry Point	DigestProcessingFilter	Prompts the user to log in via a browser dialog using HTTP Digest authentication
X509ProcessingFilterEntry Point	X509ProcessingFilter	Processes authentication using X.509 certificates

Let's take a closer look at how the authentication entry point and authentication-processing filter work together to authenticate a user. We'll examine a few of Spring Security's authentication options, starting with Spring Security's support for HTTP Basic authentication.

Basic authentication

The simplest form of web-based authentication is known as Basic authentication. Basic authentication works by sending an HTTP 401 (Unauthorized) response to the web browser. When the browser sees this response, it realizes that the server needs the user to log in. In response, the browser pops up a dialog box to prompt the user for a username and password (see figure 7.12).

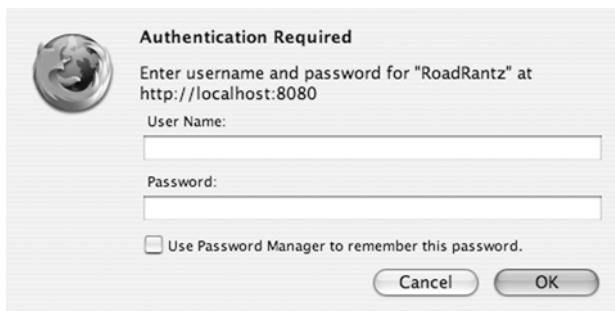


Figure 7.12
HTTP Basic authentication uses a browser-produced login dialog box to prompt a user for their credentials. This dialog box is from the Mac OS X version of Mozilla Firefox.

When the user submits the login, the browser sends it back to the server to perform the authentication. If authentication is successful, the user is sent to the desired target URL. Otherwise, the server may send back another HTTP 401 response and the browser will prompt the user again to log in.

Using Basic authentication with Spring Security starts with configuring a `BasicProcessingFilterEntryPoint` bean:

```
<bean id="authenticationEntryPoint"
      class="org.acegisecurity.ui.basicauth.
          ↗ BasicProcessingFilterEntryPoint">
    <property name="realmName" value="RoadRantz" />
</bean>
```

`BasicProcessingFilterEntryPoint` has only one property to be configured. The `realmName` property specifies an arbitrary `String` that is displayed in the login dialog box to give users some indication of what it is that they're being asked to log into. For example, the dialog box shown in figure 7.12 asks the user to enter a username and password for the RoadRantz realm.

After the user clicks the OK button in the login dialog box, the username and password are submitted via the HTTP header back to the server. At that point, `BasicProcessingFilter` picks it up and processes it:

```
<bean id="authenticationProcessingFilter"
      class="org.acegisecurity.ui.basicauth.
          ↗ BasicProcessingFilter">
    <property name="authenticationManager"
              ref="authenticationManager"/>
    <property name="authenticationEntryPoint"
              ref="authenticationEntryPoint"/>
</bean>
```

`BasicProcessingFilter` pulls the username and password from the HTTP header and sends them on to the authentication manager, which is wired in through the `authenticationManager` property. If authentication is successful, an `Authentication` object is placed into the session for future reference. Otherwise, if authentication fails, control is passed on to the authentication entry point (the `BasicProcessingFilterEntryPoint` wired in through the `authenticationEntryPoint` property) to give the user another chance.

Although Basic authentication is fine for simple applications, it has some limitations. Primarily, the login dialog box presented by the browser is neither user-friendly nor aesthetically appealing. Basic authentication doesn't fit the bill when you want a more professional-looking login.

For the RoadRantz application, we want an eye-appealing web page that shares the same look and feel as the rest of the application. Therefore, Spring Security's `AuthenticationProcessingFilterEntryPoint` is more appropriate for our needs. Let's see how it works.

Form-based authentication

`AuthenticationProcessingFilterEntryPoint` is an authentication entry point that prompts the user with an HTML-based login form. For the RoadRantz application, we'll configure it in `roaddrantz-security.xml` as follows:

```
<bean id="authenticationEntryPoint"
      class="org.acegisecurity.ui.webapp.
           ↗ AuthenticationProcessingFilterEntryPoint">
    <property name="loginFormUrl" value="/login.htm" />
    <property name="forceHttps" value="true" />
</bean>
```

`AuthenticationProcessingFilterEntryPoint` is configured here with two properties. The `loginFormUrl` property is set to a URL (relative to the web application's context) that will display the login page. The `forceHttps` property is set to true to force the login page to be displayed securely over HTTPS, even if the original request was made over HTTP.

Here we've set `loginFormUrl` to `/login.htm`. `loginFormUrl` can be configured with any URL that takes the user to an HTML form for login. In the case of the RoadRantz application, `/login.htm` is ultimately associated with a Spring MVC `UrlFilenameViewController` that displays the login page. Figure 7.13 shows what the RoadRantz login page might look like.

Regardless of how the login page is displayed, it's important that it contain an HTML form that resembles the following:

```
<form method="POST" action="j_acegi_security_check">
  <b>Username:</b><input type="text" name="j_username"><br>
  <b>Password:</b><input type="password" name="j_password"><br>
  <input type="submit" value="Login">
</form>
```

The login form must have two fields named `j_username` and `j_password` in which the user will enter the username and password, respectively. That's because those are the field names expected by `AuthenticationProcessingFilter`. As for the form's `action` attribute, it has been set to `j_acegi_security_check`, which will be intercepted by `AuthenticationProcessingFilter`.

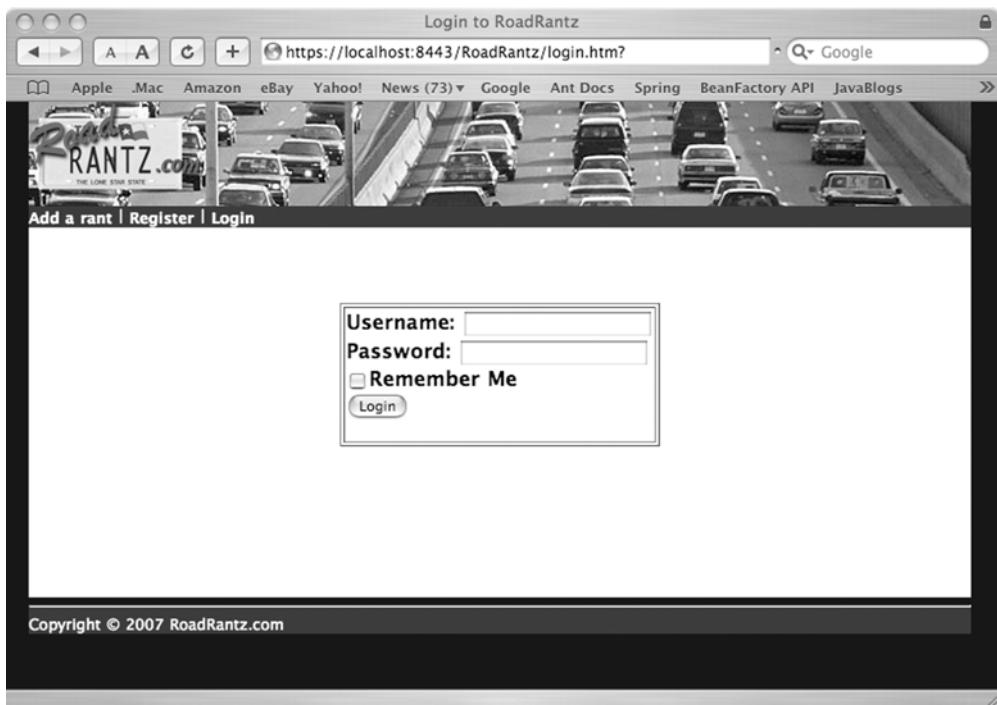


Figure 7.13 The RoadRantz login page is found at /login.htm, which is ultimately handled by Spring MVC's `UrlFilenameViewController`.

`AuthenticationProcessingFilter` is a filter that processes authentication based on the username and password information given to it in the `j_username` and `j_password` parameters. It is configured in `roaddrantz-security.xml` as follows:

```
<bean id="authenticationProcessingFilter"
    class="org.acegisecurity.ui.webapp.
        ↗ AuthenticationProcessingFilter">
    <property name="filterProcessesUrl"
        value="/j_acegi_security_check" />
    <property name="authenticationFailureUrl"
        value="/login.htm?login_error=1" />
    <property name="defaultTargetUrl" value="/" />
    <property name="authenticationManager"
        ref="authenticationManager"/>
</bean>
```

The `filterProcessesUrl` property tells `AuthenticationProcessingFilter` which URL it should intercept. This is the same URL that is in the login form's

action attribute. It defaults to /j_acegi_security_check, but I've explicitly defined it here to illustrate that you can change it if you'd like.

The authenticationFailureUrl property indicates where the user will be sent should authentication fail. In this case, we're sending them back to the login page, passing a parameter to indicate that authentication failed (so that an error message may be displayed).

Under normal circumstances, when authentication is successful, AuthenticationProcessingFilter will place an Authentication object in the session and redirect the user to their desired target page. The defaultTargetUrl property defines what will happen in the unusual circumstance where the target URL isn't known. This could happen if the user navigates directly to the login screen without first attempting to access a secured resource.

Finally, the authenticationManager property is wired with a reference to an authenticationManager bean. Just like all other authentication-processing filters, the form-based AuthenticationProcessingFilter relies on an authentication manager to help establish the user's identity.

Now we have an authentication processing filter and authentication entry point defined in the Spring configuration, ready for users to log in. But there's one loose end left to tie up. The authentication-processing filter is wired into the FilterChainProxy, but you're probably wondering what is supposed to be done with the authentication entry point. What part of Spring Security uses the authentication entry point to prompt the user for login?

I'll answer that question for you soon. But first, we'll need to look at the exception translation filter, the next filter in line to handle a secured request.

7.4.4 Handling security exceptions

In the course of providing security, any of Spring Security's filters may throw some variation of AuthenticationException or AccessDeniedException. AuthenticationException, for example, will be thrown if, for any reason, the user cannot be authenticated. This could be because the user provided an invalid username/password pair. Or it could even mean that the user hasn't even attempted to log in yet. Even if the user is successfully authenticated, they may not be granted the authority necessary to visit certain secured pages. In that case, AccessDeniedException will be thrown.

Without anything to handle Spring Security's AuthenticationException or AccessDeniedException, they'd flow up to the servlet container and be displayed in the browser as a really ugly stack trace. It goes without saying that this is less than ideal. We'd prefer to handle such exceptions in a more graceful manner.

That's where `ExceptionTranslationFilter` comes in. `ExceptionTranslationFilter` is configured at a level just outside of `FilterSecurityInterceptor` so that it will have a chance to catch the exceptions that may be thrown by `FilterSecurityInterceptor`. `ExceptionTranslationFilter` is configured in Spring as follows:

```
<bean id="exceptionTranslationFilter"
      class="org.acegisecurity.ui.ExceptionTranslationFilter">
    <property name="authenticationEntryPoint"
              ref="authenticationEntryPoint" />
</bean>
```

`ExceptionTranslationFilter` catches the exceptions thrown from `FilterSecurityInterceptor`... but what does it do with them?

Notice that `ExceptionTranslationFilter` is injected with a reference to the authentication entry point. If `ExceptionTranslationFilter` catches an `AuthenticationException`, it means that the user hasn't been successfully authenticated. In that case, the user is sent to the authentication entry point configured in the `authenticationEntryPoint` property to try to log in.

Handling authorization exceptions

An `AccessDeniedException` indicates that the user has been authenticated but has not been granted sufficient authority to access the resource that has been requested. In that case, an HTTP 403 error is returned to the browser. The HTTP 403 error means "forbidden" and indicates that the user isn't allowed to access a requested resource.

By default, `ExceptionTranslationFilter` uses an `AccessDeniedHandlerImpl` to deal with `AccessDeniedExceptions`. Unless otherwise configured, `AccessDeniedHandlerImpl` only sends an HTTP 403 error to the browser. Unfortunately, an HTTP 403 error is usually displayed in a user-unfriendly way in the browser.

But we can configure our own `AccessDeniedHandlerImpl` that will forward the user to a nicer-looking error page when `AccessDeniedException` is caught. The following XML configures an `AccessDeniedHandlerImpl` that sends the user to an error page at the URL `/error.htm`:

```
<bean id="accessDeniedHandler"
      class="org.acegisecurity.ui.AccessDeniedHandlerImpl">
    <property name="errorCode" value="403" />
    <property name="errorPage" value="/error.htm" />
</bean>
```

All that's left to do is to wire this `accessDeniedHandler` into the `ExceptionTranslationFilter`:

```
<bean id="exceptionTranslationFilter"
    class="org.acegisecurity.ui.ExceptionTranslationFilter">
    <property name="authenticationEntryPoint"
        ref="authenticationEntryPoint" />
    <property name="accessDeniedHandler"
        ref="accessDeniedHandler" />
</bean>
```

We've now declared three out of the four security filters required by Spring Security. The three filters configured thus far are the tumblers in Spring Security's lock. Now it's time to configure `FilterSecurityInterceptor`, the latch that decides whether or not to allow access to a web resource.

7.4.5 Enforcing web security

Whenever a user requests a page within a web application, that page may or may not be a page that needs to be secure. In Spring Security, a filter security interceptor handles the interception of requests, determining whether a request is secure and giving the authentication and access decision managers a chance to verify the user's identity and privileges. It is declared in the Spring configuration file as follows:

```
<bean id="filterSecurityInterceptor"
    class="org.acegisecurity.intercept.web.FilterSecurityInterceptor">
    <property name="authenticationManager"
        ref="authenticationManager" />
    <property name="accessDecisionManager"
        ref="accessDecisionManager" />
    <property name="objectDefinitionSource">
        <value>
            CONVERT_URL_TO_LOWERCASE_BEFORE_COMPARISON
            PATTERN_TYPE_APACHE_ANT
            /editProfile.htm=ROLE_MOTORIST
        </value>
    </property>
</bean>
```

`FilterSecurityInterceptor` plays the part of the security interceptor (as described in section 7.1.1) for web applications. When a request comes in for a resource (likely a web page or Spring MVC controller), `FilterSecurityInterceptor` will perform several checks to see whether the user is allowed to access the resource:

- Has the user been authenticated? If not, `FilterSecurityInterceptor` will throw an `AuthenticationException` (which will be handled by the exception translation filter, which will be handled by the exception translation filter).

- Is the requested resource secured? The `objectDefinitionSource` property defines which resources are to be secured and what privileges are required to access them. If the request's URL matches one of the URL patterns in `objectDefinitionSource` then the resource is secure.
- Has the user been granted privileges that are sufficient for accessing the resource? `FilterSecurityInterceptor` will compare the user's granted privileges with those declared as being required for the resource. If the user's privileges are sufficient, the request will be granted. If not, `FilterSecurityInterceptor` will throw an `AccessDeniedException` (that will be handled by the exception translation filter).

`FilterSecurityInterceptor` doesn't work alone when making these decisions. That's why it's wired with a reference to an authentication manager and a reference to an access decision manager.

As for the `objectDefinitionSource` property, this is how we declare which resources are secured and what privileges are required to access them. The first line indicates that we want all URL patterns to be normalized to lowercase before comparison (otherwise, the URL patterns will be case sensitive). The next line indicates that we'll be using Ant-style paths for declaring the URL patterns.

From the third line on, we can declare one or more URL patterns and what privilege is required to access each. In this case, we have one URL pattern that ensures that only authenticated users with the `ROLE_MOTORIST` role are allowed to visit the `editProfile.htm` page.

If the user has been authenticated and has appropriate privileges, `FilterSecurityInterceptor` will let the request continue. If, however, `FilterSecurityInterceptor` determines that the user doesn't have adequate privileges, either an `AuthenticationException` or an `AccessDeniedException` will be thrown.

At this point, we've configured the basic filters required to secure the RoadRantz application with Spring Security. But there's one more filter that, although not required, comes in handy for guaranteeing that secure information be transmitted securely in web requests. Next, I'll show you how to ensure that secure requests are carried over HTTPS using Spring Security's `ChannelProcessingFilter`.

7.4.6 Ensuring a secure channel

The letter "s" is the most important letter on the Internet. Anyone who has spent more than five minutes surfing the Web knows that most web pages are associated

with URLs that start with “`http://`”. That’s because most web pages are requested and sent using the HTTP protocol.

HTTP is perfect for most pages, but is woefully insufficient when confidential information is passed around on the Internet. Information sent over HTTP can be easily intercepted and read by nefarious hackers who will use it for their ill-purposed plans.

When information must be sent confidentially, the letter “`s`” goes to work. For those pages, you’ll find that the URL begins with “`https://`” instead of simply “`http://`”. With HTTPS, information is still sent using HTTP, but is sent on a different port and is encrypted so that if it is intercepted, it can’t be read by anyone for whom it isn’t meant.

Unfortunately, the problem with HTTPS is that the burden of ensuring that a page be transferred over HTTPS belongs to whoever writes the link to the secure page. In other words, for a page to be secured with encrypted HTTPS, it must be linked to with a URL that starts with “`https://`”. Without that one little “`s`” in there, the page will be sent unencrypted over HTTP.

Because it’s too easy to omit the all-important “`s`,” Spring Security offers a fool-proof way to ensure that certain pages be transferred using HTTPS, regardless of which URL was used to link to them. As illustrated in figure 7.14, `ChannelProcessingFilter` is a Spring Security filter that intercepts a request, checks to see if it needs to be secure and, if so, calls “`s`” to work by redirecting the request to an HTTPS form of the original request URL.

We’ve configured a `ChannelProcessingFilter` for the RoadRantz application in `roadrantz-security.xml` as follows:

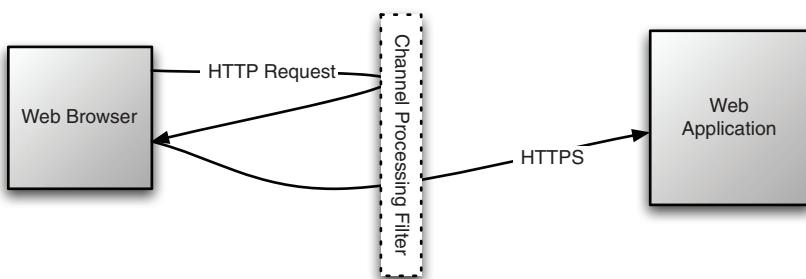


Figure 7.14 `ChannelProcessingFilter` redirects HTTP requests as HTTPS (and vice versa), ensuring the proper security for each request.

```
<bean id="channelProcessingFilter"
      class="org.acegisecurity.securechannel.
           → ChannelProcessingFilter">
    <property name="filterInvocationDefinitionSource">
      <value>
        CONVERT_URL_TO_LOWERCASE_BEFORE_COMPARISON
        PATTERN_TYPE_APACHE_ANT
        /login.htm=REQUIRES_SECURE_CHANNEL
        /j_acegi_security_check*=REQUIRES_SECURE_CHANNEL
        /**=REQUIRES_INSECURE_CHANNEL
      </value>
    </property>
    <property name="channelDecisionManager"
              ref="channelDecisionManager" />
  </bean>
```

The filterInvocationDefinitionSource property is configured to tell ChannelProcessingFilter which pages should be secured with HTTPS and which should not. It is configured with one or more URL patterns that are mapped to be either secure or not secure.

But before the URL patterns appear, we must set a few ground rules for how the URLs will be handled. The first line contains CONVERT_URL_TO_LOWERCASE_BEFORE_COMPARISON to tell Spring Security to normalize all URLs before comparing them to the URL patterns that will follow. The second line contains PATTERN_TYPE_APACHE_ANT, which indicates that the URL patterns will be presented using Apache Ant-style paths.

Each line that follows maps a URL pattern to its security requirements. In the RoadRantz application, the login page must be secure (so that nobody can intercept a user's password). Therefore, /login.htm is mapped to REQUIRES_SECURE_CHANNEL, indicating that it should be sent over HTTPS. Likewise, information sent to the URL that processes logins must also be encrypted. As you'll see soon, Spring Security's AuthenticationProcessingFilter responds to /j_acegi_security_check, so this URL pattern is also set to REQUIRES_SECURE_CHANNEL.

None of the other pages in the RoadRantz application require encryption. So the /** URL pattern (which, in Ant path syntax indicates all URLs) is set to REQUIRES_INSECURE_CHANNEL, specifying that all other pages must be sent over plain, unsecured HTTP. Notice that these pages *require* an insecure channel. That means that if these pages are accessed over HTTPS, ChannelProcessingFilter will redirect them to be sent over HTTP.

Managing channel decisions

While ChannelProcessingFilter handles the task of redirecting the HTTP requests to HTTPS (and vice versa), it doesn't necessarily need to redirect every request. Thus, it depends on a ChannelDecisionManagerImpl (wired into the channelDecisionManager property) to weigh the decision as to whether or not a request should be redirected. The ChannelDecisionManagerImpl is configured as follows:

```
<bean id="channelDecisionManager"
    class="org.acegisecurity.securechannel.
        ↪ ChannelDecisionManagerImpl">
    <property name="channelProcessors">
        <list>
            <bean class="org.acegisecurity.securechannel.
                ↪ SecureChannelProcessor"/>
            <bean class="org.acegisecurity.securechannel.
                ↪ InsecureChannelProcessor"/>
        </list>
    </property>
</bean>
```

Here we've configured ChannelDecisionManagerImpl with two channel processors—one for secure channel (HTTPS) processing and one for insecure (HTTP) channel processing.

Before we move past Spring Security's support for web-based security, let's see how to use Spring Security's tag library to enforce security rules within a page in the web application.

7.5 View-layer security

In most applications, there are certain elements that should only be displayed to a certain class of users. As you've already seen, Spring Security's filters prevent certain pages from being presented to users who are not granted a specific set of authorities.

But filters provide a coarse-grained security, limiting access at the request level. In some cases, you may want more fine-grained control over what the user is allowed to see. Maybe all users of an application will be allowed to see a certain page, but only users who are granted special authority may see certain elements on that page.

To provide fine-grained security in web applications, Spring Security comes with a small, but powerful, JSP tag library. This tag library provides only three tags, as listed in table 7.6.

Table 7.6 Spring Security's JSP tags for view-layer security.

Tag name	What it does
<authz:acl>	Conditionally renders the tag body if the user has been granted one of a set of specific permissions to a domain object
<authz:authentication>	Renders information about the user
<authz:authorize>	Conditionally renders the tag body if the user has been (or has not been) granted certain authorities

To use these tags in a JSP page, the tag library must be imported using the JSP <%@taglib%> directive:

```
<%@ taglib prefix="authz" uri="http://acegisecurity.org/authz" %>
```

Let's have a look at how to apply these tags, starting with <authz:authorize>.

7.5.1 Conditionally rendering content

The most useful of Spring Security's JSP tags is the <authz:authorize> tag. This tag effectively performs an *if* statement, evaluating whether or not the current user has been granted proper authority to view certain content. If so, the body of the tag will be rendered. Otherwise, the tag's content will be ignored.

To illustrate, let's add a welcome message and a link to logoff from the RoadRantz application. It doesn't make much sense to welcome a user who isn't authenticated and even less sense to offer them a logoff link. Therefore, we want to be certain that the user has been granted certain privileges before they're presented with that information. Using the *ifAllGranted* attribute of the <authz:authorize> tag, we might add the content to the view using this JSP snippet:

```
<authz:authorize ifAllGranted="ROLE_MOTORIST,ROLE_VIP">
    Welcome Motorist!<br/>
    <a href="j_acegi_logout">Logoff</a>
</authz:authorize>
```

Because the *ifAllGranted* attribute was used, the content contained in the body of the tag will only be rendered if the motorist has been granted both *ROLE_MOTORIST* *and* *ROLE_VIP* privileges. However, that is too restrictive, because while all users are granted *ROLE_MOTORIST* privileges, only a select few are granted *ROLE_VIP* privileges. So maybe the *ifAnyGranted* attribute would be more appropriate:

```
<authz:authorize ifAnyGranted="ROLE_MOTORIST,ROLE_VIP">
    Welcome Motorist!<br/>
```

```
<a href="j_acegi_logout">Logoff</a>
</authz:authorize>
```

In this case, the user must be granted either `ROLE_MOTORIST` or `ROLE_VIP` privileges for the welcome message and logoff link to be displayed.

Although it may seem obvious, it is worth pointing out that if you are only checking for a single privilege, the choice between `ifAllGranted` and `ifAnyGranted` is moot. Either attribute will work equally well when only one privilege is listed in the attribute value.

The final attribute option you have is `ifNotGranted`, which only renders the tag's content if the user has *not* been granted any of the authorities listed. For example, we'd use this to prevent content from being rendered to anonymous users:

```
<authz:authorize ifNotGranted="ROLE_ANONYMOUS">
    <p>This is super-secret content that anonymous users aren't
       allowed to see.</p>
</authz:authorize>
```

These three attributes cover a lot of ground by themselves. But some real security magic is conjured up when they're used together. When used together, these attributes are evaluated by logically AND'ing them together. For instance, consider the following:

```
<authz:authorize ifAllGranted="ROLE_MOTORIST"
                 ifAnyGranted="ROLE_VIP,ROLE_FAST_LANE"
                 ifNotGranted="ROLE_ADMIN">
    <p>Only special users see this content.</p>
</authz:authorize>
```

Used together this way, the tag's content will only be rendered if the user has been granted `ROLE_MOTORIST` privileges and either `ROLE_VIP` or `ROLE_FAST_LANE` privileges, and is not granted `ROLE_ADMIN` privileges. Even though this is a contrived example, you can imagine how powerful the `<authz:authorize>` tag can be by combining its three attributes.

Controlling what the user can see is only one facet of Spring Security's JSP tag library. Now let's see how to use Spring Security tags to display information about an authenticated user.

7.5.2 **Displaying user authentication information**

In the previous section, we added a welcome message to the RoadRantz application for authorized users. For simplicity's sake, the message was "Welcome

Motorist!” That’s a good start, but we’d like to make the application more personal by displaying the user’s login name instead of “Motorist.”

Fortunately, the user’s login is typically carried around in the object that is returned from the user’s `Authentication.getPrincipal()` method. All we need is a convenient way to access the principal object in the JSP. That’s what the `<authz:authentication>` tag is for.

The `<authz:authentication>` tag renders properties of the object that is returned from `Authentication.getPrincipal()` to JSP output. `Authentication.getPrincipal()` typically returns an implementation of Spring Security’s `org.acegisecurity.userdetails.UserDetails` interface, which includes a `getUsername()` method. Therefore, all we need to do to display the `username` property of the `UserDetails` object is to add the following `<authz:authentication>` tag:

```
<authz:authorize ifAnyGranted="ROLE_MOTORIST,ROLE_VIP">
    Welcome <authz:authentication operation="username"/>
</authz:authorize>
```

The `operation` attribute is a bit misleading, seeming to indicate that its purpose is to invoke a method. It’s true that it invokes a method, but more specifically, it invokes the getter method of the property whose name is specified in the `operation` attribute.

By default, the first letter of the `operation` value is capitalized and the result is prepended with `get` to produce the name of the method that will be called. In this case, the `getUsername()` method is called and its return value is rendered to the JSP output.

Now you’ve seen how to secure web applications using Spring Security’s filters. Before we are done with Spring Security, however, let’s have a quick look at how to secure method invocations using Spring Security and AOP.

7.6 **Securing method invocations**

Whereas Spring Security used servlet filters to secure web requests, Spring Security takes advantage of Spring’s AOP support to provide declarative method-level security. This means that instead of setting up a `SecurityEnforcementFilter` to enforce security, you’ll set up a Spring AOP proxy that intercepts method invocations and passes control to a security interceptor.

7.6.1 Creating a security aspect

Probably the easiest way to set up an AOP proxy is to use Spring's BeanNameAutoProxyCreator and simply list the beans that you'll want secured.⁵ For instance, suppose that you'd like to secure the courseService and billingService beans:

```
<bean id="autoProxyCreator" class=
    "org.springframework.aop.framework.autoproxy.
     → BeanNameAutoProxyCreator">
<property name="interceptorNames">
    <list>
        <value>securityInterceptor</value>
    </list>
</property>
<property name="beanNames ">
    <list>
        <value>courseService</value>
        <value>billingService</value>
    </list>
</property>
</bean>
```

Here the autoproxy creator has been instructed to proxy its beans with a single interceptor, a bean named securityInterceptor. The securityInterceptor bean is configured as follows:

```
<bean id="securityInterceptor"
    class="org.acegisecurity.intercept.method.
     → MethodSecurityInterceptor">
<property name="authenticationManager">
    <ref bean="authenticationManager"/>
</property>
<property name="accessDecisionManager">
    <ref bean="accessDecisionManager"/>
</property>
<property name="objectDefinitionSource">
    <value>
        com.springinaction.springtraining.service.
         → CourseService.createCourse=ROLE_ADMIN
        com.springinaction.springtraining.service.
         → CourseService.enroll*=ROLE_ADMIN,ROLE_REGISTRAR
    </value>
</property>
</bean>
```

⁵ This is only a suggestion. If you prefer one of the other mechanisms for proxying beans (as discussed in chapter 4), such as ProxyFactoryBean or DefaultAdvisorAutoProxyCreator, you are welcome to use those here instead.

`MethodSecurityInterceptor` does for method invocations what `FilterSecurityInterceptor` does for servlet requests. That is, it intercepts the invocation and coordinates the efforts of the authentication manager and the access decision manager to ensure that method requirements are met.

Notice that the `authenticationManager` and `accessDecisionManager` properties are the same as for `FilterSecurityInterceptor`. In fact, you may wire the same beans into these properties as you did for `FilterSecurityInterceptor`.

`MethodSecurityInterceptor` also has an `objectDefinitionSource` property just as `FilterSecurityInterceptor` does. But, although it serves the same purpose here as with `FilterSecurityInterceptor`, it is configured slightly differently. Instead of associating URL patterns with privileges, this property associates method patterns with privileges that are required to invoke the method.

A method pattern (see figure 7.15) includes the fully qualified class name and the method name of the method(s) to be secured. Note that you may use wildcards at either the beginning or the end of a method pattern to match multiple methods.

When a secured method is called, `MethodSecurityInterceptor` will determine whether the user has been authenticated and has been granted the appropriate authorities to call the method. If so, the call will proceed to the target method. If not, an `AcegiSecurityException` will be thrown. More specifically, an `AuthenticationException` will be thrown if the user cannot be authenticated. Or, if the user hasn't been granted authority to make the call, an `AccessDeniedException` will be thrown.

In keeping with Spring's exception philosophy, `AcegiSecurityException` is an unchecked exception. The calling code can either catch or ignore the exception.

Writing method security attributes in the Spring configuration file is only one way to declare method-level security. Now let's look at how to use Jakarta Commons Attributes to declare security attributes.

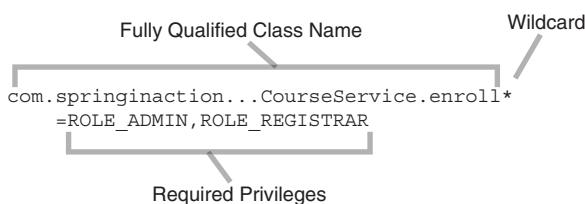


Figure 7.15 Method security rules are defined by mapping a fully qualified class name and method to the privileges required to execute that method. Wildcards may be used when specifying the method.

7.6.2 Securing methods using metadata

As with transactions and handler mappings, the first thing you must do is declare a metadata implementation to tell Spring how to load metadata. If you haven't already added a CommonsAttributes bean to your application context, add one now:

```
<bean id="attributes"
      class="org.springframework.metadata.commons.CommonsAttributes"/>
```

Next, you must declare an object definition source. In section 7.6.1, you defined an object definition source by setting the `objectDefinitionSource` property with a `String` that mapped security attributes to methods. But this time you're going to declare security attributes directly in the secured object's source code. Spring Security's `MethodDefinitionAttributes` is an object definition source that retrieves its security attributes from the secured object's metadata:

```
<bean id="objectDefinitionSource" class=
      "org.acegisecurity.intercept.method.MethodDefinitionAttributes">
    <property name="attributes"><ref bean="attributes"/></property>
</bean>
```

The `attributes` property of `MethodDefinitionAttributes` is wired with a reference to the `attributes` bean so that it will know to pull security attributes using Jakarta Commons Attributes.⁶

Now that the `objectDefinitionSource` is configured, wire it into the `objectDefinitionSource` property of `MethodSecurityInterceptor` (replacing the `String` definition from section 7.6.1):

```
<bean id="securityInterceptor"
      class="org.acegisecurity.intercept.method.
      ➔   MethodSecurityInterceptor">
  ...
    <property name="objectDefinitionSource">
      <ref bean="objectDefinitionSource"/>
    </property>
</bean>
```

Now you're ready to start tagging your code with security attributes. The only security attribute you need to know is `SecurityConfig`, which associates a privilege with a method. For example, the following snippet of code shows how to tag

⁶ When Spring supports JSR-175 annotations, you will wire the `attributes` property with a different metadata implementation.

the `enrollStudentInCourse()` method from `CourseService` to require either `ROLE_ADMIN` or `ROLE_REGISTRAR` privileges:

```
/**  
 *  @@org.acegisecurity.SecurityConfig("ROLE_ADMIN")  
 *  @@org.acegisecurity.SecurityConfig("ROLE_REGISTRAR")  
 */  
public void enrollStudentInCourse(Course course,  
        Student student) throws CourseException;
```

Declaring these security attributes on `enrollStudentInCourse()` is equivalent to the declaration of the `objectDefinitionSource` as defined in section 7.6.1.

7.7 **Summary**

Security is a very important aspect of many applications. Spring Security provides a mechanism for securing your applications that is based on Spring's philosophy of loose coupling, dependency injection, and aspect-oriented programming.

You may have noticed that this chapter presented very little Java code. We hope you weren't disappointed. The lack of Java code illustrates a key strength of Spring Security—loose coupling between an application and its security. Security is an aspect that transcends an application's core concerns. Using Spring Security, you are able to secure your applications without writing any security code directly into your application code.

Another thing you may have noticed is that much of the configuration required to secure an application with Spring Security is ignorant of the application that it is securing. The only Spring Security component that really needs to know any specifics about the secured application is the object definition source where you associate a secured resource with the authorities required to access the resource. Loose coupling runs both ways between Spring Security and its applications.



Spring and POJO-based remote services

This chapter covers

- Accessing and exposing RMI services
- Using Hessian and Burlap services
- Working with Spring's HTTP invoker
- Using Spring with web services

Imagine for a moment that you are stranded on a deserted island. This may sound like a dream come true. After all, who wouldn't want to get some solitude on a beach, blissfully ignorant of the goings-on of the outside world?

But on a deserted island, it's not piña coladas and sunbathing all the time. Even if you enjoy the peaceful seclusion, it won't be long before you'll get hungry, bored, and lonely. You can only live on coconuts and spear-caught fish for so long. You'll eventually need food, fresh clothing, and other supplies. And if you don't get in contact with another human soon, you may end up talking to a volleyball!

Many applications that you'll develop are like island castaways. On the surface they may seem self-sufficient, but in reality, they may collaborate with other systems, both within your organization and externally.

For example, consider a procurement system that needs to communicate with a vendor's supply chain system. Maybe your company's human resources system needs to integrate with the payroll system. Or even the payroll system may need to communicate with an external system that prints and mails paychecks. No matter the circumstance, your application will need to communicate with the other system to access services remotely.

Several remoting technologies are available to you, as a Java developer, including:

- Remote Method Invocation (RMI)
- Caucho's Hessian and Burlap
- Spring's own HTTP invoker
- Web services using SOAP and JAX-RPC

Regardless of which remoting technology you choose, Spring provides rich support for accessing and creating remote services. In this chapter, you'll learn how Spring both simplifies and complements these remoting services. But first, let's set the stage for this chapter with an overview of how remoting works in Spring.

8.1 An overview of Spring remoting

Remoting is a conversation between a client application and a service. On the client side, some functionality is required that isn't within the scope of the application. So, the application reaches out to another system that can provide the functionality. The remote application exposes the functionality through a *remote service*.

For example, suppose that in addition to user-entered rants, we'd like to display any traffic citations issued to a motorist. The RoadRantz application itself has

no record of traffic citations. But fortunately, we've discovered a third-party service that maintains a database of traffic citations as part of public record. The RoadRantz application could access the traffic citation service and then retrieve and display citations alongside its rants. This would involve a remote call to the traffic citation system, as illustrated in figure 8.1.

The conversation between RoadRantz and the remote service begins with a *remote procedure call (RPC)* from the RoadRantz application to the traffic citation service. On the surface, an RPC is similar to a call to a method on a local object. Both are synchronous operations, blocking execution in the calling code until the called procedure is complete.

The difference is a matter of proximity, with an analogy in human communication. If you are at the proverbial watercooler at work discussing the outcome of the weekend's football game, you are conducting a local conversation—that is, the conversation takes place between two people in the same room. Likewise, a local method call is one where execution flow is exchanged between two blocks of code within the same application.

On the other hand, if you were to pick up the phone to call a client in another city, your conversation would be conducted remotely over the telephone network. Similarly, RPC is when execution flow is handed off from one application to another application, theoretically on a different machine in a remote location over the network.

Spring supports remoting for several different RPC models, including Remote Method Invocation (RMI), Caucho's Hessian and Burlap, and Spring's own HTTP invoker. Table 8.1 outlines each of these models and briefly discusses their usefulness in various situations.

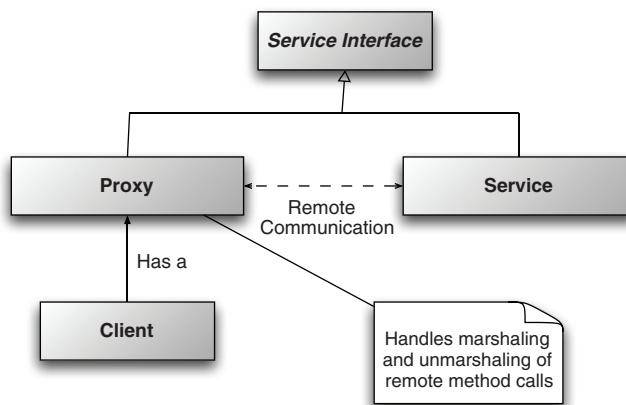


Figure 8.1
The RoadRantz application will make remote calls to the traffic citation system to dig into a motorist's records and find citations.

Table 8.1 The RPC models supported by Spring remoting.

RPC model	Useful when...
Remote Method Invocation (RMI)	Accessing/exposing Java-based services when network constraints such as firewalls aren't a factor
Hessian or Burlap	Accessing/exposing Java-based services over HTTP when network constraints are a factor
HTTP Invoker	Accessing/exposing Spring-based services when network constraints are a factor
JAX-RPC/SOAP	Accessing/exposing platform-neutral, SOAP-based web-services

Regardless of which remoting model you choose, you'll find that a common theme runs through Spring's support for each of the models. This means that once you understand how to configure Spring to work with one of the models, you'll have a very low learning curve if you decide to use a different model.

In all models, services can be configured into your application as Spring-managed beans. This is accomplished using a proxy factory bean that enables you to wire remote services into properties of your other beans as if they were local objects. Figure 8.2 illustrates how this works.

The client makes calls to the proxy as if the proxy were providing the service functionality. The proxy communicates with the remote service on behalf of the client. It handles the details of connecting and making remote calls to the remote service.

What's more, if the call to the remote service results in a `java.rmi.RemoteException`, the proxy handles that exception and rethrows it as an unchecked `org.springframework.remoting.RemoteAccessException`. Remote exceptions usually signal problems such as network or configuration issues that can't be gracefully recovered from. Since there's usually very little that a client can do to gracefully recover from a remote exception, rethrowing a `RemoteAccessException` makes it optional for the client to handle the exception.

On the service side, you are able to expose the functionality of any Spring-managed bean as a remote service using any of the models listed in table 8.1. Figure 8.3 illustrates how remote exporters expose bean methods as remote services.

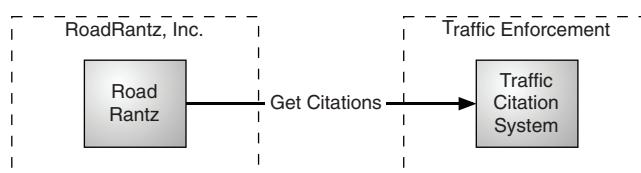


Figure 8.2
In Spring, remote services are proxied so that they can be wired into client code as if they were any other Spring bean.

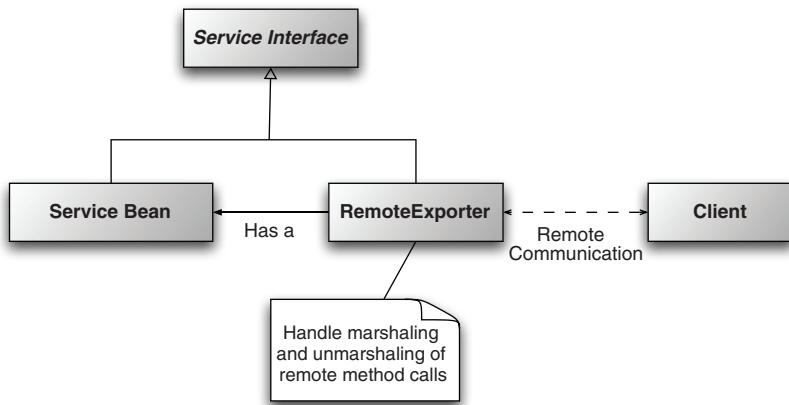


Figure 8.3 Spring-managed beans can be exported as remote services using `RemoteExporters`.

Whether you'll be developing code that consumes remote services, implements those services, or both, working with remote services in Spring is purely a matter of configuration. You won't have to write any Java code to support remoting. Your service beans don't have to be aware that they are involved in an RPC (although any beans passed to or returned from remote calls may need to implement `java.io.Serializable`).

Let's start our exploration of Spring's remoting support by looking at RMI, the original remoting technology for Java.

8.2 Working with RMI

If you've been working in Java for any length of time, you've no doubt heard of (and probably used) *Remote Method Invocation (RMI)*. RMI—first introduced into the Java platform in JDK 1.1—gives Java programmers a powerful way to conduct communication between Java programs. Before RMI, the only remoting options available to Java programmers were CORBA (which at the time required the purchase of a third-party Object Request Broker, or ORB) or handwritten socket programming.

But developing and accessing RMI services is tedious, involving several steps, both programmatic and manual. Spring simplifies the RMI model by providing a proxy factory bean that enables you to wire RMI services into your Spring

application is if they were local JavaBeans. Spring also provides a remote exporter that makes short work of converting your Spring-managed beans into RMI services.

To get started with Spring's RMI, let's see how to wire an RMI service into the RoadRantz application.

8.2.1 Wiring RMI services

As mentioned earlier, RoadRantz needs to be able to query a third-party service for traffic citations written against a vehicle. Fortunately, such a service is provided by Ticket-to-Drive, Inc. (a fictitious service-oriented company fabricated solely for purposes of this example). Conveniently, it turns out that Ticket-to-Drive's traffic citation service exposes its functionality as an RMI service.

One way to access the citation service is to write a factory method that retrieves a reference to the service in the traditional RMI way:

```
private String citationUrl = "rmi:/citation/CitationService";
public CitationService lookupCitationService()
    throws RemoteException, NotBoundException,
    MalformedURLException {
    CitationService citationService = (CitationService)
        Naming.lookup(citationUrl);
    return citationService;
}
```

The citationUrl property will need to be set to the address for the RMI service. Then, any time the RoadRantz application needs a reference to the citation service, it would need to call the lookupCitationService() method. While this would certainly work, it presents two problems:

- Conventional RMI lookups could result in any one of three exceptions (`RemoteException`, `NotBoundException`, and `MalformedURLException`) that must be caught or rethrown.
- Any code that needs the citation service is responsible for retrieving a reference to the service itself by calling `lookupCitationService()`.

The exceptions thrown in the course of an RMI lookup are the kinds that typically signal a fatal and unrecoverable condition in the application. `MalformedURLException`, for instance, indicates that the address given for the service is not valid. To recover from this exception, the application will at minimum need to be reconfigured and may have to be recompiled. No `try/catch` block will be able to recover gracefully, so why should your code be forced to catch and handle it?

But perhaps even more sinister is the fact that `lookupCitationService()` is a direct violation of dependency injection. This is bad because it means that the client of `lookupCitationService()` is also aware of where the citation service is located and of the fact that it is an RMI service. Ideally, you should be able to inject a `CitationService` object into any bean that needs one instead of having the bean look up the service itself. Using DI, any client of `CitationService` can be ignorant of where the `CitationService` comes from.

Spring's `RmiProxyFactoryBean` is a factory bean that creates a proxy to an RMI service. Using `RmiProxyFactoryBean` to reference an RMI `CitationService` is as simple as declaring the following `<bean>` in the Spring configuration file:

```
<bean id="citationService"
      class="org.springframework.remoting.rmi.RmiProxyFactoryBean">
    <property name="serviceUrl"
              value="rmi://${citationhost}/CitationService" />
    <property name="serviceInterface"
              value="com.tickettodore.CitationService" />
</bean>
```

The URL of the RMI service is set through the `serviceUrl` property. Here, the service is named `CitationService` and is hosted on a machine whose name is configured using a property placeholder (see section 3.5.3 in chapter 3). The `serviceInterface` property specifies the interface that the service implements and through which the client invokes methods on the service. The interaction between the client and the RMI proxy is illustrated in figure 8.4.

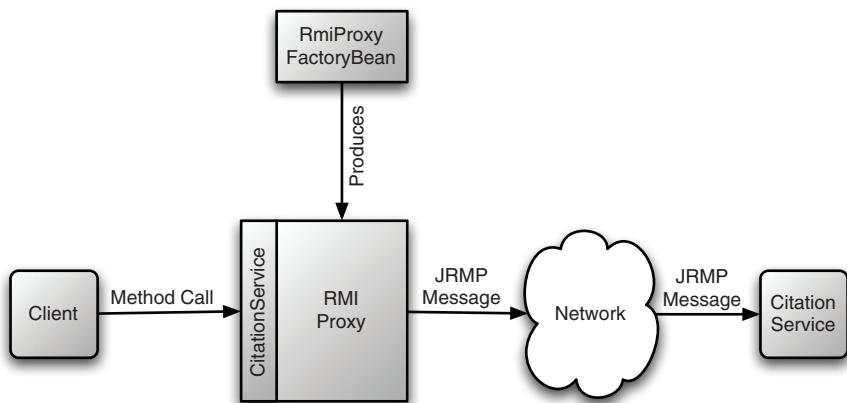


Figure 8.4 `RmiProxyFactoryBean` produces a proxy object that talks to remote RMI services on behalf of the client. The client talks to the proxy through the service's interface as if the remote service were just a local POJO.

With the service defined as a Spring-managed bean, you are able to wire it as a dependency into another bean just as you would any other non-remote bean. For example, suppose that `RantServiceImpl` needs to use the citation service to retrieve a list of citations for a vehicle. You'd use this code to wire the RMI service into `RantServiceImpl`:

```
<bean id="rantService"
      class="com.roaddrantz.service.RantServiceImpl">
  ...
    <property name="citationService">
      <ref bean="citationService"/>
    </property>
  ...
</bean>
```

What's great about accessing an RMI service in this way is that `RantServiceImpl` doesn't even know that it's dealing with an RMI service. It simply receives a `CitationService` object via injection, without any concern for where it comes from. Furthermore, the proxy catches any `RemoteExceptions` that may be thrown by the service, rethrowing them as runtime exceptions so that you may safely ignore them. This makes it possible to swap out the remote service bean with another implementation of the service—perhaps a different remote service or maybe a mock implementation used when unit testing `RantServiceImpl`.

`RmiProxyFactoryBean` certainly simplifies the use of RMI services in a Spring application. But that's only half of an RMI conversation. Let's see how Spring supports the service side of RMI.

8.2.2 Exporting RMI services

Now let's turn the tables and pretend that instead of working on the RoadRantz application, you are tasked with implementing the citation service for Ticket-to-Drive, Inc. as an RMI service.

If you've ever created an RMI service without Spring, you know that it involves the following steps:

- 1 Write the service implementation class with methods that throw `java.rmi.RemoteException`.
- 2 Create the service interface to extend `java.rmi.Remote`.
- 3 Run the RMI compiler (`rmic`) to produce client stub and server skeleton classes.
- 4 Start an RMI registry to host the services.
- 5 Register the service in the RMI registry.

Wow! That's a lot of work just to publish a simple RMI service. What's perhaps worse than all the steps required, you may have noticed that `RemoteExceptions` and `MalformedURLExceptions` are thrown around quite a bit. These exceptions usually indicate a fatal error that can't be recovered from in a catch block, but you're still expected to write boilerplate code that catches and handles those exceptions—even if there's not much you can do to fix them. Clearly a lot of code and manual work is involved to publish an RMI service without Spring.

Configuring an RMI service in Spring

Fortunately, Spring provides an easier way to publish RMI services. Instead of writing RMI-specific classes with methods that throw `RemoteException`, you simply write a POJO that performs the functionality of your service. Spring handles the rest.

To create the citation lookup service as an RMI service, we'll start by writing the service interface:

```
package com.tickettodrive;

public interface CitationService {
    Citation[] getCitationsForVehicle(
        String state, String plateNumber);
}
```

Because the service interface doesn't extend `java.rmi.Remote` and none of its methods throw `java.rmi.RemoteException`, this trims the interface down a bit. But more importantly, a client accessing the service through this interface will not have to catch exceptions that they probably won't be able to deal with.

Next you'll need to define the service implementation class. Listing 8.1 shows how this service may be implemented.

Listing 8.1 A POJO citation service

```
package com.tickettodrive;

public class CitationServiceImpl implements CitationService {
    public CitationServiceImpl() {}

    public Citation[] getCitationsForVehicle(
        String state, String plateNumber) {
        Citation[] citations;
        ...
        return citations;
    }
}
```

The code listing includes two annotations:

- A callout pointing to the line `public CitationServiceImpl() {}` with the text "Throws no RemoteException".
- A callout pointing to the line `return citations;` with the text "Looks up citations".

This time `CitationServiceImpl` is a POJO. We have no need to implement `java.rmi.Remote` and no more `java.rmi.RemoteExceptions` are being thrown around. In fact, this class has no idea that it will be used remotely. Consequently, we'll be able to reuse this exact same `CitationServiceImpl` class in the other remoting examples throughout this chapter.

The next thing you'll need to do is to configure `CitationServiceImpl` as a `<bean>` in the Spring configuration file:

```
<bean id="citationService"
      class="com.tickettodore.CitationServiceImpl">
    ...
</bean>
```

Notice that there's nothing about this version of `CitationServiceImpl` that is intrinsically RMI. It's just a simple POJO suitable for declaration in a Spring configuration file. It's no different than any other POJO that we might declare in Spring. In fact, it's entirely possible to use this implementation in a non-remote manner by wiring it directly into a client.

But we're interested in using this service remotely. So, the last thing to do is to export `CitationServiceImpl` as an RMI service. But instead of generating a server skeleton and client stub using `rmic` and manually adding it to the RMI registry (as you would in conventional RMI), we'll use Spring's `RmiServiceExporter`.

`RmiServiceExporter` exports any Spring-managed bean as an RMI service. As depicted in figure 8.5, `RmiServiceExporter` works by wrapping the bean in an adapter class. The adapter class is then bound to the RMI registry and proxies requests to the service class—in this case `CitationServiceImpl`.

The simplest way to use `RmiServiceExporter` to expose the `citationService` bean as an RMI service is to configure it in Spring with the following XML:

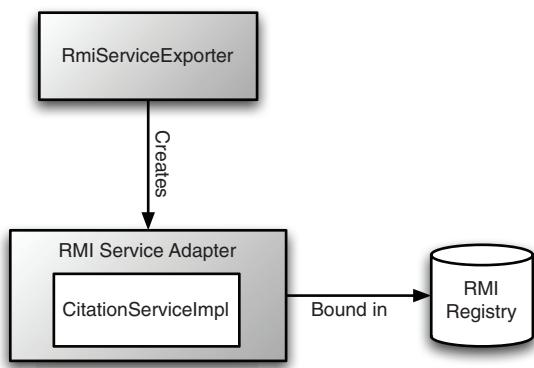


Figure 8.5
`RmiServiceExporter` turns POJOs into RMI services by wrapping them in a service adapter and binding the service adapter to the RMI registry.

```
<bean class="org.springframework.remoting.rmi.RmiServiceExporter">
    <property name="service" ref="citationService"/>
    <property name="serviceName" value="CitationService"/>
    <property name="serviceInterface"
        value="com.tickettodrive.CitationService"/>
</bean>
```

Here the citationService bean is wired into the service property to indicate that RmiServiceExporter is going to export the bean as an RMI service. Just as with RmiProxyFactoryBean described in section 8.2.1, the serviceName property names the RMI service and the serviceInterface property specifies the interface implemented by the service.

By default, RmiServiceExporter attempts to bind to an RMI registry on port 1099 of the local machine. If no RMI registry is found at that port, RmiServiceExporter will start one. If you'd like to bind to an RMI registry at a different port or host, you can specify so with the registryPort and registryHost properties. For example, the following RmiServiceExporter will attempt to bind to an RMI registry on port 1199 of rmi.tickettodrive.com:

```
<bean class="org.springframework.remoting.rmi.RmiServiceExporter">
    <property name="service" ref="citationService"/>
    <property name="serviceName" value="CitationService"/>
    <property name="serviceInterface"
        value="com.tickettodrive.CitationService"/>
    <property name="registryHost"
        value="rmi.tickettodrive.com" />
    <property name="registryPort" value="1199" />
</bean>
```

RMI is an excellent way to communicate with remote services, but it has its limitations. First, RMI has difficulty working across firewalls. That's because RMI uses arbitrary ports for communication—something firewalls typically will not allow. In an intranet environment, this usually isn't a concern, but if you are working on the “evil Internet,” you'll probably run into trouble with RMI. Even though RMI has support for tunneling through HTTP (which is usually allowed by firewalls), setting up the tunneling can be tricky.

Another thing to consider is that RMI is Java based. That means that both the client and the service must be written in Java. And since RMI uses Java serialization, the types of the objects being sent across the network must have the exact same version on both sides of the call. These may or may not be issues for your application, but it is something to bear in mind when choosing RMI for remoting.

Caucho Technology (the same people behind the Resin application server) has developed a remoting solution that addresses the limitations of RMI. Actually, they have come up with two solutions: Hessian and Burlap. Let's see how to use Hessian and Burlap to work with remote services in Spring.

8.3 Remoting with Hessian and Burlap

Hessian and Burlap are two solutions provided by Caucho Technology (<http://www.caucho.com>) that enable lightweight remote services over HTTP. They each aim to simplify web services by keeping both their API and their communication protocols as simple as possible.

You may be wondering why Caucho has two solutions to the same problem. Indeed, Hessian and Burlap are two sides of the same coin, but each serves slightly different purposes. Hessian, like RMI, uses binary messages to communicate between client and service. However, unlike other binary remoting technologies (such as RMI), the binary message is portable to languages other than Java, including PHP, Python, C++, and C#.

Burlap, on the other hand, is an XML-based remoting technology, which automatically makes it portable to any language that can parse XML. And because it's XML, it is more easily human-readable than Hessian's binary format. Unlike other XML-based remoting technologies (such as SOAP or XML-RPC), however, Burlap's message structure is as simple as possible and does not require an external definition language (e.g., WSDL or IDL).

Both Hessian and Burlap are also lightweight with regard to their size. Each is fully contained in an 84KB JAR file, with no external dependencies other than the Java runtime libraries. This makes them both perfect for use in environments that are constrained on memory, such as Java applets or handheld devices.

You may be wondering how to make a choice between Hessian and Burlap. For the most part, they are identical. The only difference is that Hessian messages are binary and Burlap messages are XML. Because Hessian messages are binary, they are more bandwidth friendly. If human-readability is important to you (for debugging purposes) or if your application will be communicating with a language for which there is no Hessian implementation, Burlap's XML messages may be preferable.

To demonstrate Hessian and Burlap services in Spring, let's revisit the citation service problem that was solved with RMI in section 8.2. This time, however, we'll look at how to solve the problem using Hessian and Burlap as the remoting models.

8.3.1 Accessing Hessian/Burlap services

As you'll recall from section 8.2.1, RantServiceImpl has no idea that the citation service is an RMI service. RantServiceImpl dealt only with the CitationService interface, while all of the RMI details were completely contained in the configuration of the beans in Spring's configuration file. The good news is that because of the client's ignorance of the service's implementation, switching from an RMI client to a Hessian client is extremely easy, requiring no changes to the client code.

The bad news is that if you really like writing code, this section may be a bit of a letdown. That's because the only difference between wiring the client side of an RMI-based service and wiring the client side of a Hessian-based service is that you'll use Spring's HessianProxyFactoryBean instead of RmiProxyFactoryBean. A Hessian-based citation service is declared in the client code like this:

```
<bean id="citationService" class="org.springframework.  
    ↪ remoting.caucho.HessianProxyFactoryBean">  
    <property name="serviceUrl">  
        <value>http://${serverName}/${contextPath}/  
            ↪ citation.service</value>  
    </property>  
    <property name="serviceInterface">  
        <value>com.tickettodore.CitationService</value>  
    </property>  
</bean>
```

Just as with an RMI-based service, the serviceInterface property specifies the interface that the service implements. And, as with RmiProxyFactoryBean, serviceUrl indicates the URL of the service. Since Hessian is HTTP based, it has been set to an HTTP URL here (you'll see how this URL is derived in the next section). Figure 8.6 shows the interaction between a client and the proxy produced by HessianProxyFactoryBean.

As it turns out, wiring a Burlap service is equally uninteresting. The only difference is that you'll use BurlapProxyFactoryBean instead of HessianProxyFactoryBean:

```
<bean id="citationService" class="org.springframework.  
    ↪ remoting.caucho.BurlapProxyFactoryBean">  
    <property name="serviceUrl">  
        <value>http://${serverName}/${contextPath}/  
            ↪ citation.service</value>  
    </property>  
    <property name="serviceInterface">  
        <value>com.tickettodore.CitationService</value>  
    </property>  
</bean>
```

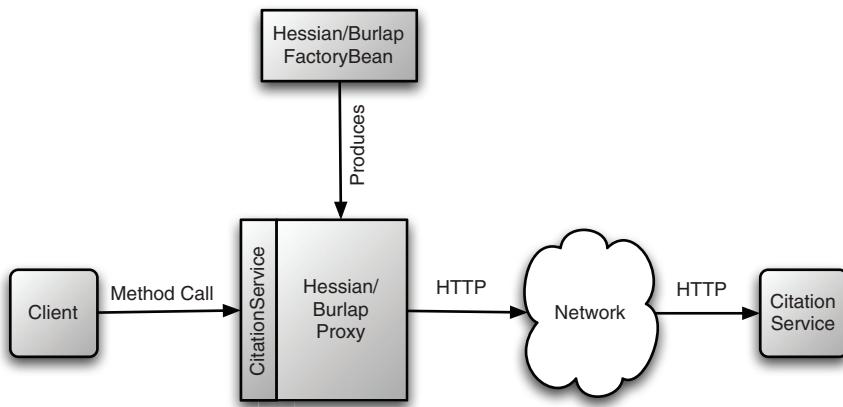


Figure 8.6 `HessianProxyFactoryBean` and `BurlapProxyFactoryBean` produce proxy objects that talk to a remote service over HTTP (Hessian in binary, Burlap in XML).

Although I've made light of how uninteresting the configuration differences are among RMI, Hessian, and Burlap services, this tedium is actually a benefit. It demonstrates that you'll be able to switch effortlessly between the various remoting technologies supported by Spring without having to learn a completely new model. Once you've configured a reference to an RMI service, it's short work to reconfigure it as a Hessian or Burlap service.

Now let's look at the other side of the conversation and expose the functionality of a Spring-managed bean as either a Hessian or Burlap service.

8.3.2 Exposing bean functionality with Hessian/Burlap

Again, let's suppose that you are tasked with implementing the citation service and exposing its functionality as a remote service. This time, however, you're going to expose it as a Hessian-based service.

Even without Spring, writing a Hessian service is fairly trivial. You simply write your service class to extend `com.caucho.hessian.server.HessianServlet` and make sure that your service methods are `public` (all public methods are considered service methods in Hessian).

Because Hessian services are already quite easy to implement, Spring doesn't do much to simplify the Hessian model any further. However, when used with Spring, a Hessian service can take full advantage of the Spring Framework in ways that a pure Hessian service cannot. This includes using Spring AOP to advise a Hessian service with systemwide services such as declarative transactions.

Exporting a Hessian service

Exporting a Hessian service in Spring is remarkably similar to implementing an RMI service in Spring. In fact, if you followed the RMI example in section 8.2.2, you've already done most of the work required to expose the citation service bean as a Hessian service.

To expose the citation service bean as an RMI service, you configured an `RmiServiceExporter` bean in the Spring configuration file. In a similar way, to expose the citation service as a Hessian service, you'll need to configure another exporter bean. This time, however, it will be a `HessianServiceExporter`.

`HessianServiceExporter` performs the exact same function for a Hessian service as `RmiServiceExporter` does for an RMI service. That is, it exposes the public methods of a POJO as methods of a Hessian service. However, as shown in figure 8.7, how it pulls off this feat is different from how `RmiServiceExporter` exports POJOs as RMI services.

`HessianServiceExporter` is a Spring MVC controller (more on that in a moment) that receives Hessian requests and translates them into method calls on the exported POJO.

The following declaration of `HessianServiceExporter` in Spring exports the `citationService` bean as a Hessian service:

```
<bean name="hessianCitationService" class="org.springframework.
    ↗ remoting caucho.HessianServiceExporter">
    <property name="service">
        <ref bean="citationService"/>
    </property>
    <property name="serviceInterface">
        <value>com.tickettodore.CitationService</value>
    </property>
</bean>
```

Just as with `RmiServiceExporter`, the `service` property is wired with a reference to the bean that implements the service. Here the `service` property is wired with

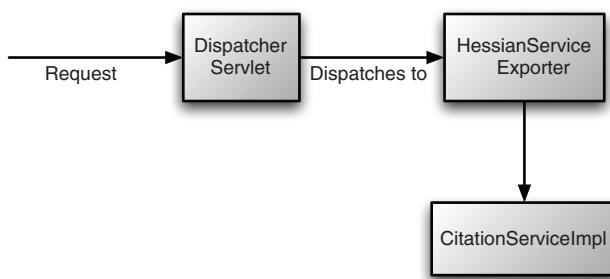


Figure 8.7
`HessianServiceExporter` is a Spring MVC controller that exports a POJO as a Hessian service by receiving Hessian requests and translating them into calls to the POJO.

a reference to the `citationService` bean. The `serviceInterface` property is set to indicate that `CitationService` is the interface that the service implements.

Unlike with `RmiServiceExporter`, however, you do not need to set a `serviceName` property. With RMI, the `serviceName` property is used to register a service in the RMI registry. Hessian doesn't have a registry and therefore there's no need to name a Hessian service.

Configuring the Hessian controller

Another major difference between `RmiServiceExporter` and `HessianServiceExporter` is that because Hessian is HTTP based, `HessianServiceExporter` is implemented as a Spring MVC Controller. This means that in order to use exported Hessian services, you'll need to perform two additional configuration steps:

- Configure a URL handler in your Spring configuration file to dispatch Hessian service URLs to the appropriate Hessian service bean.
- Configure a Spring `DispatcherServlet` in `web.xml` and deploy your application as a web application.

You'll learn the details of how Spring URL handlers and `DispatcherServlet` work in chapter 13. But for now we're only going to show you just enough to expose the Hessian citation service.

In section 8.3.1, you configured the `serviceUrl` property on the client side to point to `http://${serverName}/${contextPath}/citation.service`. The `${serverName}` and `${contextPath}` are placeholders that are configured via `PropertyPlaceholderConfigurer`. The last part of the URL, `/citation.service`, is the part we're interested in here. This is the URL pattern that you'll map the Hessian citation service to.

A URL handler maps a URL pattern to a specific Controller that will handle requests. In the case of the Hessian citation service, you want to map `/citation.service` to the `hessianCitationService` bean as follows using `SimpleUrlHandlerMapping`:

```
<bean id="urlMapping" class="org.springframework.web.  
    ↗ servlet.handler.SimpleUrlHandlerMapping">  
    <property name="mappings">  
        <props>  
            <prop key="/citation.service">hessianCitationService</prop>  
        </props>  
    </property>  
</bean>
```

You'll learn more about `SimpleUrlHandlerMapping` in chapter 13 when we see how to build web applications with Spring MVC. For now, suffice it to say that the `mappings` property takes a set of properties whose key is the URL pattern. Here it has been given a single property with a key of `/citation.service`, which is the URL pattern for the citation service. The value of the property is the name of a Spring Controller bean that will handle requests to the URL pattern—in this case, `hessianCitationService`.

Because `HessianServiceExporter` is implemented as a controller in Spring MVC, you must also configure Spring's `DispatcherServlet` in `web.xml`:

```
<servlet>
    <servlet-name>citation</servlet-name>
    <servlet-class>
        org.springframework.web.servlet.DispatcherServlet
    </servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
```

The name given to the servlet is significant because it is used by `DispatcherServlet` to locate the Spring configuration file. In this case, because the servlet is named `citation`, the configuration file must be named `citation-servlet.xml`.

One final step required to expose the Hessian service is to set up a servlet mapping:

```
<servlet-mapping>
    <servlet-name>citation</servlet-name>
    <url-pattern>*.service</url-pattern>
</servlet-mapping>
```

Configured this way, any request whose URL ends with `.service` will be given to `DispatcherServlet`, which will in turn hand off the request to the Controller that is mapped to the URL. Thus requests to `/citation.service` will ultimately be handled by the `hessianCitationService` bean (which is actually just a proxy to `CitationServiceImpl`).

If Hessian's binary messaging style is not to your liking, maybe you'd prefer to export your service as an XML-based Burlap service. Let's see how to do that with `BurlapServiceExporter`.

Exporting a Burlap service

`BurlapServiceExporter` is virtually identical to `HessianServiceExporter` in every way, except that it handles Burlap's XML-based messages instead of Hessian binary messages. The following bean definition shows how to expose the citation service as a Burlap service using `BurlapServiceExporter`:

```
<bean name="burlapCitationService" class="org.springframework.  
    ↗ remoting.caucho.BurlapServiceExporter">  
    <property name="service">  
        <ref bean="citationService"/>  
    </property>  
    <property name="serviceInterface">  
        <value>com.tickettodore.CitationService</value>  
    </property>  
</bean>
```

You'll notice that aside from the bean's name (which is purely arbitrary) and the use of BurlapServiceExporter, this `<bean>` declaration is identical to the hessianCitationService. Configuring a Burlap service is otherwise the same as configuring a Hessian service. This includes the need to set up a URL handler and the DispatcherServlet.

Because both Hessian and Burlap are based on HTTP, they do not suffer from the same firewall issues as RMI. And both are lightweight enough to be used in constrained environments where memory and space are a premium, such as applets and wireless devices.

But RMI has both Hessian and Burlap beat when it comes to serializing objects that are sent in RPC messages. Whereas Hessian and Burlap both use a proprietary serialization mechanism, RMI uses Java's own serialization mechanism. If your data model is complex, the Hessian/Burlap serialization model may not be sufficient.

There is, however, a best-of-both-worlds solution. Let's take a look at Spring's HTTP invoker, which offers RPC over HTTP (like Hessian/Burlap) while at the same time using Java serialization of objects (like RMI).

8.4 Using Spring's `HttpInvoker`

The Spring team recognized a void between RMI services and HTTP-based services like Hessian and Burlap. On one side, RMI uses Java's standard object serialization but is difficult to use across firewalls. On the other side, Hessian/Burlap work well across firewalls but use a proprietary object serialization mechanism.

Thus Spring's HTTP invoker was born. The HTTP invoker is a new remoting model created as part of the Spring Framework to perform remoting across HTTP (to make the firewalls happy) and using Java's serialization (to make programmers happy).

Working with HTTP invoker-based services is quite similar to working with Hessian/Burlap-based services. To get started with the HTTP invoker, let's take one more look at the citation service—this time implemented as an HTTP invoker service.

8.4.1 Accessing services via HTTP

To access an RMI service, you declared an `RmiProxyFactoryBean` that pointed to the service. To access a Hessian service, you declared a `HessianProxyFactoryBean`. And to access a Burlap service, you used `BurlapProxyFactoryBean`. Carrying this monotony over to the HTTP invoker, it should be of little surprise to you that to access an HTTP invoker service, you'll need to use `HttpInvokerProxyFactoryBean`.

As you can see from figure 8.8, the `HttpInvokerProxyFactoryBean` fills the same hole as the other remote service proxy factory beans we've seen in this chapter.

Had the citation service been exposed as an HTTP invoker-based service, you could configure a bean that proxies it using `HttpInvokerProxyFactoryBean` as follows:

```
<bean id="citationService" class="org.springframework.remoting.
    ↳ httpinvoker.HttpInvokerProxyFactoryBean">
    <property name="serviceUrl">
        <value>http://${serverName}/${contextPath}/
            ↳ citation.service</value>
    </property>
    <property name="serviceInterface">
        <value>com.tickettodore.CitationService</value>
    </property>
</bean>
```

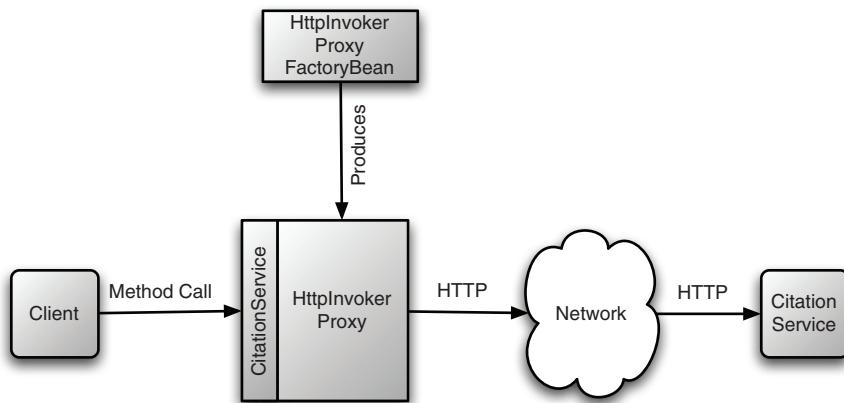


Figure 8.8 `HttpInvokerProxyFactoryBean` is a proxy factory bean that produces a proxy for remoting with a Spring-specific HTTP-based protocol.

Comparing this bean definition to those in sections 8.2.1 and 8.3.1, you'll find that little has changed. The `serviceInterface` property is still used to indicate the interface implemented by the citation service. And the `serviceUrl` property is still used to indicate the location of the remote citation service. Because HTTP invoker is HTTP-based like Hessian and Burlap, the `serviceUrl` can contain the same URL as with the Hessian and Burlap versions of the bean.

Moving on to the other side of an HTTP invoker conversation, let's now look at how to export a bean's functionality as an HTTP invoker-based service.

8.4.2 Exposing beans as HTTP Services

You've already seen how to expose the functionality of `CitationServiceImpl` as an RMI service, as a Hessian service, and as a Burlap service. Next let's rework the citation service as an HTTP invoker service using Spring's `HttpInvokerServiceExporter` to export the citation service.

At the risk of sounding like a broken record, I must tell you that exporting a bean's methods as remote method using `HttpInvokerServiceExporter` is very much like what you've already seen with the other remote service exporters. In fact, it's virtually identical. For example, the following bean definition shows how to export the `citationService` bean as a remote HTTP invoker-based service:

```
<bean id="httpCitationService" class="org.springframework.remoting.  
    ↪ httpinvoker.HttpInvokerServiceExporter">  
    <property name="service">  
        <ref bean="citationService"/>  
    </property>  
    <property name="serviceInterface">  
        <value>com.tickettodore.CitationService</value>  
    </property>  
</bean>
```

Feeling a strange sense of déjà vu? You may be having a hard time spotting the difference between this bean declaration and the ones in section 8.3.2. In case the bold text didn't help you spot it, the only difference is the class name: `HttpInvokerServiceExporter`. Otherwise, this exporter is not much different from the other remote service exporters.

As you can see in figure 8.9, `HttpInvokerServiceExporter` works very much like `HessianServiceExporter` and `BurlapServiceExporter`. It is a Spring MVC controller that receives requests from an HTTP invoker client through `DispatcherServlet` and translates those requests into method calls on the service implementation POJO.

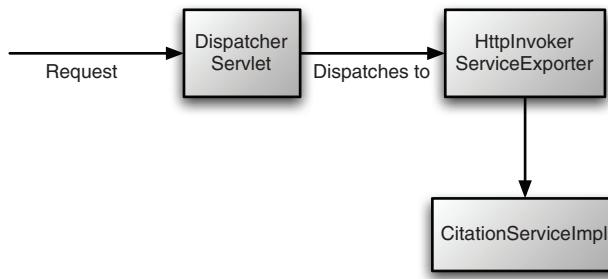


Figure 8.9
HttpInvokerServiceExporter works much like its Hessian and Burlap cousins, receiving requests from a Spring MVC **DispatcherServlet** and translating them into method calls on a POJO.

Because `HttpInvokerServiceExporter` is a Spring MVC controller, you'll need to set up a URL handler to map an HTTP URL to the service:

```

<bean id="urlMapping" class="org.springframework.web.
    ↗ servlet.handler.SimpleUrlHandlerMapping">
    <property name="mappings">
        <props>
            <prop key="/citation.service">httpCitationService</prop>
        </props>
    </property>
</bean>
  
```

And you'll also need to deploy the citation service in a web application with Spring's `DispatcherServlet` configured in `web.xml`:

```

<servlet>
    <servlet-name>citation</servlet-name>
    <servlet-class>
        org.springframework.web.servlet.DispatcherServlet
    </servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
    <servlet-name>citation</servlet-name>
    <url-pattern>*.service</url-pattern>
</servlet-mapping>
  
```

Configured this way, the citation service will be available at `/citation.service`, the same URL as it was when exposed as either a Hessian or Burlap service.

Spring's HTTP invoker presents a best-of-both-worlds remoting solution combining the simplicity of HTTP communication with Java's built-in object serialization. This makes HTTP invoker services an appealing alternative to either RMI or Hessian/Burlap.

HttpInvoker has one significant limitation that you should keep in mind: it is a remoting solution offered by the Spring Framework only. This means that both the client and the service must be Spring-enabled applications. This also implies, at least for now, that both the client and the service must be Java based. And because Java serialization is being used, both sides must have the same version of the classes (much like RMI).

RMI, Hessian, Burlap, and HTTP invoker are great remoting options. But when it comes to ubiquitous remoting, none hold a candle to web services. Next up, we'll look at how Spring supports remoting through SOAP-based web services.

8.5 **Spring and web services**

One of the most hyped TLAs (three-letter acronyms) in recent years is SOA (which stands for “service-oriented architecture”). SOA means many things to different people. But at the center of SOA is the idea that applications can and should be designed to lean on a common set of core services instead of reimplementing the same functionality for each application.

For example, a financial institution may have several applications, many of which need access to borrower account information. Rather than build account access logic into each application (much of which would be duplicated), the applications could all rely on a common service to retrieve the account information.

In this section, we'll revisit the Ticket-to-Drive citation service one more time, this time as a POJO-based web service in Spring. Let's start by turning the citation service bean into a web service.

8.5.1 **Exporting beans as web services using XFire**

Earlier in this chapter, we created remote services using Spring's service exporters. These service exporters magically turn Spring-configured POJOs into remote services. We saw how to create RMI services using `RmiServiceExporter`, Hessian services using `HessianServiceExporter`, Burlap services using `BurlapServiceExporter`, and HTTP invoker services using `HttpInvokerServiceExporter`.

At this point, I'd like to show you how to create web services using Spring's `SoapServiceExporter`. Unfortunately, I won't be able to do that because Spring doesn't come with a `SoapServiceExporter`. In fact, Spring doesn't provide any service exporter for exposing bean functionality as SOAP-based web services.

No worries, however. Even though Spring doesn't come with a service exporter for creating web services, that doesn't mean that such a service exporter doesn't exist. We'll find the service exporter we need by looking at XFire.

XFire is an open source web services platform available through Codehaus (<http://xfire.codehaus.org>). Among its many features, XFire comes with `XFireExporter`, a Spring service exporter that turns POJOs into SOAP services.

Figure 8.10 should look vaguely familiar to you by now. It's roughly the same as figures 8.7 and 8.9, except that the service exporter in question is `XFireExporter`. In fact, `XFireExporter` works very much like `HessianServiceExporter`, `BurlapServiceExporter`, and `HttpInvokerServiceExporter`, except that it deals with incoming SOAP messages to export a POJO as a web service.

For this example, I'm using XFire version 1.2.6. Adding XFire to a project is easy if you're using Maven 2 to build the application. Simply add the following `<dependency>` to the project's `pom.xml` file:

```
<dependency>
  <groupId>org.codehaus.xfire</groupId>
  <artifactId>xfire-spring</artifactId>
  <version>1.2.6</version>
  <scope>compile</scope>
</dependency>
```

This single `<dependency>` entry will load several JAR files into the application's classpath. But you won't need to worry about what those JARs are—Maven 2's transitive dependency resolution will figure out what's needed for you. If you're using Ant or another mechanism for building your application, refer to XFire's documentation for the JARs you'll need.

Once XFire is in your build's classpath, creating a web service is as simple as following these three steps:

- 1 Configure an `XFireExporter` bean to export a Spring bean as a web service.
- 2 Configure a Spring `DispatcherServlet` to handle incoming HTTP requests.
- 3 Configure a handler mapping so as to map `DispatcherServlet`-handled requests to `XFireExporter`-exported services.

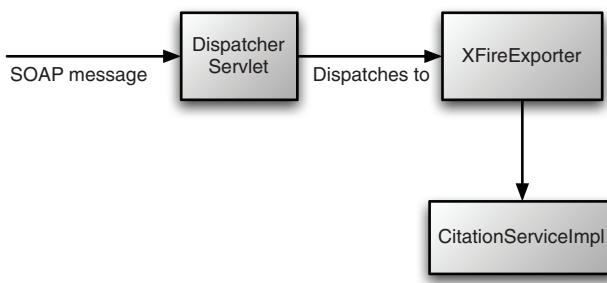


Figure 8.10
`XFireExporter` is a service exporter that exports a POJO as a web service by translating incoming SOAP messages into method invocations.

You'll recognize these steps as similar to those for configuring Hessian, Burlap, and HTTP invoker services. You may want to take a moment to review those sections from earlier in this chapter before diving into the configuration steps that follow.

We've already configured `CitationServiceImpl` as a bean in Spring, so we're ready to turn it into a web service using `XFireExporter`. Our first step: configuring the `XFireExporter` bean.

Configuring XFireExporter

The following `<bean>` definition shows the simplest way to configure `XFireExporter` to export our `citationService` bean as a web service:

```
<bean id="citationService.xfire"
      class="org.codehaus.xfire.spring.remoting.XFireExporter">
    <property name="serviceFactory"
              ref="xfire.serviceFactory" />
    <property name="xfire" ref="xfire" />

    <property name="serviceBean"
              ref="citationService" />
    <property name="serviceClass"
              value="com.tickettodrive.CitationService" />
</bean>
```

The first two properties, `serviceFactory` and `xfire`, are wired with references to `xfire.serviceFactory` and `xfire`, respectively. `XFireExporter` needs these core `XFire` beans to do its work. Fortunately, we don't have to configure those beans ourselves—they're already declared in a Spring context contained in the `XFire` JAR file. All we need to do is to import the `XFire`-defined Spring context. The following Spring `<import>` will do the trick:

```
<import resource="classpath:org/codehaus/xfire/spring/xfire.xml"/>
```

The two remaining properties are the ones that will vary for each web service we create with `XFireExporter`. The `serviceBean` property is wired with a reference to the Spring-configured bean that we want to export as a web service. Here it's wired with a reference to the `citationService` bean.

Meanwhile, the `serviceClass` property is configured with the fully qualified class name of the interface that will define the web service. The methods defined in the service interface are the ones that will be exposed as SOAP operations.

It's worth noting that the service interface's package name also determines the target namespace of the service. Since the package of the `CitationService` interface is `com.tickettodrive`, the exported service's target namespace will be `http://tickettodrive.com` (notice that the package name is reversed in the

namespace). If you want to override this behavior and specify a different namespace, you can configure XFireExporter's namespace property:

```
<bean id="citationService.xfire"
      class="org.codehaus.xfire.spring.remoting.XFireExporter">
  ...
  <property name="namespace"
            value="http://www.springinaction.com/citation"/>
</bean>
```

Using this configuration, the new target namespace for the service will be `http://www.springinaction.com/citation`.

Configuring DispatcherServlet

XFireExporter is implemented as a Spring MVC Controller. Therefore, HTTP requests destined for our XFireExporter-exported service must go through Spring's DispatcherServlet. We'll talk more about Spring MVC, controllers, and DispatcherServlet when we get to chapter 13. But for now, just know that you'll need to put the following `<servlet>` and `<servlet-mapping>` entries in the application's web.xml file:

```
<servlet>
  <servlet-name>citation</servlet-name>
  <servlet-class>
    org.springframework.web.servlet.DispatcherServlet
  </servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
  <servlet-name>citation</servlet-name>
  <url-pattern>/*</url-pattern>
</servlet-mapping>
```

This `<servlet-mapping>` will send all requests whose URL starts with `http://{host}/{app}/` to DispatcherServlet. From there, DispatcherServlet will send the request to a Controller that will handle the request. So how do we get DispatcherServlet to dispatch the request to XFireExporter?

Mapping requests to XFireExporter

In Spring MVC, handler mappings are used to map requests to their destination Controller. One of the simplest handler mappings available is SimpleUrlHandlerMapping. SimpleUrlHandlerMapping maps a URL pattern to a Controller bean. The following declaration of SimpleUrlHandlerMapping will map a URL pattern to XFireExporter:

```
<bean id="handlerMapping"
      class="org.springframework.web.servlet.handler.
```

```
    ↳ SimpleUrlHandlerMapping">
<property name="mappings">
  <props>
    <prop key="/citationService">citationService.xfire</prop>
  </props>
</property>
</bean>
```

The `mappings` property takes a `<props>` of one or more `<prop>` elements. The key of each `<prop>` element is a URL pattern and the value is a reference to a Controller—in this case the XFireExporter. The URL pattern is relative to the servlet request—it picks up where the `<servlet-mapping>` leaves off. Therefore, given the `<servlet-mapping>` and `mappings` we've configured, we can expect our exported citation lookup web service to respond to requests at `http://{host}/{app}/citationService`.

Deploy the application and ta-da! We now have a citation lookup web service. And it all started from a simple POJO. What's more, `CitationServiceImpl` is still a POJO and has absolutely no idea that it is being used as a web service.

As you can see, it's quite easy to create a web service in Spring using XFire. But as simple as XFireExporter is, XFire has something even simpler up its sleeve. Let's have a look at how to configure XFire to create web services from annotated beans.

8.5.2 Declaring web services with JSR-181 annotations

As wonderful as XFireExporter is, it suffers from one small problem: you must declare an XFireExporter bean for each bean you wish to export as a web service. This may not be a big deal if you're only exporting one or two services. But if you're planning to export several services, you're going to be writing a lot of XML.

JSR-181, also known as Web Services Metadata for the Java Platform, defines a set of eight annotations that can be applied to POJOs to declare web services. Those annotations are listed in table 8.2.

Table 8.2 Web service annotations defined by JSR-181.

Annotation	Purpose
<code>javax.jws.WebService</code>	Marks a Java class as implementing a web service or a Java interface as defining a web service interface.
<code>javax.jws.WebMethod</code>	Customizes how a method is exposed as a web service operation.

Table 8.2 Web service annotations defined by JSR-181. (continued)

Annotation	Purpose
javax.jws.Oneway	Indicates that a method has only an input message and no output. One-way methods typically return control to the calling application before executing.
javax.jws.WebParam	Customizes the mapping of an individual parameter of a web method.
javax.jws.WebResult	Customizes the mapping of a web method's return value.
javax.jws.HandlerChain	Associates a web service with an externally defined handler chain.
javax.jws.soap.SOAPBinding	Specifies the mapping of a web service onto the SOAP message protocol.
javax.jws.soap.SOAPMessageHandlers	Creates a JAX-RPC handler chain. <i>Deprecated in JSR-181 version 2.0.</i>

To use JSR-181 annotations with XFire, we'll need to add XFire's JAX-WS support to our application's classpath. In Maven 2, that means adding the following <dependency> to the pom.xml file:

```
<dependency>
<groupId>org.codehaus.xfire</groupId>
<artifactId>xfire-jaxws</artifactId>
<version>1.2.6</version>
<scope>compile</scope>
</dependency>
```

Not all web services need all the annotations from table 8.2. But we'll use a few of them to declare a citation lookup web service. To start, have a look at how we use the @WebService annotation in CitationServiceImpl:

```
package com.tickettodrive;
import java.util.Date;
import javax.jws.WebService;

@WebService(serviceName="citationService",
    endpointInterface="com.tickettodrive.CitationService")
public class CitationServiceImpl
    implements CitationService {
    ...
}
```

Just by placing the `@WebService` annotation on the `CitationServiceImpl` class, we're declaring that we want `CitationServiceImpl` to be exported as a web service. We've used the `serviceName` attribute to specify that the service should be named `citationService`. As for the `endpointInterface` property, it specifies that the `CitationService` interface will be used to define the service interface.

In the `CitationService` interface, we simply tag the interface with the `@WebService` annotation as follows:

```
package com.tickettodore;
import javax.jws.WebService;

@WebService
public interface CitationService {
    @WebMethod(operationName="getCitations")
    Citation[] getCitationsForVehicle(
        String state, String plateNumber);
}
```

By default, all public methods of the service interface will be exposed as web methods with operation names matching the Java method names. But here we've used the `@WebMethod` attribute to customize how the `getCitationsForVehicle()` method will be exposed. Instead of `getCitationsForVehicle`, the operation will be named `getCitations`.

All of these annotations are well and good, but annotations have no meaning by themselves. Therefore, we'll need to configure XFire to interpret these annotations and to expose web services from them.

Mapping requests to JSR-181 annotated beans

The first thing we'll need to do is to configure a handler mapping so that `DispatcherServlet` will dispatch requests to beans that are annotated with JSR-181 annotations.

Earlier, we configured a `SimpleUrlHandlerMapping` to map requests to XFire-Exporter beans. `SimpleUrlHandlerMapping` knows how to map URL patterns to Spring MVC controllers. But when it comes to mapping URL patterns with JSR-181-annotated beans, `SimpleUrlHandlerMapping` isn't up to the challenge.

Instead of `SimpleUrlHandlerMapping`, we'll need to use XFire's `Jsr181HandlerMapping`. As its name suggests, `Jsr181HandlerMapping` is savvy in how to map requests to beans that are annotated with JSR-181 annotations. We'll configure `Jsr181HandlerMapping` in Spring like this:

```
<bean id="annotationHandlerMapping"
    class="org.codehaus.xfire.spring.remoting.Jsr181HandlerMapping">
    <property name="xfire" ref="xfire" />
    <property name="webAnnotations">
```

```

<bean class="org.codehaus.xfire.annotations.jsr181.
    ↗ Jsr181WebAnnotations"/>
</property>
</bean>

```

The first property, `xfire`, is just a reference to the XFire handler, which is loaded by the `<import>` we used earlier with `XFireExporter`.

The `webAnnotations` property is of the most interest here. This property is wired with a reference to a `Jsr181WebAnnotations` bean (configured here as an inner bean). `Jsr181WebAnnotations` tells `Jsr181HandlerMapping` to use JSR-181 annotations in Spring beans to map to URL patterns.

And that's it. When we build and deploy the application, we'll be able to see the WSDL for our service at `http://{host}/{app}/services/citationService?wsdl`.

Now we've seen a few ways to create a web service using Spring. But creating the web service is only half the story. A web service has no purpose if it is never used. Let's justify the existence of our service by looking at how to wire the client side of a web service in Spring.

It should be noted, however, that using annotations turns these objects that are otherwise POJOs into classes that are aware (at some level) that they're to be exposed as web services. If that's a concern for you then you may wish to stick to configuring XFire strictly through XML. Otherwise, the JSR-181 annotations are a handy way to turn POJOs into web services.

8.5.3 Consuming web services

When it comes to developing a web service client, there's no shortage of options. Several web service platforms and frameworks offer an API for accessing and invoking methods on remote services.

The problem with many of these web service platforms, however, is that your client ends up being keenly aware of the fact that it is communicating with a web service. For example, consider this code snippet that uses `webMethods Glue` to access a web service:

```

CitationService cs =
    (CitationService) Registry.bind(
        "http://ws.springinaction.com/Citation/
    ↗ citationService.wsdl");

```

Or, how about this similar set of code that uses XFire's client API:

```

Service serviceModel = new ObjectServiceFactory().create(
    CitationService.class);
CitationService cs =

```

```
(CitationService) new XFireProxyFactory().create(
    serviceModel,
    "http://ws.springinaction.com/Citation/citationService");
```

There's nothing terribly wrong with either of these examples. Both are straightforward and easy ways to access a web service. But in both cases, the client code knows too much about the SOAP stack that it's using. In the first example, the code is coupled with Glue through Glue's Registry class. In the second example, it's coupled with XFire's ObjectServiceFactory and XFireProxyFactory. Moreover, in both examples the client knows that it's dealing with a web service.

By now, you should be able to guess what a better approach would be. Instead of looking up the web service through an API, the service should be injected into the client. The client would only know about the service through its interface. In fact, the client wouldn't even need to know that the service is a web service. It could be an RMI service, a local POJO, or even a mock implementation used in a unit test.

We have already seen several ways to proxy remote RMI, Hessian, Burlap, and HTTP invoker services so that they can be wired transparently into clients in Spring. In this section, we're going to see how the remote proxy concept is extended to web services.

Spring developers have two options for wiring remote references into Spring:

- `JaxRpcPortProxyFactoryBean` is Spring's own proxy factory bean for web service access.
- `XFireClientFactoryBean` is a web service proxy factory bean provided by XFire.

Let's start our exploration of web service proxying by looking at Spring's own `JaxRpcPortProxyFactoryBean`. (We'll look at `XFireClientFactoryBean` in section 8.5.4.)

Wiring JaxRpcPortProxyFactoryBean

Spring's out-of-the-box support for web service proxying comes in the form of `JaxRpcPortProxyFactoryBean`. Using `JaxRpcPortProxyFactoryBean`, we can wire the citation lookup web service in Spring as if it were any other bean. `JaxRpcPortProxyFactoryBean` is a Spring `FactoryBean` that produces a proxy that knows how to talk to a SOAP web service. The proxy itself is created to implement the service's interface (see figure 8.11). Consequently, `JaxRpcPortProxyFactoryBean` makes it possible to wire and use a remote web service as if it were just any other local POJO.

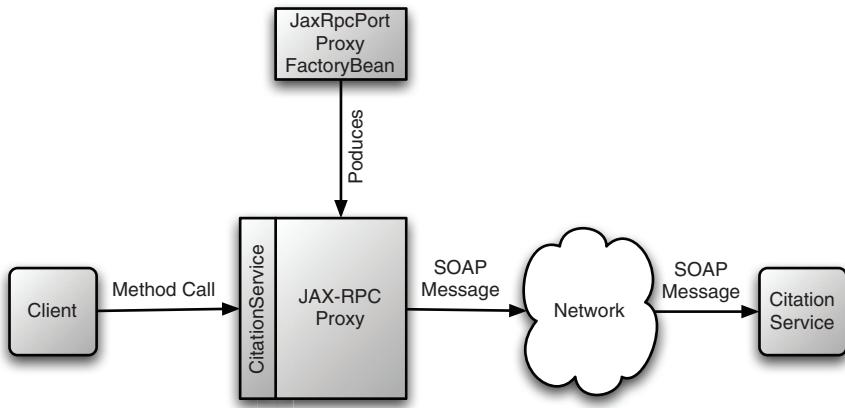


Figure 8.11 `JaxRpcPortProxyFactoryBean` produces proxies that talk to remote web services. These proxies can then be wired into other beans as if they were local POJOs.

We'll use the following XML in the Spring configuration file to declare the client side of the citation lookup web service using `JaxRpcPortProxyFactoryBean`:

```

<bean id="citationService"
      class="org.springframework.remoting.jaxrpc.
      ↳ JaxRpcPortProxyFactoryBean">
    <property name="wsdlDocumentUrl"
              value="http://localhost:8081/Citation/services/
      ↳ citationService?wsdl" />
    <property name="serviceInterface"
              value="com.tickettodrive.CitationService"/>
    <property name="portName"
              value="citationServiceHttpPort" />
    <property name="serviceName"
              value="citationService" />
    <property name="namespaceUri"
              value="http://tickettodrive.com" />
  </bean>

```

The `wsdlDocumentUrl` property identifies the location of the remote web service's definition file. `JaxRpcPortProxyFactoryBean` will use the WSDL available at that URL to construct a proxy to the service. The proxy that's produced by `JaxRpcPortProxyFactoryBean` will implement the `CitationService` interface, as specified by the `serviceInterface` property.

The values for the remaining three properties can usually be determined by looking at the service's WSDL. For illustration's sake, here's a snippet of the WSDL

file produced by XFire that shows the pertinent information for our citation lookup service:

```
<wsdl:definitions
    targetNamespace="http://tickettodore.com">
    ...
    <wsdl:service name="citationService">
        <wsdl:port name="citationServiceHttpPort"
            binding="tns:citationServiceHttpBinding">
            ...
        </wsdl:port>
    </wsdl:service>
</wsdl:definitions>
```

Although not likely, it is possible for multiple services and/or ports to be defined in the service's WSDL definition. For that reason, `JaxRpcPortProxyFactoryBean` requires that we specify the port and service names in the `portName` and `serviceName` properties. A quick glance at the `name` attribute of the `<wsdl:port>` and `<wsdl:service>` elements in the WSDL will help you figure out what these properties should be set to.

Finally, the `namespaceUri` property specifies the namespace of the service. Among other things, the namespace will help `JaxRpcPortProxyFactoryBean` locate the service definition in the WSDL. As with the port and service names, you can find the correct value for this property by looking in the WSDL. It's usually available in the `targetNamespace` attribute of the `<wsdl:definitions>` element.

If our service only communicated using primitive types (e.g., `int`, `float`, `String`, etc.), this would be all we'd need to do to create a proxy to the remote service. Most web services, however, aren't so simple—and our citation lookup service is no exception. The citation lookup service returns an array of `Citation` objects from the call to the `getCitationsForVehicle()` method. `Citation` is, itself, a complex type. What may not be obvious, however, is that an array of `Citation` objects is also a complex type (known as `ArrayOfCitation` in the service's WSDL).

Chances are good that the JAX-RPC implementation under the covers of `JaxRpcPortProxyFactoryBean` won't know how to deal with these types. Therefore, we'll need to tell it how to deal with them.

The way we teach `JaxRpcPortProxyFactoryBean` about these types is by registering JAX-RPC post processors through the `servicePostProcessors` property:

```
<bean id="citationService"
    class="org.springframework.remoting.jaxrpc.
        JaxRpcPortProxyFactoryBean">
    <property name="wsdlDocumentUrl"
        value="http://localhost:8081/Citation/services/">
```

```
    ↗ citationService?wsdl" />
<property name="serviceInterface"
          value="com.tickettodrive.CitationService"/>
<property name="portName"
          value="citationServiceHttpPort" />
<property name="serviceName"
          value="citationService" />
<property name="namespaceUri"
          value="http://tickettodrive.com" />
<property name="servicePostProcessors">
  <list>
    <ref bean="beanMappingPostProcessor" />
    <ref bean="arrayMappingPostProcessor" />
  </list>
</property>
</bean>
```

The `servicePostProcessors` property takes a list of one or more implementations of Spring's `JaxRpcServicePostProcessor` interface:

```
package org.springframework.remoting.jaxrpc;
import javax.xml.rpc.Service;

public interface JaxRpcServicePostProcessor {
    void postProcessJaxRpcService(Service service);
}
```

In the case of the citation service client, we've wired two `ServicePostProcessors` into `JaxRpcPortProxyFactoryBean`. The first references a `ServicePostProcessor` that will serialize and deserialize `Citation` objects. The second handles serialization and deserialization of arrays of `Citation` objects. Let's first look at how `beanMappingPostProcessor` is declared.

Mapping complex types

For dealing with complex types, Apache Axis (<http://ws.apache.org/axis/>) provides `BeanSerializer` and `BeanDeserializer`. These classes use reflection to break complex bean types into their simpler parts. Without Spring, you'd have to register a `BeanSerializer/BeanDeserializer` pair for each complex type used by the web service. But Spring makes things easier with `AxisBeanMappingServicePostProcessor`. The following declaration of `AxisBeanMappingServicePostProcessor` tells `JaxRpcPortProxyFactoryBean` how to serialize and deserialize `Citations`:

```
<bean id="beanMappingPostProcessor"
      class="org.springframework.remoting.jaxrpc.
        ↗ support.AxisBeanMappingServicePostProcessor">
  <property name="beanClasses">
```

```
<list>
    <value>com.tickettodrive.Citation</value>
</list>
</property>

<property name="typeNamespaceUri"
    value="http://tickettodrive.com" />
</bean>
```

AxisBeanMappingServicePostProcessor is an implementation of Spring's Jax-RpcServicePostProcessor interface that automatically registers a BeanSerializer/BeanDeserializer pair for all classes listed in its beanClasses property. Here we've asked AxisBeanMappingServicePostProcessor to handle the Citation complex type. The typeNamespaceUri property is used to specify the namespace of the types, as is usually defined in the service's WSDL.

With the beanMappingPostProcessor bean registered as a service postprocessor with JaxRpcPortProxyFactoryBean, the Citation type is covered. But we still have to contend with an array of Citations. For that, we'll need to do a bit more work.

Mapping arrays

Just as AxisBeanMappingServicePostProcessor is able to handle complex Java types, we can count on AxisArrayMappingServicePostProcessor to easily deal with arrays.

There's only one problem: Spring doesn't provide an AxisArrayMappingServicePostProcessor. Therefore, we'll need to write one for ourselves. That's perfect, because we needed a good excuse to write our own implementation of JaxRpcServicePostProcessor. You'll find an array-handling service postprocessor in listing 8.2.

Listing 8.2 A JAX-RPC post processor that serializes and deserializes arrays of Citation objects

```
package com.tickettodrive.ws;
import javax.xml.namespace.QName;
import javax.xml.rpc.Service;
import javax.xml.rpc.encoding.TypeMapping;
import javax.xml.rpc.encoding.TypeMappingRegistry;
import org.apache.axis.encoding.ser.ArrayDeserializerFactory;
import org.apache.axis.encoding.ser.ArraySerializerFactory;
import org.springframework.remoting.jaxrpc.
    ↗ JaxRpcServicePostProcessor;

public class AxisArrayOfCitationMappingServicePostProcessor
    implements JaxRpcServicePostProcessor {
```

```
public void postProcessJaxRpcService(Service service) {  
    TypeMappingRegistry registry = service.getTypeMappingRegistry();  
    TypeMapping mapping = registry.getDefaultTypeMapping();  
  
    QName xmlType = new QName(  
        "http://tickettodrive.com",  
        "ArrayOfCitation");           | Creates QName for  
                                         | ArrayOfCitation  
  
    mapping.register(Citation[].class, xmlType,  
        new ArraySerializerFactory(Citation[].class,  
            xmlType),  
        new ArrayDeserializerFactory());  | Registers  
                                         | ArraySerializerFactory  
                                         | Registers  
                                         | ArrayDeserializerFactory  
    };  
}
```

Aside from being a mouthful to say, `AxisArrayOfCitationMappingServiceProcessor` is exactly what we need for `JaxRpcPortProxyFactoryBean` to be able to receive an array of `Citation`s from the citation lookup service. As an implementation of `JaxRpcServicePostProcessor`, it only has to implement the `postProcessJaxRpcService()` method.

`postProcessJaxRpcService()` starts by getting the type mapping registry from the `Service` object that is passed in. From that it gets the default type mapping, where it will ultimately register our custom `ArrayOfCitation` mapping.

All types in SOAP are identified by their qualified name, or `QName`. So, the next thing that `postProcessJaxRpcService()` does is create a `QName` object to identify the `ArrayOfCitation` type. The `QName` is made up of two parts: a namespace and a name. In this case, the namespace is `http://tickettodrive.com` and the name is `ArrayOfCitation`. Both of these can be found by looking in the service's WSDL.

Finally, with a `TypeMapping` and a `QName` in hand, `postProcessJaxRpcService()` registers a serializer factory and a deserializer factory for `ArrayOfCitation`. Since we're dealing with an array, Axis's `ArraySerializerFactory` and `ArrayDeserializerFactory` are perfect for the job. We use each of these to map arrays of `Citation` to and from the `ArrayOfCitation` SOAP type.

Note that `AxisArrayOfCitationMappingServicePostProcessor` is very specific to our example, as it only knows how to handle `Citation` to/from `ArrayOfCitation` mappings. With a little effort, however, you could create a more general-purpose array mapping service postprocessor to handle any type of array. But I'll leave that as an exercise for you to figure out on your own.

Now that we have an array mapping service postprocessor, we simply need to declare it as a bean in Spring:

```
<bean id="arrayMappingPostProcessor"
      class="com.tickettodoreve.
          ➔ AxisArrayOfCitationMappingServicePostProcessor" />
```

Notice that I named this bean `arrayMappingPostProcessor`. That's to match the name of the bean reference wired into `JaxRpcPortProxyFactoryBean`'s `servicePostProcessors` property.

Whew! It sure seems like a lot of work to create a client for a web service with `JaxRpcPortProxyFactoryBean`. And this is a relatively simple service with only a couple of complex types. Imagine how much work we'd need to do if this service were more interesting with more operations and more types. I'm not sure I'm up to the challenge.

Although `JaxRpcPortProxyFactoryBean` is the out-of-the-box solution for wiring proxies to web services in Spring, it makes no assumptions about the services it proxies and requires a lot of configuration. If you were hoping for something simpler then read on... I'm now going to show you a similar, but much simpler, way to proxy web services using XFire.

8.5.4 Proxying web services with an XFire client

In section 8.5.2, we saw several ways that XFire can be used to export POJOs as web services. But did you know that it can also be used on the client side?

Among its Spring remoting features, XFire comes with `XFireClientFactoryBean`, a factory bean similar in spirit to Spring's own `JaxRpcPortProxyFactoryBean`, but without all of the hassle. In figure 8.12, `XFireClientFactoryBean` fits into the same position in the web service client picture.

The following XML shows how simple it is to configure a proxy to the citation lookup service:

```
<bean id="citationService"
      class="org.codehaus.xfire.spring.remoting.
          ➔ XFireClientFactoryBean">
    <property name="wsdlDocumentUrl"
              value="http://localhost:8080/Citation/services/
                  ➔ citationService?wsdl" />
    <property name="serviceInterface"
              value="com.tickettodoreve.CitationService" />
</bean>
```

Just as with `JaxRpcPortProxyFactoryBean`, we need to tell `XFireClientFactoryBean` where the service's WSDL is located through the `wsdlDocumentUrl`. And, just like `JaxRpcPortProxyFactoryBean`, `XFireClientFactoryBean` will produce a

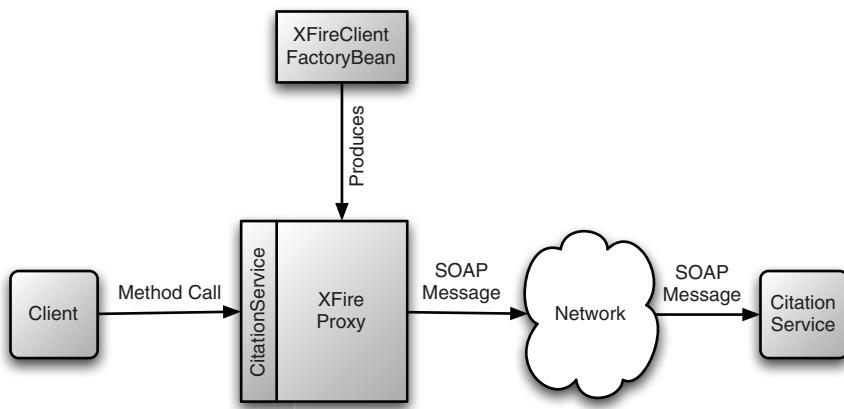


Figure 8.12 `XFireClientFactoryBean` is XFire's answer for SOAP clients in Spring. It produces a proxy that knows how to talk to a remote web service.

proxy to that service that implements the interface specified in the service-Interface property.

But very much unlike `JaxRpcPortProxyFactoryBean`, there's no more configuration required with `XFireClientFactoryBean`. In most cases, `XFireClientFactoryBean` doesn't need to be told the service name, port name, or namespace to use. It's able to figure all of that out on its own by examining the service's WSDL. Moreover, XFire is a lot smarter with regard to complex types, so there's no need to register custom type mappings. XFire is able to handle all that drudgery for us.

8.6 Summary

Working with remote services is typically a tedious chore. But Spring provides remoting support that makes working with remote services as simple as working with any regular JavaBean.

On the client side, Spring provides proxy factory beans that enable you to configure remote services in your Spring application. Regardless of whether you are using RMI, Hessian, Burlap, Spring's own HTTP invoker, or SOAP for remoting, you can wire remote services into your application as if they were POJOs. Spring even catches any `RemoteExceptions` that are thrown and rethrows runtime `RemoteAccessExceptions` in their place, freeing your code from having to deal with an exception that it probably can't recover from.

Even though Spring hides many of the details of remote services, making them appear as though they are local JavaBeans, you should bear in mind the consequences of remote services. Remote services, by their nature, are typically less efficient than local services. You should consider this when writing code that accesses remote services, limiting remote calls to avoid performance bottlenecks.

In this chapter, you saw how Spring can be used to expose and consume services based on some basic remoting technologies. Although these remoting options are useful in distributing applications, this was just a taste of what is involved in working within a service-oriented architecture (SOA).

We also looked at how to use XFire to export beans as SOAP web services. While this is certainly an easy way to develop web services, it may not be the best choice from an architectural standpoint. Coming up in the next chapter, we look at a different approach to building web services in Spring. Rather than simply exposing POJOs as remote services, we'll approach web services with a message-oriented mind-set. In doing so, we'll get our hands dirty with Spring-WS, an exciting new web services framework that enables us to create loosely coupled service endpoints to process XML messages.



Building contract-first web services in Spring

This chapter covers

- Defining XML service contracts
- Creating document-centric web services
- Marshaling and unmarshaling XML messages
- Building template-based web service clients

Imagine that it's the weekend and you've got a trip planned. Before you hit the road, you stop by your bank to deposit your paycheck and to pick up some spending cash.

This is not an unusual scenario, but what makes it interesting is that you bank at an unusual bank. When you walk in the door, there are no tellers to help you. Instead, you have full access to handle the transaction yourself. You have direct access to the ledger and to the vault, allowing you to handle all of the minute details of the transaction on your own. So, you perform the following tasks:

- 1 You place your signed paycheck in a box designated for deposited checks.
- 2 You edit your account's ledger, incrementing the balance by the amount on the check.
- 3 You take \$200 from the vault and place it in your pocket.
- 4 You edit your account's ledger, decrementing the balance by \$200.
- 5 As a thank-you for all of the hard work you did, you pay yourself a service fee by pocketing another \$50 bill on the way out the door.

Whoa! Steps 1–4 seem to be on the up and up. But isn't step 5 a bit odd?

The problem (if that's what you want to call it) with this bank is that they trust their customers with too much direct access to the internal workings of the bank. Instead of providing an appropriate interface to the inner workings of the bank (commonly known as a "teller"), they give you full access to the inner workings to do as you please. Consequently, you are able to perform an unrecorded and questionable withdrawal.

As nice as this is for the customer, most banks don't work that way (if your bank really does allow you this kind of access, please email me—I'd really like to start banking there!). Most banks have tellers, ATM machines, and websites to allow you to manipulate your account. These interfaces to the bank are customer-facing abstractions to the vault and the ledger. While they may provide service with a smile, they only allow you to perform activities that fit within the bank's business model.

Likewise, most applications do not allow direct access to the internal objects that make up the application. Take web applications, for instance. In a Spring MVC-based web application (which we'll look at when we get to chapter 13), users interact with the application through controllers. Behind the scenes, there may be dozens or even hundreds of objects that perform the core tasks of the application. But the user is only allowed to interact with the controllers, which, in turn, interact with the back-end objects.

In the previous chapter, we saw that XFire is a quick and easy way to develop web services using remote exporters. But when we export an application bean as a web service, we're exposing the application's internal API, which carries with it some consequences: as I alluded to in the banking scenario, you must be careful not to accidentally expose too much of your application's internal API. Doing so may give your web service clients more access to the inner workings of your application than they need.

In this chapter, you'll learn an alternative way of building web services using the Spring-WS framework. We'll separate the service's external contract from the application's internal API, and we'll focus on sending messages between clients and the service, not on invoking remote methods.

I won't deceive you: building web services with Spring-WS is not as simple as exporting them with XFire. However, I think you'll find that it isn't that much more difficult and that the architectural advantages that Spring-WS affords make it well worth considering.

9.1 Introducing Spring-WS

Spring Web Services (or Spring-WS, for short) is an exciting new subproject of Spring that is focused on building *contract-first web services*. What are contract-first web services? It might be easier to answer that question by first talking about their antithesis: contact-last web services.

In chapter 8 (see section 8.5.1), we used XFire to export bean functionality as a remote web service. We started by writing some Java code (the service implementation). Then we configured it as a `<bean>` in Spring. Finally, we used XFire's `XFireExporter` to turn it into a web service. We never had to explicitly define the service's contract (WSDL and XSD). Instead, XFire automatically generated the contract *after* the service was deployed. In short, the contract was the last thing defined, thus the designation of "contract-last."

Contract-last web services are a popular approach to web service development for one basic reason: they're easy. Most developers don't have the intestinal fortitude required to understand WSDL, SOAP, and XML Schema (XSD). In the contract-last approach, there's no need to manipulate complex WSDL and XSD files. You simply write a service class in Java and ask the web service framework to "SOAP-ify" it. If a web services platform such as XFire is willing to cope with the web services acronyms then why should we worry ourselves with it?

But there's one small gotcha: when a web service is developed contract last, its contract ends up being a reflection of the application's internal API. Odds are that

your application's internal API is far more volatile than you (or your service's clients) would like the external API to be. Changes to the internal API will mean changes to your service's contract, which will ultimately require changes in the clients that are consuming your service. A clever refactoring today may result in a new service contract tomorrow.

This leads to the classic web services versioning problem. It's much easier to change a web service's contract than to change the clients that consume that service. If your web service has 1,000 clients and you change your service's contract then 1,000 clients will be broken until they are changed to adhere to the new contract. A common solution to this problem is to maintain multiple versions of a service until all clients have upgraded. This, however, would multiply maintenance and support costs, as you would have to support multiple versions of the same service.

A better solution is to avoid changing the service's contract. And when the contract must be changed, the changes shouldn't break compatibility with previous versions. But this can be difficult to do when the service's contract is automatically generated.

In short, the problem with contract-last web services is that the service's most important artifact, the contract, is treated as an afterthought. The focus of a contract-last web service is on *how* the service should be implemented and not on *what* it should do.

The solution to contract-last's problems is to flip it on its head—create the contract first and then decide how it should be implemented. When you do, you end up with contract-first web services. The contract is written with little regard for what the underlying application will look like. This is a pragmatic approach, because it emphasizes what is expected of the service and not how it will be implemented.

You're probably getting an uneasy feeling about now. It could be that unusually large burrito that you had for lunch... or it could be that you're terrified that we're going to have to create a WSDL file by hand.

Don't worry. It's not going to be as bad as you think. Along the way, I'll show you several tricks that make it easy to create the service contract. (If that doesn't make you feel better, I suggest you take an antacid and cut back on the spicy food at lunch.)

The basic recipe for developing a contract-first web service with Spring-WS appears in table 9.1.

Table 9.1 The steps for developing a contract-first web service.

Step	Action	What we'll do
1	Define the service contract.	This involves designing sample XML messages that will be processed by our web service. We'll use these sample messages to create XML Schema that will later be used to create WSDL.
2	Write a service endpoint.	We'll create classes that will receive and process the messages sent to the web service.
3	Configure the endpoint and Spring-WS infrastructure.	We'll wire up our service endpoint along with a handful of Spring-WS beans that will tie everything together.

To demonstrate Spring-based web services, we're going to build a poker hand evaluation service. Figure 9.1 illustrates the requirements for this web service: given five cards, identify the poker hand in question.

Since we're creating a contract-first web service, it's only logical that the first thing we should do is define the service contract. Let's get started.

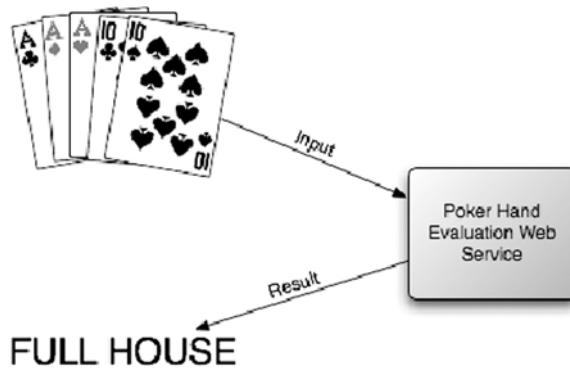


Figure 9.1
We'll build a poker hand evaluation web service. Given a poker hand made up of five cards, the web service will determine what kind of poker hand was dealt.

9.2 Defining the contract (first!)

The single most important activity in developing a contract-first web service is defining the contract itself. When defining the contract, we'll define the messages that are sent to and received from the service, with no regard for how the service is implemented or how the messages will be handled.

Even though the topic of this chapter is Spring-WS, you'll find that this section is remarkably Spring free. That's because the contract of a web service should be

defined independent of the implementation of the service. The focus is on *what* needs to be said, not *how* it needs to be done. We'll tie this all into Spring-WS starting in section 9.3. But for now, the techniques described in this section are applicable to contract-first services in general, regardless of the underlying framework.

A contract-first view of web services places emphasis on the messages that are sent to and received from services. Therefore, the first step in defining a service's contract is determining what the messages will look like. We'll start by creating sample XML messages for our web services that we'll use to define the service contract.

9.2.1 Creating sample XML messages

In simple terms, our poker hand evaluation service takes a poker hand made up of five cards as input and produces a poker hand designation (e.g., Full House, Flush, etc.) as output. Writing a sample input message for the service as XML might look a little like this:

```
<EvaluateHandRequest  
    xmlns="http://www.springinaction.com/poker/schemas">  
    <card>  
        <suit>HEARTS</suit>  
        <face>TEN</face>  
    </card>  
    <card>  
        <suit>SPADES</suit>  
        <face>KING</face>  
    </card>  
    <card>  
        <suit>HEARTS</suit>  
        <face>KING</face>  
    </card>  
    <card>  
        <suit>DIAMONDS</suit>  
        <face>TEN</face>  
    </card>  
    <card>  
        <suit>CLUBS</suit>  
        <face>TEN</face>  
    </card>  
</EvaluateHandRequest>
```

That's fairly straightforward, isn't it? There are five `<card>` elements, each with a `<suit>` and a `<face>`. That pretty much describes a poker hand. All of the `<card>` elements are contained within an `<EvaluateHandRequest>` element, which is the message we'll be sending to the service.

As simple as the input message was, the output message is even simpler:

```
<EvaluateHandResponse  
    xmlns="http://www.springinaction.com/poker/schemas">  
    <handName>Full House</handName>  
</EvaluateHandResponse>
```

The `<EvaluateHandResponse>` message simply contains a single `<handName>` element that holds the designation of the poker hand.

These sample messages will serve as the basis for our service's contract. And, although this may bring about some disbelief on your part, you should know that by defining these sample messages, we've already finished the hardest part of designing the service contract. No kidding.

Forging the data contract

Now we're ready to create the service contract. Before we do that, however, let's conceptually break the contact into two parts:

- The *data contract* will define the messages going in and out of the service. In our example, this will include the schema definition of the `<EvaluateHandRequest>` and `<EvaluateHandResponse>` messages.
- The *operational contract* will define the operations that our service will perform. Note that a SOAP operation does not necessarily correspond to a method in the service's API.

Both of these contract parts are typically (but not necessarily) defined in a single WSDL file. The WSDL file usually contains an embedded XML Schema that defines the data contract. The rest of the WSDL file defines the operational contract, including one or more `<wsdl:operation>` elements within the `<wsdl:binding>` element.

Don't worry yourself too much with the details of that last paragraph. I promised that creating the contract would be easy, so there's no need for you to know the details of what goes into a WSDL file. The key point is that there are two distinct parts of the contract.

The data contract is defined using XML Schema (XSD). XSD allows us to precisely define what should go into a message. Not only can we define what elements are in the message, but we can also specify the types of those messages and place constraints on what data goes into the message.

Although it's not terribly difficult to write an XSD file by hand, it's more work than I care to do. So, I'm going to cheat a little by using an XSD inference tool. An XSD inference tool examines one or more XML files and, based on their contents, produces an XML schema that the XML files can be validated against.

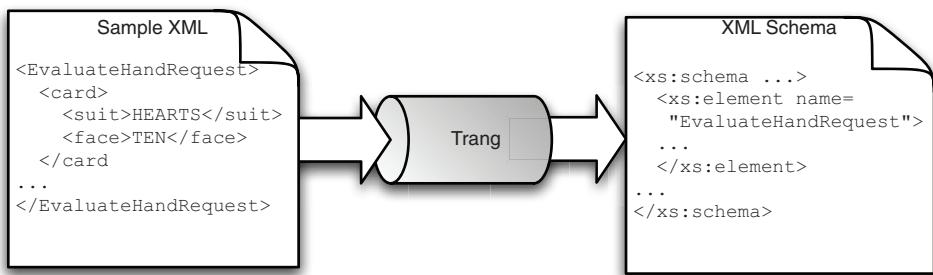


Figure 9.2 Trang is an XSD inference tool that makes simple work of producing XML Schema from sample XML files.

Several XSD inference tools are available, but one that I like is called Trang. Trang is a command-line tool (available from www.thaiopensource.com/relaxng/trang.html) that takes XML as input and produces an XSD file as output (see figure 9.2). Trang is Java based and thus can be used anywhere there's a JVM. As the URL implies, Trang is useful for generating RELAX NG schemas (an alternative schema style), but is also useful for creating XML Schema files. For Spring-WS, we'll be using Trang to generate XML Schema.

Once you've downloaded and unzipped Trang, you'll find trang.jar in the distribution. This is an executable JAR file, so running Trang is simple from the command line:

```
% java -jar trang.jar EvaluateHandRequest.xml
      ↗ EvaluateHandResponse.xml PokerTypes.xsd
```

When running Trang, I've specified three command-line arguments. The first two are the sample message XML files that we created earlier. Because we've specified both message files, Trang is able to produce an XSD file that can validate the messages in both files. The last argument is the name of the file we want Trang to write the XSD to.

When run with these arguments, Trang will generate the data contract for our service in `PokerTypes.xsd` (listing 9.1).

Listing 9.1 `PokerTypes.xsd`, which defines the data contract for the web service

```

<?xml version="1.0" encoding="UTF-8"?>
<xss:schema xmlns:xss="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified"
  targetNamespace=
    "http://www.springinaction.com/poker/schemas"
  
```

```
xmlns:schemas=
    "http://www.springinaction.com/poker/schemas">
<xs:element name="EvaluateHandRequest">
<xs:complexType>
<xs:sequence>
<xs:element maxOccurs="unbounded"
    ref="schemas:card"/>
</xs:sequence>
</xs:complexType>
</xs:element>
<xs:element name="card">
<xs:complexType>
<xs:sequence>
<xs:element ref="schemas:suit"/>
<xs:element ref="schemas:face"/>
</xs:sequence>
</xs:complexType>
</xs:element>
<xs:element name="suit" type="xs:NCName"/>
<xs:element name="face" type="xs:NCName"/>
<xs:element name="EvaluateHandResponse">
<xs:complexType>
<xs:sequence>
<xs:element ref="schemas:handName"/>
</xs:sequence>
</xs:complexType>
</xs:element>
<xs:element name="handName" type="xs:string"/>
</xs:schema>
```

Trang saved us a lot of trouble by inferring the XSD for our messages. We're not completely off the hook, though. XSD isn't perfect. As it infers the XSD, Trang makes some assumptions about what kind of data will be in your XML. Most of the time, those assumptions are okay. But often, we'll need to fine-tune the generated XSD to be more precise.

For example, Trang assumed that the values of the `<suit>` and `<face>` elements should be defined as noncolonized¹ names (`xs:NCName`). What we actually want is for those elements to be simple strings (`xs:string`). So, let's tweak the definitions of `<suit>` and `<face>` to be strings:

```
<xs:element name="suit" type="xs:string"/>
<xs:element name="face" type="xs:string"/>
```

¹ A noncolonized name is a name that isn't qualified with a namespace related prefix. Therefore, it does not have a colon (:)—it isn't “colonized.”

We also know that there are only four possible values for the `<suit>` element, so we could constrain the message a bit further:

```
<xs:element name="suit" type="schemas:Suit" />
<xs:simpleType name="Suit">
    <xsd:restriction base="xs:string">
        <xsd:enumeration value="SPADES" />
        <xsd:enumeration value="CLUBS" />
        <xsd:enumeration value="HEARTS" />
        <xsd:enumeration value="DIAMONDS" />
    </xsd:restriction>
</xs:simpleType>
```

Likewise, there are only 13 legal values for the `<face>` element, so let's define those limits in XSD:

```
<xs:element name="face" type="schemas:Face" />
<xs:simpleType name="Face">
    <xsd:restriction base="xs:string">
        <xsd:enumeration value="ACE" />
        <xsd:enumeration value="TWO" />
        <xsd:enumeration value="THREE" />
        <xsd:enumeration value="FOUR" />
        <xsd:enumeration value="FIVE" />
        <xsd:enumeration value="SIX" />
        <xsd:enumeration value="SEVEN" />
        <xsd:enumeration value="EIGHT" />
        <xsd:enumeration value="NINE" />
        <xsd:enumeration value="TEN" />
        <xsd:enumeration value="JACK" />
        <xsd:enumeration value="QUEEN" />
        <xsd:enumeration value="KING" />
    </xsd:restriction>
</xs:simpleType>
```

Also, notice that Trang incorrectly assumes that the `<EvaluateHandRequest>` may contain an unlimited number of `<card>` elements (`maxOccurs="unbounded"`). But a poker hand contains exactly five cards. Therefore, we'll need to adjust the definition of `<EvaluateHandRequest>` accordingly:

```
<xs:element name="EvaluateHandRequest">
    <xs:complexType>
        <xs:sequence>
            <xs:element minOccurs="5" maxOccurs="5"
                ref="schemas:card"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
```

As for `<EvaluateHandResponse>`, it's fine as is. We could constrain the possible values returned in the `<handName>` element, but it's not necessary. So, we'll leave it unchanged.

Now we have the data contract for the poker hand evaluation service, but what about the operational contract? Aren't we going to need some WSDL to completely define the web service?

Yes, we'll absolutely need WSDL—after all, WSDL is the standard for defining web services. We could write the WSDL by hand, but that's no fun. And, again, I promised you that this would be easy. But I'm going to have to ask you to wait awhile to see where the operational contract comes into play. I'll show you how the WSDL gets created in section 9.4.6 when we wire a WSDL definition bean in Spring.

But first, we need to create a service endpoint. The contract only defines the messages sent to and from the service, not how they're handled. Let's see how to create message endpoints in Spring-WS that will process messages from a web service client.

9.3 Handling messages with service endpoints

As you'll recall from the opening of this chapter, a well-designed application doesn't allow direct access to the internal objects that do the fine-grained tasks of a system. In Spring MVC, for example, a user interacts with the application through controllers, which in turn translate the user's requests into calls to internal objects.

It may be helpful to know that Spring MVC and Spring-WS are a lot alike. Whereas a user interacts with a Spring MVC application through one of several controllers, a web service client interacts with a Spring-WS application through one of several message endpoints.

Figure 9.3 illustrates how message endpoints interact with their client. A message endpoint is a class that receives an XML message from the client and, based on the content of the message, makes calls to internal application objects to perform the actual work. For the poker hand evaluation service, the message endpoint will process `<EvaluateHandRequest>` messages.

Once the endpoint has completed processing, it will return its response in yet another XML message. In the case of the poker hand evaluation service, the response XML is an `<EvaluateHandResponse>` document.

Spring-WS defines several abstract classes from which message endpoints can be created, as listed in table 9.2.

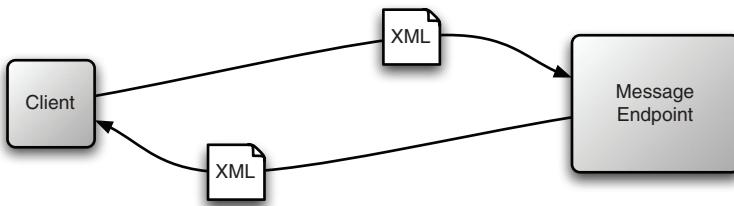


Figure 9.3 Message endpoints are the implementation of a web service in Spring-WS. Taking a message-centric approach, message endpoints process incoming XML messages and produce XML responses.

For the most part, all of the abstract endpoint classes in table 9.2 are similar. Which one you choose is mostly a matter of taste and which XML parsing technology you prefer (e.g., SAX versus DOM versus StAX, etc.). But `AbstractMarshallingPayloadEndpoint` is a bit different from the rest of the pack in that it supports automatic marshaling and unmarshaling of XML messages to and from Java objects.

Table 9.2 The message endpoint options available with Spring-WS.

Abstract endpoint class in package <code>org.springframework.ws.server.endpoint</code>	Description
<code>AbstractDom4jPayloadEndpoint</code>	Endpoint that handles message payloads as dom4j Elements
<code>AbstractDomPayloadEndpoint</code>	Endpoint that handles message payloads as DOM Elements
<code>AbstractJDomPayloadEndpoint</code>	Endpoint that handles message payloads as JDOM Elements
<code>AbstractMarshallingPayloadEndpoint</code>	Endpoint that unmarshals the request payload into an object and marshals the response object into XML
<code>AbstractSaxPayloadEndpoint</code>	Endpoint that handles message payloads through a SAX ContentHandler implementation
<code>AbstractStaxEventPayloadEndpoint</code>	Endpoint that handles message payloads using event-based StAX
<code>AbstractStaxStreamPayloadEndpoint</code>	Endpoint that handles message payloads using streaming StAX
<code>AbstractXomPayloadEndpoint</code>	Endpoint that handles message payloads as XOM Elements

We'll have a look at `AbstractMarshallingPayloadEndpoint` a little later in this chapter (in section 9.3.2). First, though, let's see how to build an endpoint that processes XML messages directly.

9.3.1 Building a JDOM-based message endpoint

Our poker hand evaluation web service takes an `<EvaluateHandRequest>` message as input and produces an `<EvaluateHandResponse>` as output. Therefore, we'll need to create a service endpoint that processes an `<EvaluateHandRequest>` element and produces an `<EvaluateHandResponse>` element.

Any of the abstract endpoint classes in table 9.2 will do, but we've chosen to base our endpoint on `AbstractJDomPayloadEndpoint`. This choice was mostly arbitrary, but I also like JDOM's XPath support, which is a simple way to extract information out of a JDOM Element. (For more information on JDOM, visit the JDOM homepage at <http://www.jdom.org>.)

`EvaluateHandJDomEndpoint` (listing 9.2) extends `AbstractJDomPayloadEndpoint` to provide the functionality required to process the `<EvaluateHandRequest>` message.

Listing 9.2 An endpoint that will process the `<EvaluateHandRequest>` message

```
package com.springinaction.poker.webservice;
import java.util.Iterator;
import java.util.List;
import org.jdom.Element;
import org.jdom.JDOMException;
import org.jdom.Namespace;
import org.jdom.xpath.XPath;
import org.springframework.beans.factory.InitializingBean;
import org.springframework.ws.server.endpoint.
    ↗ AbstractJDomPayloadEndpoint;
import com.springinaction.poker.Card;
import com.springinaction.poker.Face;
import com.springinaction.poker.PokerHand;
import com.springinaction.poker.PokerHandEvaluator;
import com.springinaction.poker.PokerHandType;
import com.springinaction.poker.Suit;

public class EvaluateHandJDomEndpoint
    extends AbstractJDomPayloadEndpoint
    implements InitializingBean {

    private Namespace namespace;
    private XPath cardsXPath;
    private XPath suitXPath;
    private XPath faceXPath;
```

```

protected Element invokeInternal(Element element)
    throws Exception {
    Card cards[] = extractCardsFromRequest(element);
    PokerHand pokerHand = new PokerHand();
    pokerHand.setCards(cards);

    PokerHandType handType =
        pokerHandEvaluator.evaluateHand(pokerHand); | Evaluates poker hand

    return createResponse(handType);
}

private Element createResponse(PokerHandType handType) {
    Element responseElement =
        new Element("EvaluateHandResponse", namespace);
    responseElement.addContent(
        new Element("handName", namespace).setText(
            handType.toString()));
    return responseElement;
} | Creates response

private Card[] extractCardsFromRequest(Element element)
    throws JDOMException {
    Card[] cards = new Card[5];

    List cardElements = cardsXPath.selectNodes(element);
    for(int i=0; i < cardElements.size(); i++) {
        Element cardElement = (Element) cardElements.get(i);
        Suit suit = Suit.valueOf(
            suitXPath.valueOf(cardElement));
        Face face = Face.valueOf(
            faceXPath.valueOf(cardElement));
        cards[i] = new Card();
        cards[i].setFace(face);
        cards[i].setSuit(suit);
    } | Extracts cards from message

    return cards;
}

public void afterPropertiesSet() throws Exception {
    namespace = Namespace.getNamespace("poker",
        "http://www.springinaction.com/poker/schemas");
    cardsXPath =
        XPath.newInstance("/poker:EvaluateHandRequest/poker:card");
    cardsXPath.addNamespace(namespace);
    faceXPath = XPath.newInstance("poker:face");
    faceXPath.addNamespace(namespace);
    suitXPath = XPath.newInstance("poker:suit");
    suitXPath.addNamespace(namespace);
} | Sets up XPath queries

// injected

```

```
private PokerHandEvaluator pokerHandEvaluator;
public void setPokerHandEvaluator(
    PokerHandEvaluator pokerHandEvaluator) {
    this.pokerHandEvaluator = pokerHandEvaluator;
}
}
```

The `invokeInternal()` method is the entry point into this endpoint. When called, it is passed a JDOM `Element` object that contains the incoming message—in this case, an `<EvaluateHandRequest>`. `invokeInternal()` hands off the `Element` to the `extractCardsFromRequest()` method, which uses JDOM XPath objects to pull card information out of the `<EvaluateHandRequest>` element.

After an array of `Card` objects is returned, `invokeInternal()` then does the right thing and passes those `Cards` to an injected `PokerHandEvaluator` to evaluate the poker hand. `PokerHandEvaluator` is defined by the following interface:

```
package com.springinaction.poker;

public interface PokerHandEvaluator {
    PokerHandType evaluateHand(PokerHand hand);
}
```

The actual implementation of `PokerHandEvaluator` isn't relevant to the discussion of building web services with Spring-WS, so I'll leave it out (but you can find it in the downloadable examples).

The fact that the endpoint calls `PokerHandEvaluator`'s `evaluateHand()` method is significant. A properly written Spring-WS endpoint shouldn't perform any business logic of its own. It should only mediate between the client and the internal API. The actual business logic is performed in the `PokerHandEvaluator` implementation. Later, in chapter 13, we'll see a similar pattern applied to Spring MVC controllers where a controller merely sits between a web user and a server-side object.

Once the `PokerHandEvaluator` has determined the type of poker hand it was given, `invokeInternal()` passes the `PokerHandType` object off to `createResponse()` to produce an `<EvaluateHandResponse>` element using JDOM. The resulting JDOM `Element` is returned and `EvaluateHandJDomEndpoint`'s job is done.

`EvaluateHandJDomEndpoint` is a fine example of how to implement a Spring-WS endpoint. But there are an awful lot of XML specifics in there. Although the messages handled by Spring-WS endpoints are XML, there's usually no reason why your endpoint needs to be written to know that. Let's see how a marshaling endpoint can help us eliminate all of that XML parsing code.

9.3.2 Marshaling message payloads

As we mentioned before, `AbstractMarshallingPayloadEndpoint` is a little different from all of the other Spring-WS abstract endpoint classes. Instead of being given an XML Element to pull apart for information, `AbstractMarshallingPayloadEndpoint` is given an object to process.

Actually, as illustrated in figure 9.4, a marshaling endpoint works with an unmarshaler that converts an incoming XML message into a POJO. Once the endpoint is finished, it simply returns a POJO and a marshaler converts it into an XML message to be returned to the client. This greatly simplifies the endpoint implementation, as it no longer has to include any XML-processing code.

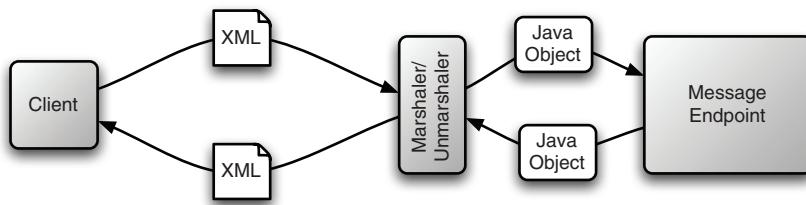


Figure 9.4 Marshaling endpoints leverage a marshaler/unmarshaler to handle XML messages so that the endpoint only has to deal with POJOs.

For example, consider listing 9.3, which shows `EvaluateHandMarshallingEndpoint`, a new implementation of our poker hand evaluation endpoint that extends `AbstractMarshallingPayloadEndpoint`.

Listing 9.3 The endpoint that will process the <EvaluateHandRequest> message

```

package com.springinaction.poker.webservice;
import org.springframework.ws.server.endpoint.*;
import com.springinaction.poker.PokerHand;
import com.springinaction.poker.PokerHandEvaluator;
import com.springinaction.poker.PokerHandType;

public class EvaluateHandMarshallingEndpoint
    extends AbstractMarshallingPayloadEndpoint {

    protected Object invokeInternal(Object object)
        throws Exception {
        EvaluateHandRequest request =
            (EvaluateHandRequest) object; | Gives EvaluateHandRequest to endpoint
        PokerHand pokerHand = new PokerHand();
        pokerHand.setCards(request.getHand());
    }
}
  
```

```
PokerHandType pokerHandType =  
    pokerHandEvaluator.evaluateHand(pokerHand); | Evaluates poker hand  
  
    return new EvaluateHandResponse(pokerHandType);  
}  
  
// injected  
private PokerHandEvaluator pokerHandEvaluator;  
public void setPokerHandEvaluator(  
    PokerHandEvaluator pokerHandEvaluator) {  
    this.pokerHandEvaluator = pokerHandEvaluator;  
}  
}
```

The first thing that you probably noticed about `EvaluateHandMarshallingEndpoint` is that it is much shorter than `EvaluateHandJDomEndpoint`. That's because `EvaluateHandMarshallingEndpoint` doesn't have any of the XML parsing code that was necessary in `EvaluateHandJDomEndpoint`.

Instead, the `invokeInternal()` method is given an `Object` to process. In this case, the `Object` is an `EvaluateHandRequest`:

```
package com.springinaction.poker.webservice;  
import com.springinaction.poker.Card;  
  
public class EvaluateHandRequest {  
    private Card[] hand;  
  
    public EvaluateHandRequest() {}  
  
    public Card[] getHand() {  
        return hand;  
    }  
  
    public void setHand(Card[] cards) {  
        this.hand = cards;  
    }  
}
```

On the other end of `invokeInternal()`, an `EvaluateHandResponse` object is returned. `EvaluateHandResponse` looks like this:

```
package com.springinaction.poker.webservice;  
import com.springinaction.poker.PokerHandType;  
  
public class EvaluateHandResponse {  
    private PokerHandType pokerHand;  
  
    public EvaluateHandResponse() {  
        this(PokerHandType.NONE);  
    }  
}
```

```
public EvaluateHandResponse(PokerHandType pokerHand) {
    this.pokerHand = pokerHand;
}

public PokerHandType getPokerHand() {
    return this.pokerHand;
}

public void setPokerHand(PokerHandType pokerHand) {
    this.pokerHand = pokerHand;
}
}
```

So how is an incoming `<EvaluateHandRequest>` XML message transformed into an `EvaluateHandRequest` object? And, while we're on the subject, how does an `EvaluateHandResponse` object end up being an `<EvaluateHandResponse>` message that gets sent to the client?

What you don't see here is that `AbstractMarshallingPayloadEndpoint` has a reference to an XML marshaler. When it receives an XML message, it uses the marshaler to turn the XML message into an object before calling `invokeInternal()`. Then, when `invokeInternal()` is finished, the marshaler turns the object returned into an XML message.

A large part of Spring-WS is an object-XML mapping (OXM) abstraction. Spring-WS's OXM comes with support for several OXM implementations, including:

- JAXB (versions 1 and 2)
- Castor XML
- JiBX
- XMLBeans
- XStream

You may be wondering which OXM I chose for `EvaluateHandMarshallingEndpoint`. I'll tell you, but not yet. The important thing to note here is that `EvaluateHandMarshallingEndpoint` has no idea where the `Object` that is passed to `invokeInternal()` came from. In fact, there's no reason why the `Object` even has to have been created from unmarshaled XML.

This highlights a key benefit of using a marshaling endpoint. Because it takes a simple object as a parameter, `EvaluateHandMarshallingEndpoint` can be unit-tested just like any other POJO. The test case can simply pass in an `EvaluateHandRequest` object and make assertions on the returned `EvaluateHandResponse`.

Now that we've written our service endpoint, we're ready to wire it up in Spring.

9.4 Wiring it all together

We're finally down to the final stage of developing a Spring-WS service. We need to configure the Spring application context with our endpoint bean and a handful of infrastructure beans required by Spring-WS.

Spring-WS is based on Spring MVC (which we'll see more of in chapter 13). In Spring MVC, all requests are handled by `DispatcherServlet`, a special servlet that dispatches requests to controller classes that process the requests. Similarly, Spring-WS can be fronted by `MessageDispatcherServlet`, a subclass of `DispatcherServlet` that knows how to dispatch SOAP requests to Spring-WS endpoints.²

`MessageDispatcherServlet` is a fairly simple servlet and can be configured in a web application's `web.xml` with the following `<servlet>` and `<servlet-mapping>` elements:

```
<servlet>
    <servlet-name>poker</servlet-name>
    <servlet-class>org.springframework.ws.transport.http.
        ↗ MessageDispatcherServlet</servlet-class>
</servlet>

<servlet-mapping>
    <servlet-name>poker</servlet-name>
    <url-pattern>/services/*</url-pattern>
</servlet-mapping>
```

We'll tweak this `MessageDispatcherServlet`'s configuration a little later, but this will get us started for now.

`MessageDispatcherServlet` is only the front end of Spring-WS. There are a handful of beans that we'll need to wire in the Spring application context. Let's see what those beans are and what they do.

9.4.1 Spring-WS: The big picture

Over the next several pages, we're going to configure several beans in the Spring context. Before we get too deep in the XML, it is probably worthwhile to have a look at the big picture to see what we're about to do. Figure 9.5 shows the beans we'll define and how they relate to one another.

² The "web" in web services seems to imply that all web services are served over HTTP. But that's not necessarily true. Spring-WS has support for JMS, email, and raw TCP/IP-based web services. Nevertheless, since most web services are, in fact, served over HTTP, that's the configuration I'll talk about here.

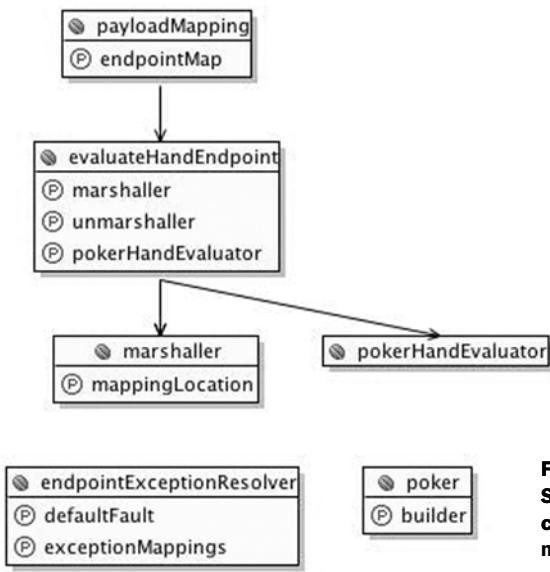


Figure 9.5
Spring-WS service configuration
 consists of several beans, including
 mappings, endpoints, marshalers, and
 other utility beans.

Figure 9.5 shows the beans that we'll configure for the poker hand evaluation service and how they relate to one another. But what are these beans and what do they do? To summarize these six beans:

- **payloadMapping**—Maps incoming XML messages to an appropriate endpoint. In this case, we'll use a mapping that looks up endpoints using the incoming XML's root element (by its qualified name).
- **evaluateHandEndpoint**—This is the endpoint that will process the incoming XML message for the poker hand evaluation service.
- **marshaller**—The `evaluateHandEndpoint` could be written to process the incoming XML as a DOM or JDOM element, or even as a SAX event handler. Instead, the `marshaller` bean will automatically convert XML to and from Java objects.
- **pokerHandEvaluator**—This is a POJO that performs the actual poker hand processing. `evaluateHandEndpoint` will use this bean to do its work.
- **endpointExceptionResolver**—This is a Spring-WS bean that will automatically convert any Java exceptions thrown while processing a request into appropriate SOAP faults.

- poker—Although it's not obvious from its name, this bean will serve the WSDL for the poker hand web service to the client. Either it can serve hand-created WSDL or it can be wired to automatically generate WSDL from the message's XML Schema.

Now that we have a roadmap of where we're going, let's dive right into configuring Spring-WS, starting with the message handler adapter.

9.4.2 Mapping messages to endpoints

When a client sends a message, how does `MessageDispatcherServlet` know which endpoint should process it? Even though we're only building one endpoint in this chapter's example (the evaluate hand endpoint), it's quite possible that `MessageDispatcherServlet` could be configured with several endpoints. We need a way to map incoming messages to the endpoints that process them.

In chapter 13, we'll see how Spring MVC's `DispatcherServlet` maps browser requests to Spring MVC controllers using handler mappings. In a similar way, `MessageDispatcherServlet` uses an *endpoint mapping* to decide which endpoint should receive an incoming XML message.

For the poker hand evaluation service, we'll use Spring-WS's `PayloadRootQNameEndpointMapping`, which is configured in Spring like this:

```
<bean id="payloadMapping"
    class="org.springframework.ws.server.endpoint.mapping.
        ↗ PayloadRootQNameEndpointMapping">
    <property name="endpointMap">
        <map>
            <entry key=
                " {http://www.springinaction.com/poker/schemas}
                    ↗ EvaluateHandRequest"
                value-ref="evaluateHandEndpoint" />
        </map>
    </property>
</bean>
```

`PayloadRootQNameEndpointMapping` maps incoming SOAP messages to endpoints by examining the qualified name (`QName`) of the message's payload and looking up the endpoint from its list of mappings (configured through the `endpointMap` property).

In our example, the root element of the message is `<EvaluateHandRequest>` with a namespace URI of `http://www.springinaction.com/poker/schemas`. This makes the `QName` of the message `{http://www.springinaction.com/poker/schemas}EvaluateHandRequest`. We've mapped this `QName` to a bean named

evaluateHandEndpoint, which is our endpoint implementation that we created in section 9.3.2.

9.4.3 Wiring the service endpoint

Now we're finally down to wiring the endpoint that will process our message. If you chose to use the JDOM-based endpoint then it is configured in Spring like this:

```
<bean id="evaluateHandEndpoint"
      class="com.springinaction.poker.webservice.
           ↗ EvaluateHandJDomEndpoint">
    <property name="pokerHandEvaluator"
              ref="pokerHandEvaluator" />
</bean>
```

The only property that must be injected is the pokerHandEvaluator property. Remember that EvaluateHandJDomEndpoint doesn't actually evaluate the poker hand, but delegates to an implementation of PokerHandEvaluator to do the dirty work. Thus, the pokerHandEvaluator bean should be configured like this:

```
<bean id="pokerHandEvaluator"
      class="com.springinaction.poker.PokerHandEvaluatorImpl" />
```

If the JDOM-based endpoint didn't suit you and instead you chose to use EvaluateHandMarshallingEndpoint, a bit of extra configuration is involved:

```
<bean id="evaluateHandEndpoint"
      class="com.springinaction.poker.webservice.
           ↗ EvaluateHandMarshallingEndpoint">
    <property name="marshaller" ref="marshaller" />
    <property name="unmarshaller" ref="marshaller" />
    <property name="pokerHandEvaluator"
              ref="pokerHandEvaluator" />
</bean>
```

Again, the pokerHandEvaluator property is injected with a reference to a PokerHandEvaluatorImpl. But the marshaling endpoint must have its marshaller and unmarshaller properties set as well. Here we've wired them with references to the same marshaller bean, which we'll configure next.

9.4.4 Configuring a message marshaler

The key to translating objects to and from XML messages is object-XML mapping (OXM). Spring-OXM is a subproject of Spring-WS that provides an abstraction layer over several popular OXM solutions, including JAXB and Castor XML.

The central elements of Spring-OXM are its Marshaller and Unmarshaller interfaces. Implementations of Marshaller are expected to generate XML

elements from Java objects. Conversely, Unmarshaller implementations are used to construct Java objects from XML elements.

`AbstractMarshallingPayloadEndpoint` takes advantage of the Spring-OXM marshalers and unmarshalers when processing messages. When `AbstractMarshallingPayloadEndpoint` receives a message, it hands it off to an Unmarshaller to unmarshal the XML message into an object that is passed to `invokeInternal()`. Then, when `invokeInternal()` is finished, the object returned is given to a Marshaller to marshal the object into XML that will be returned to the client.

Fortunately, you won't have to create your own implementations of Marshaller and Unmarshaller. Spring-OXM comes with several implementations, as listed in table 9.3.

Table 9.3 Marshalers transform objects to and from XML. Spring-OXM provides several marshaling options that can be used with Spring-WS.

OXM solution	Spring-OXM marshaler
Castor XML	<code>org.springframework.oxm.castor.CastorMarshaller</code>
JAXB v1	<code>org.springframework.oxm.jaxb.Jaxb1Marshaller</code>
JAXB v2	<code>org.springframework.oxm.jaxb.Jaxb2Marshaller</code>
JiBX	<code>org.springframework.oxm.jibx.JibxMarshaller</code>
XMLBeans	<code>org.springframework.oxm.xmlbeans.XmlBeansMarshaller</code>
XStream	<code>org.springframework.oxm.xstream.XStreamMarshaller</code>

As you can see, table 9.3 only lists marshaller classes. That's not an oversight, though. Conveniently, all of the marshaller classes in table 9.3 implement both the Marshaller and the Unmarshaller interfaces to provide one-stop solutions for OXM marshaling.

The choice of OXM solution is largely a matter of taste. Each of the OXM options offered by Spring-WS has its good and bad points. XStream, however, has limited support for XML namespaces, which are necessary in defining the types for web services. Therefore, while Spring-OXM's XStream may prove useful for general-purpose XML serialization, it should be disregarded for use with web services.

For the poker hand evaluator service, we chose to use Castor XML. Therefore, we'll need to configure a `CastorMarshaller` in Spring:

```
<bean id="marshaller"
      class="org.springframework.oxm.castor.CastorMarshaller">
    <property name="mappingLocation"
```

```
        value="classpath:mapping.xml" />
    </bean>
```

Castor XML can do some basic XML marshaling without any additional configuration. But our OXM needs are a bit more complex than what default Castor XML can handle. Consequently, we'll need to configure CastorMarshaller to use a Castor XML mapping file. The `mappingLocation` property specifies the location of a Castor XML mapping file. Here we've configured `mappingLocation` to look for a mapping file with the name `mapping.xml` in the root of the application's classpath.

As for the `mapping.xml` file itself, it is shown in listing 9.4.

Listing 9.4 Castor XML mapping file for poker hand service types

```
<?xml version="1.0"?>
<!DOCTYPE mapping PUBLIC
  "-//EXOLAB/Castor Object Mapping DTD Version 1.0//EN"
  "http://castor.exolab.org/mapping.dtd">

<mapping xmlns="http://castor.exolab.org/">
  <class name="com.springinaction.poker.webservice.
    ↗ EvaluateHandRequest">
    <map-to xml="EvaluateHandRequest" />
    <field name="hand"
      collection="array"
      type="com.springinaction.poker.Card"
      required="true">
      <bind-xml name="card" node="element" />
    </field>
  </class>

  <class name="com.springinaction.poker.Card">
    <map-to xml="card" />
    <field name="suit"
      type="com.springinaction.poker.Suit"
      required="true">
      <bind-xml name="suit" node="element" />
    </field>
    <field name="face"
      type="com.springinaction.poker.Face"
      required="true">
      <bind-xml name="face" node="element" />
    </field>
  </class>

  <class name="com.springinaction.poker.webservice.
    ↗ EvaluateHandResponse">
    <map-to xml="EvaluateHandResponse" />
  </class>
```

**Maps
<EvaluateHandRequest>
to EvaluateHandRequest**

**Maps <hand> to
array of Card**

Maps <card> to Card

Maps <suit> to Suit

Maps <face> to Face

**Maps <EvaluateHandResponse>
to EvaluateHandResponse**

```
ns-uri=
    "http://www.springinaction.com/poker/schemas"
ns-prefix="tns" />
<field name="pokerHand"
      type="com.springinaction.poker.PokerHandType"
      required="true">
<bind-xml name="tns:handName" node="element"
          QName-prefix="tns"
          xmlns:tns=
            "http://www.springinaction.com/poker/schemas"/>
</field>
</class>
</mapping>
```

Now we have configured an endpoint mapping bean, the endpoint implementation bean, and an XML marshaling bean. At this point the poker hand evaluator web service is mostly done. We could deploy it and stop for the day. But there are still a couple of beans left that will make the web service more complete. Let's see how to make our web service more robust by declaring a bean that maps Java exceptions to SOAP faults.

9.4.5 Handling endpoint exceptions

Things don't always work out as expected. What will happen if a message can't be marshaled to a Java object? What if the message isn't even valid XML? Maybe the service endpoint or one of its dependencies throws an exception—then what should we do?

If an exception is thrown in the course of processing a message, a SOAP fault will need to be sent back to the client. Unfortunately, SOAP doesn't know or care anything about Java exceptions. SOAP-based web services communicate failure using SOAP faults. We need a way to convert any Java exceptions thrown by our web service or by Spring-WS into SOAP faults.

For that purpose, Spring-WS provides `SoapFaultMappingExceptionResolver`. As shown in figure 9.6, `SoapFaultMappingExceptionResolver` will handle any uncaught exceptions that occur in the course of handling a message and produce an appropriate SOAP fault that will be sent back to the client.

For our service, we've configured a `SoapFaultMappingExceptionResolver` in Spring that looks like this:

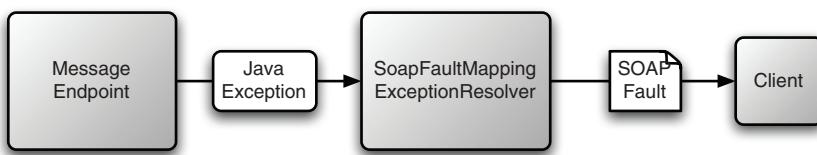


Figure 9.6 `SoapFaultMappingExceptionResolver` maps any Java exceptions thrown from a message endpoint into a SOAP fault to be returned to the client.

```

<bean id="endpointExceptionResolver"
      class="org.springframework.ws.soap.server.endpoint.
           SoapFaultMappingExceptionResolver">
    <property name="exceptionMappings">
      <props>
        <prop key="org.springframework.oxm.
                  UnmarshallingFailureException">
          SENDER, Invalid message received</prop>
        <prop key="org.springframework.oxm.
                  ValidationFailureException">
          SENDER, Invalid message received</prop>
      </props>
    </property>
    <property name="defaultFault"
              value="RECEIVER, Server error" />
  </bean>
  
```

The `exceptionMappings` property is configured with one or more SOAP fault definitions mapped to Java exceptions. The key of each `<prop>` is a Java exception that needs to be translated to a SOAP fault. The value of the `<prop>` is a two-part value where the first part is the type of fault that is to be created and the second part is a string that describes the fault.

SOAP faults come in two types: sender and receiver faults. Sender faults typically indicate that the problem is on the client (e.g., the sender) side. Receiver faults indicate that the web service (e.g., the receiver) received a message from the client but is having some problem processing the message.

For example, if a service receives an XML message that can't be unmarshaled, the marshaler will throw an `org.springframework.oxm.UnmarshallingFailureException`. Because the sender created the useless XML, this is a sender fault. As for the message, it is simply set to "Invalid message received" to indicate the nature of the problem. An `org.springframework.oxm.ValidationFailureException` is handled the same way.

Any exceptions not explicitly mapped in the exceptionMappings property will be handled by the fault definition in the defaultFault property. In this case, we're assuming that if the exception thrown doesn't match any of the mapped exceptions, it must be a problem on the receiving side. Thus, it is a receiver fault and the message simply states "Server error."

9.4.6 Serving WSDL files

Finally, I'm going to make good on my promise to show you where the WSDL file for the poker hand evaluation web service comes from. As you recall from section 9.2.1, we've already created the data portion of the contract as XML Schema in PokerTypes.xsd. Before we go any further, you may want to turn back to listing 9.1 to review the details of the data service.

Pay particular attention to the names I chose for the XML elements that make up our web service messages: EvaluateHandRequest and EvaluateHandResponse. These names weren't chosen arbitrarily. I chose them purposefully to take advantage of a convention-over-configuration feature in Spring-WS that will automatically create WSDL for the poker hand evaluation service.

To make this work, we'll need to configure Spring-WS's DynamicWsdl11Definition. DynamicWsdl11Definition is a special bean that MessageDispatcherServlet works with to generate WSDL from XML Schema. This will come in handy, as we already have some XML Schema defined for the data portion of the contract. Here's how I've configured DynamicWsdl11Definition in Spring:

```
<bean id="poker"
      class="org.springframework.ws.wsdl.wsdl11.
            ↗ DynamicWsdl11Definition">
    <property name="builder">
        <bean class="org.springframework.ws.wsdl.wsdl11.builder.
                  ↗ XsdBasedSoap11Wsdl14jDefinitionBuilder">
            <property name="schema" value="/PokerTypes.xsd"/>
            <property name="portTypeName" value="Poker"/>
            <property name="locationUri"
                      value="http://localhost:8080/Poker-WS/services"/>
        </bean>
    </property>
</bean>
```

DynamicWsdl11Definition works by reading an XML Schema definition, specified here as `PokerTypes.xsd` by the schema property. It looks through the schema file for any element definitions whose names end with Request and Response. It assumes that those suffixes indicate a message that is to be sent to or from a web service operation and creates a corresponding `<wsdl:operation>` element in the WSDL it produces, as shown in figure 9.7.

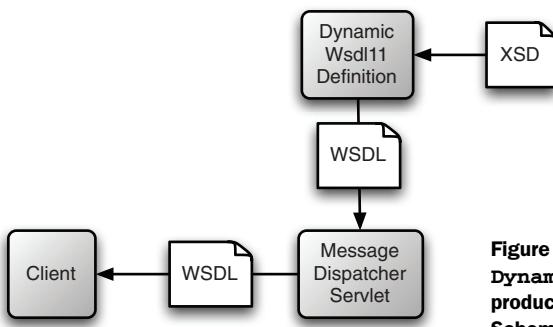


Figure 9.7

DynamicWsdl11Definition automatically produces **WSDL** for a web service based on the **XML Schema** that validates the service's messages.

For example, when **DynamicWsdl11Definition** processes the `PokerTypes.xsd` file, it assumes that the `EvaluateHandRequest` and `EvaluateHandResponse` elements are input and output messages for an operation called `EvaluateHand`. Consequently, the following definition is placed in the generated WSDL:

```

<wsdl:portType name="Poker">
    <wsdl:operation name="EvaluateHand">
        <wsdl:input message="schema:EvaluateHandRequest"
                    name="EvaluateHandRequest">
        </wsdl:input>
        <wsdl:output message="schema:EvaluateHandResponse"
                    name="EvaluateHandResponse">
        </wsdl:output>
    </wsdl:operation>
</wsdl:portType>
    
```

Notice that **DynamicWsdl11Definition** placed the `EvaluateHand` `<wsdl:operation>` within a `<wsdl:portType>` with the name `Poker`. It named the `<wsdl:portType>` using the value wired into its `portTypeName` property.

The last of **DynamicWsdl11Definition**'s properties that we've configured is `locationUri`. This property tells the client where the service can be found. The diagram in figure 9.8 breaks down the URL configured in the `locationUri` property.

In this case, I'm assuming that it will be running on the local machine, but you'll want to change the URL if you'll be running it on a different machine. Notice that the URL ends with `/services`, which matches the `<servlet-mapping>` that we created for `MessageDispatcherServlet`.

Speaking of `<servlet-mapping>`s, we'll also need to add a new `<servlet-mapping>` to `web.xml` so that `MessageDispatcherServlet` will serve WSDL definitions. The following `<servlet-mapping>` definition should do the trick:

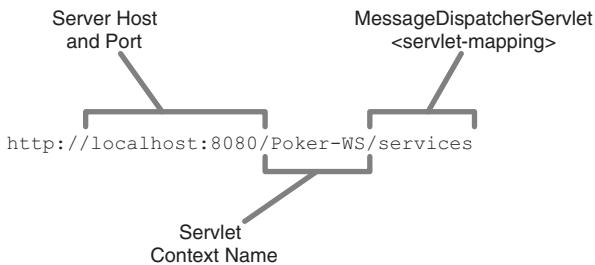


Figure 9.8
The URL configured in the `locationUri` property.

```
<servlet-mapping>
  <servlet-name>poker</servlet-name>
  <url-pattern>*.wsdl</url-pattern>
</servlet-mapping>
```

Now `MessageDispatcherServlet` has been configured (through `DynamicWsdl11Definition`) to automatically produce WSDL for the poker hand evaluation service. The only question left unanswered at this point is where to find the generated WSDL.

The generated WSDL can be found at `http://localhost:8080/Poker-WS/poker.wsdl`. How did I know that? I know that `MessageDispatcherServlet` is mapped to `*.wsdl`, so it will attempt to create WSDL for any request that matches that pattern. But how did it know to produce WSDL for our poker service at `poker.wsdl`?

The answer to that question lies in one last bit of convention followed by `MessageDispatcherServlet`. Notice that I declared the `DynamicWsdl11Definition` bean to have an ID of `poker`. When `MessageDispatcherServlet` receives a request for `/poker.wsdl`, it will look in the Spring context for a WSDL definition bean named `poker`. In this case, it will find the `DynamicWsdl11Definition` bean that I configured.

Using predefined WSDL

`DynamicWsdl11Definition` is perfect for most situations, as it keeps you from having to write the WSDL by hand. But you may have special circumstances that require you to have more control over what goes into the service's WSDL definition. In that case you'll need to create the WSDL yourself and then wire it into Spring using `SimpleWsdl11Definition`:

```
<bean id="poker"
  class="org.springframework.ws.wsdl.wsdl11.
    SimpleWsdl11Definition">
  <property name="wsdl" value="/PokerService.wsdl"/>
</bean>
```

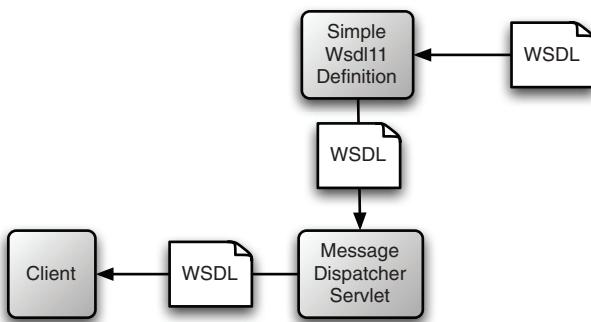


Figure 9.9 `SimpleWsdl11Definition` simply serves a predefined WSDL file through the `MessageDispatcherServlet`. It can optionally be configured to transform the service's address location to match the location of `MessageDispatcherServlet`.

`SimpleWsdl11Definition` doesn't generate WSDL (see figure 9.9); it just serves WSDL that you've provided through the `wsdl` property.

The only problem with predefined WSDL (aside from the trouble that it takes to create it) is that it is statically defined. This creates a problem for the part of the WSDL that specifies the service's location. For example, consider the following (statically defined) chunk of WSDL:

```

<wsdl:service name="PokerService">
    <wsdl:port binding="tns:PokerBinding" name="PokerPort">
        <wsdlsoap:address
            location="http://localhost:8080/Poker-WS/services" />
    </wsdl:port>
</wsdl:service>
  
```

Here the service is defined as being available at `http://localhost:8080/Poker-WS/services`. That is probably okay for development purposes, but it will need to be changed when you deploy it to another server. You could manually change the WSDL file every time you deploy it to another server, but that's cumbersome and is susceptible to mistakes.

But `MessageDispatcherServlet` knows where it's deployed and it knows the URL of requests used to access it. So, instead of tweaking the WSDL every time you deploy it to another server, why not let `MessageDispatcherServlet` rewrite it for you?

All you need to do is to set an `<init-param>` named `transformWsdlLocations` to true and `MessageDispatcherServlet` will take it from there:

```

<servlet>
    <servlet-name>poker</servlet-name>
    ...
  
```

```
<servlet-class>org.springframework.ws.transport.http.  
    ↪ MessageDispatcherServlet</servlet-class>  
<init-param>  
    <param-name>transformWsdlLocations</param-name>  
    <param-value>true</param-value>  
</init-param>  
</servlet>
```

When `transformWsdlLocations` is set to `true`, `MessageDispatcherServlet` will rewrite the WSDL served by `SimpleWsdl11Definition` to match the request's URL.

9.4.7 Deploying the service

We've defined our contract, the endpoint has been written, and all of the Spring-WS beans are in place. At this point, we're ready to package up the web service application and deploy it. Since I chose to use Maven 2 for this project, creating a deployable WAR file is as simple as typing the following at the command line:

```
% mvn package deploy
```

Once Maven's finished, there will be a `Poker-WS.war` file in the target directory, suitable for deployment in most web application servers.

Using Spring-WS to build a web service only demonstrates half of its capabilities. Spring-WS also comes with a client API based on the same message-centric paradigm that Spring-WS promotes on the service side. Let's see how to build a client to consume the poker hand evaluation service using Spring-WS client templates.

9.5 Consuming Spring-WS web services

In chapter 8, you saw how to use `JaxRpcPortProxyFactoryBean` and `XFireClientFactoryBean` to build clients that communicate with remote web services. But both of those take a remote object view of web services, treating web services as remote objects whose methods can be invoked locally. Throughout this chapter, we've been talking about a message-centric approach to web services where clients send XML messages to a web service and receive XML messages back in response. A different paradigm on the service side demands a different paradigm on the client side as well. That's where Spring-WS's `WebServiceTemplate` comes in.

`WebServiceTemplate` is the centerpiece of Spring-WS's client API. As shown in figure 9.10, it employs the Template design pattern to provide the ability to send and receive XML messages from message-centric web services. We've already seen how Spring uses the Template pattern for its data access abstractions in chapter 5.

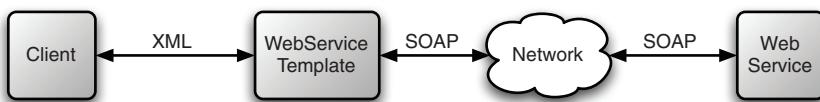


Figure 9.10 `WebServiceTemplate` is the central class in Spring-WS's client API. It sends and receives XML messages to and from web services on behalf of a client.

As we look at Spring-WS's client API, you'll find that it resembles the data access API in many ways.

To demonstrate `WebServiceTemplate`, we'll create several different implementations of the `PokerClient` interface, which is defined as follows:

```

package com.springinaction.ws.client;
import java.io.IOException;
import com.springinaction.poker.Card;
import com.springinaction.poker.PokerHandType;

public interface PokerClient {
    PokerHandType evaluateHand(Card[] cards)
        throws IOException;
}
  
```

Each implementation will show a different way of using `WebServiceTemplate` to send messages to the poker hand evaluation web service.

But first things first... Let's configure `WebServiceTemplate` as a bean in Spring.

9.5.1 Working with web service templates

As I've already mentioned, `WebServiceTemplate` is the central class in the Spring-WS client API. Sending messages to a web service involves producing SOAP envelopes and communications boilerplate code that is pretty much the same for every web service client. When sending messages to a Spring-WS client, you'll certainly want to rely on `WebServiceTemplate` to handle the grunt work so that you can focus your efforts on the business logic surrounding your client.

Configuring `WebServiceTemplate` in Spring is rather straightforward, as shown in this typical `<bean>` declaration:

```

<bean id="webServiceTemplate"
      class="org.springframework.ws.client.core.WebServiceTemplate">
    <property name="messageFactory">
      <bean class="org.springframework.ws.soap.saaj.
        SaaJSoapMessageFactory"/>
    </property>
  
```

```
<property name="messageSender" ref="messageSender" />
</bean>
```

WebServiceTemplate needs to know how to construct the message that will be sent to the service and how to send the message. The object wired into the messageFactory property handles the task of constructing the message. It should be wired with an implementation of Spring-WS's WebServiceMessageFactory interface. Fortunately, you won't have to worry about implementing WebServiceMessageFactory, as Spring-WS comes with three suitable choices (shown in table 9.4).

Table 9.4 WebServiceTemplate relies on a message factory to construct the message sent to a web service. Spring-WS provides three message factory implementations to choose from.

Message factory	What it does
AxiomSoapMessageFactory	Produces SOAP messages using the AXIOM Object Model (AXIOM). Based on the StAX streaming XML API. Useful when working with large messages and performance is a problem.
DomPoxMessageFactory	Produces Plain Old XML (POX) messages using a DOM. Use this message factory when neither the client nor the service cares to deal with SOAP.
SaajSoapMessageFactory	Produces SOAP messages using the SOAP with Attachments API for Java (SAAJ). Because SAAJ uses a DOM, large messages could consume a lot of memory. If performance becomes an issue, consider using AxiomSoapMessageFactory instead.

Since the messages sent to and from the poker hand evaluation service are rather simple, I've chosen to wire a SaajSoapMessageFactory into WebServiceTemplate's messageFactory property. (This is also the default message factory used by MessageDispatcherServlet.) If I were to decide later that performance is an

It's not all SOAP

Figure 9.10 is a bit misleading. It implies that Spring-WS only deals with SOAP-based web services. In fact, Spring-WS only uses SOAP if it is wired with an AxiomSoapMessageFactory or SaajSoapMessageFactory. The DomPoxMessageFactory supports POX messages that aren't sent in a SOAP envelope. If you have an aversion to using SOAP, maybe DomPoxMessageFactory is for you. You may be interested in knowing that the upcoming Spring-WS 1.1 release will also include support for REST.

issue, switching to AXIOM-based messages would be a simple matter of rewiring the messageFactory property.

The messageSender property should be wired with a reference to an implementation of a WebServiceMessageSender. Again, Spring-WS provides a couple of appropriate implementations, as listed in table 9.5.

Table 9.5 Message senders send the messages to a web service. Spring-WS comes with two message senders.

Message sender	What it does
CommonsHttpMessageSender	Sends the message using Jakarta Commons HTTP Client. Supports a preconfigured HTTP client, allowing advanced features such as HTTP authentication and HTTP connection pooling.
HttpURLConnectionMessageSender	Sends the message using Java's basic facilities for HTTP connections. Provides limited functionality.

The choice between CommonsHttpMessageSender and HttpURLConnectionMessageSender boils down to a trade-off between functionality and another JAR dependency. If you won't be needing the advanced features supported by CommonsHttpMessageSender (such as HTTP authentication), HttpURLConnectionMessageSender will suffice. But if you will need those features then CommonsHttpMessageSender is a must—but you'll have to be sure to include Jakarta Commons HTTP in your client's classpath.

As the advanced features aren't an issue for the poker hand evaluation web service, I've chosen HttpURLConnectionMessageSender, which is configured like this in Spring:

```
<bean id="messageSender"
      class="org.springframework.ws.transport.http.
          ↗ HttpURLConnectionMessageSender">
    <property name="url"
              value="http://localhost:8080/Poker-WS/services"/>
  </bean>
```

The url property specifies the location of the service. Notice that it matches the URL in the service's WSDL definition.

If I decide later that I'll need to authenticate to use the poker hand evaluation web service, switching to CommonsHttpMessageSender is a simple matter of changing the messageSender bean's class specification.

Sending a message

Once the `WebServiceTemplate` has been configured, it's ready to use to send and receive XML to and from the poker hand evaluation service. `WebServiceTemplate` provides several methods for sending and receiving messages. This one, however, stands out as the most basic and easiest to understand:

```
public boolean sendAndReceive(Source requestPayload,  
                           Result responseResult)  
throws IOException
```

The `sendAndReceive()` method takes a `java.xml.transform.Source` and a `java.xml.transform.Result` as parameters. The `Source` object represents the message payload to send to the web service, while the `Result` object is to be populated with the message payload returned from the service.

Listing 9.5 shows `TemplateBasedPokerClient`, an implementation of the `PokerClient` interface that uses `WebServiceTemplate`'s `sendAndReceive()` method to communicate with the poker hand evaluation service.

Listing 9.5 Client that uses an injected `WebServiceTemplate` to send and receive XML messages from the poker hand evaluation service

```
package com.springinaction.ws.client;  
import java.io.IOException;  
import org.jdom.Document;  
import org.jdom.Element;  
import org.jdom.Namespace;  
import org.jdom.transform.JDOMResult;  
import org.jdom.transform.JDOMSource;  
import org.springframework.ws.client.core.WebServiceTemplate;  
import com.springinaction.poker.Card;  
import com.springinaction.poker.PokerHandType;  
  
public class TemplateBasedPokerClient  
    implements PokerClient {  
  
    public PokerHandType evaluateHand(Card[] cards)  
        throws IOException {  
  
        Element requestElement =  
            new Element("EvaluateHandRequest");  
        Namespace ns = Namespace.getNamespace(  
            "http://www.springinaction.com/poker/schemas");  
        requestElement.setNamespace(ns);  
        Document doc = new Document(requestElement);  
  
        for(int i=0; i<cards.length; i++) {  
            Element cardElement = new Element("card");  
            Element suitElement = new Element("suit");  
            doc.appendChild(cardElement);  
            cardElement.appendChild(suitElement);  
        }  
        return (PokerHandType) webServiceTemplate.  
            sendAndReceive(doc, new JDOMResult());  
    }  
}
```

Constructs XML message

```

        suitElement.setText(cards[i].getSuit().toString());
        Element faceElement = new Element("face");
        faceElement.setText(cards[i].getFace().toString());
        cardElement.addContent(suitElement);
        cardElement.addContent(faceElement);
        doc.getRootElement().addContent(cardElement);
    }

    JDOMSource requestSource = new JDOMSource(doc);
    JDOMResult result = new JDOMResult();
    webServiceTemplate.sendAndReceive(requestSource, result);

    Document resultDocument = result.getDocument();
    Element responseElement = resultDocument.getRootElement();
    Element handNameElement =
        responseElement.getChild("handName", ns);
    return PokerHandType.valueOf(handNameElement.getText());
}

private WebServiceTemplate webServiceTemplate;
public void setWebServiceTemplate(
    WebServiceTemplate webServiceTemplate) {
    this.webServiceTemplate = webServiceTemplate;
}
}

```

▲

Constructs XML message

Sends message using template

Parses XML response

Injects template

Both Source and Result are interfaces that are a standard part of Java's XML API and are available in the Java SDK. There are countless implementations of these interfaces to choose from, but as you can see in listing 9.5, I chose to use the JDOM implementations. This choice was mostly arbitrary but influenced by the fact that I am familiar with JDOM and know how to use it to construct XML messages.

TemplateBasedPokerClient's evaluateHand() method starts by using JDOM to construct an <EvaluateHandRequest> message from the array of Card elements passed in. Once it has the request message, it calls sendAndReceive() on the WebServiceTemplate. It then uses JDOM to parse the result and find the PokerHandType that should be returned.

Notice that the WebServiceTemplate is injected through a setter method. Therefore, TemplateBasedPokerClient must be configured in Spring as follows:

```

<bean id="templateBasedClient"
    class="com.springinaction.ws.client.TemplateBasedPokerClient">
    <property name="webServiceTemplate" ref="webServiceTemplate" />
</bean>

```

The webServiceTemplate property is wired with a reference to the webServiceTemplate bean that we configured earlier.

While reading through listing 9.5, you may have noticed that the bulk of the evaluateHand() method involves creating and parsing XML. In fact, only one line deals specifically with sending a message. Manually creating and parsing XML messages may be okay when the messages are very simple, but you can probably imagine the amount of code that would be required to construct complex message payloads. Even with the poker hand evaluation service, where the message payload is far from complex, the amount of XML processing code is staggering.

Fortunately, you don't have to deal with all of that XML on your own. In section 9.4.4 you saw how an endpoint can use a marshaler to transform objects to and from XML. Now I'll show you how WebServiceTemplate can also take advantage of marshalers to eliminate the need for XML processing code on the client side.

Using marshalers on the client side

In addition to the simple sendAndReceive() method we used in listing 9.5, WebServiceTemplate also provides marshalSendAndReceive(), a method for sending and receiving XML messages that are marshaled to and from Java objects.

Using marshalSendAndReceive() is a simple matter of passing in a request object as a parameter and receiving a response object as the returned value. In the case of the poker hand evaluation service, these objects are EvaluateHandRequest and EvaluateHandResponse, respectively.

Listing 9.6 shows MarshallingPokerClient, an implementation of PokerClient that uses marshalSendAndReceive() to communicate with the poker hand evaluation service.

Listing 9.6 MarshallingPokerClient, which takes advantage of a marshaler to convert objects to and from XML

```
package com.springinaction.ws.client;
import java.io.IOException;
import org.springframework.ws.client.core.WebServiceTemplate;
import com.springinaction.poker.Card;
import com.springinaction.poker.PokerHandType;
import com.springinaction.poker.webservice.EvaluateHandRequest;
import com.springinaction.poker.webservice.EvaluateHandResponse;

public class MarshallingPokerClient
    implements PokerClient {

    public PokerHandType evaluateHand(Card[] cards)
        throws IOException {
            EvaluateHandRequest request = new EvaluateHandRequest();
            request.setHand(cards);
            Creates request object
        }
}
```

```

EvaluateHandResponse response = (EvaluateHandResponse)
    webServiceTemplate.marshalSendAndReceive(request); | Sends
                                                     | request

    return response.getPokerHand(); <— Returns poker hand response
}

private WebServiceTemplate webServiceTemplate;
public void setWebServiceTemplate(
    WebServiceTemplate webServiceTemplate) {
    this.webServiceTemplate = webServiceTemplate;
}
}

```

Wow! `MarshallingPokerClient`'s `evaluateHand()` method is much simpler and no longer involves any XML processing. Instead, it constructs an `EvaluateHandRequest` object and populates it with the `Card` array that was passed in. After calling `marshalSendAndReceive()`, passing in the `EvaluateHandRequest` object, `evaluateHand()` receives an `EvaluateHandResponse`, which it uses to retrieve the `PokerHandType` that it returns.

So, how does `WebServiceTemplate` know how to marshal/unmarshal `EvaluateHandRequest` and `EvaluateHandResponse` objects? Is it really that smart?

Well, no... not really. Actually, it doesn't know anything about marshaling and unmarshaling those objects. However, as shown in figure 9.11, it can be wired with a marshaler and an unmarshaler that know how to handle the marshaling:

```

<bean id="webServiceTemplate"
    class="org.springframework.ws.client.core.WebServiceTemplate">
    <property name="messageFactory">
        <bean class="org.springframework.ws.soap.saaj.
            ↗ SaajSoapMessageFactory"/>
    </property>
    <property name="messageSender" ref="urlMessageSender"/>

    <property name="marshaller" ref="marshaller" />
    <property name="unmarshaller" ref="marshaller" />
</bean>

```

Here I've wired both the `marshaller` and `unmarshaller` properties with a reference to a `marshaller` bean, which is the same `CastorMarshaller` configured in section 9.4.4. But it could just as easily have been any of the marshalers listed in table 9.3.

`MarshallingPokerClient` is much cleaner than `TemplateBasedPokerClient`. But there's still a little bit more we can do to trim the fat. Let's see how to use Spring-WS's `WebServiceGatewaySupport` class to eliminate the need to explicitly wire in a `WebServiceTemplate`.

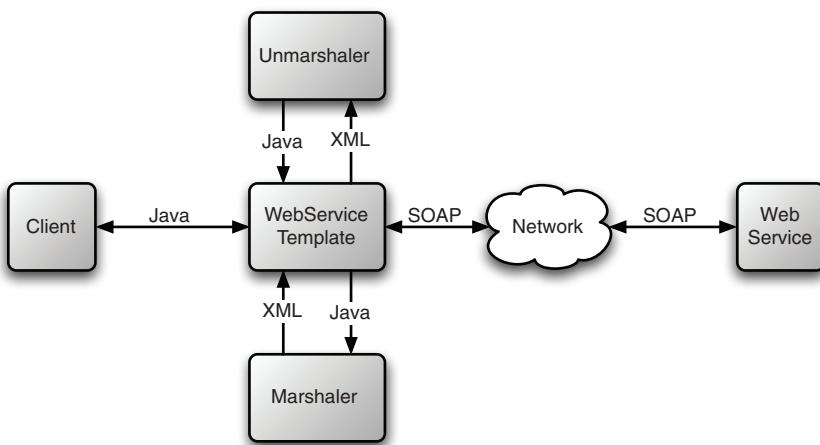


Figure 9.11 When wired with a marshaler and unmarshaler, a client can send and receive Java objects from `WebServiceTemplate`. `WebServiceTemplate` will use the marshaler and unmarshaler to transform the Java objects to and from XML.

9.5.2 Using web service gateway support

As you'll recall from chapter 5 (see sections 5.3.3, 5.4.3, 5.5.3, and 5.6.2), Spring's data access API includes convenient support classes that provide templates so that the templates themselves do not need to be configured. In a similar way, Spring-WS provides `WebServiceGatewaySupport`, a convenient support class that automatically provides a `WebServiceTemplate` to client classes that subclass it.

Listing 9.7 shows one final implementation of `PokerClient`, `PokerServiceGateway`, that extends `WebServiceGatewaySupport`.

Listing 9.7 `WebServiceGatewaySupport`, which provides a `WebServiceTemplate` through `getWebServiceTemplate()`

```
package com.springinaction.ws.client;
import java.io.IOException;
import org.springframework.ws.client.core.support.
    ↗ WebServiceGatewaySupport;
import com.springinaction.poker.Card;
import com.springinaction.poker.PokerHandType;
import com.springinaction.poker.webservice.EvaluateHandRequest;
import com.springinaction.poker.webservice.EvaluateHandResponse;

public class PokerServiceGateway
    extends WebServiceGatewaySupport
```

```

    implements PokerClient {

    public PokerHandType evaluateHand(Card[] cards)
        throws IOException {
        EvaluateHandRequest request = new EvaluateHandRequest();
        request.setHand(cards);
        EvaluateHandResponse response = (EvaluateHandResponse)
            getWebServiceTemplate().marshalSendAndReceive(request);
        return response.getPokerHand();
    }
}

```

**Uses provided
WebServiceTemplate**

As you can see, `PokerServiceGateway` isn't much different from `MarshallingPokerClient`. The key difference is that `PokerServiceGateway` isn't injected with a `WebServiceTemplate`. Instead, it gets its `WebServiceTemplate` by calling `getWebServiceTemplate()`. Under the covers, `WebServiceGatewaySupport` will create a `WebServiceTemplate` object without one being explicitly defined in Spring.

Even though `WebServiceTemplate` no longer needs to be defined in Spring, the details of how to create a `WebServiceTemplate` must still be configured through `WebServiceGatewaySupport`'s properties. For `PokerServiceGateway`, this means configuring the `messageFactory`, `messageSender`, `marshaller`, and `unmarshaller` properties:

```

<bean id="pokerClientGateway"
    class="com.springinaction.ws.client.PokerServiceGateway">
    <property name="messageFactory">
        <bean class="org.springframework.ws.soap.saaj.
            SAAJSoapMessageFactory"/>
    </property>
    <property name="messageSender" ref="messageSender"/>
    <property name="marshaller" ref="marshaller" />
    <property name="unmarshaller" ref="marshaller" />
</bean>

```

Notice that the properties are configured exactly as they were with `WebServiceTemplate`.

9.6 Summary

Traditionally, web services have been viewed as just another remoting option. In fact, some developers lovingly refer to SOAP as "CORBA with XML."

The problem with the web services as remoting view is that it leads to tight coupling between a service and its clients. When treated as remoting, a client is bound to the service's internal API. The contract with the client is a side effect of this binding. Changes to the service could break the contract with the client, requiring the client to change or requiring the service to be versioned.

In this chapter, we've looked at web services from a different angle, taking a message-centric view. This approach is known as contract-first web services, as it elevates the contract to be a first-class citizen of the service. Rather than simply being remote objects, contract-first web services are implemented as message endpoints that process messages sent by the client and defined by the contract. Consequently, the service and its API can be changed without impacting the contract.

Spring-WS is an exciting new web service framework that encourages contract-first web services. Based on Spring MVC, Spring-WS endpoints handle XML messages sent from the client, producing responses that are also XML messages.

If you're like me, you're probably a bit skeptical about all of the work that went into configuring a web service in Spring-WS. I won't deny that contract-first web services require a bit more work than using XFire to SOAP-ify a bean in contract-last style. In fact, when I first looked at Spring-WS, I initially dismissed it as too much work and no benefit... crazy talk.

But after some more thought, I realized that the benefits of decoupling the service's contract from the application's internal API far outweigh the extra effort required by Spring-WS. And that work will pay dividends in the long run as we are able to revise and refactor our application's internal API without worrying about breaking the service's contract with its clients.

Web services, especially those that are contract first, are a great way for applications to communicate with each other in a loosely coupled way. Another approach is to send messages using the Java Message Service (JMS). In the next chapter, we'll explore Spring's support for asynchronous messaging with JMS.

10

Spring messaging

This chapter covers

- Sending and receiving asynchronous messages
- Creating message-driven POJOs
- Asynchronous remoting with Lingo

It's 4:55 PM on Friday. You're only minutes away from starting a much-anticipated vacation. You have just enough time to drive to the airport and catch your flight. But before you pack up and head out, you need to be sure that your boss and colleagues know the status of the work you've been doing so that they can pick up where you left off on Monday. Unfortunately, some of your colleagues have already skipped out for an early weekend departure... and your boss is tied up in a meeting. What do you do?

You could call your boss's cell phone... but it's not necessary to interrupt his meeting for a mere status report. Maybe you could stick around and wait until he returns from the meeting. But it's anyone's guess how long the meeting will last and you have a plane to catch. Or perhaps you could leave a sticky note on his monitor... right next to 100 other sticky notes that it will blend in with.

The most practical way to communicate your status and still catch your plane is to send a quick email to your boss and your colleagues, detailing your progress and promising to send a postcard. You don't know where they are or when they'll actually read the email, but you do know that they'll eventually return to their desk and read it. Meanwhile, you're on your way to the airport.

Sometimes it's necessary to talk to someone directly. If you injure yourself and need an ambulance, you're probably going to pick up the phone—emailing the hospital just won't do. Often, however, sending a message is sufficient and offers some advantages over direct communication—such as letting you get on with your vacation.

In the past few chapters, you've seen how to use RMI, Hessian, Burlap, HTTP invoker, and web services to enable communication between applications. All of these communication mechanisms employ synchronous communication in which a client application directly contacts a remote service and waits for the remote procedure to complete before continuing.

Synchronous communication has its place, but it is not the only style of inter-application communication available to developers. Asynchronous messaging is a way of indirectly sending messages from one application to another without waiting for a response. There are several advantages of asynchronous messaging over synchronous messaging, as you'll soon see.

The Java Message Service (JMS) is a standard API for asynchronous messaging. In this chapter, we're going to look at how Spring simplifies sending and receiving messages with JMS. In addition to basic sending and receiving of messages, we'll look at Spring's support for message-driven POJOs, a way to receive messages that resembles EJB's message-driven beans (MDBs). Finally, we'll wrap up with a look at Lingo, a remoting extension for Spring that uses JMS as its transport mechanism.

But before we dive into the nuts and bolts of Spring's JMS support, let's review the basics of asynchronous messaging with JMS.

10.1 A brief introduction to JMS

Much like the remoting mechanisms we've covered in previous chapters, JMS is all about applications communicating with one another. JMS differs from those other mechanisms, however, in how information is transferred between systems.

Remoting options like RMI and Hessian/Burlap are synchronous. As illustrated in figure 10.1, when the client invokes a remote method, the client must wait for the method to complete before moving on. Even if the remote method doesn't return anything back to the client, the client will be put on hold until the service is done.

JMS, on the other hand, provides Java applications with the option of communicating asynchronously. When messages are sent asynchronously, as shown in figure 10.2, the client does not have to wait for the service to process the message or even for the message to be delivered. The client sends its message and then moves along with the assumption that the service will eventually receive and process the message.

Asynchronous communication through JMS offers several advantages over synchronous communication. We'll take a closer look at these advantages in a moment. First, let's see how messages are sent using JMS.

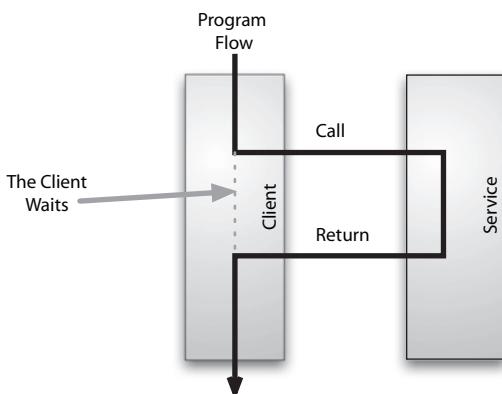


Figure 10.1
When communicating synchronously, the client must wait for the service to complete.

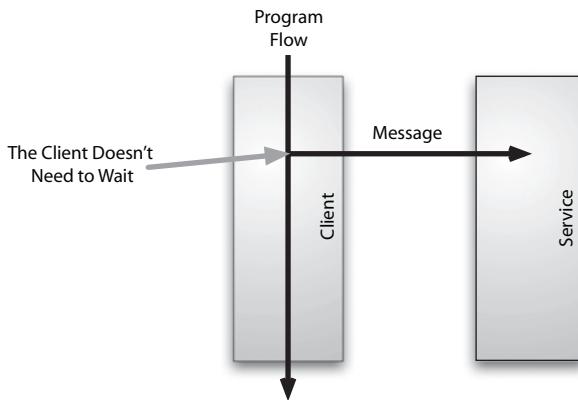


Figure 10.2
Asynchronous communication is a no-wait form of communication.

10.1.1 Architecting JMS

Most of us take the postal service for granted. Millions of times every day, people place letters, cards, and packages in the hands of postal workers, trusting that they'll get to the desired destination. The world's too big of a place for us to hand-deliver these things ourselves, so we rely on the postal system to handle it for us. We address it, place the necessary postage on it, and then drop it in the mail to be delivered without giving a second thought to how it might get there.

The key to the postal service is indirection. When Grandma's birthday comes around, it'd be very inconvenient if we had to deliver a card directly to her. Depending on where she lives, we'd have to set aside anywhere from a few hours to a few days to deliver a birthday card. Fortunately, they'll deliver the card to her while we go about our lives.

Similarly, indirection is the key to JMS. When one application sends information to another through JMS, there is no direct link between the two applications. Instead, the sending application places the message in the hands of a service that will ensure delivery to the receiving application.

There are two main concepts in JMS: *message brokers* and *destinations*.

When an application sends a message, it hands it off to a message broker. A message broker is JMS's answer to the post office. The message broker will ensure that the message is delivered to the specified destination, leaving the sender free to go about other business.

When you send a letter through the mail, it's important to address it so that the postal service knows where it should be delivered. Likewise, in JMS, messages

are addressed with a destination. Destinations are like mailboxes where the messages are placed until someone comes to pick them up.

However, unlike mail addresses, which may indicate a specific person or street address, destinations are less specific. Destinations are only concerned about where the message will be picked up—not who will pick them up. In this way, destinations are like sending a letter addressed “To resident.”

In JMS, there are two types of destination: queues and topics. Each of these is associated with a specific messaging model, either point-to-point (for queues) or publish-subscribe (for topics).

Point-to-point messaging model

In the point-to-point model, each message has exactly one sender and one receiver, as illustrated in figure 10.3. When the message broker is given a message, it places the message in a queue. When a receiver comes along and asks for the next message in the queue, the message is pulled from the queue and delivered to the receiver. Because the message is removed from the queue as it is delivered, it is guaranteed that the message will be delivered to only one receiver.

Although each message in a message queue is delivered to only one receiver, this does not imply that there is only one receiver pulling messages from the queue. In fact, it's quite likely that there are several receivers processing messages from the queue. But they'll each be given their own messages to process.

This is analogous to waiting in line at the bank. As you wait, you may notice that there are multiple tellers available to help you with your financial transaction. After each customer is helped and a teller is freed up, she will call for the next person in line. When it's your turn at the front of the line, you'll be called to the counter and helped by one teller. The other tellers will help other banking customers.

Another observation to be made at the bank is that when you get in line, you probably won't know which teller will eventually help you. You could count how

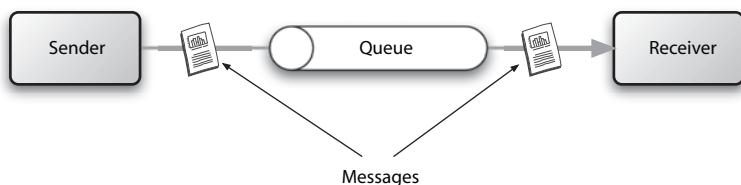


Figure 10.3 A message decouples a message sender from the message receiver. While a queue may have several receivers, each message is picked up by exactly one receiver.

many people are in line, match that up with the number of available tellers, note which teller is fastest, and then come up with a guess as to which teller will call you to their window. But chances are you'll be wrong and end up at a different teller's window.

Likewise, in JMS, if there are multiple receivers listening to a queue, there's no way of knowing which one will actually process a specific message. This uncertainty is actually a good thing because it enables an application to scale up message processing by simply adding another listener to the queue.

Publish-subscribe messaging model

In the publish-subscribe messaging model, messages are sent to a topic. As with queues, many receivers may be listening to a topic. However, unlike queues where a message is delivered to exactly one receiver, all subscribers to a topic will receive a copy of the message, as shown in figure 10.4.

As implied by its name, the publish-subscribe message model is very much like the model of a magazine publisher and its subscribers. The magazine (a message) is published, sent to the postal service, and then all subscribers receive their own copy.

The magazine publisher analogy breaks down, however, when you realize that in JMS, the publisher has no idea of who its subscribers are. The publisher only knows that its message will be published to a particular topic—not who is listening to that topic. This also implies that the publisher has no idea of how the message will be processed.

Now that we've covered the basics of JMS, let's see how JMS messaging compares to synchronous RPC.

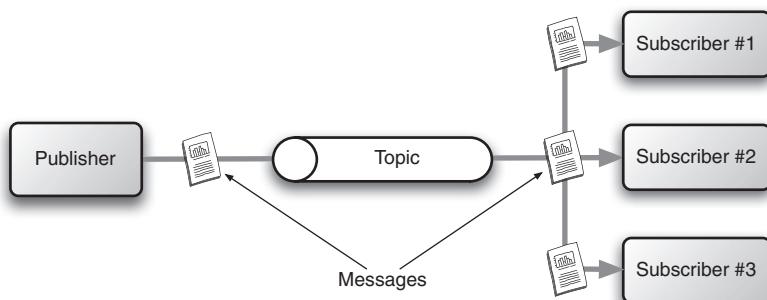


Figure 10.4 Like queues, topics decouple message senders from message receivers. Unlike queues, however, a topic message could be delivered to many topic subscribers.

10.1.2 Assessing the benefits of JMS

Even though it's very intuitive and simple to set up, synchronous communication imposes several limitations on the client of a remote service. Most significantly:

- Synchronous communication implies waiting. When a client invokes a method on a remote service, it must wait for the remote method to complete before the client can continue. If the client communicates frequently with the remote service and/or the remote service is slow to respond, this could negatively impact performance of the client application.
- The client is coupled to the service through the service's interface. If the interface of the service changes, all of the service's clients will also need to change accordingly.
- The client is coupled to the service's location. A client must be configured with the service's network location so that it knows how to contact the service. If the network topology changes, the client will need to be reconfigured with the new location.
- The client is coupled to the service's availability. If the service becomes unavailable, the client is effectively crippled.

While synchronous communication has its place, these shortcomings should be taken into account when deciding what communication mechanism is a best fit for your application's needs. If these constraints are a concern for you, you may want to consider how asynchronous communication with JMS addresses these issues.

No waiting

When a message is sent with JMS, the client doesn't need to wait around for it to be processed or even delivered. The client drops the message off with the message broker and moves along with faith that the message will make it to the appropriate destination.

Since it doesn't have to wait, the client will be freed up to perform other activities. With all of this free time, the client's performance can be dramatically improved.

Message-oriented

Unlike RPC communication that is typically oriented around a method call, messages sent with JMS are data-centric. This means that the client isn't fixed to a specific method signature. Any queue or topic subscriber that can process the data sent by the client can process the message. The client doesn't need to be aware of any service specifics.

Location independence

Synchronous RPC services are typically located by their network address. The implication of this is that clients are not resilient to changes in network topology. If a service's IP address changes or if it's configured to listen on a different port, the client must be changed accordingly or the client will be unable to access the service.

In contrast, JMS clients have no idea who will process their messages or where the service is located. The client only knows the queue or topic through which the messages will be sent. As a result, it doesn't matter where the service is located, as long as it can retrieve messages from the queue or topic.

In the point-to-point model, it's possible to take advantage of location independence to create a cluster of services. If the client is unaware of the service's location and if the service's only requirement is that it must be able to access the message broker, there's no reason why multiple services can't be configured to pull messages from the same queue. If the service is being overburdened and falling behind in its processing, all we need to do is turn up a few more instances of the service to listen to the same queue.

Location independence takes on another interesting side effect in the publish-subscribe model. Multiple services could all be subscribed to a single topic, receiving duplicate copies of the same message. But each service could process that message differently. For example, let's say you have a set of services that together process a message that details the new hire of an employee. One service might add the employee to the payroll system, another to the HR portal, and yet another makes sure that the employee is given access to the systems they'll need to do their job. Each service works independently on the same data that they each received from a topic.

Guaranteed delivery

In order for a client to communicate with a synchronous service, the service must be listening at the IP address and port specified. If the service were to go down or otherwise become unavailable, the client wouldn't be able to proceed.

However, when sending messages with JMS, the client can rest assured that its messages will be delivered. Even if the service is unavailable when a message is sent, it will be stored until the service is available again.

Now that you have a feel for the basics of JMS and asynchronous messaging, let's set up a JMS message broker that we'll use in our examples. Although you're free to use any JMS message broker you'd like, we're going to use the popular ActiveMQ message broker.

10.1.3 Setting up ActiveMQ in Spring

ActiveMQ is a great open source message broker and a wonderful option for asynchronous messaging with JMS. Although ActiveMQ began its life as a Codehaus project, it is in the process of moving to Apache. As this is being written, ActiveMQ 4.1.0 is in Apache's incubator program.

To get started with ActiveMQ, you'll need to download the binary distribution from www.activemq.org. Once you've downloaded ActiveMQ, unzip it to your local hard drive. In the base directory of the unzipped distribution, you'll find incubator-activemq-4.1.0.jar. This is the JAR file you'll need to add to the application's classpath to be able to use ActiveMQ's API.

In the bin directory, you'll find a script that starts ActiveMQ: activemq for Unix users or activemq.bat for Windows users. Run the script and within moments ActiveMQ will be ready and waiting to broker your messages.

Creating a connection factory

Throughout this chapter, we're going to see different ways that Spring can be used to both send and receive messages through JMS. In all cases, we'll need a JMS connection factory to be able to send messages through the message broker. Since we're using ActiveMQ as our message broker, we'll have to configure the JMS connection factory so that it knows how to connect to ActiveMQ. ActiveMQConnectionFactory is the JMS connection factory that comes with ActiveMQ, and it is configured in Spring like this:

```
<bean id="connectionFactory"
      class="org.apache.activemq.ActiveMQConnectionFactory">
    <property name="brokerURL" value="tcp://localhost:61616"/>
</bean>
```

Later in this chapter we'll use this connectionFactory bean a lot. But for now, suffice it to say that the brokerURL property tells the connection factory where the message broker is located. In this case, the URL in the brokerURL property tells ActiveMQConnectionFactory to connect to ActiveMQ on the local machine at port 61616 (which is the port that ActiveMQ listens to by default).

Declaring an ActiveMQ message destination

In addition to a connection factory, we'll need a destination for the messages to be passed along on to. The destination can be either a queue or a topic, depending on the needs of the application.

Regardless of whether you are using a queue or a topic, you must configure the destination bean in Spring using a message broker-specific implementation class. For example, the following <bean> declaration declares an ActiveMQ queue:

```
<bean id="rantzDestination"
      class="org.apache.activemq.command.ActiveMQQueue">
    <constructor-arg index="0" value="rantz.marketing.queue"/>
</bean>
```

Similarly, the following `<bean>` declares a topic for ActiveMQ:

```
<bean id="rantzDestination"
      class="org.apache.activemq.command.ActiveMQTopic">
    <constructor-arg index="0" value="rantz.marketing.topic"/>
</bean>
```

Again, these beans illustrate how to configure destinations for ActiveMQ 4.1.0. If you're using a different message broker, be sure to configure destination beans using implementations that are appropriate to the message broker you're using.

With the common beans out of the way, we're ready to start sending and receiving messages. For that, we'll use `JmsTemplate`, the centerpiece of Spring's JMS support. But first, let's gain an appreciation for what `JmsTemplate` provides by looking at what JMS is like without `JmsTemplate`.

10.2 Using JMS with Spring

As you've seen, JMS gives Java developers a standard API for interacting with message brokers and for sending and receiving messages. Furthermore, virtually every message broker implementation available supports JMS, so there's no reason to learn a proprietary messaging API for every message broker you deal with.

But while JMS offers a universal interface to all message brokers, its convenience comes at a cost. Sending and receiving messages with JMS is not a simple matter of licking a stamp and placing it on an envelope. As you'll see, JMS demands that you also fuel up the mail carrier's truck.

10.2.1 Tackling runaway JMS code

In chapter 5 (see section 5.3.1) you saw how conventional JDBC code can be an unwieldy mess of code to handle connections, statements, result sets, and exceptions. Unfortunately, conventional JMS follows a similar model, as you'll observe in listing 10.1.

Listing 10.1 Sending a message using conventional (non-Spring) JMS

```
ConnectionFactory cf =
  new ActiveMQConnectionFactory("tcp://localhost:61616");
Connection conn = null;
Session session = null;
```

Gets connection factory

```

try {
    conn = cf.createConnection();   ↪ Creates connection
    session = conn.createSession(false,   | Creates
        Session.AUTO_ACKNOWLEDGE);     | session

    Destination destination = new ActiveMQQueue("myQueue");   ↪ Creates queue

    MessageProducer producer = session.createProducer(destination);
    TextMessage message = session.createTextMessage();
    message.setText("Hello world!");                                | Sets up message

    producer.send(message);   ↪ Sends message
} catch (JMSEException e) {   | Handles exception—
...
} finally {
    try {
        if(session != null) { session.close(); }   | Cleans up
        if(conn != null) { conn.close(); }           | resource
    } catch (JMSEException ex) {}
}

```

At the risk of sounding repetitive—holly runaway code, Batman! Just as with the JDBC example, there are almost 20 lines of code here just to send a “Hello world!” message. Actually, only a few lines actually send the message. The rest are merely setting the stage for sending a message.

As you can see in listing 10.2, it’s not any prettier for the receiving end, either.

Listing 10.2 Receiving a message using conventional (non-Spring) JMS

```

ConnectionFactory cf =
    new ActiveMQConnectionFactory("tcp://localhost:61616");
Connection conn = null;
Session session = null;
try {
    conn = cf.createConnection();   ↪ Creates connection
    session = conn.createSession(false,   | Creates session
        Session.AUTO_ACKNOWLEDGE);

    Destination destination = new ActiveMQQueue("myQueue");   ↪ Selects destination

    MessageConsumer consumer = session.createConsumer(destination);
    conn.start();

    Message message = consumer.receive();   ↪ Receives message

    TextMessage textMessage = (TextMessage) message;
    System.out.println("GOT A MESSAGE: " + textMessage.getText());
} catch (JMSEException e) {   | Handles any
    System.out.println(e);           | JMSEExceptions
}

```

```

} finally {
    try {
        if(session != null) { session.close(); }
        if(conn != null) { conn.close(); }
    } catch (JMSEException ex) {}
}

```

Closes session and connection

Again, just as in listing 10.1, there's a lot of code here to do something so simple. If you take a line-by-line comparison, you'll find that they're almost identical. And if you were to look at a thousand other JMS examples, you'd find them all to be strikingly similar. Some may retrieve their connection factories from JNDI and some may use a topic instead of a queue. Nevertheless, they all follow roughly the same pattern.

A consequence of all of this boilerplate code is that you'll find that you repeat yourself every time you work with JMS. Worse still, you'll find yourself repeating other developers' JMS code.

We've already seen in chapter 5 how Spring's `JdbcTemplate` handles runaway JDBC boilerplate. Now let's look at how Spring's `JmsTemplate` can do the same thing for JMS boilerplate code.

10.2.2 Working with JMS templates

`JmsTemplate` is Spring's answer to verbose and repetitive JMS code. `JmsTemplate` takes care of creating a connection, obtaining a session, and the actual sending and receiving of messages. This leaves you to focus your development efforts on constructing the message to send or processing messages that are received.

What's more, `JmsTemplate` can handle any clumsy `JMSEException` that may be thrown along the way. If a `JMSEException` is thrown in the course of working with `JmsTemplate`, `JmsTemplate` will catch it and rethrow it as one of the unchecked subclasses of `JmsException` in the first column of table 10.1.

Table 10.1 The subclasses of Spring's `JmsException` compared to the subclasses of `JMSEException`.

Spring (<code>org.springframework.jms.*</code>)	JMS (<code>javax.jms.*</code>)
<code>support.destination.DestinationResolutionException</code>	Spring specific—Thrown when Spring can't resolve a destination name.
<code>IllegalStateException</code>	<code>IllegalStateException</code>
<code>InvalidClientIDException</code>	<code>InvalidClientIDException</code>
<code>InvalidDestinationException</code>	<code>InvalidDestinationException</code>

Table 10.1 The subclasses of Spring's `JmsException` compared to the subclasses of `JMSEException`. (continued)

Spring (<code>org.springframework.jms.*</code>)	JMS (<code>javax.jms.*</code>)
<code>InvalidSelectorException</code>	<code>InvalidSelectorException</code>
<code>JmsSecurityException</code>	<code>JMSecurityException</code>
<code>listener.adapter.ListenerExecutionFailedException</code>	Spring specific—Thrown when execution of a listener method fails.
<code>support.converter.MessageConversionException</code>	Spring specific—Thrown when message conversion fails.
<code>MessageEOFException</code>	<code>MessageEOFException</code>
<code>MessageFormatException</code>	<code>MessageFormatException</code>
<code>MessageNotReadableException</code>	<code>MessageNotReadableException</code>
<code>MessageNotWritableException</code>	<code>MessageNotWritableException</code>
<code>ResourceAllocationException</code>	<code>ResourceAllocationException</code>
<code>TransactionInProgressException</code>	<code>TransactionInProgressException</code>
<code>TransactionRolledBackException</code>	<code>TransactionRolledBackException</code>
<code>UncategorizedJmsException</code>	Spring specific—Thrown when no other exception applies.

In fairness to the JMS API, `JMSEException` does come with a rich and descriptive set of subclasses that give you a better sense of what went wrong. Nevertheless, all of these subclasses of `JMSEException` are checked exceptions and thus must be caught. `JmsTemplate` will attend to that for you.

A tale of two `JmsTemplates`

Spring actually comes with two JMS template classes: `JmsTemplate` and `JmsTemplate102`. `JmsTemplate102` is a special version of `JmsTemplate` for JMS 1.0.2 providers. In JMS 1.0.2, topics and queues are treated as completely different concepts known as *domains*. In JMS 1.1+, however, topics and queues are unified under a domain-independent API. Because topics and queues are treated so differently in JMS 1.0.2, there has to be a special `JmsTemplate102` for interacting with older JMS implementations. In this chapter, we're going to assume a modern JMS provider and therefore will focus our attention on `JmsTemplate`.

Wiring JmsTemplate

To use `JmsTemplate`, we'll need to declare it as a bean in the Spring configuration file. The following XML should do the trick:

```
<bean id="jmsTemplate"
      class="org.springframework.jms.core.JmsTemplate">
    <property name="connectionFactory" ref="connectionFactory" />
</bean>
```

Because `JmsTemplate` needs to know how to get connections to the message broker, we had to set the `connectionFactory` property with a bean that implements JMS's `ConnectionFactory` interface. Here we've wired it with a reference to the `connectionFactory` bean that we declared earlier in section 10.1.3.

`JmsTemplate` has a few other properties that come in handy in certain circumstances. But we'll defer discussion of those until we need them. For now, the `JmsTemplate` we've configured is good enough to get started.

That's all you need to do to configure `JmsTemplate`—it is now ready to go. Let's send a message!

Sending messages

One of the benefits of signing up as a motorist in the RoadRantz application is that you can opt in to be sent coupons and great deals on car washes, oil changes, and other automotive products and services. Motorists who are interested in third-party offers are tracked in a separate marketing system from the main RoadRantz application. When a user registers as a RoadRantz motorist, if they elect to receive third-party offers, their name and email address is sent to the RoadRantz marketing system as a JMS message.

On the RoadRantz side, we're going to use `JmsTemplate` to send the motorist information to the RoadRantz marketing system. Listing 10.3 shows `RantzMarketingGatewayImpl`, which is the class through which RoadRantz will interact with the marketing system.

Listing 10.3 Sending a motorist message using `JmsTemplate`

```
package com.roaddrantz.marketing;
import javax.jms.Destination;
import javax.jms.JMSEException;
import javax.jms.MapMessage;
import javax.jms.Message;
import javax.jms.Session;
import org.springframework.jms.core.JmsTemplate;
import org.springframework.jms.core.MessageCreator;
import com.roaddrantz.domain.Motorist;
```

```

public class RantzMarketingGatewayImpl
    implements RantzMarketingGateway {
    public RantzMarketingGatewayImpl() {}

    public void sendMotoristInfo(final Motorist motorist) {
        jmsTemplate.send(
            destination,           ←———— Specifies destination
            new MessageCreator() {
                public Message createMessage(Session session)
                    throws JMSException {
                    MapMessage message = session.createMapMessage();

                    message.setString("lastName", motorist.getLastName());
                    message.setString("firstName", motorist.getFirstName());
                    message.setString("email", motorist.getEmail());

                    return message;
                }
            });
    }

    private JmsTemplate jmsTemplate;
    public void setJmsTemplate(JmsTemplate jmsTemplate) {
        this.jmsTemplate = jmsTemplate;
    }                                     ←———— Injects JmsTemplate

    private Destination destination;
    public void setDestination(Destination destination) {
        this.destination = destination;
    }                                     ←———— Injects Destination
}

```

The `sendMotoristInfo()` method is the centerpiece of `RantzMarketingGatewayImpl`. It does nothing more than use the `JmsTemplate`'s `send()` method to send the message.

The first parameter to the `send()` method is the name of the JMS Destination that the message will be sent to. Here we're using the `destination` property that will be given to `RantzMarketingGatewayImpl` through setter injection. When the `send()` method is called, `JmsTemplate` will deal with obtaining a JMS connection and session and will send the message on behalf of the sender (see figure 10.5).

As for the message itself, it is constructed using a `MessageCreator`, implemented here as an anonymous inner class. In `MessageCreator`'s `createMessage()` method, we simply assemble a JMS `MapMessage` and populate it with the motorist's name and email address. The `createMessage()` method is a callback method that `JmsTemplate` will use to construct the message that will be sent.

The `JmsTemplate` and the `Destination` are injected into `RantzMarketingGatewayImpl` using setter injection. Therefore, when we configure the `RantzMar-`



Figure 10.5 *JmsTemplate deals with the complexities of sending a message on behalf of the sender.*

MarketingGatewayImpl class in Spring, we must wire in references to the jmsTemplate and rantzDestination beans:

```

<bean id="marketingGateway"
      class="com.rodrantz.marketing.RantzMarketingGatewayImpl">
    <property name="jmsTemplate" ref="jmsTemplate" />
    <property name="destination" ref="rantzDestination" />
</bean>

```

And that's it! Notice that the sendMotoristInfo() method is focused entirely on assembling and sending a message. There's no connection or session management code—JmsTemplate handles all of that for us. And there's no need to catch JMSException—JmsTemplate will catch any JMSException that is thrown and then rethrow it as one of Spring's unchecked exceptions from table 10.1.

Setting a default destination

In listing 10.3, we explicitly specified a specific Destination that the motorist message would be sent to in the send() method. That form of the send() method comes in handy when we want to programmatically choose a destination. But in the case of RantzMarketingGatewayImpl, we will always be sending the motorist messages to the same destination, so the benefits of that form of send() aren't as clear.

Instead of explicitly specifying a destination each time we send a message, we could opt for wiring a default destination into JmsTemplate:

```

<bean id="jmsTemplate"
      class="org.springframework.jms.core.JmsTemplate">
    <property name="connectionFactory" ref="connectionFactory" />
    <property name="defaultDestination" ref="rantzDestination" />
</bean>

```

Now the call to JmsTemplate's send() method can be simplified slightly by removing the first parameter:

```

jmsTemplate.send(
    new MessageCreator() {
        ...
    });

```

This form of the `send()` method only takes a `MessageCreator`. With no destination specified, `JmsTemplate` will assume that you want the message sent to the default destination. Since `JmsTemplate` will always assume the correct destination, we no longer need to inject a destination into `RantzMarketingGatewayImpl`. Its declaration can be simplified to this:

```
<bean id="marketingGateway"
      class="com.roaddrantz.marketing.RantzMarketingGatewayImpl">
    <property name="jmsTemplate" ref="jmsTemplate" />
</bean>
```

Because we're no longer injecting the destination into `RantzMarketingGatewayImpl`, the `destination` property and its setter method can also be removed.

Consuming messages

Now you've seen how to send a message using `JmsTemplate`. But what if you're on the receiving end? Can `JmsTemplate` be used to receive messages?

Yes, it can. In fact, it's even easier to receive a message with `JmsTemplate`. All you need to do is call `JmsTemplate`'s `receive()` method, as shown in listing 10.4.

Listing 10.4 Receiving a message using `JmsTemplate`

```
package com.roaddrantz.marketing;
import javax.jms.JMSEException;
import javax.jms.MapMessage;
import org.springframework.jms.core.JmsTemplate;
import org.springframework.jms.support.JmsUtils;

public class MarketingReceiverGatewayImpl {
    public MarketingReceiverGatewayImpl() {}

    public SpammedMotorist receiveSpammedMotorist() {
        MapMessage message = (MapMessage) jmsTemplate.receive(); ← Receives message

        SpammedMotorist motorist = new SpammedMotorist();
        try {
            motorist.setFirstName(message.getString("firstName"));
            motorist.setLastName(message.getString("lastName"));
            motorist.setEmail(message.getString("email"));

        } catch (JMSEException e) {
            throw JmsUtils.convertJmsAccessException(e); ← Converts any JMSEException
        }
        return motorist;
    }

    //injected
    private JmsTemplate jmsTemplate;
    public void setJmsTemplate(JmsTemplate jmsTemplate) { ← Creates object from message
    }
}
```

```
        this.jmsTemplate = jmsTemplate;
    }
}
```

When the `receive()` method is called, `JmsTemplate` will once again piece together all of the parts needed to interact with the message broker—the JMS connection, session, and message consumer. Then it will call the `receive()` method on the message consumer on behalf of the receiving application, as illustrated in figure 10.6.

`JmsTemplate`'s `receive()` method is synchronous. By default, a call to `receive()` will block until a message appears on the destination—waiting forever, if necessary. To avoid an eternal wait for messages, you can specify a receive timeout by setting the `receiveTimeout` property when configuring the `JmsTemplate`. The following configuration configures `JmsTemplate` to time out on receives after one minute (60,000 milliseconds):

```
<bean id="jmsTemplate"
      class="org.springframework.jms.core.JmsTemplate">
    <property name="connectionFactory" ref="connectionFactory" />
    <property name="defaultDestination" ref="rantzDestination" />
    <property name="receiveTimeout" value="60000" />
</bean>
```

As used in listing 10.4, the `receive()` method receives a message from the default destination. If you'd prefer to specify a destination, you can do so by passing in the destination:

```
MapMessage message =
  (MapMessage) jmsTemplate.receive(destination);
```

Alternatively, you can specify a destination by name and let Spring's destination resolver automatically resolve the destination:

```
MapMessage message =
  (MapMessage) jmsTemplate.receive("rantz.marketing.queue");
```



Figure 10.6 Receiving messages with `JmsTemplate` is as simple as calling the `receive()` method. `JmsTemplate` takes care of the rest.

Synchronous receipt of messages is not the only option offered by Spring. We'll look at how Spring supports asynchronous receiving in section 10.3. But first, let's explore `JmsTemplate` a bit more by looking at how it can be configured to automatically convert messages to and from Java objects.

10.2.3 Converting messages

In listing 10.3, the message that is sent is constructed by the anonymous `MessageCreator` instance in its `createMessage()` method. The message is constructed by taking the properties of the `Motorist` object and placing them in a `MapMessage` object. Meanwhile, on the receiving end, `MarketingReceiverGatewayImpl`'s `receiveSpammedMotorist()` method pulls values out of the received message to populate a `SpammedMotorist` object.

For our simple example, it's not that big of a deal to place the message conversion code directly alongside the code that sends and receives the message. But if you find yourself sending and/or receiving the same message at multiple points in your application, you may want to avoid unnecessary duplication of the mapping code by consolidating it into a message converter.

Although it wouldn't be hard to extract the message conversion code into a utility class of your own design, you'd still need to explicitly invoke the utility class to do the conversion. Fortunately, Spring supports message conversion through its `MessageConverter` interface:

```
public interface MessageConverter {  
    public Message toMessage(Object object, Session session);  
    public Object fromMessage(Message message);  
}
```

The `MessageConverter` interface is very straightforward. It has only two methods that must be implemented. For sending messages, the `toMessage()` method converts an `Object` to a `Message`. On the receiving end, the `fromMessage()` method converts an incoming `Message` into an `Object`.

Because `MessageConverter` is an interface, we'll need to provide an implementation for our application. Listing 10.5 shows `MotoristMessageConverter`, an implementation of `MessageConverter` that transforms `Motorist` objects into messages and messages into `SpammedMotorist` objects.

Listing 10.5 A message converter that converts a Motorist to a message and a message to a SpammedMotorist

```
package com.roadrantz.marketing;
import javax.jms.JMSEException;
import javax.jms.MapMessage;
import javax.jms.Message;
import javax.jms.Session;
import org.springframework.jms.support.converter.
    ↗ MessageConversionException;
import org.springframework.jms.support.converter.MessageConverter;
import com.roadrantz.domain.Motorist;
import com.roadrantz.marketing.SpammedMotorist;

public class MotoristMessageConverter implements MessageConverter {
    public MotoristMessageConverter() {}

    public Object fromMessage(Message message)
        throws JMSEException, MessageConversionException {
        if(!(message instanceof MapMessage)) {
            throw new MessageConversionException(
                "Message isn't a MapMessage");
        }

        MapMessage mapMessage = (MapMessage) message;
        SpammedMotorist motorist = new SpammedMotorist();

        motorist.setFirstName(mapMessage.getString("firstName"));
        motorist.setLastName(mapMessage.getString("lastName"));
        motorist.setEmail(mapMessage.getString("email"));

        return motorist;
    }

    public Message toMessage(Object object, Session session)
        throws JMSEException, MessageConversionException {
        if(!(object instanceof Motorist)) {
            throw new MessageConversionException("Object isn't
                ↗ a Motorist");
        }

        Motorist motorist = (Motorist) object;
        MapMessage message = session.createMapMessage();
        message.setString("firstName", motorist.getFirstName());
        message.setString("lastName", motorist.getLastName());
        message.setString("email", motorist.getEmail());

        return message;
    }
}
```

Converts message
to SpammedMotorist

Converts
Motorist
to message

When the RoadRantz application needs to send a Motorist to the marketing system, the `toMessage()` method will be used to convert the `Motorist` object into a `MapMessage` object. In the marketing system, the `fromMessage()` method will convert the received message into a `SpammedMotorist` object that the marketing system will use to send special offers to the user.

So now when we send and receive messages, all we'll need to do is call the `toMessage()` and `fromMessage()` methods on the converter object, right? Well... not exactly.

Sending and receiving converted messages

While we could call the `toMessage()` method before sending a message and `fromMessage()` upon receipt of a message, Spring offers a better way.

Instead of explicitly calling the `toMessage()` method before sending a message, we can simply call the `convertAndSend()` method of `JmsTemplate`. That way, the `sendMotoristInfo()` method in `RantzMarketingGatewayImpl` becomes significantly simpler:

```
public void sendMotoristInfo(final Motorist motorist) {  
    jmsTemplate.convertAndSend(motorist);  
}
```

`JmsTemplate`'s `convertAndSend()` method automatically calls the `toMessage()` method before sending the message to the destination. As used here, the message will be sent to the `JmsTemplate`'s default destination (assuming one has been specified). But we could also select a specific destination when calling `convertAndSend()`:

```
jmsTemplate.convertAndSend(destination, motorist);
```

Optionally, we can specify the destination by name:

```
jmsTemplate.convertAndSend("rantz.marketing.queue", motorist);
```

On the receiving end, we won't need to call `fromMessage()` to convert the message returned from `JmsTemplate`'s `receive()`. Instead, we'll replace the call to `receive()` with a call to `receiveAndConvert()`:

```
public SpammedMotorist receiveSpammedMotorist() {  
    return (SpammedMotorist) jmsTemplate.receiveAndConvert();  
}
```

Again, unless otherwise specified, `receiveAndConvert()` receives messages from the default destination. But we can also choose a destination by passing it as a parameter to `receiveAndConvert()`:

```
return (SpammedMotorist) jmsTemplate.receiveAndConvert(destination);
```

or, using the destination's name:

```
return (SpammedMotorist)
    jmsTemplate.receiveAndConvert("rantz.marketing.queue");
```

There's but one small detail that we've left out. If `JmsTemplate`'s `convertAndSend()` and `receiveAndConvert()` methods use the message converter to convert the messages then how does `JmsTemplate` know about the message converter?

Wiring a message converter

For the message converter to work, we'll need to configure it as a `<bean>` in Spring. The following XML will handle that:

```
<bean id="motoristConverter"
      class="com.roaddrantz.marketing.MotoristMessageConverter" /
```

Finally, the `JmsTemplate` needs to know about the message converter. To accommodate that, we'll wire the `motoristConverter` bean into `JmsTemplate`'s `messageConverter` property:

```
<bean id="jmsTemplate"
      class="org.springframework.jms.core.JmsTemplate">
    <property name="connectionFactory" ref="connectionFactory" />
    <property name="defaultDestination" ref="rantzDestination" />
    <property name="messageConverter" ref="motoristConverter" />
</bean>
```

You've already seen many parallels between how `JdbcTemplate` and `JmsTemplate` work. But there's still one more parallel that you may be interested in. Let's look at how Spring provides support for building JMS gateways through its `JmsGatewaySupport` class.

10.2.4 Using Spring's gateway support classes for JMS

You may remember from chapter 5 that Spring makes working with `JdbcTemplate` a little easier by providing `JdbcDaoSupport`, a base class for writing JDBC-based DAOs. Similarly, Spring comes with `JmsGatewaySupport`, a base class for JMS gateway classes.

Thus far, we have explicitly created `jmsTemplate` and `destination` properties in `RantzMarketingGatewayImpl` to hold the `JmsTemplate` and `Destination`. Although this didn't add a lot of extra code, just imagine how many times those properties (and their setter methods) would be duplicated in an application that has several JMS gateways. To clean things up a bit, we could have written `RantzMarketingGatewayImpl` to subclass `JmsGatewaySupport` instead, as shown in listing 10.6.

Listing 10.6 A new version of RantzMarketingGatewayImpl rewritten to use JmsGatewaySupport

```

package com.roadrantz.marketing;
// imports omitted

public class RantzMarketingGatewayImpl
    extends JmsGatewaySupport
    implements RantzMarketingGateway {
    public RantzMarketingGatewayImpl() {}

    public void sendMotoristInfo(final Motorist motorist) {
        getJmsTemplate().send(
            "rantz.marketing.queue",           Sends message
            new MessageCreator() {           to queue
                public Message createMessage(Session session)
                    throws JMSException {
                    MapMessage message = session.createMapMessage();
                    message.setString("lastName", motorist.getLastName());
                    message.setString("firstName", motorist.getFirstName());
                    message.setString("email", motorist.getEmail());
                    return message;
                }
            });
    }
}

```

Sets message properties

Notice that the `jmsTemplate` property and the `setJmsTemplate()` method are gone. Instead of using an injected `jmsTemplate` property as before, this version of `RantzMarketingGatewayImpl` makes a call to `getJmsTemplate()` to receive the `JmsTemplate` that's managed by `JmsGatewaySupport`. Therefore, `jmsTemplate` and its setter method are no longer needed.¹

Where does `JmsGatewaySupport` get its `JmsTemplate`? It depends. You can inject a `JmsTemplate` directly into the `jmsTemplate` property, as we did with the original `RantzMarketingGatewayImpl`. Or, you can short-circuit the need for a `JmsTemplate` bean altogether by wiring the connection factory into the `connectionFactory` property:

```

<bean id="marketingGateway"
    class="com.roadrantz.marketing.RantzMarketingGatewayImpl">
    <property name="connectionFactory" ref="connectionFactory" />
</bean>

```

¹ As a matter of fact, the `setJmsTemplate()` method is final in `JmsGatewaySupport`. As a result it can't appear in any class that extends `JmsGatewaySupport`.

When configured this way, `JmsGatewaySupport` will automatically create a `JmsTemplate` object based on the injected connection factory. You no longer need to declare a `JmsTemplate` bean in Spring.

Before you get too excited about wiring a connection factory directly into the gateway class, you should be aware that there are a couple of shortcomings to this approach:

- You can only specify a default destination on a `JmsTemplate`. If `JmsGatewaySupport` creates its own `JmsTemplate`, you won't get a chance to specify a default destination. You'll need to always explicitly choose a destination when calling `send()` or `receive()`.
- You can only wire a message converter into a `JmsTemplate`. If `JmsGatewaySupport` creates its own `JmsTemplate`, you won't be able to use a message converter. You'll need to explicitly handle message conversion in your gateway code.

You've already seen how to receive messages using `JmsTemplate`. But as you saw, the `receive()` method blocks until a message is available. Coming up next, let's have a look at a new feature in Spring 2.0 that makes it possible to asynchronously receive messages.

10.3 Creating message-driven POJOs

During one summer in college, I had the great privilege of working in Yellowstone National Park. The job wasn't one of the high-profile jobs like park ranger or the guy who turns Old Faithful on and off.² Instead, I held a position in housekeeping at Old Faithful Inn, changing sheets, cleaning bathrooms, and vacuuming floors. Not glamorous, but at least I was working in one of the most beautiful places on earth.

Every day after work, I would head over to the local post office to see if I had any mail. I was away from home for several weeks, so it was nice to receive a letter or card from my friends back at school. I didn't have my own post box, so I'd walk up and ask the man sitting on the stool behind the counter if I had received any mail. That's when the wait would begin.

² Before I get emails, I'm quite aware that Old Faithful doesn't have a cut-off valve. Old Faithful is a geyser, a natural phenomenon where incredibly hot water shoots out of the ground periodically—without being turned on or off. The valve comment was a joke.

You see, the man behind the counter was approximately 195 years old. And like most people that age he had a difficult time getting around. He'd drag his keister off the stool, slowly scoot his feet across the floor, and then disappear behind a partition. After a few moments, he'd emerge, shuffle his way back to the counter, and lift himself back up onto the stool. Then he would look at me and say, "No mail today."

JmsTemplate's receive() method is a lot like that aged postal employee. When you call receive(), it goes away and looks for a message in the queue or topic and doesn't return until a message arrives or until the timeout has passed. Meanwhile, your application is sitting there doing nothing, waiting to see if there's a message. Wouldn't it be better if your application could go about its business and be notified when a message arrives?

One of the highlights of the EJB 2 specification was the inclusion of the message-driven bean (MDB). MDBs are EJBs that process messages asynchronously. In other words, MDBs react to messages in a JMS destination as events and respond to those events. This is in contrast to synchronous message receivers that block until a message is available.

MDBs were a bright spot in the EJB landscape. Even many of the most rabid detractors of EJB would concede that MDBs were an elegant way of handling messages. The only blemish to be found in EJB 2 MDBs was that they had to implement javax.ejb.MessageDrivenBean. In doing so, they also had to implement a few EJB lifecycle callback methods. Put simply, EJB 2 MDBs were very un-POJO.

With the EJB 3 specification, MDBs were cleaned up to have a slightly more POJO feel to them. No longer must you implement the MessageDrivenBean interface. Instead, you implement the more generic javax.jms.MessageListener interface and annotate MDBs with @MessageDriven.

Spring 2.0 addresses the need for asynchronous consumption of messages by providing its own form of message-driven bean that is quite similar to EJB 3's MDBs. In this section, you'll learn how Spring supports asynchronous message consumption using message-driven POJOs (we'll call them MDPs, for short).

10.3.1 Creating a message listener

For a moment, try to imagine a simpler world where the MarketingMdp doesn't have to implement the MessageDrivenBean interface. In such a happy place, the sky would be the brightest of blues, the birds would always whistle your favorite song, and you wouldn't have to implement the setMessageDrivenContext() method or the empty ejbRemove() method—all demands placed on an MDB

developer by the EJB 2 MDB programming model. In such a world, the MarketingMdp class might look a little like listing 10.7.

Listing 10.7 Simplifying the message-driven paradigm

```
package com.roadrantz.marketing;
import javax.jms.JMSException;
import javax.jms.MapMessage;
import javax.jms.Message;
import javax.jms.MessageListener;

public class MarketingMdp implements MessageListener {
    public void onMessage(Message message) { ← [Reacts to message]
        MapMessage mapMessage = (MapMessage) message;
        try {
            SpammedMotorist motorist = new SpammedMotorist();
            motorist.setFirstName(mapMessage.getString("firstName"));
            motorist.setLastName(mapMessage.getString("lastName"));
            motorist.setEmail(mapMessage.getString("email"));

            processMotoristInfo(motorist); ← [Converts message to SpammedMotorist]
        } catch (JMSException e) {
            // handle-somehow
        }
    }

    private void processMotoristInfo(SpammedMotorist motorist) {
        ...
    }
}
```

Although the color of the sky and training birds to sing are a bit out of scope for Spring, the dream world I just described is much closer to reality with the release of Spring 2.0. In fact, the MarketingMdp class in listing 10.7 is exactly how a Spring 2.0 message-driven POJO³ would be written to process motorist information messages in the RoadRantz marketing engine.

By itself, MarketingMdp doesn't do much. It has an `onMessage()` method ready to process messages, but until we configure it in Spring, it's just dead weight. So let's go ahead and configure MarketingMdp as a `<bean>` in the Spring application context:

³ Okay, okay... I know... it's still not a POJO because it implements the `MessageListener` interface. Nevertheless, many in the Spring community are satisfied to call this a message-driven POJO despite the dependence on `MessageListener`. If you're still not convinced then hang on... we'll look at pure-POJO MDPs in section 10.2.3.

```
<bean id="rantzMdp"
      class="com.roaddrantz.marketing.MarketingMdp" />
```

So far, this is nothing special. The most striking thing about listing 10.7 is that it looks very much like what an equivalent EJB 3 MDB might look like. The only real difference is that an EJB 3 MDB would also be annotated with @MessageDriven to indicate to the container that it is an MDB. In Spring, however, we'll indicate that this is an MDP by wiring it into a message listener container.

Containing message listeners

A message listener container is a special bean that watches a JMS destination, waiting for a message to arrive. Once a message arrives, it retrieves the message and then passes it on to a `MessageListener` implementation by calling the `onMessage()` method. Figure 10.7 illustrates this interaction.



Figure 10.7 `MessageListenerContainer` listens to a queue/topic. When a message arrives, it is forwarded to a `MessageListener`.

Because our `MarketingMdp` class implements the `MessageListener` interface, it seems that a message listener container is in order. Table 10.2 lists the message listener containers offered by Spring.

Table 10.2 Spring's message listener containers.

Container class (<code>org.springframework.jms.listener.*</code>)	What it does
<code>SimpleMessageListenerContainer</code>	This is the simplest message listener container. It works with a fixed number of JMS sessions and does not support transactions.
<code>DefaultMessageListenerContainer</code>	This message listener container builds upon <code>SimpleMessageListenerContainer</code> by adding support for transactions.
<code>serversession.ServerSessionMessageListenerContainer</code>	This is the most powerful of the message listener containers. Like <code>DefaultMessageListenerContainer</code> , it supports transactions. However it is unique in that it allows for dynamic management of JMS sessions.

As its name implies, `SimpleMessageListenerContainer` is the simplest of the message listener containers. It can be configured in Spring as follows:

```
<bean class="org.springframework.jms.listener.
    ↪ SimpleMessageListenerContainer">
    <property name="connectionFactory" ref="connectionFactory" />
    <property name="destination" ref="rantzDestination" />
    <property name="messageListener" ref="rantzMdp" />
</bean>
```

The `connectionFactory` and `destination` properties are wired with the same `connectionFactory` and `destination` properties that we wired into `JmsTemplate` in section 10.2. As for the `messageListener` property, we've wired it with a reference to our MDP implementation so that the `onMessage()` method will be invoked upon receipt of a message.

Working with transactional MDPs

`SimpleMessageListenerContainer` is great for basic messaging needs. But if you need messages to be received in a transaction, you'll want to look at `DefaultMessageListenerContainer` instead.

`DefaultMessageListenerContainer` is configured similarly to `SimpleMessageListenerContainer`, as you can see here:

```
<bean class="org.springframework.jms.listener.
    ↪ DefaultMessageListenerContainer">
    <property name="connectionFactory" ref="connectionFactory" />
    <property name="destination" ref="rantzDestination" />
    <property name="messageListener" ref="rantzMdp" />
    <property name="transactionManager" ref="jmsTransactionManager" />
</bean>
```

The only difference here is that `DefaultMessageListenerContainer`'s `transactionManager` property is wired with a reference to a transaction manager. If you need the MDP to participate in transactions along with other transactional elements (a data access object, for example), you'll want to wire a `JtaTransactionManager` into the `transactionManager` property. Review section 6.2.5 in chapter 6 for details on how to configure a `JtaTransactionManager`.

If your transactional needs are simpler, a `JmsTransactionManager` will do:

```
<bean id="jmsTransactionManager"
    class="org.springframework.jms.connection.
        ↪ JmsTransactionManager">
    <property name="connectionFactory" ref="connectionFactory" />
</bean>
```

It's worth mentioning that the `transactionManager` property of `DefaultMessageListenerContainer` is optional. If you don't inject a transaction manager into it, the MDP will not be transactional. Leaving out the transaction manager will effectively degrade `DefaultMessageListenerContainer` to be equivalent to `SimpleMessageListenerContainer`.

At this point you may be thinking that we're trying to pull a fast one on you. We keep referring to `MarketingMdp` as a message-driven POJO. But there's nothing very POJO about a class that is required to implement a `MessageListener` interface. It's certainly simpler than the EJB 2 MDB, but it's still not quite a POJO. If this discrepancy is keeping you up at night then read on as I show you how to create a MDP that is truly a POJO.

10.3.2 Writing pure-POJO MDPs

In `MarketingMdp`, the `onMessage()` method is really just plumbing code. It's only there to receive and translate the message. What's more, it's the `onMessage()` method and its defining `MessageListener` interface that keep `MarketingMdp` from truly being a POJO.

The real work occurs in the `processMotoristInfo()` method. If there were only some way we could bypass the `onMessage()` method and go straight to the `processMotoristInfo()` method, `MarketingMdp` would be much simpler. And, more importantly, `MarketingMdp` would be a honest-to-goodness message-driven *POJO*.

But the `MessageListener` interface and its `onMessage()` method serve a very important purpose. If the message listener container can count on its `messageListener` property being wired with an implementation of `MessageListener` then it knows that it only needs to call the `onMessage()` method when a message arrives.

Fortunately, Spring offers an alternative in `MessageListenerAdapter`. `MessageListenerAdapter` is a `MessageListener` that delegates to a bean and method of your choosing, as shown in figure 10.8.

Instead of wiring your own implementation of `MessageListener` into the message listener container, you can wire in a `MessageListenerAdapter`:

```
<bean class="org.springframework.jms.listener.  
      ↗ SimpleMessageListenerContainer">  
  <property name="connectionFactory" ref="connectionFactory" />  
  <property name="destination" ref="rantzDestination" />  
  <property name="messageListener" ref="purePojoMdp" />  
</bean>
```

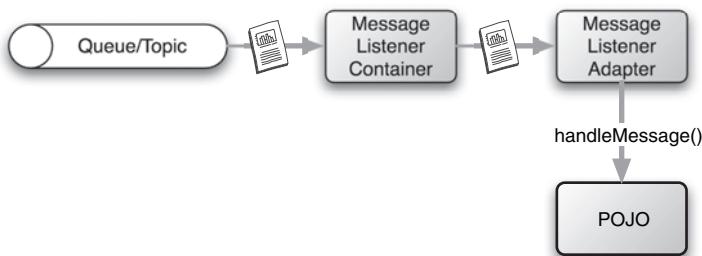


Figure 10.8 A `MessageListenerAdapter` plays the role of `MessageListener`. When it receives a message, it invokes a method on a `POJO`.

Here we've wired a reference to the `purePojoMdp` bean into the `messageListener` property. Otherwise, this message listener container is configured like any other message listener container. As for the `purePojoMdp` bean, it's a `MessageListenerAdapter`:

```

<bean id="purePojoMdp"
      class="org.springframework.jms.listener.adapter.
      <-- MessageListenerAdapter">
      <property name="delegate" ref="rantzMdp" />
      <property name="defaultListenerMethod"
              value="processMotoristInfo" />
</bean>
  
```

As you can see, we've configured the `MessageListenerAdapter` to call the `processMotoristInfo()` method on the `rantzMdp` bean when a message arrives on the destination. By default, `MessageListenerAdapter` calls a `handleMessage()` method when a message arrives. But we want our `MarketingMdp` bean to handle messages through its `processMotoristInfo()` method, so we've set `defaultListenerMethod` to `processMotoristInfo`.

Because we have chosen a specific method to be invoked, there's no need to implement `MessageListener` or the `onMessage()` method. As a result, `MarketingMdp` can now be simplified to look like listing 10.8.

Listing 10.8 A near-POJO implementation of `MarketingMdp.java`

```

package com.roadrantz.marketing;
import javax.jms.JMSException;
import javax.jms.MapMessage;

public class MarketingMdp {
    public MarketingMdp () {}
    public void processMotoristInfo(MapMessage message) { 
        <--> Specifies  
message arrives  
as Map
    }
}
  
```

```
try {  
    SpammedMotorist motorist = new SpammedMotorist();  
    motorist.setFirstName(message.getString("firstName"));  
    motorist.setLastName(message.getString("lastName"));  
    motorist.setEmail(message.getString("email"));  
    ...  
} catch (JMSEException e) {  
    // handle this-somehow  
}  
}
```

Processes
motorist info

How to contend
with this
exception?

This version of MarketingMdp is somewhat simpler than before. There's no longer an `onMessage()` method and `MarketingMdp` no longer needs to implement a `MessageListener` interface. When a message arrives in the destination, the `processMotoristInfo()` method will be called. Because the message is a `MapMessage`, the `processMotoristInfo()` method translated it into a `SpammedMotorist` object, which is then processed.

One thing about `MarketingMdp` that still feels wrong is that the `processMotoristInfo()` method is still called with a JMS-specific `MapMessage`. Although it's a POJO, the dependency on `MapMessage` unnecessarily couples `MarketingMdp` with JMS. What's more, `MapMessage`'s `getString()` method throws a `JMSException` that must be dealt with somehow. Ideally, `MarketingMdp` wouldn't depend on any framework-specific types.

When `MessageListenerAdapter` receives a message, it considers the message type and the value of the `defaultListenerMethod` and tries to find an appropriate listener method signature to invoke. Table 10.3 describes how `MessageListenerAdapter` maps a JMS message to listener method parameters.

Table 10.3 How JMS messages are mapped to MDP message parameters.

Message type	Method parameter
TextMessage	String or TextMessage
MapMessage	java.util.Map or MapMessage
BytesMessage	byte[] or BytesMessage
ObjectMessage	java.io.Serializable or ObjectMessage

Something to notice about table 10.3 is that the listener method can be written to take either the original JMS message or a simpler type. For example, the `processMotoristInfo()` method could be made even simpler by using a `java.util.Map`:

```
public void processMotoristInfo(Map map) {  
    SpammedMotorist motorist = new SpammedMotorist();  
    motorist.setFirstName((String) map.get("firstName"));  
    motorist.setLastName((String) map.get("lastName"));  
    motorist.setEmail((String) map.get("email"));  
    ...  
}
```

Now `MarketingMdp` is truly a POJO. It no longer has any dependency on any JMS type. And since the message arrives as a simple `java.util.Map`, there's no need to catch `JMSEException` when retrieving the messages' values. This is a lot better, but something still doesn't feel right.

Converting MDP messages

In the original `MarketingMdp` class, the `processMotoristInfo()` method took a `SpammedMotorist` object as a parameter. But in the latest version, it takes a `Map`, which it translates into a `SpammedMotorist` before processing. This means that the first few lines of `processMotoristInfo()` still include some plumbing code to do the translation into `SpammedMotorist`. Wouldn't it be great if `processMotoristInfo()` would be given a `SpammedMotorist` object, ready for processing, when a message arrives?

As you'll recall from earlier in this chapter (see section 10.2.3), Spring message converters can be used to translate messages to and from domain-specific Java types. And from listing 10.5, we already have a message converter that converts a `MapMessage` into a `SpammedMotorist` object. All we need to do is tell `MessageListenerAdapter` about the message converter. As it turns out, `MessageListenerAdapter` has a `messageConverter` property for that purpose:

```
<bean id="purePojoMdp"  
      class="org.springframework.jms.listener.adapter.  
            ↗ MessageListenerAdapter">  
    <property name="delegate" ref="rantzMdp" />  
    <property name="defaultListenerMethod"  
              value="processMotoristInfo" />  
    <property name="messageConverter" ref="motoristConverter" />  
  </bean>
```

We've wired the messageConverter property with a reference to the motorist-Converter bean (which is configured as a MotoristMessageConverter from listing 10.5). Now we can write the ultimate version of MarketingMdp, as shown in listing 10.9.

Listing 10.9 MarketingMdp, now a pure POJO (with no hint of JMS)

```
package com.roadrantz.marketing;

public class MarketingMdp {
    public MarketingMdp() {}

    public void processMotoristInfo (SpammedMotorist motorist) {
        ...
    }
}
```

Wow! MarketingMdp has come a long way from the original MessageListener version in listing 10.7. MarketingMdp started as a class purposed for processing JMS messages, but the final version has no hint of JMS. In fact, MarketingMdp doesn't even have to be used as an MDP. Because it's just a POJO, its processMotoristInfo() method could be called directly without a message broker involved. Keep this fact in mind, because we're not quite done with MarketingMdp. Soon you'll learn how we can export this same POJO as a remote service.

Now you've seen how Spring supports messaging through JMS abstraction. We've both sent and received messages without having to succumb to the complexities of the JMS API or resorting to using EJB MDBs. But there's still one more way to use messaging in Spring that may appeal to you if you're more comfortable with the RPC model. Before we leave the topic of messaging behind, let's look at how to make remote procedure calls using asynchronous messaging as a transport.

10.4 Using message-based RPC

In chapter 8, we explored several of Spring's options for making remote procedure calls on remote objects. Those were all fantastic options for communication between applications. But all of those options were synchronous. That is, the client would invoke a method on the server object and then wait for the server to respond. Moreover, if the server process weren't available, the request would immediately end with an exception being thrown.

As you've seen in this chapter, there are benefits to asynchronous communication through messaging. With asynchronous messaging, the message sender

doesn't have to wait for the receiver to finish processing the message before moving on. This has a positive impact on the performance of the "client" application. Also, message delivery is guaranteed—even if the receiver isn't available when the message is sent.

But there's also something appealing about the RPC model. The RPC programming model makes interacting with remote services as straightforward as invoking local object methods. If there were only some way to have the simplicity of the RPC programming model with the benefits of asynchronous messaging...

10.4.1 Introducing Lingo

Lingo is a Spring-based remoting option that bridges the gap between RPC and asynchronous messaging. As with other Spring remoting options, Lingo provides a service exporter to export bean functionality as a Lingo service and a client-side proxy to transparently wire a reference to a remote Lingo service on the calling side.

What makes Lingo different from the other remoting options we looked at in chapter 8 is in how it communicates. In all of Spring's other remoting options, the client communicates directly with the service via sockets. Consequently, the service must be available when the client makes a call or else the call will fail.

Lingo remoting, however, carries information over a JMS queue or topic. As such, if the service is unavailable when a call is made, the call will be held in the queue/topic until the service is available again. Also, if the client call is one-way (that is, there is no return value), the client call can immediately return without having to wait for the service to process the call.

Although Lingo is based on Spring remoting, it is not part of the Spring Framework. You can download Lingo from the Lingo homepage at <http://lingo.codehaus.org/Download>. Be sure to get the latest version. We're building our examples against version 1.3.

Alternatively, if you're using Maven 2 to build your project, you can add Lingo as a dependency in pom.xml with the following:

```
<dependency>
  <groupId>org.logicblaze.lingo</groupId>
  <artifactId>lingo</artifactId>
  <version>1.3</version>
  <scope>compile</scope>
</dependency>
```

Maven will take care of ensuring that Lingo is in the build and runtime classpath for you.

Now that you have a basic idea of what Lingo does, let's revisit the RoadRantz marketing system. This time we'll build the interaction between the main RoadRantz application and the marketing engine using Lingo for communication.

10.4.2 Exporting the service

When we last visited `MarketingMdp` in listing 10.9, it was a pure POJO. Sure, we were wiring it into a `MessageListenerAdapter` so that it could be used as a message-driven POJO. But there is nothing about the class in listing 10.9 that makes it message-driven. It's just a POJO and we can do any POJO kind of thing we want to with it.

As a POJO, it is suitable for export as a remote service using any of Spring's remote service exporters we discussed in chapter 8. However, instead of rehashing those conventional RPC exporters again, this time we're going to export `MarketingMdp` as a Lingo service.

On the service side, Lingo provides `JmsServiceExporter`, a service exporter not entirely unlike those in chapter 8. Instead of exporting a service that is available for direct RPC access, however, Lingo-exported services are made available through JMS, as shown in figure 10.9.

The following XML configures a `JmsServiceExporter` that exports the `rantz-Mdp` bean (which is a `MarketingMdp`) as an RPC-over-JMS service:

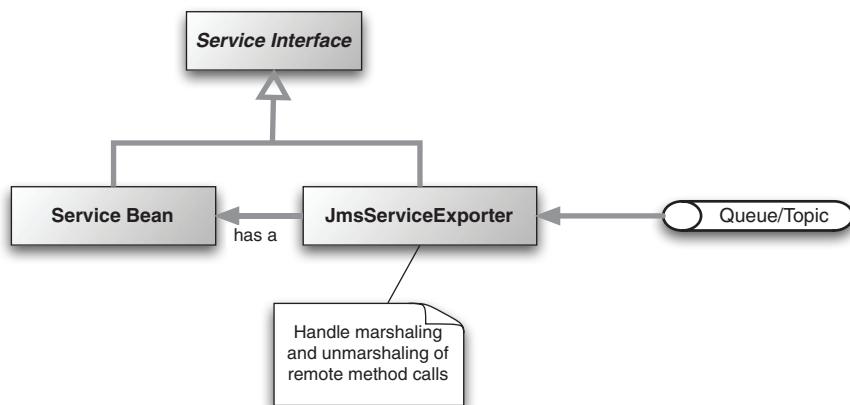


Figure 10.9 `JmsServiceExporter` exports a POJO as an RPC service that receives messages from a JMS destination.

```
<bean id="server"
      class="org.logicblaze.lingo.jms.JmsServiceExporter">
    <property name="connectionFactory" ref="connectionFactory" />
    <property name="destination" ref="destination" />
    <property name="service" ref="rantzMdp" />
    <property name="serviceInterface"
              value="com.roadrantz.marketing.MarketingService" />
</bean>
```

The first two properties wired in `JmsServiceExporter` are our old friends, the `connectionFactory` and `destination` properties. Unlike conventional RPC services, which typically use TCP/IP as the transport, a Lingo-exported service is available through a JMS destination (either a topic or a queue). Therefore, instead of configuring `JmsServiceExporter` with a URL or a port number, we will need to configure `JmsServiceExporter` with a JMS connection factory and a destination so that it will know where to export the service.

The `service` property is wired with a reference to the `rantzMdp` bean, which is our `MarketingMdp`. Finally, the `serviceInterface` property is configured with the class name of an interface that defines the exported service. We're declaring that our service will be exported with the `MarketingService` interface, which is defined here:

```
package com.roadrantz.marketing;
public interface MarketingService {
    void processMotoristInfo(SpammedMotorist motorist);
}
```

Because we defined the service with the `MarketingService` interface, this means that we'll need to make one small tweak to the `MarketingMdp` class. We'll need to change it to implement the `MarketingService` interface:

```
package com.roadrantz.marketing;

public class MarketingMdp implements MarketingService {
    public MarketingMdp () {}

    public void processMotoristInfo(SpammedMotorist motorist) {
        ...
    }
}
```

That's all there is to exporting a service using Lingo. Once the application is started, the `JmsServiceExporter` will kick in and we'll be ready to start using it. Now let's look at the client side to see how the RoadRantz application will make calls to the exported marketing service.

10.4.3 Proxying JMS

In the RoadRantz application, we're going to need to call the `processMotoristInfo()` method every time a user registers and elects to receive special offers from RoadRantz. Therefore, we'll have to wire a reference to the Lingo-exported service into the RoadRantz application somehow.

Wiring JmsProxyFactoryBean

Lingo provides `JmsProxyFactoryBean`, a proxy factory bean that produces proxies to remote Lingo-exported services. Conceptually, this is no different than the proxy factory beans we looked at in chapter 8. As you can see in figure 10.10, however, the service proxied by `JmsProxyFactoryBean` is available through a JMS destination (either a queue or a topic) instead of through TCP/IP.

The following `<bean>` declaration configures a `JmsProxyFactoryBean` that makes the marketing service exported in section 10.4.2 available on the client side:

```
<bean id="marketing"
      class="org.logicblaze.lingo.jms.JmsProxyFactoryBean">
    <property name="connectionFactory" ref="connectionFactory" />
    <property name="destination" ref="destination" />
    <property name="serviceInterface"
              value="com.roadrantz.marketing.MarketingService" />
</bean>
```

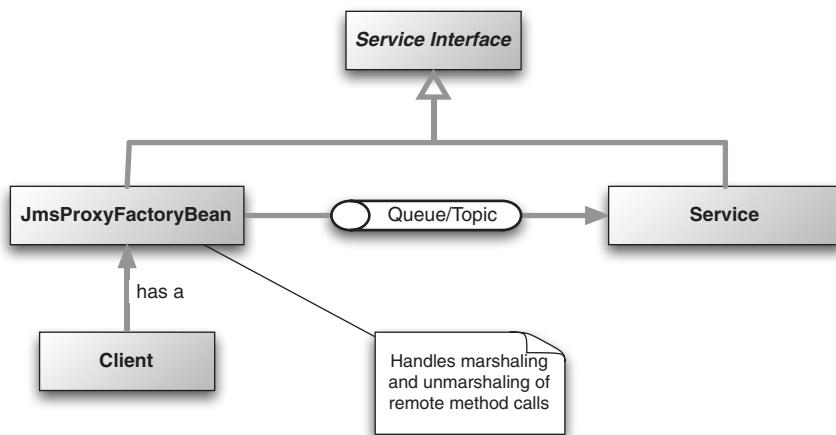


Figure 10.10 `JmsProxyFactoryBean` proxies a remote service that is listening on a JMS destination.

You've already been introduced to the `connectionFactory` and `destination` properties—they serve the same purpose here as they have throughout this chapter. And the `serviceInterface` property specifies the Java interface that the proxy will implement. It is through this interface that `RoadRantz` will invoke the `processMotoristInfo()` method.

The most significant thing to notice about how `JmsProxyFactoryBean` is configured is the conspicuous absence of anything telling where the service is located. Unlike the proxy factory beans in chapter 8, there's no IP address, host-name, or port number—no clue as to the service's whereabouts. That's because the service's location isn't important. You don't need to know where it lives—only where it picks up its mail.

In fact, there's nothing here indicating that there's only one instance of the remote service. If we wanted to set up the marketing service for high availability, we would only have to start up two or more instances, with all of them listening to the same destination. Each instance could potentially process a request. Meanwhile, the client has no idea that there's a pool of services waiting to respond to its request.

Making the call

With the `JmsProxyFactoryBean` wired, we're ready to start making calls to the remote service. All we need to do is wire it into `RantServiceImpl`:

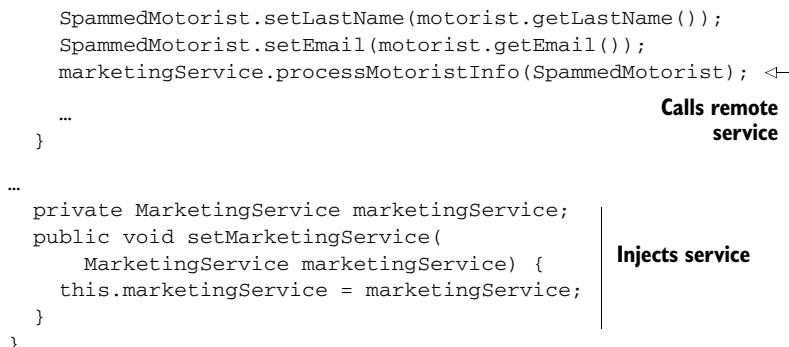
```
<bean id="rantService"
      class="com.roaddrantz.service.RantServiceImpl">
    <property name="rantDao" ref="rantDao" />
    <property name="marketingService" ref="marketing" />
</bean>
```

Then we can use it in the `addMotorist()` method to send a `SpammedMotorist` object to the marketing service. Listing 10.10 shows the pertinent changes to `RantServiceImpl` to call the remote marketing service.

Listing 10.10 Changing RantServiceImpl to send SpammedMotorists to the marketing service

```
public class RantServiceImpl implements RantService {
  ...
  public void addMotorist(Motorist motorist)
    throws MotoristAlreadyExistsException {
  ...
    SpammedMotorist SpammedMotorist = new SpammedMotorist();
    SpammedMotorist.setFirstName(motorist.getFirstName());
```

```
SpammedMotorist.setLastName(motorist.getLastName());
SpammedMotorist.setEmail(motorist.getEmail());
marketingService.processMotoristInfo(SpammedMotorist); <--  
    ...  
    }  
  
...  
private MarketingService marketingService;  
public void setMarketingService(  
    MarketingService marketingService) {  
    this.marketingService = marketingService;  
}  
}
```



Calls remote service

Injects service

As you can see, invoking a Lingo-exported service is no different from invoking an RMI service, a web service, or even a method on another bean in the same process. Nothing in listing 10.10 indicates that JMS is involved.

The only thing that is different is the Spring configuration. In this way, switching from JMS to another communication mechanism is a simple matter of changing the configuration. You could easily replace Lingo with one of the synchronous options from chapter 8. Or perhaps you could inject the `marketingService` property with a mock implementation of the `MarketingService` interface in the context of a unit test.

10.5 Summary

Asynchronous messaging presents several advantages over synchronous RPC. Indirect communication results in applications that are loosely coupled with respect to one another and thus reduces the impact of any one system going down. Additionally, because messages are forwarded to their recipients, there's no need for a sender to wait for a response. In many circumstances, this can be a boost to application performance.

Although JMS provides a standard API for all Java applications wishing to participate in asynchronous communication, it can be a little cumbersome to use. Spring eliminates the need for JMS boilerplate code and exception-handling code and makes asynchronous messaging easier to use.

Coming up in the next chapter, we'll watch worlds collide as we see how Spring supports the use and development of Enterprise JavaBeans.

Spring and Enterprise JavaBeans

This chapter covers

- Wiring EJBs into a Spring context
- Building Spring-aware EJBs
- Using EJB 3 annotations with Spring beans

Several years ago, a series of advertisements ran on television for Reese's peanut butter cups. In these advertisements, one character would be enjoying a chocolate bar while another would be enjoying some peanut butter. Ultimately, the characters would collide, accidentally mixing their snacks. One would proclaim, "You've got chocolate in my peanut butter!" while the other would declare, "You've got peanut butter on my chocolate!" After each would taste the mixture, they would decide it was "two great tastes that taste great together."

You may be surprised to find a section on how to use Spring with EJBs in this book. Much of this book so far has shown you how to implement enterprise-class applications without resorting to EJBs. Many developers liken Spring and EJB more to oil and water than to chocolate and peanut butter.

The fact is that although Spring provides a lot of functionality that gives POJOs the power of EJBs, you may not always have the luxury of working on a project that is completely EJB free. On the one hand, you may have to interface with other systems that expose their functionality through stateless session EJBs. On the other hand, you may be placed in a project where for legitimate technical (or perhaps political) reasons you must write EJB code.

Whether your application is the client of an EJB or you must write the EJB itself, you don't have to completely abandon all the benefits of Spring in order to work with EJBs. Spring offers three ways to mix the Spring and EJB to reap the benefits of both frameworks:

- If your application will be consuming the services of a session EJB, Spring enables you to declare EJBs as beans within the Spring application context. This makes it possible to wire references to EJB session beans (both EJB 2.x and EJB 3) into the properties of your other beans as though the EJB is just another POJO.
- If you're developing an EJB 2.x session bean, you can write your EJB to be Spring aware. That is, your EJB will have access to the Spring application context so that it can access and use beans that are configured in Spring.
- A Spring-related framework called Pitchfork makes it possible to use EJB 3 annotations to perform dependency injection and simple AOP on beans within the Spring context.

This chapter represents the collision of EJBs with Spring. Whether you want to write some Spring-flavored EJBs or mix EJB into your Spring application, there's something here for you. We'll explore all of the ways that Spring and EJB can be mixed, starting with wiring EJBs in a Spring application context.

11.1 Wiring EJBs in Spring

If you've ever written a client for a 2.x EJB, you are probably familiar with how you gain access to an EJB reference. First you would look up the EJB's home interface from JNDI using code that looks a little like this:

```
private TrafficServiceHome trafficServiceHome;
private TrafficServiceHome getTrafficServiceHome () throws javax.naming.NamingException {
    if(trafficServiceHome != null)
        return trafficServiceHome;
    javax.naming.InitialContext ctx =
        new javax.naming.InitialContext();
    try {
        Object objHome = ctx.lookup("trafficService");
        TrafficServiceHome home =
            (TrafficServiceHome) javax.rmi.PortableRemoteObject.narrow(
                objHome, TrafficServiceHome.class);
        trafficServiceHome = home;
        return home;
    } finally {
        ctx.close();
    }
}
```

Once you've got the reference to the home interface, you'll then need to get a reference to the EJB's business interface (either remote or local) so that you can call its methods. For example, the following code shows how you might call the `getTrafficConditions()` method on the traffic service EJB:

```
try {
    TrafficServiceHome home = getTrafficServiceHome();
    TrafficService trafficService = home.create();
    TrafficConditions conditions =
        trafficService.getTrafficConditions(city, state);
} catch (java.rmi.RemoteException e) {
    throw new TrafficException();
} catch (CreateException e) {
    throw new TrafficException();
}
```

Wow! That's a lot of code just to look up traffic conditions. What's most unsettling is that only a few lines have anything directly to do with retrieving the traffic conditions. Most of it is boilerplate EJB plumbing that is used just to retrieve a

reference to the EJB. That's an awful lot of work just to make a single call to the EJB's `getTrafficConditions()` method.

EJB 3 makes things a little bit easier. Instead of looking up the EJB's home interface from JNDI, you look up EJB 3 session beans directly from JNDI. But that still involves a lot of boilerplate JNDI lookup code.

Hold on. Throughout this book, you've seen ways to inject your application beans with the services they need. Beans don't look up other beans—beans are *given* to other beans. But this whole exercise of looking up an EJB via JNDI and its home interface doesn't seem to fit how the rest of the RoadRantz application is constructed. If we proceed to interact with EJB in the traditional EJB way, we'll end up muddying up everything with ugly lookup code and will definitely couple the code with the EJB. Isn't there a better way?

11.1.1 Proxied session beans (EJB 2.x)

As you've probably guessed from this lead-up, yes, there is a better way. In chapter 8 we showed you how to configure proxies to access various remote services, including services based in RMI, Hessian, Burlap, and Spring's own HTTP invoker. Spring offers much the same kind of proxy support for accessing EJBs.

Spring comes with two proxies suitable for accessing session EJBs:

- `LocalStatelessSessionProxyFactoryBean`—Used to access local EJBs (EJBs that are in the same container as their clients)
- `SimpleRemoteStatelessSessionProxyFactoryBean`—Used to access remote EJBs (EJBs that are in a separate container from their clients)

As illustrated in figure 11.1, these proxy factory beans produce proxies that handle the details of looking up an EJB's home interface and invoking the EJB's business methods. They make it possible to configure references to EJBs in the Spring application context that can be wired as if they were any other Spring-managed bean.

For illustration's sake, let's assume that the traffic service EJB is a local stateless session EJB. To wire a traffic service EJB in Spring, you would use `LocalStatelessSessionProxyFactoryBean` like this:

```
<bean id="trafficService"
    class="org.springframework.ejb.access.
        LocalStatelessSessionProxyFactoryBean"
    lazy-init="true">

    <property name="jndiName" value="ejb/TrafficService" />
    <property name="businessInterface"
        value="com.roaddrantz.ejb.TrafficServiceEjb" />
</bean>
```

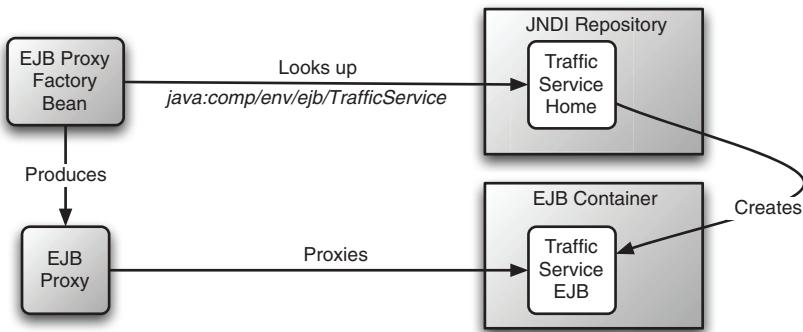


Figure 11.1 Spring’s EJB proxy factory beans look up a session bean’s home interface and then produce a proxy that delegates to the actual EJB.

The `jndiName` property, here set to `trafficService`, is used to identify the name of the EJB home interface in JNDI. Meanwhile the `businessInterface` property identifies the EJB’s business interface. The proxy will adhere to this interface.

Pay particular attention to the `lazy-init` attribute on the `<bean>` element. The Spring application context will normally pre-instantiate singleton beans once the Spring configuration file is loaded. This is usually a good thing, but it could cause problems with EJB proxies. That’s because the Spring application context may load and instantiate the proxy before the EJB’s home interface is bound in the naming service. By setting `lazy-init` to true, we are telling Spring to not look up the home interface until the `trafficService` bean is first used—which should be plenty of time for the EJB to be bound in the naming service.

Now let’s switch gears and see how we would configure the `trafficService` bean if the traffic service were a remote stateless session bean. Take a close look at the following XML:

```

<bean id="trafficService"
      class="org.springframework.ejb.access.
           SimpleRemoteStatelessSessionProxyFactoryBean"
      lazy-init="true">
    <property name="jndiName" value="trafficService" />
    <property name="businessInterface"
              value="com.roaddrantz.ejb.TrafficServiceEjb" />
</bean>
  
```

See the difference? The only thing that changed was the name of the proxy factory bean class. Nothing else needs to change. Spring makes the choice between local and remote EJBs almost transparent.

But you're probably wondering about `java.rmi.RemoteException`. How can the choice between local and remote EJBs be completely transparent if invoking a remote EJB method could throw a `RemoteException`? Doesn't someone need to catch that exception?

This illustrates one more benefit of using Spring's EJB support for accessing EJBs. As with RMI services, any `RemoteException` that may be thrown from EJBs are caught and then rethrown as `org.springframework.remoting.RemoteAccessException`. Since `RemoteAccessException` is an unchecked exception, catching it is optional for the EJB client.

Declaring EJB proxies with Spring's JEE namespace

Spring's proxy factory beans for EJB access greatly simplify EJB and make it possible to wire EJBs into Spring beans as if they were any other Spring bean. Life couldn't be much easier for EJB, could it?

Well, Spring 2 makes things even easier by supplying EJB configuration elements in the new JEE namespace. The JEE namespace includes two configuration elements specifically for EJB:

- `<jee:local-slsb>`—Configures a proxy to a local stateless session bean in the Spring application context
- `<jee:remote-slsb>`—Configures a proxy to a remote stateless session bean in the Spring application context

In order to use these two elements, you'll need to declare the JEE namespace in your Spring configuration by including the following in the `<beans>` element:

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jee="http://www.springframework.org/schema/jee"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/
            spring-beans-2.0.xsd
        http://www.springframework.org/schema/jee
        http://www.springframework.org/schema/jee/
            spring-jee-2.0.xsd">
```

With the JEE namespace declared, you're ready to use its elements to configure an EJB reference in Spring. For example, to configure a reference to a local stateless session EJB, you would use the `<jee:local-slsb>` as follows:

```
<jee:local-slsb id="trafficService"
    jndi-name="trafficService"
    business-interface="com.roadrantz.ejb.TrafficServiceEjb"/>
```

Under the covers, `<jee:local-slsb>` configures a `LocalStatelessSessionProxyFactoryBean` in the Spring context. The end result is the same, even though the amount of XML is less.

Similarly, a remote stateless session EJB can be wired using the `<jee:remote-slsb>` element:

```
<jee:remote-slsb id="trafficService"
    jndi-name="trafficService"
    business-interface="com.roadrantz.ejb.TrafficServiceEjb"/>
```

Just as `<jee:local-slsb>` is a shortcut for `LocalStatelessSessionProxyFactoryBean`, `<jee:remote-slsb>` is a shortcut for `SimpleRemoteStatelessSessionProxyFactoryBean`.

Declaring EJB 3 session beans in Spring

As I mentioned earlier, EJB 3 simplifies the session bean lookup process by eliminating the notion of a home interface. Instead of looking up a session bean through a home interface that was retrieved through JNDI, EJB 3 session beans are retrieved directly from JNDI.

But as you've already seen, JNDI lookup code is rather complex and is mostly boilerplate. Furthermore, looking up an EJB is in stark contrast to Spring's principle of dependency injection. In Spring, session beans should be injected, not retrieved.

Fortunately, Spring provides the ability to wire JNDI-stored objects just like any other bean in the application context. The trick involves using Spring's `JndiObjectFactoryBean`. The following snippet of XML from the Spring application context shows how you might declare an EJB 3 traffic service session bean:

```
<bean id="trafficService"
    class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiName" value="ejb/TrafficService" />
    <property name="resourceRef" value="true" />
</bean>
```

As a factory bean, `JndiObjectFactoryBean` produces a proxy to the real session bean (see figure 11.2), which it looks up from a JNDI repository using the `jndiName` property. The `resourceRef` property indicates that the EJB should be looked up as a Java resource, in effect prefixing the value of `jndiName` with `java:comp/env/` before performing the lookup.

Optionally, using Spring 2.0's JEE namespace, you can declare the EJB using the `<jee:jndi-lookup>`:

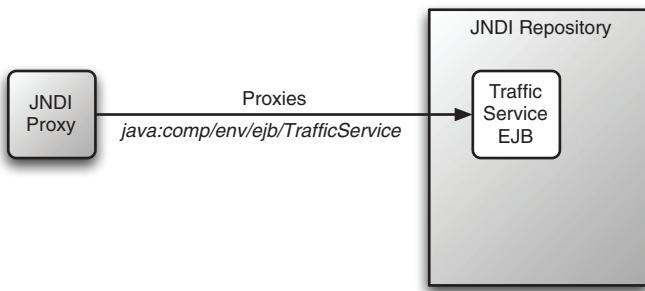


Figure 11.2
EJB 3 session beans can be configured in Spring using a JNDI proxy.

```
<jee:jndi-lookup id="trafficService"
    jndi-name="ejb/TrafficService"
    resource-ref="true" />
```

This snippet of XML is equivalent to the `<bean>` declaration above. Under the covers, `<jee:jndi-lookup>` creates a `JndiObjectFactoryBean`. We'll talk more about `JndiObjectFactoryBean` and `<jee:jndi-lookup>` in chapter 12. But for now just know that `JndiObjectFactoryBean` creates a proxy to the session bean that's stored in JNDI.

Now that the session bean proxy has been declared, it's time to put it to work. Let's see how to wire an EJB into a Spring-configured POJO.

11.1.2 Wiring EJBs into Spring beans

As it turns out, wiring an EJB into a Spring-configured POJO isn't any different than wiring any other POJO. For example, to wire the traffic service session bean into the `rantService` bean, you could use the following XML:

```
<bean id="rantService"
    class="com.roadrantz.service.RantServiceImpl">
    ...
    <property name="trafficService" ref="trafficService" />
    ...
</bean>
```

Did you see that? There is nothing EJB about that. The `trafficService` bean (which happens to be a proxy to an EJB) is simply injected into the `trafficService` property. There's no indication that EJB is involved at all. As illustrated in figure 11.3, proxied EJBs can be injected into other beans just like any other Spring-configured POJO.

The wonderful thing about using a proxy factory bean to access the traffic service EJB is that you don't have to write your own service locator or business delegate code. In fact, you don't have to write any JNDI code of any sort. Nor must you

deal with the EJB's home interface (or local home interface).

Furthermore, by hiding it all behind the `TrafficService` business interface, the `trafficService` bean isn't even aware that it's dealing with an EJB. As far as it knows, it's collaborating with just another POJO. This is significant because it means that you are free to swap out the EJB implementation of `TrafficService` with any other implementation (perhaps even a mock implementation that's used when unit testing `RantServiceImpl`).

Now that you've seen how to wire EJBs into a Spring application, let's look at how Spring supports EJB development.

11.2 Developing Spring-enabled EJBs (EJB 2.x)

Although Spring provides many capabilities that make it possible to implement enterprise applications without EJBs, you may still find yourself needing to develop your components as EJBs.

In chapter 8, you saw how Spring exporters can turn any Spring-configured POJO into a remote service. I hate to disappoint you, but unfortunately, Spring doesn't provide an `EjbServiceExporter` class that exports POJOs as EJBs. (But I do agree that such an exporter would be really cool.)

Nevertheless, Spring can make developing EJBs a little bit easier. Spring comes with four abstract support classes that bring regular EJBs into the world of Spring:

- `AbstractMessageDrivenBean`—Useful for developing message-driven beans that accept messages from sources other than JMS (as allowed by the EJB 2.1 specification)
- `AbstractJmsMessageDrivenBean`—Useful for developing message-driven beans that accept messages from JMS sources
- `AbstractStatefulSessionBean`—Useful for developing stateful session EJBs
- `AbstractStatelessSessionBean`—Useful for developing stateless session EJBs

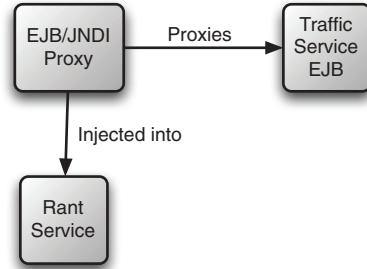


Figure 11.3 EJB and JNDI proxies can be injected into a Spring beans just as if they were any other Spring bean.

These abstract classes simplify EJB development in two ways:

- They provide default empty implementations of EJB lifecycle methods (e.g., `ejbActivate()`, `ejbPassivate()`, `ejbRemove()`). These methods are required per the EJB specification but are typically implemented as empty methods.
- They provide access to a Spring bean factory. This makes it possible for you to implement an EJB that delegates responsibility for the business logic to Spring-configured POJOs. Effectively, the EJB can be developed as an EJB façade to Spring-managed POJO functionality.

For example, suppose that you were to expose the functionality of the `RantService` bean as a stateless session EJB. Listing 11.1 shows how you might implement this EJB.

Listing 11.1 A stateless session EJB that delegates responsibility for business logic to a Spring-managed POJO

```
package com.roaddrantz.ejb;
import java.util.Date;
import java.util.List;
import javax.ejb.CreateException;
import org.springframework.ejb.support.AbstractStatelessSessionBean;
import com.roaddrantz.domain.Motorist;
import com.roaddrantz.domain.Rant;
import com.roaddrantz.domain.Vehicle;
import com.roaddrantz.service.MotoristAlreadyExistsException;
import com.roaddrantz.service.RantService;

public class RantServiceEjb
    extends AbstractStatelessSessionBean
    implements RantService {
    public RantServiceEjb() {}

    private RantService rantService;
    protected void onEjbCreate() throws CreateException {
        rantService = (RantService)
            getBeanFactory().getBean("rantService");
    }

    public void addMotorist(Motorist motorist)
        throws MotoristAlreadyExistsException {
        rantService.addMotorist(motorist);
    }

    public void addRant(Rant rant) {
        rantService.addRant(rant);
    }
}
```

```
public List<Rant> getRantsForDay(Date date) {  
    return rantService.getRantsForDay(date);  
}  
  
public List<Rant> getRantsForVehicle(Vehicle vehicle) {  
    return rantService.getRantsForVehicle(vehicle);  
}  
  
public List<Rant> getRecentRants() {  
    return rantService.getRecentRants();  
}  
  
public void sendDailyRantEmails() {  
    rantService.sendDailyRantEmails();  
}  
  
public void sendEmailForVehicle(Vehicle vehicle) {  
    rantService.sendEmailForVehicle(vehicle);  
}  
}
```

↑
**Delegates
to POJO**

When the `RantServiceEjb` is created, its `onEjbCreate()` method retrieves the `rantService` bean from the Spring bean factory. Then, when any of its methods are invoked, they delegate responsibility to the `rantService` bean, as illustrated in figure 11.4.

The big unanswered question regarding the EJB in listing 11.1 is where the bean factory comes from. In typical JEE fashion, the abstract EJB classes retrieve the bean factory from JNDI. By default, they expect to find the Spring bean factory in JNDI with the name `java:comp/env/ejb/BeanFactoryPath`. This means that you'll need to configure the bean factory in JNDI at that name.

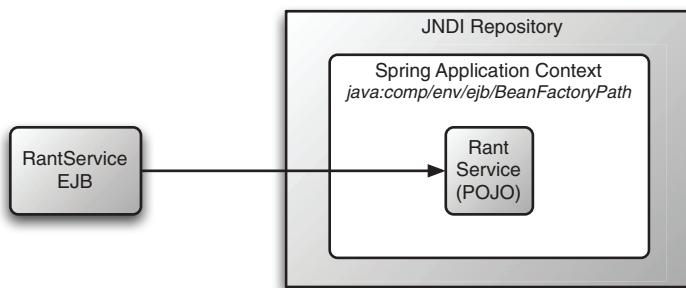


Figure 11.4 Spring's EJB support classes enable development of EJBs that have access to a Spring application context (stored in JNDI).

If you'd rather configure the bean factory under a different JNDI name, set the `beanFactoryLocatorKey` property before the bean factory is loaded (in either the constructor or in the `setSessionContext()` method). For example:

```
public void setSessionContext(SessionContext sessionContext) {  
    super.setSessionContext(sessionContext);  
  
    setBeanFactoryLocatorKey("java:comp/env/ejb/SpringContext");  
}
```

With this `setSessionContext()` method in place, the Spring context will be located using `java:comp/env/ejb/SpringContext` instead of the default JNDI name.

Spring's support for developing EJBs is focused on the EJB 2.x specifications. However, the EJB 3 specification changed things dramatically. EJB 3 borrows several ideas from Spring, such as dependency injection and AOP, to make EJB development much simpler than in previous specifications. Let's have a look at EJB 3 and how it fits into Spring.

11.3 Spring and EJB3

Although EJBs have enjoyed a great amount of popularity among Java developers since their debut, they also suffer from several complexities, including:

- Retrieving access to an EJB involves complex JNDI code to look up the EJB's home (or local home) interface, which is then used to create the EJB's business interface.
- With EJB 2.x, EJBs had to implement special interfaces that mandated that certain lifecycle callback methods be implemented. Because most applications have no use for these methods, they are often implemented as empty methods.
- The method signatures of remote EJBs are required to throw `java.rmi.RemoteException`, even when the method implementation does not actually throw the exception.

These and other problems have caused many developers to lose interest in EJBs and to start looking for simpler alternatives such as Spring. Reacting to the backlash against EJBs, the Java Community Process revisited the EJB specification, producing the most significant change in EJBs since their initial introduction: the EJB 3 specification.

The EJB 3 specification addresses the concerns of its heavyweight predecessor by supporting dependency injection of EJBs and resources instead of complex JNDI lookups. However, EJB 3 encourages the use of Java 5 annotations for declaring dependencies that are to be injected into bean properties.

Moreover, EJB 3 doesn't require that EJBs implement any specific interface or implement needless lifecycle methods. And remote methods no longer need to be declared to throw `RemoteException`. In short, the EJB 3 programming model is a POJO-based model.

Spring doesn't provide any direct support for the EJB 3 specification. However, there is a Spring add-on that makes it possible to use EJB 3 annotations to perform dependency injection and AOP in Spring.

11.3.1 Introducing Pitchfork

Pitchfork is an add-on for Spring that supports EJB 3 annotations. It is co-developed by Interface 21 (the Spring team) and BEA and is used within BEA's WebLogic Server 10 to support EJB 3. But you don't have to use WebLogic to use Pitchfork. Pitchfork is open sourced under the Apache 2.0 license and can be used in any Spring 2.0 application. You can download Pitchfork from Interface 21's site at <http://www.springframework.com/pitchfork>.

Pitchfork is not intended to be a complete implementation of the EJB 3 specification. However, it does support dependency injection and AOP through EJB 3 annotations, including the annotations listed in table 11.1.

In this section, you'll see how to use a few of these annotations within a Spring context using Pitchfork. We will assume, however, that you already have some understanding of EJB 3 and these annotations. For a more detailed discussion of EJB 3, we suggest that you have a look at *EJB 3 in Action* (Manning, 2006).

Table 11.1 EJB 3 annotations supported by Pitchfork.

Annotation	What it does
<code>@ApplicationException</code>	Declares an exception to be an application exception, which, by default, does not roll back a transaction
<code>@AroundInvoke</code>	Declares a method to be an interceptor method
<code>@EJB</code>	Declares a dependency to an EJB
<code>@ExcludeClassInterceptors</code>	Declares that a method should not be intercepted by a class interceptor

Table 11.1 EJB 3 annotations supported by Pitchfork. (continued)

Annotation	What it does
@ExcludeDefaultInterceptors	Declares that a method should not be intercepted by a default interceptor
@Interceptors	Specifies one or more interceptors classes to associate with a bean class or method
@PostConstruct	Specifies a method to be executed after a bean is constructed and all dependency injection is done to perform initialization
@PreDestroy	Specifies a method to be executed prior to bean being removed from the container
@Resource	Declares a dependency to an external resource
@Stateless	Declares a bean to be a stateless session bean
@TransactionAttribute	Specifies that a method should be invoked within a transaction context

11.3.2 Getting started with Pitchfork

Pitchfork uses a bean factory postprocessor to perform dependency injection on beans that are annotated with EJB 3 annotations. Pitchfork comes with two bean factory postprocessors to choose from:

- `org.springframework.jee.config.JeeBeanFactoryPostProcessor`
- `org.springframework.jee.ejb.config.JeeEjbBeanFactoryPostProcessor`

For the most part, these two bean factory postprocessors are identical. Both support all of the annotations in table 11.1, except for the `@EJB` annotation, which is only supported by `JeeEjbBeanFactoryPostProcessor`. If you want to use the `@EJB` annotation, be sure to choose `JeeEjbBeanFactoryPostProcessor`. Otherwise, the choice is arbitrary.

To configure one of these bean factory postprocessors in Spring, simply add it as a bean in the Spring application context. For example, to use `JeeBeanFactoryPostProcessor`, add the following to your Spring configuration:

```
<bean class="org.springframework.jee.config.  
       ↵ JeeBeanFactoryPostProcessor" />
```

With `JeeBeanFactoryPostProcessor` in place, we're now ready to start using the EJB 3 annotations. Next I'll show you how to apply these annotations in Spring-managed beans.

11.3.3 Injecting resources by annotation

To illustrate the use of EJB annotations in Pitchfork, we're going to revisit the knight example from chapter 1. Imagine that we were to rewrite the `KnightOfTheRoundTable` class from chapter 1 to use the `@Resource` attribute for dependency injection. It might look a little like listing 11.2.

Listing 11.2 An annotation-injected KnightOfTheRoundTable

```
package com.springinaction.knight;
import javax.annotation.Resource;

public class KnightOfTheRoundTable implements Knight {
    @Resource(name = "quest")    ← Injects
    private Quest quest;          | quest
    public String name;
    public KnightOfTheRoundTable(String name) {
        this.name = name;
    }
    public String getName() {
        return name;
    }
    public void embarkOnQuest() {
        quest.embark();    ← Uses injected
    }                                | quest
}
```

This `KnightOfTheRoundTable` is then configured in Spring using the following XML:

```
<bean id="knight" class=
    ↗ "com.springinaction.knight.KnightOfTheRoundTable">
    <constructor-arg value="Bedivere" />
</bean>
```

In chapter 1, we injected the knight's `quest` property using XML in the Spring configuration. But here we're letting the `@Resource` annotation do all of the work. `@Resource` will try to find an object named `quest` and, if it's found, wire it into the

quest property. Notice that there wasn't a need for a `setQuest()` method—`@Resource` can inject directly into private properties!

But where does the quest object come from? Well, that depends. Pitchfork will first look in JNDI for an object named `quest` and if it finds it, that object will be wired into the `quest` property. If JNDI turns up nothing, it will look in the Spring application context for a bean named `quest`.

Therefore, you'll either need to make sure that a `Quest` implementation is available through JNDI or you'll need to declare a Spring bean named `quest`:

```
<bean id="quest"
      class="com.springinaction.knight.SlayDragonQuest" />
```

That demonstrates Pitchfork's capability to do EJB 3 dependency injection. Now let's see how Pitchfork provides for EJB 3 AOP using interceptors.

11.3.4 Declaring interceptors using annotations

In addition to dependency injection, Pitchfork also supports EJB 3 interceptor annotations. EJB 3 interceptors are a simple form of AOP around advice that can be applied using annotations.

For example, the `Minstrel` advisor from chapter 1 could be rewritten as in listing 11.3 to use the `@AroundInvoke` annotation.

Listing 11.3 An EJB-annotated minstrel interceptor

```
package com.springinaction.knight;
// imports omitted
public class Minstrel {
    @AroundInvoke
    public Object singAboutQuest(InvocationContext ctx)
        throws Exception {
        Knight knight = (Knight) ctx.getTarget();
        Logger song =
            Logger.getLogger(knight.getClass());
        Method method = ctx.getMethod();
        song.debug("Brave " + knight.getName() +
            " did " + method.getName());
        Object rtn = ctx.proceed(); // Proceeds to target method
        return rtn;
    }
}
```

The code diagram highlights two annotations:

- `@AroundInvoke`: Annotates the `singAboutQuest` method, with a callout pointing to it labeled "Declares interceptor method".
- `ctx.proceed()`: A call to the proceed method of the `InvocationContext`, with a callout pointing to it labeled "Proceeds to target method".

The `@AroundInvoke` method declares a method that will be invoked when an advised method is intercepted. In this case, the `singAboutQuest()` method will be called before a target method is called so that the Minstrel can sing about the knight's exploits.

By itself the `@AroundInvoke` annotation only defines an interceptor method. We still need to apply it to the `KnightOfTheRoundTable` class. That's what the `@Interceptors` annotation is for:

```
@Interceptors({Minstrel.class})
public class KnightOfTheRoundTable implements Knight {
    ...
}
```

The `@Interceptors` annotation takes an array of one or more interceptor classes (e.g., classes that have methods that are annotated with `@AroundInvoke`). When a class is annotated with `@Interceptors`, all methods of the class are intercepted by the interceptors listed. Since `KnightOfTheRoundTable` only has an `embarkOnQuest()` method, that will be the only method intercepted. However, if you want finer control over which methods are intercepted and which are not, you may want to place the `@Interceptors` annotation at the method level:

```
@Interceptors({Minstrel.class})
public void embarkOnQuest() {
    quest.embark();
}
```

When used at the method level, only those methods that are annotated will be intercepted by the interceptors.

Another option for limiting classwide interception is to use the `@ExcludeClassInterceptors` annotation. When applied to a method, `@ExcludeClassInterceptors` will prevent class interceptors from intercepting the method. For example, to prevent `embarkOnQuest()` from being intercepted, annotate it like this:

```
@ExcludeClassInterceptors
public void embarkOnQuest() {
    quest.embark();
}
```

Pitchfork represents a choice for Spring developers. You can either use conventional Spring dependency injection and AOP, or you can use EJB 3 annotations for dependency injection and AOP. In virtually all circumstances, however, the pure Spring option is likely the best choice. Spring AOP, for instance, is far more flexible than EJB 3's interceptors are. Nevertheless, with Pitchfork, the choice is yours to make.

11.4 Summary

Although Spring's POJO-based development model offers a compelling alternative to Enterprise JavaBeans, there may be factors (either technical, political, or historical) that force a project to choose EJBs. In those cases, there's no need to dismiss Spring entirely, as Spring supports both EJB development and consumption.

In this chapter, you've seen how Spring and EJB can work together, starting with how Spring beans can be made into clients of EJBs. Using EJB proxies, we declared references to EJBs in a Spring application context. Once configured in Spring, the EJB could then be wired into other Spring beans that will consume the EJB's services. We also looked at how EJB proxy declaration is made simpler using Spring 2.0's `<jee:local-slsb>` and `<jee:remote-slsb>` configuration elements.

We then turned our attention to developing EJBs. Even though Spring doesn't provide a mechanism for directly hosting 2.x EJBs in the Spring container, Spring does provide a set of Spring-aware base classes from which EJBs can be developed. These base classes expose the Spring application context to the EJB so that the EJB can delegate its work to Spring-managed POJOs.

Finally, we peeked at Pitchfork, an intriguing Spring add-on that enables the use of EJB 3 annotations for dependency injection and AOP within a Spring container.

With the Spring-EJB story behind us, we now turn to look at a hodgepodge of other enterprise features available in Spring. In the next chapter, we'll explore Spring's support for JNDI, sending email, scheduling, and JMX.

12

Accessing enterprise services

This chapter covers

- Wiring JNDI resources in Spring
- Sending email messages
- Scheduling tasks
- Exporting and using MBeans

Contrary to what you may have heard, Spring's manifesto does not include a wholesale ousting of JEE technologies. Spring recognizes that the JEE specification is a collection of several effective subspecifications that still have their place in enterprise Java development. However, JEE does have a few rough edges that encourage less-than-ideal programming practices. Therefore, rather than subvert and replace JEE, Spring aims to complement JEE with a set of abstractions that round off the rough edges.

As it stands, a motorist can log into the RoadRantz application and view any rants that have been posted against their vehicles. That's great, but it requires the motorist to play an active role, logging in to check for new rants. More often than not, there won't be any new rants posted for them (unless they're an exceptionally bad driver). Suppose that instead of making the motorist come to RoadRantz to check for rants, we take RoadRantz to the motorist whenever they receive new rants.

In this chapter, we're going to add an email feature to RoadRantz to alert motorists that they have new rants. Along the way, we'll explore some of Spring's useful abstraction APIs, including Spring's support for Java Naming and Directory Interface (JNDI), JavaMail, scheduling, and Java Management Extensions (JMX).

To get started, let's see how to wire JNDI-managed objects into a Spring application context.

12.1 **Wiring objects from JNDI**

Throughout this book, you've seen how to use Spring to configure and wire your application objects. But what if you need to wire in objects that aren't configured in Spring? What if you need to wire in objects that are stored in JNDI?

What? Objects that aren't configured in Spring? How can that be? After all, this is a book about Spring. Why would I even *suggest* that some objects wouldn't be configured in Spring?

Before you chastise me for putting forward such deplorable thoughts, consider the following `dataSource` bean (taken from chapter 5):

```
<bean id="dataSource"
      class="org.springframework.jdbc.datasource.
          ➔ DriverManagerDataSource">
    <property name="driverClassName"
        value="org.hsqldb.jdbcDriver" />
    <property name="url"
        value="jdbc:hsqldb:hsq1://localhost/roaddrantz/roaddrantz" />
```

```

<property name="username" value="sa" />
<property name="password" value="" />
</bean>

```

This bean configures a JDBC DataSource in Spring, perfectly suitable for wiring into a JdbcTemplate, HibernateTemplate, or some other data access object. It will certainly work anywhere access to the RoadRantz database is required, but it presents a couple of problems:

- All of the database information is configured directly in the Spring application context. For purposes of change control and security, system administrators may prefer to configure and control the data source when the application is deployed, rather than allowing the developers to configure it themselves in the Spring context.
- Changing the database connection information can be inconvenient. Should the database connection need to change (perhaps the database is moved to a different server or the password must be changed), the application will likely need to be rebuilt and redeployed.

To address these concerns, you can configure the data source external to Spring, perhaps in a JNDI-accessible directory.

JNDI is a Java API that enables lookup of objects by name in a directory (often, but not necessarily, an LDAP directory). JNDI provides Java applications with access to a central repository for storing and retrieving application objects. JNDI is typically used in JEE applications to store and retrieve JDBC data sources and JTA transaction managers.

But if some of our application objects are configured in JNDI, external to Spring, how can we inject them into the Spring-managed objects that need them?

In this section, we'll be looking at how Spring supports JNDI by providing a simplified abstraction layer above the standard JNDI API. Spring's JNDI abstraction makes it possible to declare JNDI lookup information in your Spring context definition file. Then you can wire a JNDI-managed object into the properties of other Spring beans as though the JNDI object were just another POJO.

To gain a deeper appreciation of what Spring's JNDI abstraction provides, let's start by looking at how to look up an object from JNDI without Spring.

12.1.1 Working with conventional JNDI

Looking up objects in JNDI can be a tedious chore. For example, suppose you need to perform the very common task of retrieving a javax.sql.DataSource

from JNDI. Using the conventional JNDI APIs, your might write some code that looks like this:

```
InitialContext ctx = null;
try {
    ctx = new InitialContext();

    DataSource ds =
        (DataSource)ctx.lookup("java:comp/env/jdbc/RantzDatasource");
} catch (NamingException ne) {
    // handle naming exception
    ...
} finally {
    if(ctx != null) {
        try {
            ctx.close();
        } catch (NamingException ne) {}
    }
}
```

If you've ever written JNDI lookup code before, you're probably very familiar with what's going on in this code snippet. You may have written a similar incantation dozens of times before to raise an object out of JNDI. Before you repeat it again, however, take a closer look at what is going on:

- You must create and close an initial context for no other reason than to look up a `DataSource`. This may not seem like a lot of extra code, but it is extra plumbing code that is not directly in line with the goal of retrieving a data source.
- You must catch or, at the very least, rethrow a `javax.naming.NamingException`. If you choose to catch it, you must deal with it appropriately. If you choose to rethrow it, the calling code will be forced to deal with it. Ultimately, someone somewhere will have to deal with the exception.
- Your code is tightly coupled with a JNDI lookup. All your code needs is a `DataSource`. It doesn't matter whether it comes from JNDI. But if your code contains code like that shown earlier, you're stuck retrieving the `DataSource` from JNDI.
- Your code is tightly coupled with a specific JNDI name—in this case `java:comp/env/jdbc/RantzDatasource`. Sure, you could extract that name into a properties file, but then you'll have to add even more plumbing code to look up the JNDI name from the properties file.

Upon closer inspection we find that most of the code is boilerplate JNDI lookup that looks pretty much the same for all JNDI lookups. The actual JNDI lookup happens in just one line:

```
DataSource ds =
    (DataSource)ctx.lookup("java:comp/env/jdbc/RantzDatasource");
```

Even more disquieting than boilerplate JNDI code is the fact that the application knows where the data source comes from. It is coded to *always* retrieve a data source from JNDI. As illustrated in figure 12.1, the DAO that uses the data source will be coupled to JNDI. This makes it almost impossible to use this code in a setting where JNDI isn't available or desirable.

For instance, imagine that the data source lookup code is embedded in a class that is being unit tested. In an ideal unit test, we're testing an object in isolation without any direct dependence on specific objects. Although the class is decoupled from the data source through JNDI, it is coupled to JNDI itself. Therefore, our unit test has a direct dependence on JNDI and a JNDI server must be available for the unit test to run.

Regardless, this doesn't change the fact that sometimes you need to be able to look up objects in JNDI. DataSources are often configured in an application server to take advantage of the application server's connection pooling and then retrieved by the application code to access the database. How can your code get an object from JNDI without being dependent on JNDI?

The answer is found in dependency injection (DI). Instead of asking for a data source from JNDI, you should write your code to accept a data source from anywhere. That is, your code should have a `DataSource` property that is injected either through a setter method or through a constructor. Where the object comes from is of no concern to the class that needs it.

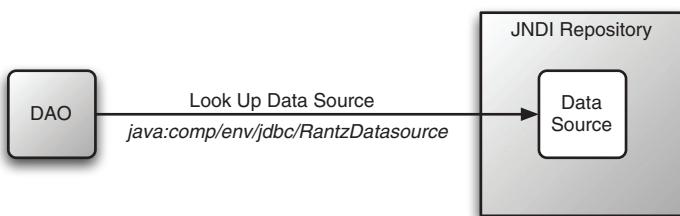


Figure 12.1 Using conventional JNDI to get dependencies means that an object is coupled to JNDI, making it difficult to use the object anywhere that JNDI is not available.

The data source object still lives in JNDI, however. So how can we configure Spring to inject an object that is stored in JNDI?

12.1.2 Injecting JNDI objects

Spring's `JndiObjectFactoryBean` gives you the best of both JNDI and DI. It is a factory bean, which means that when it is wired into a property, it will actually create some other type of object that will wire into that property. In the case of `JndiObjectFactoryBean`, it will wire an object retrieved from JNDI.

To illustrate how this works, let's revisit an example from chapter 5 (section 5.2.1). There I used `JndiObjectFactoryBean` to retrieve a `DataSource` from JNDI:

```
<bean id="dataSource"
      class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiName" value="jdbc/RantzDatasource" />
</bean>
```

The `jndiName` property specifies the name of the object in JNDI. By default, the name is used as is to look up the object in JNDI. But if the lookup is occurring in a JEE container then a `java:comp/env/` prefix needs to be added. You can manually add the prefix to the value specified in `jndiName`. Or you can set the `resourceRef` property to true to have `JndiObjectFactoryBean` automatically prepend `jndiName` with `java:comp/env/`:

```
<bean id="dataSource"
      class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiName" value="jdbc/RantzDatasource" />
    <property name="resourceRef" value="true" />
</bean>
```

With the `dataSource` bean declared, you may now inject it into a `dataSource` property. For instance, you may use it to configure a Hibernate session factory as follows:

```
<bean id="sessionFactory" class="org.springframework.orm.
  ↗ hibernate.LocalSessionFactoryBean">
  <property name="dataSource" ref="dataSource" />
  ...
</bean>
```

As shown in figure 12.2, when Spring wires the `sessionFactory` bean, it will inject the `DataSource` object retrieved from JNDI into the session factory's `dataSource` property.

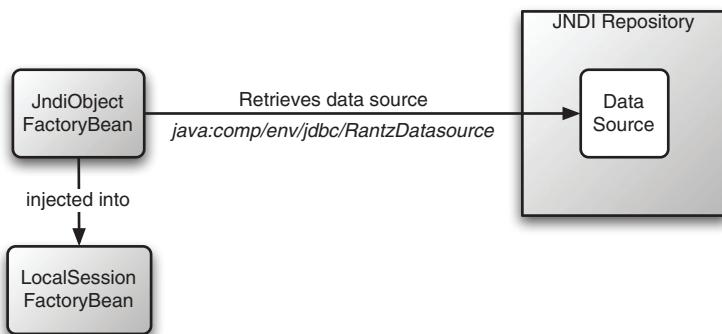


Figure 12.2 `JndiObjectFactoryBean` looks up an object from JNDI on behalf of a Spring object that it is wired into.

The great thing about using `JndiObjectFactoryBean` to look up an object in JNDI is that the only part of the code that knows that the `DataSource` is retrieved from JNDI is the XML declaration of the `dataSource` bean. The `sessionFactory` bean doesn't know (or care) where the `DataSource` came from. This means that if you decide that you would rather get your `DataSource` from a JDBC driver manager, all you need to do is redefine the `dataSource` bean to be a `DriverManagerDataSource`.

Now our data source is retrieved from JNDI and then injected into the session factory. No more explicit JNDI lookup code! Whenever we need it, the data source is always handy in the Spring application context as the `dataSource` bean.

As you have seen, wiring a JNDI-managed bean in Spring is fairly simple. Now let's explore a few ways that we can influence when and how the object is retrieved from JNDI, starting with caching.

Caching JNDI objects

Oftentimes, the objects retrieved from JNDI will be used more than once. A data source, for example, will be needed every time you access the database. It would be inefficient to repeatedly retrieve the data source from JNDI every time that it is needed. For that reason, `JndiObjectFactoryBean` caches the object that it retrieves from JNDI by default.

Caching is good in most circumstances. However, it precludes hot redeployment of objects in JNDI. If you were to change the object in JNDI, the Spring application would need to be restarted so that the new object can be retrieved.

If your application is retrieving an object from JNDI that will change frequently, you'll want to turn caching off for `JndiObjectFactoryBean`. To turn caching off, you'll need to set the `cache` property to false when declaring the bean:

```
<bean id="dataSource"
      class="org.springframework.jndi.JndiObjectFactoryBean"
      <property name="jndiName" value="jdbc/RantzDatasource" />
      <property name="cache" value="false" />
      <property name="proxyInterface" value="javax.sql.DataSource" />
</bean>
```

Setting the `cache` property to false tells `JndiObjectFactoryBean` to always fetch the object from JNDI. Notice that the `proxyInterface` property has also been set. Since the JNDI object can be changed at any time, there's no way for `JndiObjectFactoryBean` to know the actual type of the object. The `proxyInterface` property specifies a type that is expected for the object retrieved from JNDI.

Lazily loading JNDI objects

Sometimes your application won't need to retrieve the JNDI object right away. For instance, suppose that a JNDI object is only used in an obscure branch of your application's code. In that situation, it may not be desirable to load the object until it is actually needed.

By default, `JndiObjectFactoryBean` fetches objects from JNDI when the application context is started. Nevertheless, you can configure it to wait to retrieve the object until it's needed by setting the `lookupOnStartup` property to false:

```
<bean id="dataSource"
      class="org.springframework.jndi.JndiObjectFactoryBean">
      <property name="jndiName" value="jdbc/RantzDatasource" />
      <property name="lookupOnStartup" value="false" />
      <property name="proxyInterface" value="javax.sql.DataSource" />
</bean>
```

As with the `cache` property, you'll need to set the `proxyInterface` property when setting `lookupOnStartup` to false. That's because `JndiObjectFactoryBean` won't know the type of the object being retrieved until it is actually retrieved. The `proxyInterface` property tells it what type to expect from the fetched object.

Fallback objects

You now know how to wire JNDI objects in Spring and have a JNDI-loaded data source to show for it. Life is good. But what if the object can't be found in JNDI?

For instance, maybe your application can count on a data source being available in JNDI when running in a production environment. But that arrangement

may not be practical in a development environment. If Spring is configured to retrieve its data source from JNDI for production, the lookup will fail in development. How can we make sure that a data source bean is always available from JNDI in production and explicitly configured in development?

As you've seen, `JndiObjectFactoryBean` is great for retrieving objects from JNDI and wiring them in a Spring application context. But it also has a fallback mechanism that can account for situations where the requested object can't be found in JNDI. All you must do is configure its `defaultObject` property.

For example, suppose that you've declared a data source in Spring using `DriverManagerDataSource` as follows:

```
<bean id="devDataSource"
      class="org.springframework.jdbc.datasource.
      ➔ DriverManagerDataSource">
    <property name="driverClassName"
              value="org.hsqldb.jdbcDriver" />
    <property name="url"
              value="jdbc:hsqldb:hsq://localhost/roadrantz/roadrantz" />
    <property name="username" value="sa" />
    <property name="password" value="" />
</bean>
```

This is the data source that you'll use in development. But in production, you'd rather use a data source configured in JNDI by the system administrators. If that's the case, you'll configure the `JndiObjectFactoryBean` like this:

```
<bean id="dataSource"
      class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiName" value="jdbc/RantzDatasource" />
    <property name="defaultObject" ref="devDataSource" />
</bean>
```

Here I've wired the `defaultObject` property with a reference to the `devDataSource` bean. If `JndiObjectFactoryBean` can't find an object in JNDI at `jdbc/RantzDatasource`, it will use the `devDataSource` bean as its object.

As you can see, it's reasonably simple to use `JndiObjectFactoryBean` to wire JNDI-managed objects into a Spring application context. What we've covered so far works in all versions of Spring. But if you're using Spring 2, there's an even easier way. Let's see how Spring 2's namespace support makes JNDI wiring even simpler.

12.1.3 Wiring JNDI objects in Spring 2

As easy as it is to wire JNDI objects into your Spring context using `JndiObjectFactoryBean`, it's even easier if you're using Spring 2. Spring 2's `jee` namespace provides the `<jee:jndi-lookup>` configuration element for retrieving objects from

JNDI. To use the `<jee:jndi-lookup>` element, you must declare the `jee` namespace in your Spring XML using the following preamble:

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:jee="http://www.springframework.org/schema/jee"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/
                           spring-beans-2.0.xsd
                           http://www.springframework.org/schema/jee
                           http://www.springframework.org/schema/jee/spring-jee-2.0.xsd">
```

Now you can look up JNDI objects using `<jee:jndi-lookup>`. For example, the following XML snippet retrieves a data source from JNDI:

```
<jee:jndi-lookup id="dataSource"
                 jndi-name="jdbc/RantzDatasource"
                 resource-ref="true" />
```

Under the covers, `<jee:jndi-lookup>` configures a `JndiObjectFactoryBean` in the Spring context. This makes `<jee:jndi-lookup>` a shortcut for referencing objects in JNDI. The `jndi-name` attribute maps to the `jndiName` property of `JndiObjectFactoryBean` to identify the name of the object to look up. Likewise, the `resource-ref` attribute maps to the `resourceRef` property and is used to indicate that the object should be looked up as a JEE resource by prepending the `jndi-name` with `java:comp/env/`.

Looking up objects in JNDI comes in handy when you need access to objects that are configured external to Spring. As you've seen, data sources may be configured through an application server and accessed through JNDI. And as you'll see next, Spring's JNDI lookup capability can be useful when sending email. Let's take a look at Spring's email abstraction layer next.

12.2 **Sending email**

As a registered RoadRantz user, you might want to know when new rants are posted for your vehicles. Although you could visit the RoadRantz site over and over again, it may get a bit old to check every day only to find out that there's nothing new most of the time. Wouldn't it be nice if, instead of having to check the site for new rants, the site would contact you if there were new rants?

In this section, we'll add email functionality to the RoadRantz application so that an email is generated when a new rant is posted. In doing so, we're going to take advantage of the email support provided by Spring.

12.2.1 Configuring a mail sender

Spring comes with an email abstraction API that makes simple work of sending emails. At the heart of Spring's email abstraction is the `MailSender` interface.

As its name implies and as illustrated in figure 12.3, a `MailSender` implementation sends email.

Spring comes with two implementations of this interface, as described in table 12.1.

Table 12.1 Mail senders handle the intricacies of sending email. Spring comes with two mail sender implementations.

Mail sender implementation	Description
<code>CosMailSenderImpl</code>	Simple implementation of an SMTP mail sender based on Jason Hunter's COS (<code>com.oreilly.servlet</code>) implementation from his <i>Java Servlet Programming</i> book (O'Reilly, 1998).
<code>JavaMailSenderImpl</code>	A mail sender based on the JavaMail API. Allows for sending of MIME messages as well as non-SMTP mail (such as Lotus Notes).

Either mail sender is capable of meeting the needs of the RoadRantz email. But we'll choose `JavaMailSenderImpl` since it is the more versatile and more standard of the two options. We'll declare it as a `<bean>` in `roaddrantz-services.xml` as follows:

```
<bean id="mailSender"
      class="org.springframework.mail.javamail.JavaMailSenderImpl">
    <property name="host" value="mail.roaddrantz.com" />
</bean>
```

The `host` property specifies the hostname of the mail server that we'll use to send the email. By default, `JavaMailSenderImpl` assumes that the mail server is listening on port 25 (the standard SMTP port). However, if your mail server is listening on a different port, specify the correct port number using the `port` property:

```
<bean id="mailSender"
      class="org.springframework.mail.javamail.JavaMailSenderImpl">
    <property name="host" value="mail.roaddrantz.com" />
    <property name="port" value="1025" />
</bean>
```



Figure 12.3 Spring's `MailSender` interface is primary component of Spring's email abstraction API. It simply sends an email to a mail server for delivery.

If the mail server requires authentication, also set values for the `username` and `password` properties:

```
<bean id="mailSender"
      class="org.springframework.mail.javamail.JavaMailSenderImpl">
    <property name="host" value="mail.roadrantz.com" />
    <property name="username" value="rantzuser" />
    <property name="password" value="changeme" />
</bean>
```

As declared, this `mailSender` bean spells out the details of accessing the mail server. The mail server's hostname and the `username/password` pair are explicitly configured in Spring. However, this setup may raise red flags for you with regard to security. Maybe you don't want to hard-code this information in the Spring configuration.

You may already have a `javax.mail.MailSession` configured in JNDI (or perhaps one was placed there by your application server). If so then Spring's `JavaMailSenderImpl` offers you an option to use the `MailSender` in JNDI.

Using a JNDI mail session

You've already seen how to retrieve objects from JNDI in section 12.1. To use a mail session from JNDI, you can retrieve it using `JndiObjectFactoryBean`:

```
<bean id="mailSession"
      class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiName" value="mail/Session" />
    <property name="resourceRef" value="true" />
</bean>
```

Or you can use the `<jee:jndi-lookup>` configuration element:

```
<jee:jndi-lookup id="mailSession"
  jndi-name="mail/Session"
  resource-ref="true" />
```

In either event, the mail session object retrieved from JNDI can be wired into the `mailSender` bean as follows:

```
<bean id="mailSender"
      class="org.springframework.mail.javamail.JavaMailSenderImpl">
    <property name="session" ref="mailSession" />
</bean>
```

The mail session wired into the `session` property of `JavaMailSenderImpl` completely replaces the explicit server (and `username/password`) configuration from before. Now the mail session is completely configured in JNDI.

Wiring the mail sender into a service bean

Now that the mail sender has been configured, it's time to wire it into the bean that will use it. In the RoadRantz application, the `RantServiceImpl` class is the most appropriate place to send the email from. To make the mail sender available to the service bean, add a `mailSender` property (and its setter method) to `RantServiceImpl`:

```
private MailSender mailSender;
public void setMailSender(MailSender mailSender) {
    this.mailSender = mailSender;
}
```

Now we can use Spring DI to wire the `mailSender` bean into the `mailSender` property:

```
<bean id="rantService"
      class="com.roaddrantz.service.RantServiceImpl">
    <property name="rantDao" ref="rantDao" />
    <property name="mailSender" ref="mailSender" />
</bean>
```

With the `mailSender` bean wired into the `rantService` bean, we're ready to construct and send the email.

12.2.2 Constructing the email

Since we want to send an email to a motorist to alert the motorist of new rants for their vehicle, it seems that we'll need a method that sends an email for a particular vehicle. The `sendEmailForVehicle()` method in listing 12.1 uses the mail sender to send the email.

Listing 12.1 Sending an email to tell a motorist that they have new rants

```
public void sendEmailForVehicle(Vehicle vehicle) {
    Motorist motorist = vehicle.getMotorist();
    if(motorist == null) { return; }

    SimpleMailMessage message = new SimpleMailMessage(mailMessage); ← Constructs copy  
of message

    message.setTo(motorist.getEmail()); ← Sets recipient's address

    String text = message.getText();
    text = StringUtils.replace(text, "%STATE%",
        vehicle.getState());
    text = StringUtils.replace(text, "%PLATE%",
        vehicle.getPlateNumber());
    message.setText(text); | Fills in  
blanks

    mailSender.send(message); ← Sends email
}
```

The first thing that `sendEmailForVehicle()` does is verify that the vehicle has a motorist associated with it. If the motorist of the vehicle hasn't registered with RoadRantz, the motorist will be null and we won't be able to send the email.

Next, the details of the message are set. The motorist's email address is given to the `setTo()` method to specify the recipient of the email. And the message's text is set through the `setText()` method.

Finally, `sendEmailForVehicle()` uses the mail sender to send the email.

Most of the code in listing 12.1 is straightforward. But where does the email message come from? And what is going on with those calls to `String-Utils.replace()`?

Although we could explicitly define the entire email message within the scope of the `sendEmailForVehicle()` method, it would involve a lot of hard-coded values. It would be better to extract that message definition to a Spring-configured bean. The following bean declaration captures the common properties of the mail message:

```
<bean id="mailMessage"
      class="org.springframework.mail.SimpleMailMessage">
    <property name="from">
      <value><![CDATA[RoadRantz <notify@roadrantz.com>]]></value>
    </property>
    <property name="subject" value="You've got new Rantz!" />
    <property name="text">
      <value>
        <![CDATA[
Someone's been ranting about you! Log in to RoadRantz.com or
click on the link below to see what they had to say.

http://www.roadrantz.com/rantsForVehicle.htm?  
➡ state=%STATE&plateNumber=%PLATE%
]]>
      </value>
    </property>
  </bean>
```

The `mailMessage` bean serves as a template for all of the beans sent from the RoadRantz application. The `from` and `subject` values will be the same for all emails sent from RoadRantz, so there's no reason why we shouldn't configure them in this bean. The contents of the message will be mostly the same for all emails sent, so we can configure it here using the `text` property. On the other hand, the recipient will vary from email to email, so it doesn't make much sense to set the `to` property in the `mailMessage` bean.

To make the `mailMessage` bean available to the `sendEmailForVehicle()` method, we'll need to wire it into the `RantServiceImpl` class. First add a property to hold the bean and a setter that will be used to inject it into `RantServiceImpl`:

```
private SimpleMailMessage mailMessage;
public void setMailMessage(SimpleMailMessage mailMessage) {
    this.mailMessage = mailMessage;
}
```

Then configure the `rantService` bean to wire the `mailMessage` bean into the `mailMessage` property:

```
<bean id="rantService"
      class="com.roaddrantz.service.RantServiceImpl">
    <property name="rantDao" ref="rantDao" />
    <property name="mailSender" ref="mailSender" />
    <property name="mailMessage" ref="mailMessage" />
</bean>
```

The only thing left to discuss is what is going on in `sendEmailForVehicle()` where the `StringUtil.replace()` methods are used. Although the text of the email is mostly the same for all emails that are sent, it will vary slightly. That's because the link that is included in the email has parameters that are specific to the vehicle in question.

While we could have constructed the email text in the `sendEmailForVehicle()` method, we'd prefer to configure it externally. By configuring it externally, we are afforded the opportunity to tweak the message without having to change the method's source code.

So, the message configured in the `mailMessage` bean has two placeholders—`%STATE%` and `%PLATE%`—that are replaced in `sendEmailForVehicle()` with the vehicle's state and license plate number. This makes it possible for the message to be somewhat dynamic and still be configured in Spring.

Now that we have a `sendEmailForVehicle()` method, we should figure out how to best use it. Since we'd like to send an email alerting registered motorists of new rants for their vehicles, it would seem best to make the call to `sendEmailForVehicle()` from within `RantServiceImpl`'s `addRant()` method:

```
public void addRant(Rant rant) {  
    rant.setPostedDate(new Date());  
  
    Vehicle rantVehicle = rant.getVehicle();  
  
    // check for existing vehicle with same plates  
    Vehicle existingVehicle =  
        rantDao.findVehicleByPlate(rantVehicle.getState(),  
            rantVehicle.getPlateNumber());
```

```
if(existingVehicle != null) {  
    rant.setVehicle(existingVehicle);  
} else {  
    rantDao.saveVehicle(rantVehicle);  
}  
  
rantDao.saveRant(rant);  
  
sendEmailForVehicle(existingVehicle);  
}
```

Used this way, `sendEmailForVehicle()` will send an email to the motorist of the vehicle within moments of a new rant being added. This is good, but it could result in the motorist being bombarded with emails from RoadRantz. Maybe there is some justice in filling a bad motorist's inbox with frequent emails, but it probably will only result in the motorist discontinuing their relationship with RoadRantz—something we'd rather not have happen.

Rather than email the user after every rant, maybe it would be better to send only one email per motorist per day. For that we'll need a way to schedule the sending of emails. As it so happens, Spring provides support for task scheduling, as you'll see in the next section.

12.3 Scheduling tasks

Much of the code we've written thus far is triggered as the result of some user action. However, even though much of an application's functionality is triggered through user activity, sometimes it's necessary to kick off some activity based on a schedule.

For example, in the RoadRantz application we'd like to send a single email to every motorist who has received a rant within a given day. Even if the motorist has been ranted about multiple times, only one email should be sent per day. The `sendDailyRantEmails()` method in listing 12.2 pulls together a unique list of vehicles that have been ranted about and sends an email to their motorist.

Listing 12.2 Sending a daily email to motorists who have been ranted about

```
public void sendDailyRantEmails() {  
    List<Rant> rantsForToday = getRantsForToday(new Date()); ← Gets today's  
rants  
  
    Set<Vehicle> vehiclesRantedAboutToday = new HashSet<Vehicle>();  
  
    for (Rant rant : rantsForToday) {  
        vehiclesRantedAboutToday.add(rant.getVehicle());  
    }
```

```
for (Vehicle vehicle : vehiclesRantedAboutToday) {  
    sendEmailForVehicle(vehicle);  
}  
}  
  
|  
Sends emails
```

The first thing `sendDailyRantEmails()` does is call `getRantsForDay()` to retrieve a list of rants for the current day. Because a motorist may have received several rants on a given day, `sendDailyRantEmails()` then goes through the rants, placing each vehicle into a `java.util.Set`. By putting them into a `Set`, we know that there won't be any duplicates (and thus we won't send duplicate emails). Finally, `sendDailyRantEmails()` iterates over the set of vehicles and uses `sendEmailForVehicle()` to send an email to each vehicle's motorist.

Now we have a method that can be used to send a daily email to all the motorists who have been ranted about. What we need now is to set up a schedule for when that method will be called.

12.3.1 Scheduling with Java's Timer

Starting with Java 1.3, the Java SDK has included rudimentary scheduling functionality through its `java.util.Timer` class. This class lets you schedule a task (defined by a subclass `java.util.TimerTask`) to occur every so often.

Spring provides application context support for Java's Timer through `TimerFactoryBean`. `TimerFactoryBean` is a Spring factory bean that produces a Java Timer in the application context that kicks off a `TimerTask`. Figure 12.4 illustrates how `TimerFactoryBean` works.

I'll show you how to configure a `TimerFactoryBean` in a moment. But first, we need a `TimerTask` to send the email.

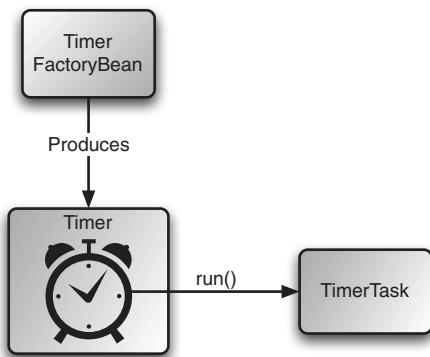


Figure 12.4
`Spring's TimerFactoryBean` produces a Java Timer, scheduled to kick off a `TimerTask` after a specified time has passed.

Creating a timer task

The first step in scheduling the daily rant email using Java's Timer is to create the email task by subclassing `java.util.TimerTask`, as shown in listing 12.3.

Listing 12.3 A timer task for sending the daily rant emails

```
package com.roaddrantz.service;
import java.util.TimerTask;

public class DailyRantEmailTask extends TimerTask {
    public DailyRantEmailTask() {}

    public void run() {
        rantService.sendDailyRantEmails();  ← Sends emails
    }

    // injected
    private RantService rantService;

    public void setRantService(RantService rantService) {
        this.rantService = rantService;
    }
}
```

Injects rant service

The `run()` method defines what the task is to do when it is run. In this case, it calls the `sendDailyRantEmails()` method on the injected `RantService` object.

The `DailyRantEmailTask` can be configured in Spring like this:

```
<bean id="dailyRantEmailTask"
      class="com.roaddrantz.service.DailyRantEmailTask">
    <property name="rantService" ref="rantService" />
</bean>
```

By itself, this `<bean>` declaration only places an instance of `DailyRantEmailTask` into the Spring application context and wires the `rantService` bean into its `rantService` property. But this bean is only a task—it won't do anything interesting until you schedule it.

Scheduling the timer task

Spring's `ScheduledTimerTask` defines how often a timer task is to be run. Since the rant email is daily, we should schedule it to run every 24 hours. The following `ScheduledTimerTask` bean should do the trick:

```
<bean id="scheduledEmailTask"
      class="org.springframework.scheduling.timer.ScheduledTimerTask">
    <property name="timerTask" ref="dailyRantEmailTask" />
    <property name="period" value="86400000" />
</bean>
```

The `timerTask` property tells the `ScheduledTimerTask` which `TimerTask` to run. Here it is wired with a reference to the `dailyRantEmailTask` bean, which is the `DailyRantEmailTask`. The `period` property is what tells the `ScheduledTimerTask` how often the `TimerTask`'s `run()` method should be called. This property, specified in milliseconds, has been set to `86400000` to indicate that the task should be kicked off every 24 hours.

Starting the timer

The last thing you'll need to do is to start the timer. Spring's `TimerFactoryBean` is responsible for starting timer tasks. You can declare the `TimerFactoryBean` in Spring like this:

```
<bean class="org.springframework.scheduling.timer.  
       ↗ TimerFactoryBean">  
  <property name="scheduledTimerTasks">  
    <list>  
      <ref bean="scheduledEmailTask"/>  
    </list>  
  </property>  
</bean>
```

The `scheduledTimerTasks` property takes an array of timer tasks that it should start. By default, `TimerFactoryBean` will start these tasks immediately upon application startup. Since we only have one timer task right now, the list contains a single reference to the `scheduledEmailTask` bean.

Delaying the start of the timer

Unfortunately, even though the task will be run every 24 hours, there is no way to specify what time of the day it should be run. `ScheduledTimerTask` does have a `delay` property that lets you specify how long to wait before the task is first run.

For example, to delay the first run of `DailyRantEmailTask` by an hour, you'd use this:

```
<bean id="scheduledEmailTask"  
      class="org.springframework.scheduling.timer.  
            ↗ ScheduledTimerTask">  
  <property name="timerTask" ref="reportTimerTask"/>  
  <property name="period" value="86400000" />  
  <property name="delay" value="3600000" />  
</bean>
```

Even with the delay, however, the time that the `DailyRantEmailTask` will run will be relative to when the application starts. And each successive run will be relative to the end time of the previous run. How can you have it sent at midnight every night (aside from starting the application at 11:00 p.m.)?

This highlights a limitation of using Java's Timer. Although it's great for running tasks at a regular interval, it's difficult to schedule tasks to run at a specific time. In order to specify precisely when the email is sent, you'll need to use the Quartz scheduler instead.

12.3.2 Using the Quartz scheduler

Suppose that we want the daily rant email to be sent at 11:59 p.m. every night. Unfortunately, Java's Timer is limited to scheduling *how often* the task is performed, not *when* it is performed.

The Quartz scheduler provides richer support for scheduling jobs. Just as with Java's Timer, you can use Quartz to run a job every so many milliseconds. But Quartz goes beyond Java's Timer by enabling you to schedule a job to run at a particular time and/or day. This makes Quartz more suitable for sending the daily rant email than Java's Timer.

For more information about Quartz, visit the Quartz home page at <http://www.opensymphony.com/quartz>.

Creating a Quartz job

The first step in defining a Quartz job is to create the class that defines the job. For that, we'll subclass Spring's QuartzJobBean, as shown in listing 12.4.

Listing 12.4 Defining a Quartz job for sending a daily rant email

```
package com.roadrantz.service;
import org.quartz.JobExecutionContext;
import org.quartz.JobExecutionException;
import org.springframework.scheduling.quartz.QuartzJobBean;

public class DailyRantEmailJob extends QuartzJobBean {
    public DailyRantEmailJob() {}

    protected void executeInternal(JobExecutionContext jobContext)
        throws JobExecutionException {
        rantService.sendDailyRantEmails(); ← Sends enrollment report
    }

    private RantService rantService;
    public void setRantService(RantService rantService) {
        this.rantService = rantService;
    }
}
```

Injects RantService

A QuartzJobBean is the Quartz equivalent of Java's TimerTask. It is an implementation of Quartz's org.quartz.Job interface. The executeInternal() method defines the actions that the job will do when its time comes. Here, just as with DailyRantEmailTask, the task simply makes a call to the sendDailyRantEmails() method of the injected RantService bean.

Declare the job in the Spring configuration file as follows:

```
<bean id="dailyRantEmailJob"
      class="org.springframework.scheduling.quartz.JobDetailBean">
    <property name="jobClass"
              value="com.rodrantz.service.DailyRantEmailJob" />
    <property name="jobDataAsMap">
      <map>
        <entry key="rantService" value-ref="rantService" />
      </map>
    </property>
  </bean>
```

Notice that although the job is defined in the DailyRantEmailJob class, it's not this class that is declared in the Spring context. Instead, a JobDetailBean is declared. This is an idiosyncrasy of working with Quartz. JobDetailBean is a subclass of Quartz's org.quartz.JobDetail, which requires that the Job implementation be set through the jobClass property.

Another quirk of working with Quartz's JobDetail is that the rantService property of DailyRantEmailJob isn't set directly. Instead, JobDetail's jobDataAsMap property takes a java.util.Map that contains properties that are to be set on the object specified by jobClass. Here, the map contains a reference to the rantService bean with a key of rantService. When JobDetailBean is instantiated, it will inject the rantService bean into the rantService property of DailyRantEmailJob.

Scheduling the job

Now that the job is defined, you'll need to schedule the job. As shown in figure 12.5, Quartz's org.quartz.Trigger class decides when and how often a Quartz job should run.

Spring comes with two subclasses of Trigger: SimpleTriggerBean and CronTriggerBean. Which one should you use? Let's have a look at each of them, starting with SimpleTriggerBean.

SimpleTriggerBean is very similar to ScheduledTimerTask, which we discussed in the last section. Using it, you can specify how often a job should run and (optionally) how long to wait before running the job for the first time. For

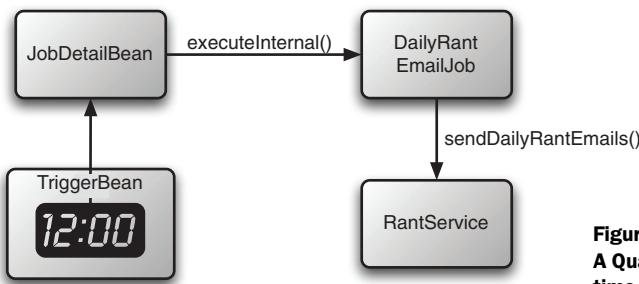


Figure 12.5
A Quartz trigger determines the exact time that a job will be kicked off.

example, to schedule the report job to run every 24 hours, with the first run starting one hour after the application starts, declare the following bean:

```

<bean id="simpleReportTrigger"
      class="org.springframework.scheduling.quartz.
      ↗ SimpleTriggerBean">
    <property name="jobDetail" ref="dailyRantEmailJob" />
    <property name="startDelay" value="3600000" />
    <property name="repeatInterval" value="86400000" />
</bean>
  
```

The jobDetail property is wired with the job that is to be scheduled. Here it is the dailyRantEmailJob bean, which we declared earlier. The repeatInterval property tells the trigger how often to run the job (in milliseconds). Here we've specified that the job should run every 86,400,000 milliseconds—or every 24 hours. Finally, the optional startDelay property has been set to delay the first run of the job to one hour (or 3,600,000 milliseconds) after the application is started.

Although you can probably think of many applications for which SimpleTriggerBean is perfectly suitable, it isn't sufficient for emailing the daily rant email. Just as with the DailyRantEmailTask (which is based on Java's Timer), we can only specify how often the job is run—not exactly when it's run. Therefore, we can't rely on SimpleTriggerBean to send out the daily emails at midnight as we want. For more precise control over scheduling, Quartz provides cron jobs.

Scheduling a cron job

CronTriggerBean is another Spring subclass of Quartz's Trigger class. This trigger class, however, lets you specify exact times and days when the job will run. If you're familiar with the Unix cron tool, you'll feel right at home with CronTriggerBean. Instead of declaring how often a job is run, CronTriggerBean lets you use a cron expression to specify exact times (and days) that a job will run.

For example, to declare that `DailyRantEmailJob` be run every day at 11:59 p.m., declare a `CronTriggerBean` in Spring as follows:

```
<bean id="cronEmailTrigger"
    class="org.springframework.scheduling.quartz.CronTriggerBean">
    <property name="jobDetail" ref="dailyRantEmailJob"/>
    <property name="cronExpression" value="0 59 23 * * ?" />
</bean>
```

As with `SimpleTriggerBean`, the `jobDetail` property tells the trigger which job to schedule. Again, we've wired it with a reference to the `dailyRantEmailJob` bean. The `cronExpression` property tells the trigger when to fire. If you're a cron fanatic, you will have no trouble deciphering this property's value (and we're guessing that you have little trouble setting the timer on your VCR).

But for those of you who aren't as well versed in cron expressions, let's break down the `cronExpression` property a bit. It is made up of six (or possibly seven) time elements, separated by spaces. In order from left to right, the elements are defined as follows:

- 1 Seconds (0–59)
- 2 Minutes (0–59)
- 3 Hours (0–23)
- 4 Day of month (1–31)
- 5 Month (1–12 or JAN–DEC)
- 6 Day of week (1–7 or SUN–SAT)
- 7 Year (1970–2099)

Each of these elements can be specified with an explicit value (e.g., 6), a range (e.g., 9–12), a list (e.g., 9,11,13), or a wildcard (e.g., *). The day of the month and day of the week elements are mutually exclusive, so you should also indicate which one of these fields you don't want to set by specifying it with a question mark (?). Table 12.2 shows some example cron expressions and what they mean.

In the case of the `cronEmailTrigger` bean, we've set the `cronExpression` property to 0 59 23 * * ?. You can read this as the zero second of the 59th minute of the 23rd hour of any day of the month of any month (regardless of the day of the week). In other words, the trigger is fired at a minute before midnight every night.

With this kind of precision in timing, it's clear that `CronTriggerBean` is better suited for our daily email than `SimpleTriggerBean`. Now all that's left is to start the job.

Table 12.2 Some sample cron expressions.

Expression	What it means
0 0 10,14,16 * * ?	Every day at 10 a.m., 2 p.m., and 4 p.m.
0 0,15,30,45 * 1-30 * ?	Every 15 minutes on the first 30 days of the month
30 0 0 1 1 ? 2012	30 seconds after midnight on January 1, 2012
0 0 8-17 ? * MON-FRI	Every working hour of every business day

Starting the job

To start a Quartz job, we'll use Spring's `SchedulerFactoryBean`. `SchedulerFactoryBean` is the Quartz equivalent to `TimerFactoryBean`. It is declared in the Spring configuration as follows:

```
<bean class="org.springframework.scheduling.
    ↪ quartz.SchedulerFactoryBean">
<property name="triggers">
    <list>
        <ref bean="cronEmailTrigger"/>
    </list>
</property>
</bean>
```

The `triggers` property takes an array of references to trigger beans. Since we only have a single trigger at the moment, we simply need to wire it with a list containing a single reference to the `cronEmailTrigger` bean.

At this point, we should have a nightly email generated at just before midnight every night. But in doing so, perhaps we've done a bit too much work. Before we let this go, let's take a look at a slightly easier way to schedule the nightly email.

12.3.3 Invoking methods on a schedule

In scheduling the nightly rant email we wrote a `DailyRantEmailJob` bean (or the `DailyRantEmailTask` bean in the case of the timer tasks). But this bean doesn't do much more than make a simple call to the `sendDailyRantEmails()` method of the `RantService`. In this light, both `DailyRantEmailJob` and `DailyRantEmailTask` seem a bit superfluous. Wouldn't it be great if we could just ask Spring to call `sendDailyRantEmails()` without having to write the extra task or job class?

Good news! If all you want to do is schedule a single method call, you can do that without writing a separate `TimerTask` or `QuartzJobBean` class. To accomplish this, Spring has provided `MethodInvokingTimerTaskFactoryBean` and

`MethodInvokingJobDetailFactoryBean` to schedule method calls with Java's timer support and the Quartz scheduler, respectively.

For example, to schedule a call to `sendDailyRantEmails()` using Java's timer service, redeclare the `scheduledEmailTask` bean as follows:

```
<bean id="scheduledEmailTask"
      class="org.springframework.scheduling.timer.
          MethodInvokingTimerTaskFactoryBean">
    <property name="targetObject" ref="rantService" />
    <property name="targetMethod" value="sendDailyRantEmails" />
</bean>
```

Behind the scenes, `MethodInvokingTimerTaskFactoryBean` will create a `TimerTask` that calls the method specified by the `targetMethod` property on the object that is referenced by the `targetObject` property (as shown in figure 12.6). This is effectively the same as the `DailyRantEmailTask`. Now you can eliminate the `DailyRantEmailTask` class and its declaration in the `dailyRantEmailTask` bean.

`MethodInvokingTimerTaskFactoryBean` is good when scheduling simple one-method calls using a `ScheduledTimerTask`. But `ScheduledTimerTask` didn't provide us with the precision needed to schedule the email at just before midnight every night. So instead of using `MethodInvokingTimerTaskFactoryBean`, let's redeclare the `dailyRantEmailJob` bean as follows:

```
<bean id="dailyRantEmailJob"
      class="org.springframework.scheduling.quartz.
          MethodInvokingJobDetailFactoryBean">
    <property name="targetObject" ref="rantService" />
    <property name="targetMethod" value="sendDailyRantEmails" />
</bean>
```

As you may have guessed, `MethodInvokingJobDetailFactoryBean` is the Quartz equivalent of `MethodInvokingTimerTaskFactoryBean`. Under the covers it will

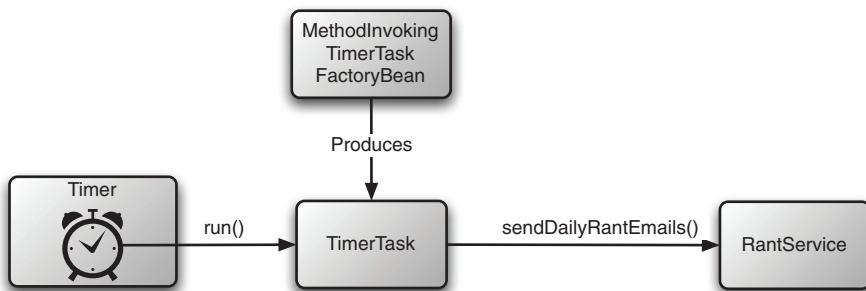


Figure 12.6 `MethodInvokingTimerTaskFactoryBean` produces a Java `TimerTask` that is configured to invoke a specific method on a specific bean in the application context.

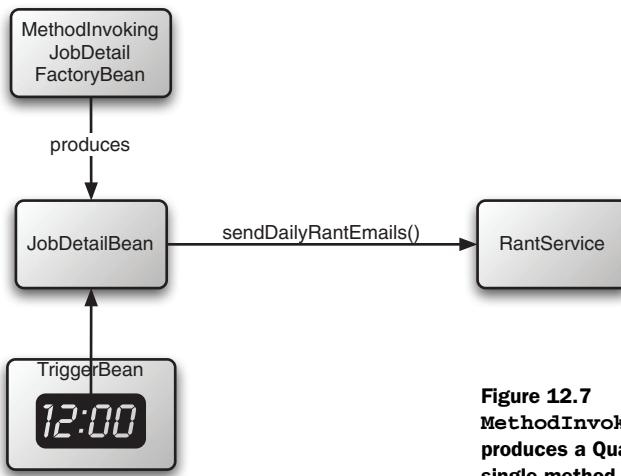


Figure 12.7
MethodInvokingJobDetailFactoryBean
 produces a Quartz JobDetail that invokes a
 single method on a specified bean.

create a Quartz JobDetail object that makes a single method call to the object and method specified by the `targetObject` and `targetMethod` properties (see figure 12.7).

Now that we've scheduled our email, we can sit back and enjoy the fact that our registered motorists are receiving their rant notification emails. But wait... what if the mail server is down at the time when the scheduler goes off? Is there a way that we can manually trigger the email? Or what if we decide to change the scheduler's time? Do we need to redeploy the application to enact changes to the scheduler?

To address these concerns, let's now look at how Spring's support for JMX enables us to create a management interface for our application's beans, letting us change and invoke them on the fly.

12.4 Managing Spring beans with JMX

Spring's support for DI is a great way to configure bean properties in an application. But once the application has been deployed and is running, there's not much that DI alone can do to help you change that configuration. Suppose that you want to dig into a running application and change its configuration on the fly. That's where Java Management Extensions (JMX) comes in.

JMX is a technology that enables you to instrument applications for management, monitoring, and configuration. Originally available as a separate extension to Java, JMX is now a standard part of the Java 5 distribution.

In this section we'll focus on how JMX is supported in Spring. If you want to learn more about JMX, we recommend that you have a look at *JMX in Action* (Manning, 2002).

The key component of an application that is instrumented for management with JMX is the MBean (managed bean). An MBean is a JavaBean that exposes certain methods that define the management interface. The JMX specification defines four types of MBeans:

- *Standard MBeans*—Standard MBeans are MBeans whose management interface is determined by reflection on a fixed Java interface that is implemented by the bean class.
- *Dynamic MBeans*—Dynamic MBeans are MBeans whose management interface is determined at runtime by invoking methods of the `DynamicMBean` interface. Because the management interface isn't defined by a static interface, it can vary at runtime.
- *Open MBeans*—Open MBeans are a special kind of dynamic MBean whose attributes and operations are limited to primitive types, class wrappers for primitive types, and any type that can be decomposed into primitives or primitive wrappers.
- *Model MBeans*—A model MBean is a special kind of dynamic MBean that bridges a management interface to the managed resource. Model MBeans aren't written as much as they are declared. Model MBeans are typically produced by a factory that uses some meta-information to assemble the management interface.

Spring's JMX module enables you to export Spring beans as Model MBeans so that you can see inside your application and tweak the configuration—even while the application is running. Let's see how to use Spring's JMX support to manage the beans within a Spring application.

12.4.1 Exporting Spring beans as MBeans

In the previous section, we used Spring's scheduling support to schedule the generation of emails at 11:59 p.m. every night. Although just before midnight is usually best for sending the emails, it would also be nice to be able to adjust the timing of the email without having to redeploy the RoadRantz application. To accommodate reconfiguration of the scheduler, we're going to use Spring's JMX support to export the timer bean as an MBean.

Spring's `MBeanExporter` is a bean that exports one or more Spring beans as Model MBeans in an *MBean server*. An MBean server (sometimes called an MBean

agent) is a container where MBeans live and through which the MBeans are accessed. For MBeans to be of any use for management and configuration, they must be registered in an MBean server. As illustrated in figure 12.8, exporting Spring beans as JMX MBeans makes it possible for a JMX-based management tool such as MC4J (<http://mc4j.org>) to peer inside a running application to view the beans' properties and invoke their methods.

The following <bean> declares an MBeanExporter bean in Spring to export the cronEmailTrigger bean as a Model MBean:

```
<bean class="org.springframework.jmx.export.MBeanExporter">
    <property name="beans">
        <map>
            <entry key="rantz:name=emailSchedule"
                  value-ref="cronEmailTrigger"/>
        </map>
    </property>
</bean>
```

In its simplest form, MBeanExporter can be configured through its beans property with a <map> of one or more beans that you'd like to expose as a model MBean through JMX. The key of each <entry> is the name of the MBean. The value of the <entry> is a reference to the Spring-managed bean that is to be exported. Here we're exporting the cronEmailTrigger bean so that the timer can be managed through JMX.

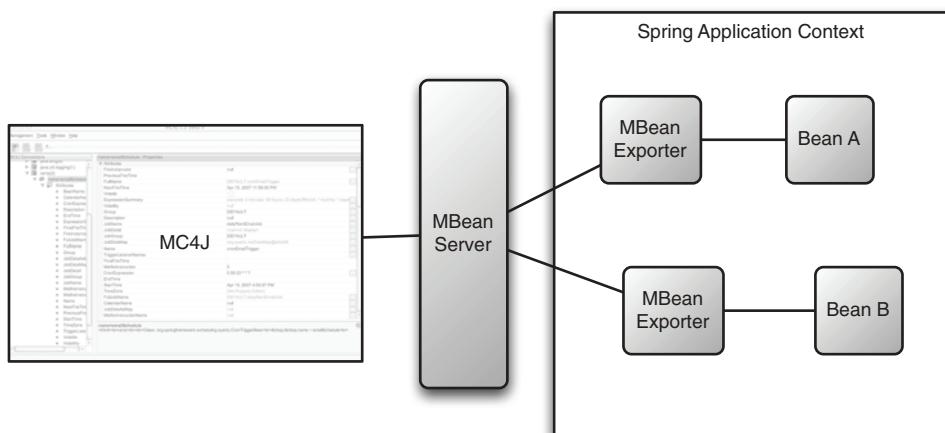


Figure 12.8 Spring's MBeanExporter exports the properties and methods of Spring beans as JMX attributes and operations in an MBean server. From there, a JMX management tool such as MC4J can look inside the running application.

NOTE

As configured above, `MBeanExporter` assumes that it is running within an application server that provides an MBean server (such as Tomcat or JBoss). But if your Spring application will be running stand-alone or in a container that doesn't provide an MBean server, you'll want to configure an `MBeanServerFactoryBean`:

```
<bean id="jmxServer"
      class="org.springframework.jmx.support.
      ↪ MBeanServerFactoryBean">
    <property name="defaultDomain" value="rantz" />
</bean>
```

Then you'll need to wire the `MBeanServerFactoryBean` into the `MBeanExporter`'s `server` property:

```
<bean id="mbeanExporter"
      class="org.springframework.jmx.export.MBeanExporter">
  <property name="beans">
    <map>
      <entry key="rantz:name=emailSchedule"
            value-ref="cronEmailTrigger" />
    </map>
  </property>
  <property name="server" ref="jmxServer" />
</bean>
```

With the `MBeanExporter` in place, the `cronEmailTrigger` bean will be exported as a Model MBean to the MBean server for management under the name `emailSchedule`. Figure 12.9 shows how the `cronEmailTrigger` MBean appears when viewed through MC4J.

As you can see from figure 12.9, all public members of the `cronEmailTrigger` bean are exported as MBean operations and attributes. This is probably not what we want. All we really want to do is to be able to configure the timing of the daily email. The `cronExpression` property tells `CronTriggerBean` when to trigger jobs. But even if we change this property, its schedule won't take effect until after the next job is fired. If we want to control when the next job is fired, we'll also need to be able to configure the `nextFireTime` property.

All of the other attributes and operations, however, are superfluous and just get in the way. Furthermore, we may want to purposefully restrict those other attributes and operations from appearing in the management interface to avoid accidental changes to the bean. Thus, we need a way to select which attributes and operations end up in the management.

Recall that a model MBean is an MBean whose management interface is assembled using some form of meta-information. In Spring, when it comes to

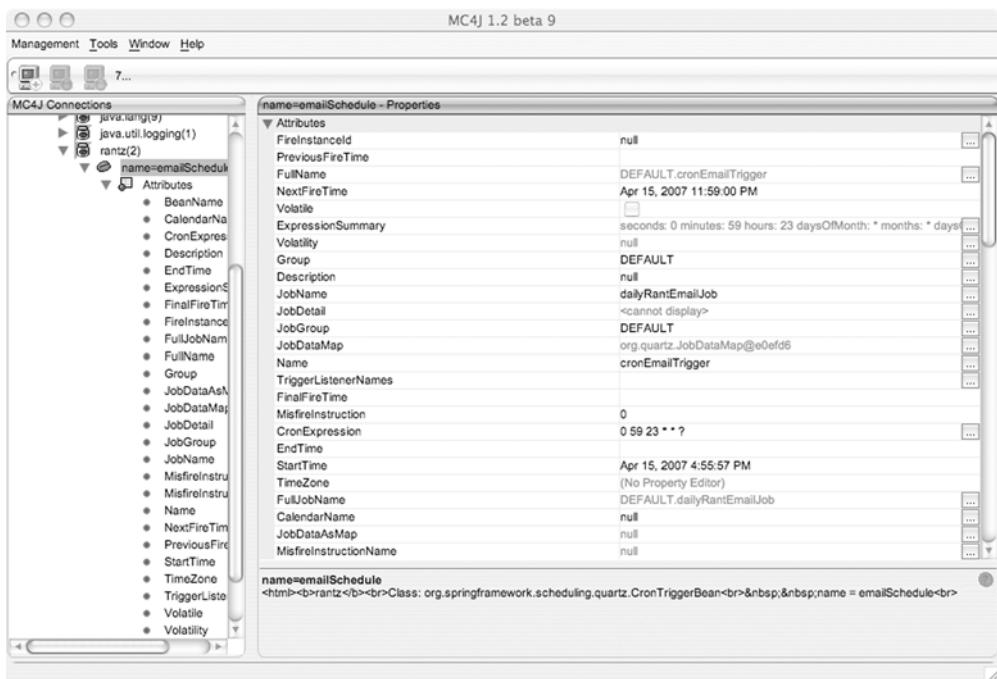


Figure 12.9 `CronTriggerBean` exported as an MBean and seen through the eyes of MC4J. Notice that all of `CronTriggerBean`'s public methods and properties are exported as MBean operations and attributes.

picking and choosing which methods and properties of a bean become operations and attributes of a model MBean, we must specify an MBean *assembler*:

```
<bean id="mbeanExporter"
      class="org.springframework.jmx.export.MBeanExporter">
    <property name="beans">
      <map>
        <entry key="rantz:name=emailSchedule"
              value-ref="cronEmailTrigger"/>
      </map>
    </property>

    <property name="assembler" ref="assembler" />
</bean>
```

An assembler is a bean whose job is to assemble the management interface for MBeans that are exported by `MBeanExporter`. We have three assemblers to choose from, each using a different source of meta-information to define the management interface.

- `MethodNameBasedMBeanInfoAssembler`—Lets you explicitly configure methods to expose by name
- `InterfaceBasedMBeanInfoAssembler`—Exposes bean methods based on what is contained in an interface
- `MetadataMBeanInfoAssembler`—Exposes bean methods and properties that are annotated with `@ManagedOperation` and `@ManagedAttribute`

Let's look at how to use each of these MBean assemblers one by one starting with the `MethodNameBasedMBeanInfoAssembler`.

Exposing methods by name

`MethodNameBasedMBeanInfoAssembler` is an MBean assembler that decides which bean methods and properties to expose on the MBean's management interface based on a list of method names. The following `<bean>` declaration shows an example of using `MethodNameBasedMBeanInfoAssembler` to expose the `cronExpression` and `nextFireTime` properties of the `cronEmailTrigger` bean:

```
<bean id="assembler"
      class="org.springframework.jmx.export.assembler.
           ↪ MethodNameBasedMBeanInfoAssembler">
  <property name="managedMethods">
    <list>
      <value>setCronExpression</value>
      <value>getCronExpression</value>
      <value>setNextFireTime</value>
      <value>getNextFireTime</value>
    </list>
  </property>
</bean>
```

The `managedMethods` property takes a list of method names that are to be exposed as managed operations. Notice that to expose the `cronExpression` and `nextFireTime` properties as MBean attributes, we had to declare their setter and getter methods. Although `managedMethods` is meant to expose methods as managed operations, the corresponding properties are exposed as managed attributes if the methods are getter and setter methods.

Now when we look at the `emailSchedule` MBean in a JMX client, we see only the attributes and operations we specified to be exported. Figure 12.10 shows how the newly assembled `emailSchedule` MBean looks in MC4J.

As you've seen, the `MethodNameBasedMBeanInfoAssembler` is the simplest of all of Spring's MBean assemblers. It lets you succinctly list all the methods that you wish to expose in the management interfaces of the exported MBeans.

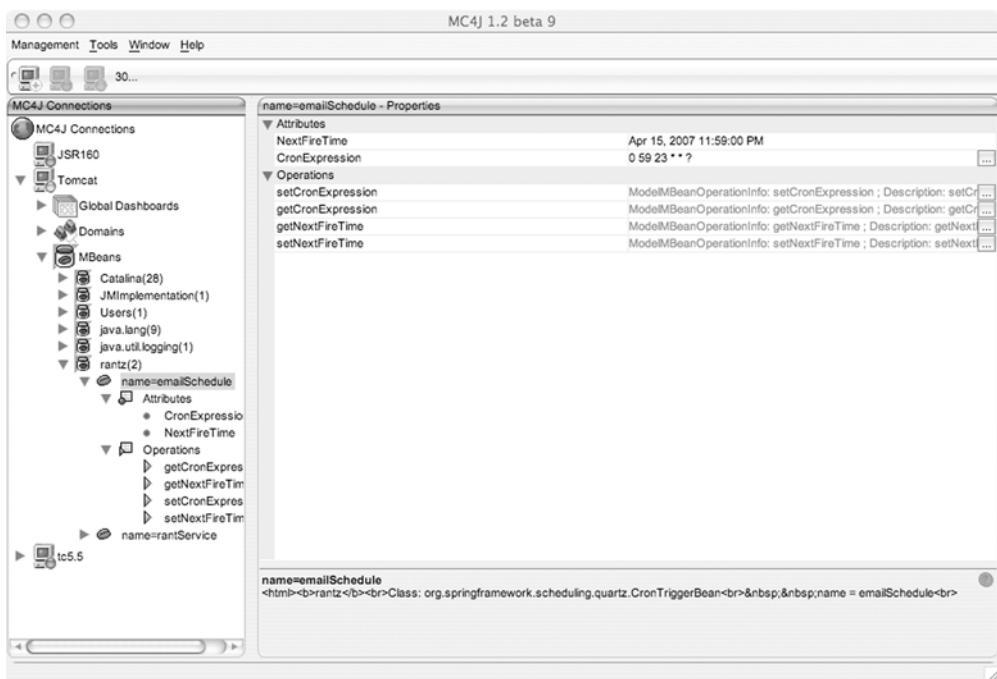


Figure 12.10 Once we use an MBean assembler, only selected methods and properties are exposed through the exported MBean.

On the other hand, `MethodNameBasedMBeanInfoAssembler` is also the most cumbersome to use because it requires that you specify all of the methods that you wish to expose. If you are using `MBeanExporter` to export several beans, each with their own set of methods to be exposed, the list of method names given to `managedMethods` will likely grow very large. And because the method names are all listed together, it will be difficult to know which methods belong to which exported beans.

Using interfaces to define MBean operations and attributes

Another of Spring's MBean assemblers is `InterfaceBasedMBeanInfoAssembler`. `InterfaceBasedMBeanInfoAssembler` is similar to `MethodNameBasedMBeanInfoAssembler`, except that instead of declaring which methods to expose on the management interface, you declare one or more Java interfaces that define the management methods.

To illustrate, suppose that we've defined the following interface:

```
package com.roadrantz.service.mbean;
import java.util.Date;

public interface ManagedCronTrigger {
    void setCronExpression(String ce);
    String getCronExpression();
    void setNextFireTime(Date date);
    Date getNextFireTime();
}
```

The `ManagedCronTrigger` interface contains declarations of the methods we'd like to expose from the Quartz `CronTriggerBean`. With this interface, we can declare an interface-based assembler as follows:

```
<bean id="assembler"
      class="org.springframework.jmx.export.assembler.
           ↗ InterfaceBasedMBeanInfoAssembler">
    <property name="managedInterfaces">
        <list>
            <value>com.roadrantz.service.mbean.ManagedCronTrigger</value>
        </list>
    </property>
</bean>
```

With `ManagedCronTrigger` being the only managed interface specified, this assembler is effectively equivalent to the `MethodNameBasedMBeanInfoAssembler` we declared earlier. The difference is that we're now able to use Java interfaces to define our MBean managed interfaces.

Before we move on to the next type of assembler, you may want to take note of the fact that although `ManagedCronTrigger` declares methods that we'd like to expose on the exported MBean, `CronTriggerBean` doesn't directly implement this interface. Oftentimes, the interfaces specified in the `managedInterfaces` property will be interfaces that are actually implemented by the exported beans. But as you can see here in the case of `CronTriggerBean` and `ManagedCronTrigger`, they do not have to be.

Both `MethodNameBasedMBeanInfoAssembler` and `InterfaceBasedMBeanInfoAssembler` are suitable for assembling an MBean's managed interface, especially when you do not have access to the source code for the beans that are to be exported. But there's one more MBean information assembler that is great when you have access to the MBean's source code.

Working with metadata-driven MBeans

If you are lucky enough to have access to the bean's source code, you may want to consider exporting your beans using a metadata-driven MBean assembler.

For example, suppose that we'd like to export the `rantService` bean as an MBean so that we can invoke the `sendDailyRantEmails()` method as a managed operation. We could certainly use either `MethodNameBasedMBeanInfoAssembler` or `InterfaceBasedMBeanInfoAssembler` to assemble the MBean's managed interface. But with either of those assemblers we'd have to write some interface or declaration separate from the `rantService` bean or the `RantServiceImpl` class. What if we could take advantage of Java 5 annotations instead?

Spring's `MetadataMBeanInfoAssembler` is an MBean assembler that assembles an MBean's managed interface based on source-level metadata placed on the methods and properties to be exposed. The following `<bean>` declaration sets up the assembler bean to use source-level metadata:

```
<bean id="assembler"
      class="org.springframework.jmx.export.assembler.
            ➔ MetadataMBeanInfoAssembler">
    <property name="attributeSource" ref="attributeSource" />
</bean>
```

The `attributeSource` property is used to tell `MetadataMBeanInfoAssembler` what kind of metadata to look for. In theory, `MetadataMBeanInfoAssembler` can be configured to read MBean metadata from virtually any number of metadata sources, so long as the `attributeSource` property is configured with an implementation of `org.springframework.jmx.export.metadata.JmxAttributeSource`. Spring comes with two such implementations to choose from:

- `AttributesJmxAttributeSource`—Reads MBean metadata that is precompiled into source code using Jakarta Commons Attributes
- `AnnotationJmxAttributeSource`— Reads MBean metadata from JDK 1.5 annotations

Since we're targeting Java 5, we'll use `AnnotationJmxAttributeSource` so that we can use annotations. It is declared in Spring with the following `<bean>`:

```
<bean id="attributeSource"
      class="org.springframework.jmx.export.annotation.
            ➔ AnnotationJmxAttributeSource" />
```

Meanwhile, `MBeanExporter` is wired with a reference to the `MetadataMBeanInfoAssembler` along with a couple of other useful properties:

```
<bean id="mbeanExporter"
      class="org.springframework.jmx.export.MBeanExporter">
    <property name="assembler" ref="assembler" />
    <property name="autodetectModeName"
              value="AUTODETECT_ASSEMBLER" />
    <property name="namingStrategy" ref="namingStrategy" />
</bean>
```

Rather than explicitly list all beans that are to be exposed as MBeans through the beans property, we'd like Spring to figure out which beans to expose as MBeans based on their annotations. Therefore, we've configured the autodetectModeName property with AUTODETECT_ASSEMBLER. This tells the MBeanExporter to use the MetadataMBeanInfoAssembler to look for all beans in the Spring application context that are annotated with the @ManagedResource annotation.

Moreover, MetadataMBeanInfoAssembler determines a bean's managed attributes and operations by looking for properties and methods annotated with @ManagedAttribute and @ManagedOperation (respectively). For example, consider the annotated RantService interface in Listing 12.5.

Listing 12.5 Using annotations to declaratively create MBeans

```
package com.roaddrantz.service;
import java.util.Date;
import java.util.List;
import org.springframework.jmx.export.annotation.
    ↗ ManagedOperation;
import org.springframework.jmx.export.annotation.
    ↗ ManagedResource;
import com.roaddrantz.domain.Rant;
import com.roaddrantz.domain.Motorist;
import com.roaddrantz.domain.Vehicle;

@ManagedResource(objectName="rantz:name=RantService")    ↘ Declare as an
public interface RantService {                                MBean
    ↗
    public void addRant(Rant rant);
    public List<Rant> getRecentRants();
    public void addMotorist(Motorist motorist)
        throws MotoristAlreadyExistsException;
    public List<Rant> getRantsForVehicle(Vehicle vehicle);
    public List<Rant> getRantsForDay(Date date);
    public void sendEmailForVehicle(Vehicle vehicle);

    @ManagedOperation(
        description="Send the daily rant e-mail.")    ↘ Expose as a
        ↗
        ↗ managed operation
    public void sendDailyRantEmails();
}
```

I've annotated the `RantService` interface with `@ManagedResource` to indicate that any class that implements it should be exposed as an MBean. I've also annotated the `sendDailyRantEmails()` method with `@ManagedOperation` to indicate that this method should be exposed as a managed operation.

However, the `beans` property did more than just list the beans to expose as MBeans; it also gave the MBeans their names. If we're not explicitly listing the beans anymore, how can we make sure that the MBeans are named appropriately?

That's what the `namingStrategy` property is for. By default, `MBeanExporter` uses `KeyNamingStrategy`, which draws the MBean name from the key value in the map that is wired into the `beans` property. Since we're not using the `beans` map, `KeyNamingStrategy` won't work. Instead, we'll use `MetadataNamingStrategy`, which is declared as follows:

```
<bean id="namingStrategy"
      class="org.springframework.jmx.export.naming.
           ↗ MetadataNamingStrategy">
    <property name="attributeSource" ref="attributeSource" />
</bean>
```

As you might guess, `MetadataNamingStrategy` determines MBean names from metadata placed in the bean class. In this case, we've wired the `attributeSource` with a reference to the `AnnotationJmxAttributeSource` bean we defined earlier. Thus, each MBean's name will be specified by the `objectName` attribute of the `@ManagedResource` annotation.

Handling MBean object name collisions

So far you've seen how to publish an MBean into an MBean server using several approaches. In all cases, we've given the MBean an object name that is made up of a managed domain name and a key-value pair. Assuming that there's not already an MBean published with the name we've given our MBean, we should have no trouble publishing our MBean. But what happens if there's a name collision?

By default, `MBeanExporter` will throw an `InstanceAlreadyExistsException` should you try to export an MBean that is named the same as an MBean that's already in the MBean server. But you can change that behavior by setting the `registrationBehaviorName` property on the `MBeanExporter`. For example, the following `<bean>` declares an `MBeanExporter` that replaces the existing MBean with the new MBean being registered:

```
<bean class="org.springframework.jmx.export.MBeanExporter">
  <property name="beans">
    <map>
      <entry key="rantz:name=emailSchedule"
```

```

        value-ref="cronEmailTrigger"/>
    </map>
</property>

<property name="registrationBehaviorName"
    value="REGISTRATION_REPLACE_EXISTING" />
</bean>
```

There are three options for the `registrationBehaviorName` property, as described in table 12.3.

As we mentioned, `MBeanExporter`'s default behavior is to throw an `InstanceAlreadyExistsException` in the event of an MBean object name collision. Therefore, it's unnecessary to explicitly set the `registrationBehaviorName` property with `REGISTRATION_FAIL_ON_EXISTING`.

Table 12.3 MBean registration behavior options.

Behavior name	Use it when...
REGISTRATION_FAIL_ON_EXISTING	You'd like to be notified (via an exception) when an MBean registration fails due to a name collision
REGISTRATION_IGNORE_EXISTING	You'd like to attempt to register an MBean but fail silently on a name collision
REGISTRATION_REPLACE_EXISTING	You'd like to replace an existing MBean with a new MBean

Now that we've registered our MBeans using `MBeanExporter`, we'll need a way to access them for management. As you've seen already, most JMX implementations support an HTML interface for the MBean server. But the HTML interface varies across all JMX implementations and doesn't lend itself to programmatic management of MBeans. Fortunately, there's another way to access MBeans as remote objects. Let's explore how Spring's support for remote MBeans will enable us to access our MBeans in a standard way.

12.4.2 Remoting MBeans

Although the original JMX specification referred to remote management of applications through MBeans, it didn't define the actual remoting protocol or API. Consequently, it fell to JMX vendors to define their own, often proprietary, remoting solutions.

In response to the need for a standard for remote JMX, the Java Community Process produced JSR-160, the Java Management Extensions (JMX) Remote API Specification. This specification defines a standard for JMX remoting, which at

minimum requires an RMI binding and optionally the JMX Messaging Protocol (JMXMP).

Exposing remote MBeans

The simplest thing we can do to make our MBeans available as remote objects is to configure Spring's `ConnectorServerFactoryBean`:

```
<bean class="org.springframework.jmx.support.  
      ↪ ConnectorServerFactoryBean" />
```

`ConnectorServerFactoryBean` creates and starts a JSR-160 `JMXConnectorServer`. By default, the server listens for the JMXMP protocol on port 9875—thus it is bound to `service:jmx:jmxmp://localhost:9875`. The problem with this is that most JMX implementations do not support JMXMP. Therefore, we'll need to choose some other protocol for accessing our MBeans.

Depending on your JMX provider, you may have several remoting protocol options to choose from. MX4J's (<http://mx4j.sourceforge.net/>) remoting support includes RMI, SOAP, Hessian/Burlap, and IIOP. RMI is sufficient for our needs, so let's configure `ConnectorServerFactoryBean` to expose its beans remotely with RMI:

```
<bean class="org.springframework.jmx.support.  
      ↪ ConnectorServerFactoryBean">  
  <property name="serviceUrl"  
    value="service:jmx:rmi://localhost/jndi/rmi://localhost:1099/  
          ↪ rantz" />  
</bean>
```

The `serviceUrl` property is used to specify the remote binding for the `JMXConnectorServer`. In this case, we're binding it to an RMI registry listening on port 1099 of the localhost. That means we'll also need an RMI registry running. As you'll recall from chapter 8, an RMI registry can be started through Spring with the following `<bean>` declaration:

```
<bean class="org.springframework.remoting.rmi.  
      ↪ RmiRegistryFactoryBean">  
  <property name="port" value="1099" />  
</bean>
```

And that's it! Now our MBeans are available through RMI. But there's little point in doing this if nobody will ever access the MBeans over RMI. So, let's now turn our attention to the client side of JMX remoting and see how to wire up a remote MBean in Spring.

Accessing remote MBeans

Accessing a remote MBean server involves configuring an `MBeanServerConnectionFactoryBean` in Spring. The following `<bean>` declares an `MBeanServerConnectionFactoryBean` that is used to access the remote server we created in the previous section:

```
<bean id="mBeanServerClient"
      class="org.springframework.jmx.support.
           ↗ MBeanServerConnectionFactoryBean">
    <property name="serviceUrl"
              value="service:jmx:rmi://localhost/jndi/rmi://localhost:1099/
                   ↗ rantz" />
</bean>
```

As its name implies, `MBeanServerConnectionFactoryBean` is a factory bean that creates an `MBeanServerConnection`. The `MBeanServerConnection` produced by `MBeanServerConnectionFactoryBean` acts as a local proxy to the remote MBean server. It can be wired into a property just like any other bean:

```
<bean id="jmxClient"
      class="com.springinaction.jmx.JmxClient">
    <property name="mbeanServerConnection"
              ref="mBeanServerClient" />
</bean>
```

`MBeanServerConnection` provides several methods that let you query the remote MBean server and invoke methods on the MBeans contained within it. For example, say that we'd like to know how many MBeans are registered in the remote MBean server. The following code snippet will print that information:

```
int mbeanCount = mbeanServerConnection.getMBeanCount();
print ("There are " + mbeanCount + " MBeans");
```

And you may also query the remote server for all of the MBean names using the `queryNames()` method:

```
java.util.Set mbeanNames =
  mbeanServerConnection.queryNames(null, null);
```

The two parameters passed to `queryNames()` are used to refine the results. By passing in `null` for both parameters, we're asking for the names of all registered MBeans.

Querying the remote MBean server for bean counts and names is interesting, but doesn't get much work done. The real value of accessing an MBean server remotely is found in accessing attributes and invoking operations on the MBeans that are registered in the remote server.

For accessing MBean attributes, you'll want to use the `getAttribute()` and `setAttribute()` methods. For example, to retrieve the value of an MBean, you'd call the `getAttribute()` method like so:

```
String cronExpression = mbeanServerConnection.getAttribute(
    new ObjectName("rantz:name=emailSchedule"),
    "cronExpression");
```

Changing the value of an MBean's property is similarly done using the `setAttribute()` method:

```
mbeanServerConnection.setAttribute(
    new ObjectName("rantz:name=emailSchedule"),
    new Attribute("cronExpression", "0 59 23 * * ?"));
```

If you'd like to invoke a method on a remote MBean, the `invoke()` method is what you'll call. Here's how you might invoke the `sendDailyRantEmails()` method on the RantService MBean:

```
mbeanServerConnection.invoke(
    new ObjectName("rantz:name=rantService"),
    "sendDailyRantEmails",
    new Object[] {},
    new String[] {""]);
```

And there are dozens of other things you can do with remote MBeans using the `MBeanServerConnection` provided by `MBeanServerConnectionFactoryBean`. I'll leave it to you to explore the possibilities.

However, invoking methods and setting attributes on remote MBeans is awkward through the API offered through `MBeanServerConnection`. Doing something as simple as calling the `sendDailyRantEmails()` method involves creating an `ObjectName` instance, and passing in several parameters to the `invoke()` method is not nearly as intuitive as simply calling the `sendDailyRantEmails()` method directly. For a more direct approach, we'll need to proxy the remote MBean.

Proxying MBeans

Spring's `MBeanProxyFactoryBean` is a proxy factory bean in the same vein as the remoting proxy factory beans we examined in chapter 8. But instead of providing proxy-based access to remote beans via RMI or Hessian/Burlap, `MBeanProxyFactoryBean` lets you access remote MBeans directly (as if they were any other locally configured bean). Figure 12.11 illustrates how this works.

For example, consider the following declaration of `MBeanProxyFactoryBean`:

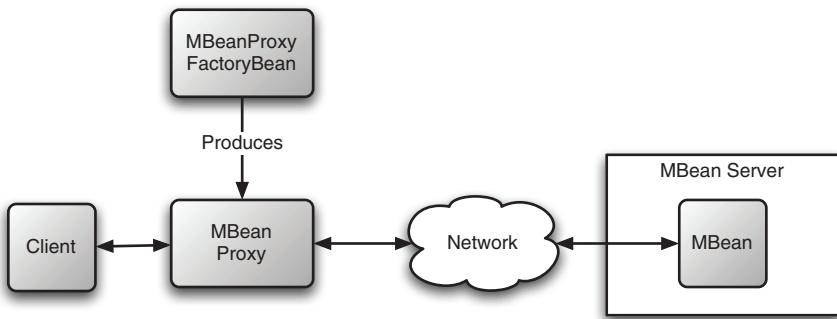


Figure 12.11 `MBeanProxyFactoryBean` produces a proxy to a remote MBean. The proxy's client can then interact with the remote MBean as if it were a locally configured POJO.

```

<bean id="remoteRantServiceMBean"
      class="org.springframework.jmx.access.MBeanProxyFactoryBean">
    <property name="objectName" value="rantz:name=RantService" />
    <property name="server" ref="mBeanServerClient" />
    <property name="proxyInterface"
              value="com.rodrantz.service.mbean.RantServiceRemoteMBean" />
</bean>
  
```

The `objectName` property specifies the object name of the remote MBean that is to be proxied locally. Here it's referring to the MBean that is exported from the `RantServiceImpl` bean.

The `server` property refers to an `MBeanServerConnection` through which all communication with the MBean is routed. Here I've wired in the `MBeanServerConnectionFactoryBean` that we configured a page or so ago.

Finally, the `proxyInterface` property specifies the interface that will be implemented by the proxy. In this case, it is the `RantServiceRemoteMBean` interface:

```

public interface RantServiceRemoteMBean {
    void sendDailyRantEmails();
}
  
```

With the `remoteRantServiceMBean` bean declared, you can wire it into a `RantServiceRemoteMBean` property of any class that needs to access the remote MBean. Then you'll be able to invoke the `sendDailyRantEmails()` method directly like this:

```

remoteRantServiceMBean.sendDailyRantEmails();
  
```

We've now seen several ways that we can communicate with MBeans and are now able to view and tweak our Spring bean configuration while the application is running. But thus far it's been a one-sided conversation. We've talked to the MBeans, but the MBeans haven't been able to get a word in edgewise. It's now time for us to hear to what they have to say by listening for notifications.

12.4.3 Handling notifications

Querying an MBean for information is only one way of keeping an eye on the state of an application. It is not, however, the most efficient way to be informed of significant events within the application.

For example, suppose you want to know the precise moment that the one-millionth motorist registers to the RoadRantz applications. You could add an appropriate managed attribute to the RantService MBean and continually query the MBean, waiting for the one-millionth motorist. But you know what they say about a watched pot and when it boils—you could end up wasting a great deal of time querying the MBean only to have the one-millionth motorist arrive while you're away at lunch.

Instead of asking the MBean if the one-millionth motorist has registered, a better approach would be to have the MBean notify you once the momentous milestone has been achieved. JMX notifications, as shown in figure 12.12, are a way that MBeans can communicate with the outside world proactively, instead of waiting for an external application to query them.

Spring's support for sending notifications comes in the form of the `NotificationPublisherAware` interface. Any bean-turned-MBean that wishes to send notifications should implement this interface. Here are the pertinent changes required to enable the `RantServiceImpl` class to publish notifications:

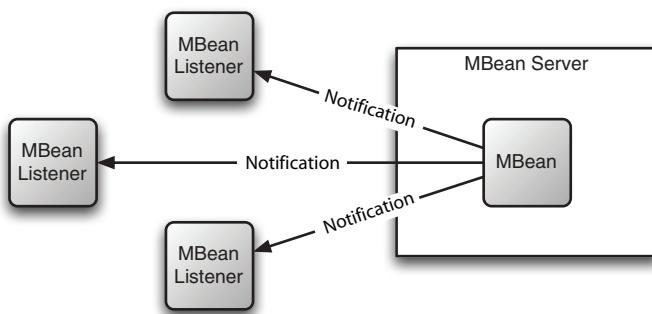


Figure 12.12
JMX notifications enable MBeans to communicate proactively with the outside world.

```

public class RantServiceImpl implements RantService,
    NotificationPublisherAware {
    ...
    private NotificationPublisher notificationPublisher;
    public void setNotificationPublisher(
        NotificationPublisher notificationPublisher) {
        this.notificationPublisher = notificationPublisher;
    }
}

```

The `NotificationPublisherAware` interface only demands a single method to be implemented: `setNotificationPublisher()`. The `setNotificationPublisher()` method is used to inject a `NotificationPublisher` into the `RantServiceImpl`. Here we've wired `ModelMBeanNotificationPublisher` as an inner bean to the `notificationPublisher` property:

```

<bean id="rantService"
    class="com.roaddrantz.service.RantServiceImpl">
    ...
    <property name="notificationPublisher">
        <bean class="org.springframework.jmx.export.notification.
            ModelMBeanNotificationPublisher" />
    </property>
</bean>

```

With a `NotificationPublisher` object at hand, we are now able to write the code that sends a notification when the one-millionth motorist registers. The following `checkForOneMillionthMotorist()` method should do the trick:

```

private void checkForOneMillionthMotorist() {
    if(rantDao.getMotoristCount() == 1000000) {
        notificationPublisher.sendNotification(
            new Notification(
                "RantService.OneMillionMotorists", this, 0));
    }
}

```

After determining that, in fact, the one-millionth motorist has just been added, the `checkForOneMillionthMotorist()` method constructs a new JMX `Notification` object and uses the `NotificationPublisher's sendNotification()` method to publish the notification.

Once the `sendNotification()` method is called, the notification is on its way to... hmm... it seems that we haven't decided who will receive the notification yet. Let's set up a notification listener to listen to and react to the notification.

Listening for notifications

The standard way to receive MBean notifications is to implement the javax.management.NotificationListener interface. For example, consider PagingNotificationListener:

```
package com.roadrantz.service.mbean;
import javax.management.Notification;
import javax.management.NotificationListener;

public class PagingNotificationListener
    implements NotificationListener {
    public PagingNotificationListener() {}

    public void handleNotification(Notification notification,
        Object handback) {
        ... // send pager message
    }
}
```

PagingNotificationListener is a typical JMX notification listener. When a notification is received, the handleNotification() method will be invoked to react to the notification. Presumably, PagingNotificationListener's handleNotification() method will send a message to a pager or cell phone about the one-millionth motorist. (I've left the actual implementation to the reader's imagination.)

The only thing left to do is register PagingNotificationListener with the MBeanExporter:

```
<bean class="org.springframework.jmx.export.MBeanExporter">
...
<property name="notificationListenerMappings">
    <map>
        <entry key="rantz:name=rantService">
            <bean class=
                "com.roadrantz.service.mbean.
                    ↗ PagingNotificationListener" />
        </entry>
    </map>
</property>
</bean>
```

MBeanExporter's notificationListenerMappings property is used to map notification listeners to the MBeans that they'll be listening to. In this case, we've set up PagingNotificationListener to listen to any notifications published by the rantService MBean.

12.5 **Summary**

In this chapter, we've added new email capabilities to the RoadRantz application. In doing so, we've explored a few of tidbits from among Spring's enterprise abstraction APIs.

Although configuration through dependency injection is one of Spring's strong points, sometimes it's preferable to store certain configuration information or application objects outside of the Spring application context. JDBC data sources, for example, are often configured within an application server and made accessible through JNDI. Fortunately, as you've seen, Spring's JNDI abstraction makes short work of injecting JNDI-managed objects into Spring-managed beans.

You've also seen how Spring's JavaMail abstraction simplifies the sending of emails by enabling you to configure a mail sender bean in Spring. The mail sender can then be injected into any application object that needs to send email.

Oftentimes it is necessary for an application to perform specific tasks on a schedule. In the RoadRantz application, we used Spring's scheduling support to periodically send emails to registered motorists. Keeping with the Spring theme of choice, Spring supports scheduling through a variety of scheduling APIs, including Java's Timer object and OpenSymphony's Quartz.

Finally, you saw how Spring enables management of beans through its JMX abstraction API. Using Spring JMX, we were able to expose Spring-configured beans as MBeans suitable for management through JMX.

Our journey through Spring's enterprise APIs and abstractions is now at an end. We've covered a lot of ground, including database persistence, declarative transactions and security, remoting, web services, asynchronous messaging, and EJBs. These are the heart and mind of many applications, working behind the scenes to get the job done.

The enterprise technologies may keep an application humming, but it's what's on the screen that's of the most concern to your application's users. In the next part of this book, we're going to look at several ways to build the user-facing portion of an application using Spring. We'll start in the next chapter by looking at Spring's own web framework, Spring MVC.

Part 3

Client-side Spring

Now that you've seen how to build the business layer of a Spring application, it's time to put a face on it.

In chapter 13, "Handling web requests," you'll learn the basics of using Spring MVC, a web framework built on the principles of the Spring Framework. You'll discover Spring MVC's vast selection of controllers for handling web requests and see how to transparently bind request parameters to your business objects while providing validation and error handling at the same time.

Once a request has been handled, you'll likely want to show the results to the user. Chapter 14, "Rendering web views," will pick up where chapter 13 left off by showing you how to match controller output to JSP, Velocity, and FreeMarker views. You'll also learn how to lay out your pages using Tiles and how to produce PDF, Excel, and RSS output in Spring.

Building on what you learned in chapters 13 and 14, chapter 15, "Using Spring Web Flow," will show you how to build conversational, flow-based web applications using the Spring Web Flow framework.

Although Spring MVC is a fantastic web framework, you may already have a different framework in mind for your application's web layer. In chapter 16, "Integrating with other web frameworks," you'll see how to use frameworks such as Struts, WebWork, Tapestry, and JSF to front your Spring application.

Handling web requests

13

This chapter covers

- Mapping requests to Spring controllers
- Transparently binding form parameters
- Validating form submissions
- Mapping exceptions to views

As a JEE developer, you have more than likely developed a web-based application. In fact, for many Java developers, web-based applications are their primary focus. If you do have this type of experience, you are well aware of the challenges that come with these systems. Specifically, state management, workflow, and validation are all important features that need to be addressed. None of these is made any easier given the HTTP protocol's stateless nature.

Spring's web framework is designed to help you address these concerns. Based on the Model-View-Controller (MVC) pattern, Spring MVC helps you build web-based applications that are as flexible and as loosely coupled as the Spring Framework itself.

In this chapter and the one that follows, we'll explore the Spring MVC web framework. In this chapter, we'll focus on the parts of Spring MVC that process requests. You'll see how to extend Spring's rich set of controller objects to handle virtually any web functionality required in your application. You'll also see how Spring's handler mapping makes easy work of associating URL patterns with specific controller implementations. Chapter 14 will pick up where this chapter leaves off by showing you how to use Spring MVC views to send a response back to the user.

Before we go too deep with the specifics of Spring MVC's controllers and handler mappings, let's start with a high-level view of Spring MVC and build our first complete bit of web functionality.

13.1 Getting started with Spring MVC

Have you ever seen the children's game Mousetrap? It's a crazy game. The goal is to send a small steel ball over a series of wacky contraptions in order to trigger a mousetrap. The ball goes over all kinds of intricate gadgets, from rolling down a curvy ramp to getting sprung off a teeter-totter to spinning on a miniature Ferris wheel to being kicked out of a bucket by a rubber boot. It goes through all of this to spring a trap on a poor, unsuspecting plastic mouse.

At first glance, you may think that Spring's MVC framework is a lot like Mousetrap. Instead of moving a ball around through various ramps, teeter-totters, and wheels, Spring moves requests around between a dispatcher servlet, handler mappings, controllers, and view resolvers.

But don't draw too strong of a comparison between Spring MVC and the Rube Goldberg-esque game of Mousetrap. Each of the components in Spring MVC performs a specific purpose. Let's start the exploration of Spring MVC by examining the lifecycle of a typical request.

13.1.1 A day in the life of a request

Every time that a user clicks a link or submits a form in their web browser, a request goes to work. A request's job description is that of a courier. Just like a postal carrier or a Federal Express delivery person, a request lives to carry information from one place to another.

The request is a busy fellow. From the time that it leaves the browser until the time that it returns a response, it will make several stops, each time dropping off a bit of information and picking up some more. Figure 13.1 shows all the stops that the request makes.

When the request leaves the browser, it carries information about what the user is asking for. At very least, the request will be carrying the requested URL. But it may also carry additional data such as the information submitted in a form by the user.

The first stop in the request's travels is Spring's DispatcherServlet ①. Like most Java-based MVC frameworks, Spring MVC funnels requests through a single front controller servlet. A front controller is a common web-application pattern where a single servlet delegates responsibility for a request to other components of an application to perform the actual processing. In the case of Spring MVC, DispatcherServlet is the front controller.

The DispatcherServlet's job is to send the request on to a Spring MVC controller. A controller is a Spring component that processes the request. But a typical application may have several controllers and DispatcherServlet needs help deciding which controller to send the request to. So, the DispatcherServlet

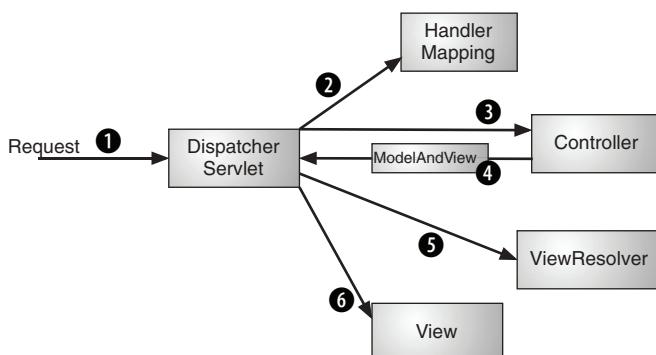


Figure 13.1 A request is dispatched by `DispatcherServlet` to a controller (which is chosen through a handler mapping). Once the controller is finished, the request is then sent to a view (which is chosen through a `ViewResolver`) to render output.

consults one or more handler mappings ② to figure out where the request’s next stop will be. The handler mapping will pay particular attention to the URL carried by the request when making its decision.

Once an appropriate controller has been chosen, `DispatcherServlet` sends the request on its merry way to the chosen controller. ③ At the controller, the request will drop off its payload (the information submitted by the user) and patiently wait for the controller to process that information. (Actually, a well-designed Controller performs little or no processing itself and instead delegates responsibility for the business logic to one or more service objects.)

The logic performed by a controller often results in some information that needs to be carried back to the user and displayed in the browser. This information is referred to as the *model*. But sending raw information back to the user isn’t sufficient—it needs to be formatted in a user-friendly format, typically HTML. For that the information needs to be given to a *view*, typically a JSP.

So, the last thing that the controller will do is package up the model data and the name of a view into a `ModelAndView` object. ④ It then sends the request, along with its new `ModelAndView` parcel, back to the `DispatcherServlet`. As its name implies, the `ModelAndView` object contains both the model data as well as a hint to what view should render the results.

So that the controller isn’t coupled to a particular view, the `ModelAndView` doesn’t carry a reference to the actual JSP. Instead it only carries a logical name that will be used to look up the actual view that will produce the resulting HTML. Once the `ModelAndView` is delivered to the `DispatcherServlet`, the `DispatcherServlet` asks a view resolver to help find the actual JSP. ⑤

Now that the `DispatcherServlet` knows which view will render the results, the request’s job is almost over. Its final stop is at the view implementation (probably a JSP) where it delivers the model data. ⑥ With the model data delivered to the view, the request’s job is done. The view will use the model data to render a page that will be carried back to the browser by the (not-so-hard-working) response object.

We’ll discuss each of these steps in more detail throughout this and the next chapter. But first things first—you’ll need to configure `DispatcherServlet` to use Spring MVC.

13.1.2 Configuring `DispatcherServlet`

At the heart of Spring MVC is `DispatcherServlet`, a servlet that functions as Spring MVC’s front controller. Like any servlet, `DispatcherServlet` must be configured in your web application’s `web.xml` file. Place the following `<servlet>` declaration in your application’s `web.xml` file:

```
<servlet>
  <servlet-name>roadrantz</servlet-name>
  <servlet-class>org.springframework.web.servlet.DispatcherServlet
  </servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>
```

The `<servlet-name>` given to the servlet is significant. By default, when `DispatcherServlet` is loaded, it will load the Spring application context from an XML file whose name is based on the name of the servlet. In this case, because the servlet is named `roadrantz`, `DispatcherServlet` will try to load the application context from a file named `roadrantz-servlet.xml`.

Next you must indicate what URLs will be handled by the `DispatcherServlet`. Add the following `<servlet-mapping>` to `web.xml` to let `DispatcherServlet` handle all URLs that end in `.htm`:

```
<servlet-mapping>
  <servlet-name>roadrantz</servlet-name>
  <url-pattern>*.htm</url-pattern>
</servlet-mapping>
```

So, you're probably wondering why we chose this particular URL pattern. It could be because all of the content produced by our application is HTML. It could also be because we want to fool our friends into thinking that our entire application is composed of static HTML files. And it could be that we think `.do` is a silly extension.

But the truth of the matter is that the URL pattern is somewhat arbitrary and we could've chosen any URL pattern for `DispatcherServlet`. Our main reason for choosing `*.htm` is that this pattern is the one used by convention in most Spring MVC applications that produce HTML content. The reasoning behind this convention is that the content being produced is HTML and so the URL should reflect that fact.

Now that `DispatcherServlet` is configured in `web.xml` and given a URL mapping, you are ready to start writing the web layer of your application. However, there's still one more thing that we recommend you add to `web.xml`.

Breaking up the application context

As we mentioned earlier, `DispatcherServlet` will load the Spring application context from a single XML file whose name is based on its `<servlet-name>`. But this doesn't mean that you can't split your application context across multiple XML files. In fact, we recommend that you split your application context across application layers, as shown in figure 13.2.

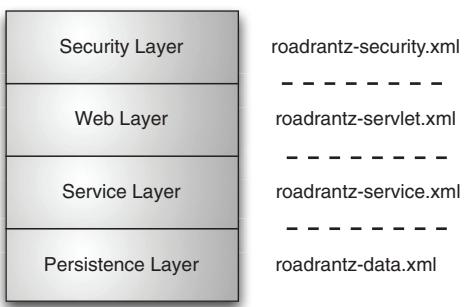


Figure 13.2
Breaking an application into separate tiers helps to cleanly divide responsibility. Security-layer code secures the application, web-layer code is focused on user interaction, service-layer code is focused on business logic, and persistence-layer code deals with database concerns.

As configured, `DispatcherServlet` already loads `roadrantz-servlet.xml`. You could put all of your application's `<bean>` definitions in `roadrantz-servlet.xml`, but eventually that file would become quite unwieldy. Splitting it into logical pieces across application layers can make maintenance easier by keeping each of the Spring configuration files focused on a single layer of the application. It also makes it easy to swap out a layer configuration without affecting other layers (swapping out a `roadrantz-data.xml` file that uses Hibernate with one that uses iBATIS, for example).

Because `DispatcherServlet`'s configuration file is `roadrantz-servlet.xml`, it makes sense for this file to contain `<bean>` definitions pertaining to controllers and other Spring MVC components. As for beans in the service and data layers, we'd like those beans to be placed in `roadrantz-service.xml` and `roadrantz-data.xml`, respectively.

Configuring a context loader

To ensure that all of these configuration files are loaded, you'll need to configure a context loader in your `web.xml` file. A context loader loads context configuration files in addition to the one that `DispatcherServlet` loads. The most commonly used context loader is a servlet listener called `ContextLoaderListener` that is configured in `web.xml` as follows:

```
<listener>
  <listener-class>org.springframework.
    web.context.ContextLoaderListener</listener-class>
</listener>
```

NOTE Some web containers do not initialize servlet listeners before servlets—which is important when loading Spring context definitions. If your application is going to be deployed to an older web container that adheres to Servlet 2.2 or if the web container is a Servlet 2.3 container

that does not initialize listeners before servlets, you'll want to use `ContextLoaderServlet` instead of `ContextLoaderListener`.

With `ContextLoaderListener` configured, you'll need to tell it the location of the Spring configuration file(s) to load. If not specified otherwise, the context loader will look for a Spring configuration file at `/WEB-INF/applicationContext.xml`. But this location doesn't lend itself to breaking up the application context across application layers, so you'll probably want to override this default.

You can specify one or more Spring configuration files for the context loader to load by setting the `contextConfigLocation` parameter in the servlet context:

```
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>
    /WEB-INF/roadrantz-service.xml
    /WEB-INF/roadrantz-data.xml
    /WEB-INF/roadrantz-security.xml
  </param-value>
</context-param>
```

The `contextConfigLocation` parameter is specified as a list of paths (relative to the web application root). As configured here, the context loader will use `contextConfigLocation` to load three context configuration files—one for the security layer, one for the service layer, and one for the data layer.

`DispatcherServlet` is now configured and ready to dispatch requests to the web layer of your application. But the web layer hasn't been built yet! Don't fret. We'll build much of the web layer in this chapter. Let's start by getting an overview of how all the pieces of Spring MVC are assembled to produce web functionality.

13.1.3 Spring MVC in a nutshell

Every web application has a homepage. It's necessary to have a starting point in the application. It gives the user a place to launch from and a familiar place to return when they get lost. Otherwise, they would flail around, clicking links, getting frustrated, and probably ending up leaving and going to some other website.

The RoadRantz application is no exception to the homepage phenomenon. There's no better place to start developing the web layer of our application than with the homepage. In building the homepage, we get a quick introduction to the nuts and bolts of Spring MVC.

As you'll recall from the requirements for RoadRantz, the homepage should display a list of the most recently entered rants. The following list of steps defines the bare minimum that you must do to build the homepage in Spring MVC:

- 1 Write the controller class that performs the logic behind the homepage. The logic involves using a RantService to retrieve the list of recent rants.
- 2 Configure the controller in the DispatcherServlet's context configuration file (roadrantz-servlet.xml).
- 3 Configure a view resolver to tie the controller to the JSP.
- 4 Write the JSP that will render the homepage to the user.

The first step is to build a controller object that will handle the homepage request. So with no further delay, let's write our first Spring MVC controller.

Building the controller

When you go out to eat at a nice restaurant, the person you'll interact with the most is the waiter or waitress. They'll take your order, hand it off to the cooks in the kitchen to prepare, and ultimately bring out your meal. And if they want a decent tip, they'll offer a friendly smile and keep the drinks filled. Although you know that other people are involved in making your meal a pleasant experience, the waiter or waitress is your interface to the kitchen.

Similarly, in Spring MVC, a controller is a class that is your interface to the application's functionality. As shown in figure 13.3, a controller receives the request, hands it off to service classes for processing, then ultimately collect the results in a page that is returned to you in your web browser. In this respect, a controller isn't much different than an HttpServlet or a Struts Action.

The homepage controller of the RoadRantz application is relatively simple. It takes no request parameters and simply produces a list of recently entered rants for display on the homepage. Listing 13.1 shows HomePageController, a Spring MVC controller that implements the homepage functionality.

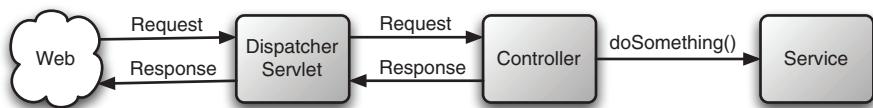


Figure 13.3 A controller handles web requests on behalf of the DispatcherServlet. A well-designed controller doesn't do all of the work itself—it delegates to a service layer object for business logic.

Listing 13.1 `HomePageController`, which retrieves a list of recent rants for display on the homepage

```
package com.roaddrantz.mvc;
import java.util.List;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.AbstractController;
import com.roaddrantz.service.RantService;

public class HomePageController extends AbstractController {
    public HomePageController() {}

    protected ModelAndView handleRequestInternal(
        HttpServletRequest request, HttpServletResponse response)
        throws Exception {
        List recentRants = rantService.getRecentRants();   ↪ Retrieves list of rants
        return new ModelAndView("home",   ↪ Goes to "home" view
            "rants", recentRants);   ↪ Returns rants in model
    }

    private RantService rantService;
    public void setRantService(RantService rantService) {
        this.rantService = rantService;
    }
}
```

**Injects
RantService**

Where a Spring MVC controller differs from a servlet or a Struts Action is that it is configured as just another JavaBean in the Spring application context. This means you can take full advantage of dependency injection (DI) and Spring AOP with a controller class just as you would with any other bean.

In the case of `HomePageController`, DI is used to wire in a `RantService`. `HomePageController` delegates responsibility for retrieving the list of recent rants to the `RantService`.

Introducing ModelAndView

After the chef has prepared your meal, the waiter/waitress will pick it up and bring it to your table. On the way out, the last thing that they may do is add some final garnishments—perhaps a sprig of parsley.

Once the business logic has been completed by the service objects, it's time for the controller to send the results back to the browser. The last thing that `handleRequestInternal()` does is to return a `ModelAndView` object. The `ModelAndView` class represents an important concept in Spring MVC. In fact, every controller

execution method must return a `ModelAndView`. So, let's take a moment to understand how this important class works.

A `ModelAndView` object, as its name implies, fully encapsulates the view and model data that is to be displayed by the view. In the case of `HomePageController`, the `ModelAndView` object is constructed as follows:

```
new ModelAndView("home", "rants", recentRants);
```

The first parameter of this `ModelAndView` constructor is the logical name of a view component that will be used to display the output from this controller. Here the logical name of the view is `home`. A view resolver will use this name to look up the actual `View` object (you'll learn more about `Views` and view resolvers later in chapter 14).

The next two parameters represent the model object that will be passed to the view. These two parameters act as a name-value pair. The second parameter is the name of the model object given as the third parameter. In this case, the list of rants in the `recentRants` variable will be passed to the view with a name of `rants`.

Configuring the controller bean

Now that `HomePageController` has been written, it is time to configure it in the `DispatcherServlet`'s context configuration file (which is `roadrantz-servlet.xml` for the `RoadRantz` application). The following chunk of XML declares the `HomePageController`:

```
<bean name="/home.htm"
      class="com.roadrantz.mvc.HomePageController">
    <property name="rantService" ref="rantService" />
</bean>
```

As mentioned before, the `rantService` property is to be injected with an implementation of the `RantService` interface. In this `<bean>` declaration, we've wired the `rantService` property with a reference to another bean named `rantService`. The `rantService` bean itself is declared elsewhere (in `roadrantz-service.xml`, to be precise).

One thing that may have struck you as odd is that instead of specifying a `bean id` for the `HomePageController` bean, we've specified a name. And to make things even weirder, instead of giving it a real name, we've given it a URL pattern of `/home.htm`. Here the `name` attribute is serving double duty as both the name of the bean and a URL pattern for requests that should be handled by this controller. Because the URL pattern has special characters that are not valid in

an XML `id` attribute—specifically, the slash (/) character—the `name` attribute had to be used instead of `id`.

When a request comes to `DispatcherServlet` with a URL that ends with `/home.htm`, `DispatcherServlet` will dispatch the request to `HomePageController` for handling. Note, however, that the only reason that the bean's `name` attribute is used as the URL pattern is because we haven't configured a handler-mapping bean. The default handler mapping used by `DispatcherServlet` is `BeanNameUrlHandlerMapping`, which uses the base name as the URL pattern. Later (in section 13.2), you'll see how to use some of Spring's other handler mappings that let you decouple a controller's bean name from its URL pattern.

Declaring a view resolver

On the way back to the web browser, the results of the web operation need to be presented in a human-friendly format. Just like a waiter may place a sprig of parsley on a plate to make it more presentable, the resulting list of rants needs to be dressed up a bit before presenting it to the client. For that, we'll use a JSP page that will render the results in a user-friendly format.

But how does Spring know which JSP to use for rendering the results? As you'll recall, one of the values returned in the `ModelAndView` object is a logical view name. While the logical view name doesn't directly reference a specific JSP, it can be used to indirectly deduce which JSP to use.

To help Spring MVC figure out which JSP to use, you'll need to declare one more bean in `rodrantz-servlet.xml`: a view resolver. Put simply, a view resolver's job is to take the view name returned in the `ModelAndView` and map it to a view. In the case of `HomePageController`, we need a view resolver to resolve `home` (the logical view name returned in the `ModelAndView`) to a JSP file that renders the homepage.

As you'll see in section 13.4, Spring MVC comes with several view resolvers from which to choose. But for views that are rendered by JSP, there's none simpler than `InternalResourceViewResolver`:

```
<bean id="viewResolver"
      class="org.springframework.web.
      ↗ servlet.view.InternalResourceViewResolver">
  <property name="prefix">
    <value>/WEB-INF/jsp/</value>
  </property>
  <property name="suffix">
    <value>.jsp</value>
  </property>
</bean>
```

InternalResourceViewResolver prefixes the view name returned in the ModelAndView with the value of its prefix property and suffixes it with the value from its suffix property. Since HomePageController returns a view name of home in the ModelAndView, InternalResourceViewResolver will find the view at /WEB-INF/jsp/home.jsp.

Creating the JSP

We've written a controller that will handle the homepage request and have configured it in the Spring application context. It will consult with a RantService bean to look up the most recently added rants. And when it's done, it will send the results on to a JSP. So now we only have to create the JSP that renders the homepage. The JSP in listing 13.2 iterates over the list of rants and displays them on the home page.

Listing 13.2 home.jsp, which displays a list of recent rants

```
<%@ page contentType="text/html" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jstl/core" %>

<html>
    <head><title>Rantz</title></head>

    <body>
        <h2>Welcome to RoadRantz!</h2>

        <h3>Recent rantz:</h3>
        <ul>
            <c:forEach items="${rants}" var="rant">
                <li><c:out value="${rant.vehicle.state}"/> /
                    <c:out value="${rant.vehicle.plateNumber}"/> --
                    <c:out value="${rant.rantText}"/>
            </li>
            </c:forEach>
        </ul>
    </body>
</html>
```

**Iterates over
list of rants**

Although we've left out any aesthetic elements in home.jsp for brevity's sake, it still serves to illustrate how the model data returned in ModelAndView can be used in the view. In HomePageController, we placed the list of rants in a model property named rants. When home.jsp is rendering the homepage, it references the list of rants as \${rants}.

Be sure to name this JSP `home.jsp` and to place it in the `/WEB-INF/jsp` folder in your web application. That's where `InternalResourceViewResolver` will try to find it.

Putting it all together

The homepage is now complete. You've written a controller to handle requests for the homepage, configured it to rely on `BeanNameUrlHandlerMapping` to have a URL pattern of `/home.htm`, written a simple JSP that represents the homepage, and configured a view resolver to find the JSP. Now, how does this all fit together?

Figure 13.4 shows the steps that a request for `/home.htm` will go through given the work done so far.

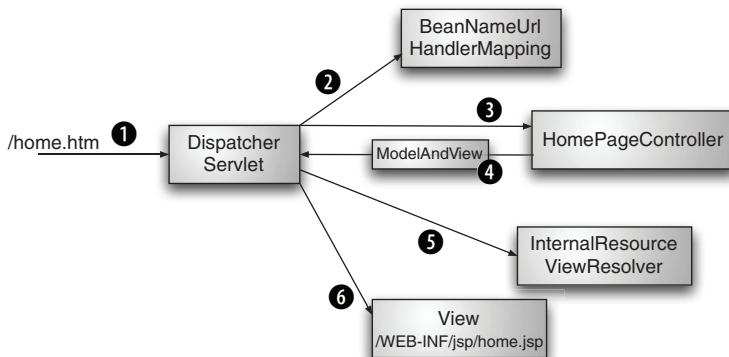


Figure 13.4 A homepage request is sent by `DispatcherServlet` to the `HomePageController` (as directed by `BeanNameUrlHandlerMapping`). When finished, `InternalResourceViewResolver` directs the request to `home.jsp` to render the homepage.

To recap this process:

- 1 `DispatcherServlet` receives a request whose URL pattern is `/home.htm`.
- 2 `DispatcherServlet` consults `BeanNameUrlHandlerMapping` to find a controller whose bean name is `/home.htm`; it finds the `HomePageController` bean.
- 3 `DispatcherServlet` dispatches the request to `HomePageController` for processing.
- 4 `HomePageController` returns a `ModelAndView` object with a logical view name of `home` and a list of rants in a property called `rants`.
- 5 `InternalResourceViewResolver` directs the request to `home.jsp` to render the homepage.

- 5 DispatcherServlet consults its view resolver (configured as InternalResourceViewResolver) to find a view whose logical name is home. InternalResourceViewResolver returns the path to /WEB-INF/jsp/home.jsp.
- 6 DispatcherServlet forwards the request to the JSP at /WEB-INF/jsp/home.jsp to render the homepage to the user.

Now that you've seen the big picture of Spring MVC, let's slow down a bit and take a closer look at each of the moving parts involved in servicing a request. We'll start where it all begins—with handler mappings.

13.2 **Mapping requests to controllers**

When a courier has a package that is to be delivered to a particular office within a large office building, they'll need to know how to find the office. In a large office building with many tenants, this would be tricky if it weren't for a building directory. The building directory is often located near the elevators and helps anyone unfamiliar with the building locate the floor and suite number of the office they're looking for.

In the same way, when a request arrives at the DispatcherServlet, there needs to be some directory to help figure out how the request should be dispatched. Handler mappings help DispatcherServlet figure out which controller the request should be sent to. Handler mappings typically map a specific controller bean to a URL pattern. This is similar to how URLs are mapped to servlets using a <servlet-mapping> element in a web application's web.xml file or how Actions in Jakarta Struts are mapped to URLs using the path attribute of <action> in struts-config.xml.

In the previous section, we relied on the fact that DispatcherServlet defaults to use BeanNameUrlHandlerMapping. BeanNameUrlHandlerMapping was fine to get started, but it may not be suitable in all cases. Fortunately, Spring MVC offers several handler-mapping implementations to choose from.

All of Spring MVC's handler mappings implement the org.springframework.web.servlet.HandlerMapping interface. Spring comes prepackaged with four useful implementations of HandlerMapping, as listed in table 13.1.

You've already seen an example of how BeanNameUrlHandlerMapping works (as the default handler mapping used by DispatcherServlet). Let's look at how to use each of the other handler mappings, starting with SimpleUrlHandlerMapping.

Table 13.1 Handler mappings help DispatcherServlet find the right controller to handle a request.

Handler mapping	How it maps requests to controllers
BeanNameUrlHandlerMapping	Maps controllers to URLs that are based on the controllers' bean name.
SimpleUrlHandlerMapping	Maps controllers to URLs using a property collection defined in the Spring application context.
ControllerClassNameHandlerMapping	Maps controllers to URLs by using the controller's class name as the basis for the URL.
CommonsPathMapHandlerMapping	Maps controllers to URLs using source-level metadata placed in the controller code. The metadata is defined using Jakarta Commons Attributes (http://jakarta.apache.org/commons/attributes).

13.2.1 Using SimpleUrlHandlerMapping

SimpleUrlHandlerMapping is probably one of the most straightforward of Spring's handler mappings. It lets you map URL patterns directly to controllers without having to name your beans in a special way.

For example, consider the following declaration of SimpleUrlHandlerMapping that associates several of the RoadRantz application's controllers with their URL patterns:

```
<bean id="simpleUrlMapping" class=
    "org.springframework.web.servlet.handler.
     SimpleUrlHandlerMapping">
    <property name="mappings">
        <props>
            <prop key="/home.htm">homePageController</prop>
            <prop key="/rantsForVehicle.htm">
                <!-- rantsForVehicleController-->
            <prop key="/rantsForVehicle.rss">
                <!-- rantsForVehicleControllerRss-->
            <prop key="/rantsForDay.htm">rantsForDayController</prop>
            <prop key="/login.htm">loginController</prop>
            <prop key="/register.htm">registerMotoristController</prop>
            <prop key="/addRant.htm">addRantController</prop>
        </props>
    </property>
</bean>
```

SimpleUrlHandlerMapping's `mappings` property is wired with a `java.util.Properties` using `<props>`. The `key` attribute of each `<prop>` element is a URL pattern.

Just as with `BeanNameUrlHandlerMapping`, all URL patterns are relative to `DispatcherServlet`'s `<servlet-mapping>`. URL. The value of each `<prop>` is the bean name of a controller that will handle requests to the URL pattern.

In case you're wondering where all of those other controllers came from, just hang tight. By the time this chapter is done, we'll have seen most of them. But first, let's explore another way to declare controller mappings using the class names of the controllers.

13.2.2 Using `ControllerClassNameHandlerMapping`

Oftentimes you'll find yourself mapping your controllers to URL patterns that are quite similar to the class names of the controllers. For example, in the `RoadRantz` application, we're mapping `rantsForVehicle.htm` to `RantsForVehicleController` and `rantsForDay.htm` to `RantsForDayController`.

Notice a pattern? In those cases, the URL pattern is the same as the name of the controller class, dropping the `Controller` portion and adding `.htm`. It seems that with a pattern like that it would be possible to assume a certain default for the mappings and not require explicit mappings.

In fact, that's roughly what `ControllerClassNameHandlerMapping` does:

```
<bean id="urlMapping"
      class="org.springframework.web.servlet.mvc.
      ↗ ControllerClassNameHandlerMapping" />
```

By configuring `ControllerClassNameHandlerMapping`, you are telling Spring's `DispatcherServlet` to map URL patterns to controllers following a simple convention. Instead of explicitly mapping each controller to a URL pattern, Spring will automatically map controllers to URL patterns that are based on the controller's class name. Figure 13.5 illustrates how `RantsForVehicleController` will be mapped.

Put simply, to produce the URL pattern, the `Controller` portion of the controller's class name is removed (if it exists), the remaining text is lowercased, a slash (/) is added to the beginning, and ".htm" is added to the end to produce the URL pattern. Consequently, a controller bean whose class is `RantsForVehicleController` will be mapped to `/rantsforvehicle.htm`. Notice that the entire

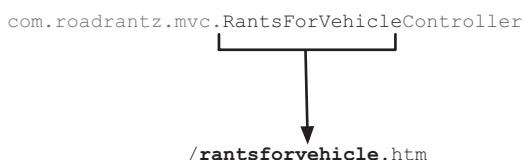


Figure 13.5
`ControllerClassNameHandlerMapping` maps a request to a controller by stripping `Controller` from the end of the class name and normalizing it to lowercase.

URL pattern is lowercased, which is slightly different from the convention we were following with `SimpleUrlHandlerMapping`.

13.2.3 Using metadata to map controllers

The final handler mapping we'll look at is `CommonsPathMapHandlerMapping`. This handler mapping considers source-level metadata placed in a controller's source code to determine the URL mapping. In particular, the metadata is expected to be an `org.springframework.web.servlet.handler.commonattributes.PathMap` attribute compiled into the controller using the Jakarta Commons Attributes compiler.

To use `CommonsPathMapHandlerMapping`, simply declare it as a `<bean>` in your context configuration file as follows:

```
<bean id="urlMapping" class="org.springframework.web.  
    servlet.handler.metadata.CommonsPathMapHandlerMapping"/>
```

Then tag each of your controllers with a `PathMap` attribute to declare the URL pattern for the controller. For example, to map `HomePageController` to `/home.htm`, tag `HomePageController` as follows:

```
/**  
 * @org.springframework.web.servlet.handler.  
 *   commonsattributes.PathMap("/home.htm")  
 */  
public class HomePageController  
    extends AbstractController {  
    ...  
}
```

Finally, you'll need to set up your build to include the Commons Attributes compiler so that the attributes will be compiled into your application code. We refer you to the Commons Attributes homepage (<http://jakarta.apache.org/commons/attributes>) for details on how to set up the Commons Attributes compiler in either Ant or Maven.

13.2.4 Working with multiple handler mappings

As you've seen, Spring comes with several useful handler mappings. But what if you can't decide which to use? For instance, suppose your application has been simple and you've been using `BeanNameUrlHandlerMapping`. But it is starting to grow and you'd like to start using `SimpleUrlHandlerMapping` going forward. How can you mix-'n'-match handler mappings during the transition?

As it turns out, all of the handler mapping classes implement Spring's Ordered interface. This means that you can declare multiple handler mappings in your application and set their order properties to indicate which has precedence with relation to the others.

For example, suppose you want to use both BeanNameUrlHandlerMapping and SimpleUrlHandlerMapping alongside each other in the same application. You'd need to declare the handler mapping beans as follows:

```
<bean id="beanNameUrlMapping" class="org.springframework.web.  
    ↪ servlet.handler.BeanNameUrlHandlerMapping">  
    <property name="order"><value>1</value></property>  
</bean>  
<bean id="simpleUrlMapping" class="org.springframework.web.  
    ↪ servlet.handler.SimpleUrlHandlerMapping">  
    <property name="order"><value>0</value></property>  
    <property name="mappings">  
    ...  
    </property>  
</bean>
```

Note that the lower the value of the order property, the higher the priority. In this case, SimpleUrlHandlerMapping's order is lower than that of BeanNameUrlHandlerMapping. This means that DispatcherServlet will consult SimpleUrlHandlerMapping first when trying to map a URL to a controller. BeanNameUrlHandlerMapping will only be consulted if SimpleUrlHandlerMapping turns up no results.

Spring's handler mappings help DispatcherServlet know which controller a request should be directed to. After DispatcherServlet has figured out where to send the request, it's up to a controller to process it. Next up, let's have a look at how to create controllers in Spring MVC.

13.3 **Handling requests with controllers**

If DispatcherServlet is the heart of Spring MVC then controllers are the brains. When implementing the behavior of your Spring MVC application, you extend one of Spring's controller classes. The controller receives requests from DispatcherServlet and performs some business functionality on behalf of the user.

If you're familiar with other web frameworks such as Struts or WebWork, you may recognize controllers as being roughly equivalent in purpose to a Struts or WebWork action. One huge difference between Spring controllers and Struts/WebWork actions, however, is that Spring provides a rich controller hierarchy (as

shown in figure 13.6) in contrast to the rather flat action hierarchy of Struts or WebWork.

At first glance, figure 13.6 may seem somewhat daunting. Indeed, when compared to other MVC frameworks such as Jakarta Struts or WebWork, there's a lot more to swallow with Spring's controller hierarchy. In reality, however, this perceived complexity is actually quite simple and flexible.

At the top of the controller hierarchy is the `Controller` interface. Any class implementing this interface can be used to handle requests through the Spring

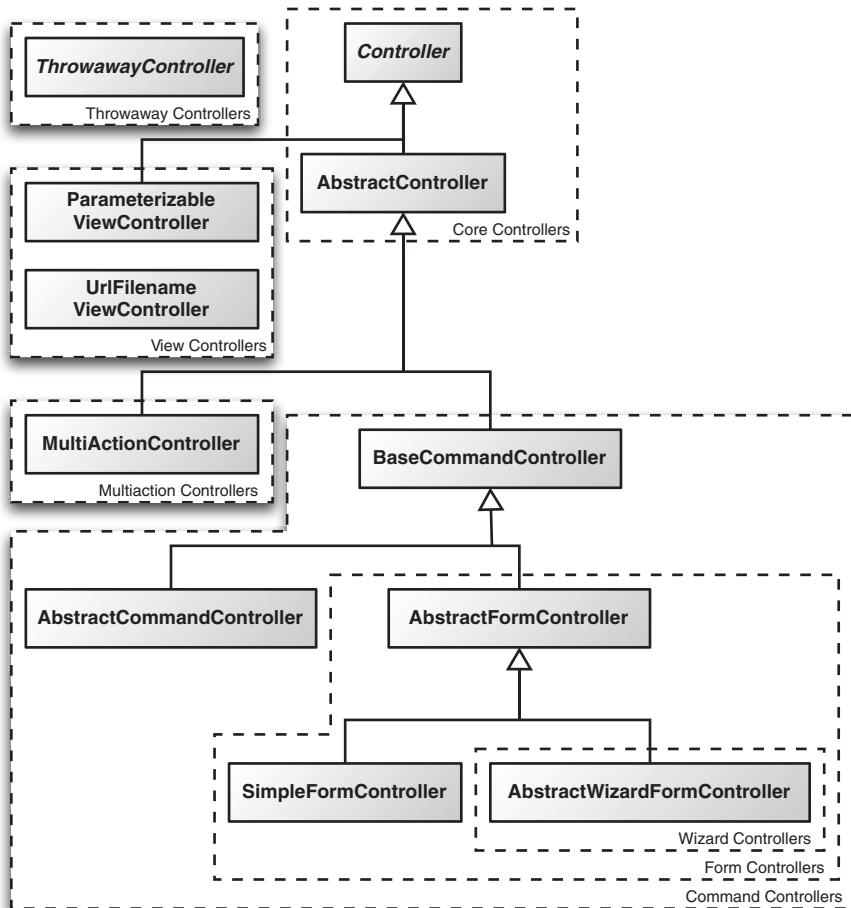


Figure 13.6 Spring MVC's controller hierarchy includes controllers for every occasion—from the simplest requests to more complex form processing.

MVC framework. To create your own controller, all you must do is write a class that implements this interface.

While you could write a class that directly implements the Controller interface, you're more likely to extend one of the classes lower in the hierarchy. Whereas the Controller interface defines the basic contract between a controller and Spring MVC, the various controller classes provide additional functionality beyond the basics.

The wide selection of controller classes is both a blessing and a curse. Unlike other frameworks that force you to work with a single type of controller object (such as Struts's Action class), Spring lets you choose the controller that is most appropriate for your needs. However, with so many controller classes to choose from, many developers find themselves overwhelmed and don't know how to decide.

To help you decide which controller class to extend for your application's controllers, consider table 13.2. As you can see, Spring's controller classes can be grouped into six categories that provide more functionality (and introduce more complexity) as you progress down the table. You may also notice from figure 13.5 that (with the exception of ThrowawayController) as you move down the controller hierarchy, each controller builds on the functionality of the controllers above it.

Table 13.2 Spring MVC's selection of controller classes.

Controller type	Classes	Useful when...
View	ParameterizableViewController UrlFilenameViewController	Your controller only needs to display a static view—no processing or data retrieval is needed.
Simple	Controller (interface) AbstractController	Your controller is extremely simple, requiring little more functionality than is afforded by basic Java servlets.
Throwaway	ThrowawayController	You want a simple way to handle requests as commands (in a manner similar to WebWork Actions).
Multiaction	MultiActionController	Your application has several actions that perform similar or related logic.

Table 13.2 Spring MVC's selection of controller classes. (continued)

Controller type	Classes	Useful when...
Command	BaseCommandController AbstractCommandController	Your controller will accept one or more parameters from the request and bind them to an object. Also capable of performing parameter validation.
Form	AbstractFormController SimpleFormController	You need to display an entry form to the user and also process the data entered into the form.
Wizard	AbstractWizardFormController	You want to walk your user through a complex, multipage entry form that ultimately gets processed as a single form.

You've already seen an example of a simple controller that extends `AbstractController`. In listing 13.1, `HomePageController` extends `AbstractController` and retrieves a list of the most recent rants for display on the home page. `AbstractController` is a perfect choice because the homepage is so simple and takes no input.

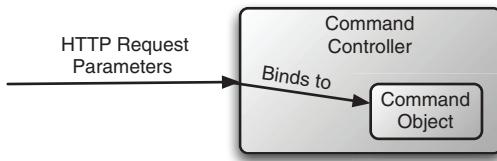
Basing your controller on `AbstractController` is fine when you don't need a lot of power. Most controllers, however, are going to be more interesting, taking parameters and requiring validation of those parameters. In the sections that follow, we're going to build several controllers that define the web layer of the RoadRantz application by extending the other implementations of the Controller classes in figure 13.6, starting with command controllers.

13.3.1 Processing commands

It's common for a web request to take one or more parameters that are used to determine the results. For instance, one of the requirements for the RoadRantz application is to display a list of rants for a particular vehicle.

Of course, you could extend `AbstractController` and retrieve the parameters your controller needs from the `HttpServletRequest`. But you would also have to write the logic that binds the parameters to business objects and you'd have to put validation logic in the controller itself. Binding and validation logic really don't belong in the controller.

In the event that your controller will need to perform work based on parameters, your controller class should extend a command controller class such as

**Figure 13.7**

Command controllers relieve you from the hassle of dealing with request parameters directly. They bind the request parameters to a command object that you'll work with instead.

`AbstractCommandController`. As shown in figure 13.7, command controllers automatically bind request parameters to a command object. They can also be wired to plug in validators to ensure that the parameters are valid.

Listing 13.3 shows `RantsForVehicleController`, a command controller that is used to display a list of rants that have been entered for a specific vehicle.

Listing 13.3 RantsForVehicleController, which lists all rants for a particular vehicle

```

package com.roaddrantz.mvc;
import java.util.List;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.springframework.validation.BindException;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.
    ↗ AbstractCommandController;
import com.roaddrantz.domain.Vehicle;
import com.roaddrantz.service.RantService;

public class RantsForVehicleController
    extends AbstractCommandController {

    public RantsForVehicleController() {
        setCommandClass(Vehicle.class);
        setCommandName("vehicle");
    }

    protected ModelAndView handle(HttpServletRequest request,
        HttpServletResponse response, Object command,
        BindException errors) throws Exception {
        Vehicle vehicle = (Vehicle) command;
        List vehicleRants =
            rantService.getRantsForVehicle(vehicle);
        Map model = errors.getModel();
        model.put("rants", 4
            rantService.getRantsForVehicle(vehicle));
        model.put("vehicle", vehicle);

        return new ModelAndView("vehicleRants", model);
    }
}
  
```

Creates the model

Returns model

Uses RantService to retrieve list of rants

Casts command object to Vehicle

Sets command class, name

```
private RantService rantService;
public void setRantService(RantService rantService) {
    this.rantService = rantService;
}
}
```

The `handle()` method of `RantsForVehicleController` is the main execution method for `AbstractCommandController`. This method is a bit more interesting than the `handleRequestInternal()` method from `AbstractController`. In addition to an `HttpServletRequest` and an `HttpServletResponse`, `handle()` takes an `Object` that is the controller's command.

A command object is a bean that is meant to hold request parameters for easy access. If you are familiar with Jakarta Struts, you may recognize a command object as being similar to a Struts `ActionForm`. The key difference is that unlike a Struts form bean that must extend `ActionForm`, a Spring command object is a POJO that doesn't need to extend any Spring-specific classes.

In this case, the command object is an instance of `Vehicle`, as set in the controller's constructor. You may recognize `Vehicle` as the domain class that describes a vehicle from chapter 5. Although command classes don't have to be instances of domain classes, it is sure handy when they are. `Vehicle` already defines the same data needed by `RantsForVehicleController`. Conveniently, it's also the exact same type needed by the `getRantsForVehicle()` method of `RantService`. This makes it a perfect choice for a command class.

Before the `handle()` method is called, Spring will attempt to match any parameters passed in the request to properties in the command object. `Vehicle` has two properties: `state` and `plateNumber`. If the request has parameters with these names, the parameter values will automatically be bound to the `Vehicle`'s properties.

As with `HomePageController`, you'll also need to register `RantsForVehicleController` in `roadrantz-servlet.xml`:

```
<bean id="rantsForVehicleController"
      class="com.roadrantz.mvc.RantsForVehicleController">
    <property name="rantService" ref="rantService" />
</bean>
```

Command controllers make it easy to handle requests with request parameters by binding the request parameters to command objects. The request parameters could be given as URL parameters (as is likely the case with `RantsForVehicleController`) or as fields from a web-based form. Although command controllers can process input from a form, Spring provides another type of controller with better support for form handling. Let's have a look at Spring's form controllers next.

13.3.2 Processing form submissions

In a typical web-based application, you're likely to encounter at least one form that you must fill out. When you submit that form, the data that you enter is sent to the server for processing, and once the processing is completed, you are either presented with a success page or are given the form page with errors in your submission that you must correct.

The core functionality of the RoadRantz application is the ability to enter a rant about a particular vehicle. In the application, the user will be presented with a form to enter their rant. Upon submission of that form, the expectation is that the rant will be saved to the database for later viewing.

When implementing the rant submission process, you might be tempted to extend `AbstractController` to display the form and to extend `AbstractCommandController` to process the form. This could certainly work, but would end up being more difficult than necessary. You would have to maintain two different controllers that work in tandem to process rant submissions. Wouldn't it be simpler to have a single controller handle both form display and form processing?

What you'll need in this case is a form controller. Form controllers take the concept of command controllers a step further, as shown in figure 13.8, by adding functionality to display a form when an HTTP GET request is received and process the form when an HTTP POST is received. Furthermore, if any errors occur in processing the form, the controller will know to redisplay the form so that the user can correct the errors and resubmit.

To illustrate how form controllers work, consider `AddRantFormController` in listing 13.4.

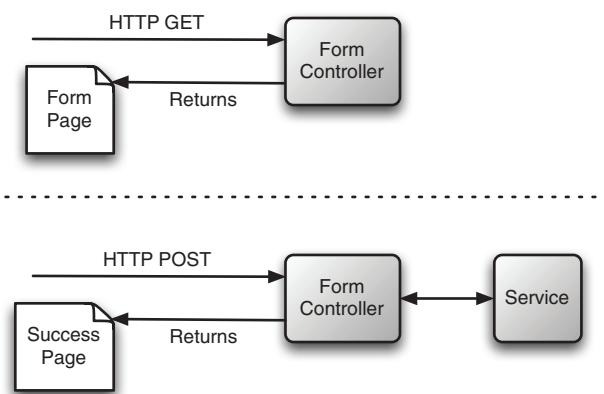


Figure 13.8
On an HTTP GET request, form controllers display a form to collect user input. Upon submitting the form with an HTTP POST, the form controller processes the input and returns a success page.

Listing 13.4 A controller for adding new rants

```

package com.roaddrantz.mvc;
import java.util.HashMap;
import java.util.Map;
import javax.servlet.http.HttpServletRequest;
import org.springframework.validation.BindException;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.SimpleFormController;
import com.roaddrantz.domain.Rant;
import com.roaddrantz.domain.Vehicle;
import com.roaddrantz.service.RantService;

public class AddRantFormController extends SimpleFormController {
    private static final String[] ALL_STATES = {
        "AL", "AK", "AZ", "AR", "CA", "CO", "CT", "DE", "DC", "FL",
        "GA", "HI", "ID", "IL", "IN", "IA", "KS", "KY", "LA", "ME",
        "MD", "MA", "MI", "MN", "MS", "MO", "MT", "NE", "NV", "NH",
        "NJ", "NM", "NY", "NC", "ND", "OH", "OK", "OR", "PA", "RI",
        "SC", "SD", "TN", "TX", "UT", "VA", "VT", "WA", "WV", "WI",
        "WY"
    };
    public AddRantFormController() {
        setCommandClass(Rant.class);
        setCommandName("rant");
    }

    protected Object formBackingObject(HttpServletRequest request)
        throws Exception {
        Rant rantForm = (Rant) super.formBackingObject(request);
        rantForm.setVehicle(new Vehicle());
        return rantForm;
    }

    protected Map referenceData(HttpServletRequest request)
        throws Exception {
        Map referenceData = new HashMap();
        referenceData.put("states", ALL_STATES);
        return referenceData;
    }

    protected ModelAndView onSubmit(Object command,
        BindException bindException) throws Exception {
        Rant rant = (Rant) command;
        rantService.addRant(rant); ← Adds new rant
        return new ModelAndView(getSuccessView());
    }

    private RantService rantService;
    public void setRantService(RantService rantService) {

```

Sets command class, name

Sets up Rant command with blank Vehicle

Provides list of states for form

```
        this.rantService = rantService;
    }
}
```

Although it may not be obvious, `AddRantFormController` is responsible for both displaying a rant entry form and processing the results of that form. When this controller receives an HTTP GET request, it will direct the request to the form view. And when it receives an HTTP POST request, the `onSubmit()` method will process the form submission.

The `referenceData()` method is optional, but is handy when you need to provide any additional information for displaying the form. In this case, our form will need a list of states that will be displayed (presumably in a drop-down selection list). So, the `referenceData()` method of `AddRantFormController` adds an array of `String`s that contains all 50 U.S. states as well as the District of Columbia.

Under normal circumstances, the command object that backs the form is simply an instance of the command class. In the case of `AddRantFormController`, however, a simple `Rant` instance will not do. The form is going to use the nested `Vehicle` property within a `Rant` as part of the form-backing object. Therefore, it was necessary to override the `formBackingObject()` method to set the `vehicle` property. Otherwise, a `NullPointerException` would be thrown when the controller attempts to bind the `state` and `plateNumber` properties.

The `onSubmit()` method handles the form submission (an HTTP POST request) by passing the command object (which is an instance of `Rant`) to the `addRant()` method of the injected `RantService` reference.

What's not clear from listing 13.4 is how this controller knows to display the rant entry form. It's also not clear where the user will be taken after the rant has been successfully added. The only hint is that the result of a call to `getSuccessView()` is given to the `ModelAndView`. But where does the success view come from?

`SimpleFormController` is designed to keep view details out of the controller's Java code as much as possible. Instead of hard-coding a `ModelAndView` object, you configure the form controller in the context configuration file as follows:

```
<bean id="addRantController"
      class="com.roaddrantz.mvc.AddRantFormController">
    <property name="formView" value="addRant" />
    <property name="successView" value="rantAdded" />
    <property name="rantService" ref="rantService" />
</bean>
```

Just as with the other controllers, the `addRantController` bean is wired with any services that it may need (e.g., `rantService`). But here you also specify a `formView` property and a `successView` property. The `formView` property is the logical name of a view to display when the controller receives an HTTP GET request or when any errors are encountered. Likewise, the `successView` is the logical name of a view to display when the form has been submitted successfully. A view resolver (see section 13.4) will use these values to locate the `View` object that will render the output to the user.

Validating form input

When `AddRantFormController` calls `addRant()`, it's important to ensure that all of the data in the `Rant` command is valid and complete. You don't want to let users enter only a state and no plate number (or vice versa). Likewise, what's the point in specifying a state and plate number but not providing any text in the rant? And it's important that the user not enter a plate number that isn't valid.

The `org.springframework.validation.Validator` interface accommodates validation for Spring MVC. It is defined as follows:

```
public interface Validator {  
    void validate(Object obj, Errors errors);  
    boolean supports(Class clazz);  
}
```

Implementations of this interface should examine the fields of the object passed into the `validate()` method and reject any invalid values via the `Errors` object. The `supports()` method is used to help Spring determine whether the validator can be used for a given class.

`RantValidator` (listing 13.5) is a Validator implementation used to validate a `Rant` object.

Listing 13.5 Validating a Rant entry

```
package com.roaddrantz.mvc;  
import org.apache.oro.text.perl.Perl5Util;  
import org.springframework.validation.Errors;  
import org.springframework.validation.ValidationUtils;  
import org.springframework.validation.Validator;  
import com.roaddrantz.domain.Rant;  
  
public class RantValidator implements Validator {  
    public boolean supports(Class clazz) {  
        return clazz.equals(Rant.class);  
    }  
  
    public void validate(Object command, Errors errors) {
```

```

Rant rant = (Rant) command;

ValidationUtils.rejectIfEmpty(
    errors, "vehicle.state", "required.state",
    "State is required.");

ValidationUtils.rejectIfEmpty(
    errors, "vehicle.plateNumber", "required.plateNumber",
    "The license plate number is required.");

ValidationUtils.rejectIfEmptyOrWhitespace(
    errors, "rantText", "required.rantText",
    "You must enter some rant text.");

validatePlateNumber(
    rant.getVehicle().getPlateNumber(), errors);
}

private static final String PLATE_REGEX =
    "/[a-z0-9]{2,6}/i";

private void validatePlateNumber(
    String plateNumber, Errors errors) {
    Perl5Util perl5Util = new Perl5Util();
    if(!perl5Util.match(PLATE_REGEX, plateNumber)) {
        errors.reject("invalid.plateNumber",
            "Invalid license plate number.");
    }
}
}

```

Validates required fields

Validates plate numbers

The only other thing to do is to configure AddRantFormController to use RantValidator. You can do this by wiring a RantValidator bean into the AddRantFormController bean (shown here as an inner bean):

```

<bean id="addRantController"
    class="com.roaddrantz.mvc.AddRantFormController">
    <property name="formView" value="addRant" />
    <property name="successView" value="rantAdded" />
    <property name="rantService" ref="rantService" />
    <property name="validator">
        <bean class="com.roaddrantz.mvc.RantValidator" />
    </property>
</bean>

```

When a rant is entered, if all of the required properties are set and if the plate number passes validation, AddRantFormController's onSubmit() will be called and the rant will be added. However, if RantValidator rejects any of the fields, the user will be returned to the form view to correct the mistakes.

By implementing the Validator interface, you are able to programmatically take full control over the validation of your application's command objects. This may be perfect if your validation needs are complex and require special logic.

However, in simple cases such as ensuring required fields and basic formatting, writing our own implementation of the Validator interface is a bit too involved. It'd be nice if we could write validation rules declaratively instead of having to write validation rules in Java code. Let's have a look at how to use declarative validation with Spring MVC.

Validating with Commons Validator

One complaint that we've heard about Spring MVC is that validation with the Validator interface doesn't even compare to the kind of validation possible with Jakarta Struts. We can't argue with that complaint. Jakarta Struts has a very nice facility for declaring validation rules outside of Java code. The good news is that we can do declarative validation with Spring MVC, too.

But before you go digging around in Spring's JavaDoc for a declarative Validator implementation, you should know that Spring doesn't come with such a validator. In fact, Spring doesn't come with any implementations of the Validator interface and leaves it up to you to write your own.

However, you don't have to go very far to find an implementation of Validator that supports declarative validation. The Spring Modules project (<https://springmodules.dev.java.net>) is a sister project of Spring that provides several extensions to Spring whose scope exceeds that of the main Spring project. One of those extensions is a validation module that makes use of Jakarta Commons Validator (<http://jakarta.apache.org/commons/validator>) to provide declarative validation.

To use the validation module in your application, you start by making the springmodules-validator.jar file available in the application's classpath. If you're using Ant to do your builds, you'll need to download the Spring Modules distribution (I'm using version 0.6) and find the spring-modules-0.6.jar file in the dist directory. Add this JAR to the <war> task's <lib> to ensure that it gets placed in the WEB-INF/lib directory of the application's WAR file.

If you're using Maven 2 to do your build (as I'm doing), you'll need to add the following <dependency> to pom.xml:

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>springmodules-validation</artifactId>
  <version>0.6</version>
```

```
<scope>compile</scope>
</dependency>
```

You'll also need to add the Jakarta Commons Validator JAR to your application's classpath. In Maven 2, it will look like this:

```
<dependency>
  <groupId>commons-validator</groupId>
  <artifactId>commons-validator</artifactId>
  <version>1.1.4</version>
  <scope>compile</scope>
</dependency>
```

Spring Modules provides an implementation of Validator called DefaultBeanValidator. DefaultBeanValidator is configured in roadrantz-servlet.xml as follows:

```
<bean id="beanValidator" class=
  "org.springmodules.commons.validator.DefaultBeanValidator">
  <property name="validatorFactory" ref="validatorFactory" />
</bean>
```

DefaultBeanValidator doesn't do any actual validation work. Instead, it delegates to Commons Validator to validate field values. As you can see, DefaultBeanValidator has a validatorFactory property that is wired with a reference to a validatorFactory bean. The validatorFactory bean is declared using the following XML:

```
<bean id="validatorFactory" class=
  "org.springmodules.commons.validator.DefaultValidatorFactory">
  <property name="validationConfigLocations">
    <list>
      <value>WEB-INF/validator-rules.xml</value>
      <value>WEB-INF/validation.xml</value>
    </list>
  </property>
</bean>
```

DefaultValidatorFactory is a class that loads the Commons Validator configuration on behalf of DefaultBeanValidator. The validationConfigLocations property takes a list of one or more validation configurations. Here we've asked it to load two configurations: validator-rules.xml and validation.xml.

The validator-rules.xml file contains a set of predefined validation rules for common validation needs such as email and credit card numbers. This file comes with the Commons Validator distribution, so you won't have to write it yourself—simply add it to the WEB-INF directory of your application. Table 13.3 lists all of the validation rules that come in validator-rules.xml.

Table 13.3 The validation rules available in Commons Validator's validator-rules.xml.

Validation rule	What it validates
byte	That the field contains a value that is assignable to byte
creditCard	That the field contains a String that passes a LUHN check and thus is a valid credit card number
date	That the field contains a value that fits a Date format
double	That the field contains a value that is assignable to double
email	That the field contains a String that appears to be an email address
float	That the field contains a value that is assignable to float
floatRange	That the field contains a value that falls within a range of float values
intRange	That the field contains a value that falls within a range of int values
integer	That the field contains a value that is assignable to int
long	That the field contains a value that is assignable to long
mask	That the field contains a String value that matches a given mask
maxlength	That the field has no more than a specified number of characters
minlength	That the field has at least a specific number of characters
required	That the field is not empty
requiredif	That the field is not empty, but only if another criterion is met
short	That the field contains a value that is assignable to short

The other file, validation.xml, defines application-specific validation rules that apply directly to the RoadRantz application. Listing 13.6 shows the contents of validation.xml as applied to RoadRantz.

Listing 13.6 Declaring validations in RoadRantz

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE form-validation PUBLIC
    "-//Apache Software Foundation//DTD
        Commons Validator Rules Configuration 1.1//EN"
    "http://jakarta.apache.org/commons/dtds/validator_1_1.dtd">

<form-validation>
    <formset>
        <form name="rant">
            <field property="rantText" depends="required"> <!--
                Requires rant text
            -->
```

```

        <arg0 key="required.rantText" />
    </field>
    <field property="vehicle.state" depends="required"> ←
        <arg0 key="required.state" /> ←
    </field>
    <field property="vehicle.plateNumber"
        depends="required,mask"> ←
        <arg0 key="invalid.plateNumber" /> ←
        <var> ←
            <var-name>mask</var-name>
            <var-value>^ [0-9A-Za-z] {2,6} $</var-value>
        </var>
    </field>
    </form>
</formset>
</form-validation>

```

█ Requires vehicle state

█ Requires and masks plate number

If the contents of validation.xml look strangely familiar to you, it's probably because Struts uses the same validation file XML. Under the covers, Struts is using Commons Validator to do its validation. Now Spring Modules brings the same declarative validation to Spring.

One last thing to do is change the controller's declaration to wire in the new declarative implementation of Validator:

```

<bean id="addRantController"
    class="com.roaddrantz.mvc.AddRantFormController">
    <property name="formView" value="addRant" />
    <property name="successView" value="rantAdded" />
    <property name="rantService" ref="rantService" />
    <property name="validator" ref="beanValidator" />
</bean>

```

A basic assumption with SimpleFormController is that a form is a single page. That may be fine when you're doing something simple such as adding a rant. But what if your forms are complex, requiring the user to answer several questions? In that case, it may make sense to break the form into several subsections and walk users through using a wizard. Let's see how Spring MVC can help you construct wizard forms.

13.3.3 Processing complex forms with wizards

Another feature of RoadRantz is that anyone can register as a user (known as a *motorist* in RoadRantz's terms) and be notified if any rants are entered for their vehicles. We developed the rant notification email in chapter 12. But we also need to provide a means for users to register themselves and their vehicles.

We could put the entire motorist registration form into a single JSP and extend `SimpleFormController` to process and save the data. However, we don't know how many vehicles the user will be registering and it gets tricky to ask the user for an unknown number of vehicle data in a single form.

Instead of creating one form, let's break motorist registration into several subsections and walk the user through the form using a wizard. Suppose that we partition the registration process questions into three pages:

- General user information such as first name, last name, password, and email address
- Vehicle information (state and plate number)
- Confirmation (for the user to review before committing their information)

Fortunately, Spring MVC provides `AbstractWizardFormController` to help out. `AbstractWizardFormController` is the most powerful of the controllers that come with Spring. As illustrated in figure 13.9, a wizard form controller is a special type of form controller that collects form data from multiple pages into a single command object for processing.

Let's see how to build a multipage registration form using `AbstractWizardFormController`.

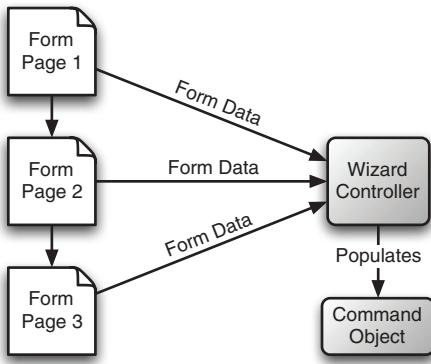


Figure 13.9
A wizard form controller is a special form controller that helps to split long and complex forms across multiple pages.

Building a basic wizard controller

To construct a wizard controller, you must extend the `AbstractWizardFormController` class. `MotoristRegistrationController` (listing 13.7) shows a minimal wizard controller to be used for registering a user in RoadRantz.

Listing 13.7 Registering motorists through a wizard

```

package com.roadrantz.mvc;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.HashSet;
import java.util.List;
import java.util.Map;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.springframework.validation.BindException;
import org.springframework.validation.Errors;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.

    ↗ AbstractWizardFormController;
import com.roadrantz.domain.Motorist;
import com.roadrantz.domain.Vehicle;
import com.roadrantz.service.RantService;

public class MotoristRegistrationController
    extends AbstractWizardFormController {
    public MotoristRegistrationController() {
        setCommandClass(Motorist.class);
        setCommandName("motorist");
    }
}

protected Object formBackingObject(HttpServletRequest request)
    throws Exception {
    Motorist formMotorist = new Motorist();
    List<Vehicle> vehicles = new ArrayList<Vehicle>();
    vehicles.add(new Vehicle());
    formMotorist.setVehicles(vehicles);
    return formMotorist;
}

protected Map referenceData(HttpServletRequest request,
    Object command, Errors errors, int page) throws Exception {
    Motorist motorist = (motorist) command;
    Map refData = new HashMap();

    if(page == 1 && request.getParameter("_target1") != null) {
        refData.put("nextVehicle",
            motorist.getVehicles().size() - 1);
    }
}

protected void postProcessPage(HttpServletRequest request,
    Object command, Errors errors, int page) throws Exception {
    Motorist motorist = (Motorist) command;
}

```

Sets command class, name

Creates form backing object

Increments next vehicle pointer

```
if(page == 1 && request.getParameter("_target1") != null) {
    motorist.getVehicles().add(new Vehicle());    <-- Adds new
}                                         blank vehicle

protected ModelAndView processFinish(HttpServletRequest request,
    HttpServletResponse response, Object command,
    BindException errors)
throws Exception {

    Motorist motorist = (motorist) command;
    // the last vehicle is always blank...remove it
    motorist.getVehicles().remove(
        motorist.getVehicles().size() - 1);    <-- Adds motorist
    rantService.addMotorist(motorist);

    return new ModelAndView(getSuccessView(),
        "motorist", motorist);
}

// injected
private RantService rantService;
public void setRantService(RantService rantService) {
    this.rantService = rantService;
}

// returns the last page as the success view
private String getSuccessView() {
    return getPages()[getPages().length-1];
}
}
```

Just as with any command controller, you must set the command class when using a wizard controller. Here `MotoristRegistrationController` has been set to use `Motorist` as the command class. But because the motorist will also be registering one or more vehicles, the `formBackingObject()` method is overridden to set the `vehicles` property with a list of `Vehicle` objects. The list is also started with a blank `Vehicle` object for the form to populate.

Since the user can register any number of vehicles and since the vehicles list will grow with each vehicle added, the form view needs a way of knowing which entry in the list is the next entry. So, `referenceData()` is overridden to make the index of the next vehicle available to the form.

The only compulsory method of `AbstractWizardFormController` is `processFinish()`. This method is called to finalize the form when the user has finished completing it (presumably by clicking a Finish button). In `MotoristRegistrationController`, the `processFinish()` method adds a new blank vehicle to the list of vehicles and removes the last vehicle from the list. It also adds the motorist to the `RantService`.

tionController, processFinish() sends the data in the Motorist object to addMotorist() on the injected RantService object.

Notice there's nothing in MotoristRegistrationController that gives any indication of what pages make up the form or in what order the pages appear. That's because AbstractWizardFormController handles most of the work involved to manage the workflow of the wizard under the covers. But how does AbstractWizardFormController know what pages make up the form?

Some of this may become more apparent when you see how MotoristRegistrationController is declared in roadrantz-servlet.xml:

```
<bean id="registerMotoristController"
      class="com.rodrantz.mvc.MotoristRegistrationController">
    <property name="rantService" ref="rantService" />
    <property name="pages">
      <list>
        <value>motoristDetailForm</value>
        <value>motoristVehicleForm</value>
        <value>motoristConfirmation</value>
        <value>redirect:home.htm</value>
      </list>
    </property>
  </bean>
```

So that the wizard knows which pages make up the form, a list of logical view names is given to the pages property. These names will ultimately be resolved into a View object by a view resolver (see section 13.4). But for now, just assume that these names will be resolved into the base filename of a JSP.

While this clears up how MotoristRegistrationController knows which pages to show, it doesn't tell us how it knows what order to show them in.

Stepping through form pages

The first page to be shown in any wizard controller will be the first page in the list given to the pages property. In the case of the motorist registration wizard, the first page shown will be the motoristDetailForm page.

To determine which page to go to next, AbstractWizardFormController consults its getTargetPage() method. This method returns an int, which is an index into the zero-based list of pages given to the pages property.

The default implementation of getTargetPage() determines which page to go to next based on a parameter in the request whose name begins with _target and ends with a number. getTargetPage() removes the _target prefix from the parameter and uses the remaining number as an index into the pages list. For

example, if the request has a parameter whose name is `_target2`, the user will be taken to the page rendered by the `motoristConfirmation` view.

Knowing how `getTargetPage()` works helps you to know how to construct your Next and Back buttons in your wizard's HTML pages. For example, suppose that your user is on the `motoristVehicleForm` page (`index = 1`). To create Next and Back buttons on the page, all you must do is create submit buttons that are appropriately named with the `_target` prefix:

```
<form method="POST" action="feedback.htm">
...
<input type="submit" value="Back" name="_target0">
<input type="submit" value="Next" name="_target2">
</form>
```

When the Back button is clicked, a parameter with its name, `_target0`, is placed into the request back to `MotoristRegistrationController`. The `getTargetPage()` method will process this parameter's name and send the user to the `motoristDetailForm` page (`index = 0`). Likewise, if the Next button is clicked, `getTargetPage()` will process a parameter named `_target2` and decide to send the user to the `motoristConfirmation` page (`index = 2`).

The default behavior of `getTargetPage()` is sufficient for most projects. However, if you would like to define a custom workflow for your wizard, you may override this method.

Finishing the wizard

That explains how to step back and forth through a wizard form. But how can you tell the controller that you have finished and that the `processFinish()` method should be called?

There's another special request parameter called `_finish` that indicates to `AbstractWizardFormController` that the user has finished filling out the form and wants to submit the information for processing. Just like the `_targetX` parameters, `_finish` can be used to create a Finish button on the page:

```
<form method="POST" action="feedback.htm">
...
<input type="submit" value="Finish" name="_finish">
</form>
```

When `AbstractWizardFormController` sees the `_finish` parameter in the request, it will pass control to the `processFinish()` method for final processing of the form.

Unlike other form controllers, `AbstractWizardFormController` doesn't provide a means for setting the success view page. So, we've added a `getSuccessView()` method in `MotoristRegistrationController` to return the last page in the pages list. So, when the form has been submitted as finished, the `processFinish()` method returns a `ModelAndView` with the last view in the pages list as the view.

Cancelling the wizard

What if your user is partially through with registration and decides that they don't want to complete it at this time? How can they abandon their input without finishing the form?

Aside from the obvious answer—they could close their browser—you can add a Cancel button to the form:

```
<form method="POST" action="feedback.htm">
...
<input type="submit" value="Cancel" name="_cancel">
</form>
```

As you can see, a Cancel button should have `_cancel` as its name so that, when clicked, the browser will place a parameter into the request called `_cancel`. When `AbstractWizardFormController` receives this parameter, it will pass control to the `processCancel()` method.

By default, `processCancel()` throws an exception indicating that the cancel operation is not supported. So, you'll need to override this method so that it (at a minimum) sends the user to whatever page you'd like them to go to when they click Cancel. The following implementation of `processCancel()` sends the user to the success view:

```
protected ModelAndView processCancel(HttpServletRequest request,
    HttpServletResponse response, Object command,
    BindException bindException) throws Exception {
    return new ModelAndView(getSuccessView());
}
```

If there is any cleanup work to perform upon a cancel, you could also place that code in the `processCancel()` method before the `ModelAndView` is returned.

Validating a wizard form a page at a time

As with any command controller, the data in a wizard controller's command object can be validated using a `Validator` object. However, there's a slight twist.

With other command controllers, the command object is completely populated at once. But with wizard controllers, the command object is populated a bit at a time as the user steps through the wizard's pages. With a wizard, it doesn't make much sense to validate all at once because if you validate too early, you will probably find validation problems that stem from the fact that the user isn't finished with the wizard. Conversely, it is too late to validate when the Finish button is clicked because any errors found may span multiple pages (which form page should the user go back to?).

Instead of validating the command object all at once, wizard controllers validate the command object a page at a time. This is done every time that a page transition occurs by calling the `validatePage()` method. The default implementation of `validatePage()` is empty (i.e., no validation), but you can override it to do your bidding.

To illustrate, on the `motoristDetailForm` page you ask the user for their email address. This field is optional, but if it is entered, it should be in a valid email address format. The following `validatePage()` method shows how to validate the email address when the user transitions away from the `motoristDetailForm` page:

```
protected void validatePage(Object command, Errors errors,
    int page) {

    Motorist motorist = (Motorist) command;
    MotoristValidator validator =
        (MotoristValidator) getValidator();

    if(page == 0) {
        validator.validateEmail(motorist.getEmail(), errors);
    }
}
```

When the user transitions from the `motoristDetailForm` page (`index = 0`), the `validatePage()` method will be called with `0` passed in to the `page` argument. The first thing `validatePage()` does is get a reference to the `Motorist` command object and a reference to the `MotoristValidator` object. Because there's no need to do email validation from any other page, `validatePage()` checks to see that the user is coming from page `0`.

At this point, you could perform the email validation directly in the `validatePage()` method. However, a typical wizard will have several fields that will need to be validated. As such, the `validatePage()` method can become quite unwieldy. We recommend that you delegate responsibility for validation to a fine-grained field-level validation method in the controller's `Validator` object, as we've done here with the call to `MotoristValidator`'s `validateEmail()` method.

All of this implies that you'll need to set the `validator` property when you configure the controller:

```
<bean id="registerMotoristController"
      class="com.roadrantz.mvc.MotoristRegistrationController">
    <property name="rantService" ref="rantService" />
    <property name="pages">
      <list>
        <value>motoristDetailForm</value>
        <value>motoristVehicleForm</value>
        <value>motoristConfirmation</value>
        <value>redirect:home.htm</value>
      </list>
    </property>
    <property name="validator">
      <bean class="com.roadrantz.mvc.MotoristValidator" />
    </property>
  </bean>
```

It's important to be aware that unlike the other command controllers, wizard controllers never call the standard `validate()` method of their `Validator` object. That's because the `validate()` method validates the entire command object as a whole, whereas it is understood that the command objects in a wizard will be validated a page at a time.

The controllers you've seen up until now are all part of the same hierarchy that is rooted with the `Controller` interface. Even though the controllers all get a bit more complex (and more powerful) as you move down the hierarchy, all of the controllers that implement the `Controller` interface are somewhat similar. But before we end our discussion of controllers, let's have a look at another controller that's very different than the others—the throwaway controller.

13.3.4 Working with throwaway controllers

One last controller that you may find useful is a throwaway controller. Despite the dubious name, throwaway controllers can be quite useful and easy to use. Throwaway controllers are significantly simpler than the other controllers, as evidenced by the `ThrowawayController` interface:

```
public interface ThrowawayController {
  ModelAndView execute() throws Exception;
}
```

To create your own throwaway controller, all you must do is implement this interface and place the program logic in the `execute()` method. Quite simple, isn't it?

But hold on. How are parameters passed to the controller? The execution methods of the other controllers are given `HttpServletRequest` and `command`

objects from which to pull the parameters. If the `execute()` method doesn't take any arguments, how can your controller process user input?

You may have noticed in figure 13.5 that the `ThrowawayController` interface is not even in the same hierarchy as the `Controller` interface. This is because throwaway controllers are very different from the other controllers. Instead of being given parameters through an `HttpServletRequest` or a command object, throwaway controllers act as their own command object. If you have ever worked with WebWork, this may seem quite natural because WebWork actions behave in a similar way.

From the requirements for `RoadRantz`, we know that we'll need to display a list of rants for a given month, day, and year. We could implement this using a command controller, as we did with `RantsForVehicleController` (listing 13.3). Unfortunately, no domain object exists that takes a month, day, and year. This means we'd need to create a special command class to carry this data. It wouldn't be so hard to create such a POJO, but maybe there's a better way.

Instead of implementing `RantsForDayController` as a command controller, let's implement it as a `ThrowawayController`, as shown in listing 13.8.

Listing 13.8 A throwaway controller that produces a list of rants for a given day

```
package com.roaddrantz.mvc;
import java.util.Date;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.throwaway.
    ↗ ThrowawayController;
import com.roaddrantz.service.RantService;

public class RantsForDayController implements ThrowawayController {
    private Date day;

    public ModelAndView execute() throws Exception {
        List<Rant> dayRants = rantService.getRantsForDay(day);
        return new ModelAndView("dayRants", "rants", dayRants); ←
    }

    public void setDay(Date day) { ←
        this.day = day;
    }

    private RantService rantService;
    public void setRantService(RantService rantService) {
        this.rantService = rantService;
    }
}
```

Binds day to request

Gets list of rants for day

Before `RantsForDayController` handles the request, Spring will call the `setDay()` method, passing in the value of the day request parameter. Once in the `execute()` method, `RantsForDayController` simply passes `day` to `rantService.getRantsForDay()` to retrieve the list of rants for that day. One thing that remains the same as the other controllers is that the `execute()` method must return a `ModelAndView` object when it has finished.

Just as with any controller, you also must declare throwaway controllers in the `DispatcherServlet`'s context configuration file. There's only one small difference, as you can see in this configuration of `RantsForDayController`:

```
<bean id="rantsForDayController"
      class="com.roaddrantz.mvc.RantsForDayController"
      scope="prototype">
    <property name="rantService" ref="rantService" />
</bean>
```

Notice that the `scope` attribute has been set to `prototype`. This is where throwaway controllers get their name. By default all beans are singletons, and so unless you set `scope` to `prototype`, `RantsForDayController` will end up being recycled between requests. This means its properties (which should reflect the request parameter values) may also be reused. Setting `scope` to `prototype` tells Spring to throw the controller away after it has been used and to instantiate a fresh instance for each request.

There's just one more thing to be done before we can use our throwaway controller. `DispatcherServlet` knows how to dispatch requests to controllers by using a *handler adapter*. The concept of handler adapters is something that you usually don't need to worry about because `DispatcherServlet` uses a default handler adapter that dispatches to controllers in the `Controller` interface hierarchy.

But because `ThrowawayController` isn't in the same hierarchy as `Controller`, `DispatcherServlet` doesn't know how to talk to `ThrowawayController`. To make it work, you must tell `DispatcherServlet` to use a different handler adapter. Specifically, you must configure `ThrowawayControllerHandlerAdapter` as follows:

```
<bean id="throwawayHandler" class="org.springframework.web.
  ↪ servlet.mvc.throwaway.ThrowawayControllerHandlerAdapter" />
```

By just declaring this bean, you are telling `DispatcherServlet` to replace its default handler adapter with `ThrowawayControllerHandlerAdapter`.

This is fine if your application is made up of nothing but throwaway controllers. But the `RoadRantz` application will use both throwaway and regular controllers alongside each other in the same application. Consequently, you still need

DispatcherServlet to use its regular handler adapter as well. Thus, you should also declare SimpleControllerHandlerAdapter as follows:

```
<bean id="simpleHandler" class="org.springframework.web.  
    ↳ servlet.mvc.SimpleControllerHandlerAdapter"/>
```

Declaring both handler adapters lets you mix both types of controllers in the same application.

Regardless of what functionality your controllers perform, ultimately they'll need to return some results to the user. The result pages are rendered by views, which are selected by their logical name when creating a `ModelAndView` object. But there needs to be a mechanism to map logical view names to the actual view that will render the response. We'll see that in chapter 14 when we turn our attention to Spring's view resolvers.

But first, did you notice that all of Spring MVC's controllers have method signatures that throw exceptions? It's possible that things could go awry as a controller processes a request. If an exception is thrown from a controller, what will the user see? Let's find out how to control the behavior of errant controllers with an exception resolver.

13.4 Handling exceptions

There's a bumper sticker that says "Failure is not an option: it comes with the software." Behind the humor of this message is a universal truth. Things don't always go well in software. When an error happens (and it inevitably will happen), do you want your application's users to see a stack trace or a friendlier message? How can you gracefully communicate the error to your users?

`SimpleMappingExceptionResolver` comes to the rescue when an exception is thrown from a controller. Use the following `<bean>` definition to configure `SimpleMappingExceptionResolver` to gracefully handle any `java.lang.Exception` thrown from Spring MVC controllers:

```
<bean id="exceptionResolver" class="org.springframework.web.  
    ↳ servlet.handler.SimpleMappingExceptionResolver">  
    <property name="exceptionMappings">  
        <props>  
            <prop key="java.lang.Exception">friendlyError</prop>  
        </props>  
    </property>  
</bean>
```

The `exceptionMappings` property takes a `java.util.Properties` that contains a mapping of fully qualified exception class names to logical view names. In this

case, the base `Exception` class is mapped to the view whose logical name is `friendlyError` so that if any errors are thrown, users won't have to see an ugly stack trace in their browser.

When a controller throws an `Exception`, `SimpleMappingExceptionResolver` will resolve it to `friendlyError`, which in turn will be resolved to a View using whatever view resolver(s) are configured. If the `InternalResourceViewResolver` from section 13.4.1 is configured then perhaps the user will be sent to the page defined in `/WEB-INF/jsp/friendlyError.jsp`.

13.5 Summary

The Spring Framework comes with a powerful and flexible web framework that is itself based on Spring's tenets of loose coupling, dependency injection, and extensibility.

At the beginning of a request, Spring offers a variety of handler mappings that help to choose a controller to process the request. You are given a choice to map URLs to controllers based on the controller bean's name, a simple URL-to-controller mapping, the controller class's name, or source-level metadata.

To process a request, Spring provides a wide selection of controller classes with complexity ranging from the very simple `Controller` interface all the way to the very powerful wizard controller and several layers in between, letting you choose a controller with an appropriate amount of power (and no more complexity than required). This sets Spring apart from other MVC web frameworks such as Struts and WebWork, where your choices are limited to only one or two `Action` classes.

All in all, Spring MVC maintains a loose coupling between how a controller is chosen to handle a request and how a view is chosen to display output. This is a powerful concept, allowing you to mix-'n'-match different Spring MVC parts to build a web layer most appropriate to your application.

In this chapter, you've been taken on a whirlwind tour of how Spring MVC handles requests. Along the way, you've also seen how most of the web layer of the RoadRantz application is constructed.

Regardless of what functionality is provided by a controller, you'll ultimately want the results of the controller to be presented to the user. So, in the next chapter, we'll build on Spring MVC by creating the view layer of the RoadRantz application. In addition to JSP, you'll learn how to use alternate template languages such as Velocity and FreeMarker. And you'll also learn how to dynamically produce non-HTML output such as Excel spreadsheets, PDF documents, and RSS feeds.

Rendering web views

14

This chapter covers

- Matching controllers to views
- Rendering views with JSP, Velocity, and FreeMarker
- Laying out pages with Tiles
- Generating PDF, Excel, and RSS output

The controllers, services, and DAOs in an application live in a box we call the *server*. Those application components may be doing something very important, but unless we can see what they’re doing, we can only guess what’s going on in there. For an application to be useful, it needs a way to communicate with the user.

That’s where the V of MVC comes into play. The view of an application communicates information back to the user and prompts the user to communicate with the application. Without the view layer of an application, we can only guess what’s inside.

In chapter 13, we looked at how to build the web layer of the RoadRantz application using Spring MVC. You learned how to configure Spring MVC and how to write controllers that would process user input and produce model data to present to the user. But at that time we conveniently ignored how that information would be presented to the user.

Now it’s time to see how to show the user what’s going on in the application by building the application’s view layer. We’ll start by looking at how Spring chooses a view based on the logical view name returned from a controller. Then we’ll explore various ways that Spring can produce output for the user, including template-based HTML, Excel spreadsheets, PDFs, and Rich Site Summary (RSS) feeds.

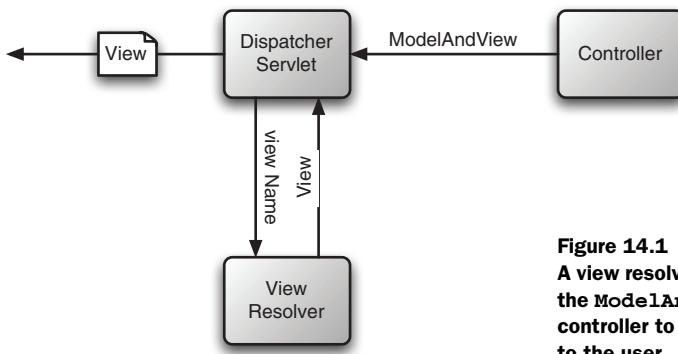
14.1 Resolving views

As you saw in the previous chapter, most of Spring MVC’s controllers return `ModelAndView` objects from their main execution method. You saw how model objects are passed to the view through the `ModelAndView` object, but we deferred discussion of how the logical view name is used to determine which view will render the results to the user.

In Spring MVC, a view is a bean that renders results to the user. How it performs the rendering depends on the type of view you’ll use. Most likely, you’ll want to use JavaServer Pages (JSP) to render the results. But you may also want to use alternate view technologies such as Velocity and FreeMarker templates or even views that produce PDF and Microsoft Excel documents. We’ll talk about all of these options later in this chapter.

The big question at this point is how a logical view name given to a `ModelAndView` object gets resolved into a `View` bean that will render output to the user. That’s where view resolvers come into play. Figure 14.1 shows how view resolvers work.

A view resolver is any bean that implements `org.springframework.web.servlet.ViewResolver`. Spring MVC regards these beans as special and consults them

**Figure 14.1**

A view resolver uses the logical view name in the `ModelAndView` returned from a controller to look up a view to render results to the user.

when trying to determine which View bean to use. Spring comes with several implementations of `ViewResolver`, as listed in table 14.1.

Let's have a look at a few of the most useful view resolvers, starting with `InternalResourceViewResolver`.

Table 14.1 Spring MVC's view resolvers help `DispatcherServlet` find view implementations based on a logical view name that was returned from a controller.

View resolver	How it works
<code>InternalResourceViewResolver</code>	Resolves logical view names into View objects that are rendered using template file resources (such as JSPs and Velocity templates)
<code>BeanNameViewResolver</code>	Looks up implementations of the View interface as beans in the Spring context, assuming that the bean name is the logical view name
<code> ResourceBundleViewResolver</code>	Uses a resource bundle (e.g., a properties file) that maps logical view names to implementations of the View interface
<code>XmlViewResolver</code>	Resolves View beans from an XML file that is defined separately from the application context definition files

14.1.1 Using template views

Odds are good that most of the time templates will render the results of your controllers. For example, suppose that after `RantsForVehicleController` is finished, you'd like to display the list of rants using the following JSP:

```
<%@ page contentType="text/html" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jstl/core" %>
```

```
<html>
    <head><title>Rantz For Vehicle</title></head>

    <body>
        <h2>Rantz for: ${vehicle.state} ${vehicle.plateNumber}</h2>

        <ul>
            <c:forEach items="${rants}" var="rant">
                <li><c:out value="${rant.vehicle.state}" /> --
                    <c:out value="${rant.vehicle.plateNumber}" /> --
                    <c:out value="${rant.rantText}" /></li>
            </c:forEach>
        </ul>
    </body>
</html>
```

Aesthetics aside, this JSP renders a list of the rants retrieved by `RantsForVehicleController`. Knowing that `RantsForVehicleController`'s `handle()` method concludes with the following return:

```
return new ModelAndView("vehicleRants", "rants", vehicleRants);
```

how can you tell Spring MVC that the logical view name `vehicleRants` means to use the rant-listing JSP to render the results?

`InternalResourceViewResolver` resolves a logical view name into a `View` object that delegates rendering responsibility to a template located in the web application's context. As illustrated in figure 14.2, it does this by taking the logical view name returned in a `ModelAndView` object and surrounding it with a prefix and a suffix to arrive at the path of a template within the web application.

Let's say that you've placed all of the JSPs for the `RoadRantz` application in the `/WEB-INF/jsp/` directory. Given that arrangement, you'll need to configure an `InternalResourceViewResolver` bean in `roadrantz-servlet.xml` as follows:

```
<bean id="viewResolver" class=
    "org.springframework.web.servlet.view.
     InternalResourceViewResolver">
    <property name="prefix" value="/WEB-INF/jsp/" />
    <property name="suffix" value=".jsp" />
</bean>
```

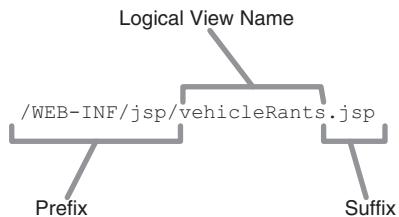


Figure 14.2
`InternalResourceViewResolver`
 resolves a view template's path by
 attaching a specified prefix and suffix to
 the logical view name.

When `InternalResourceViewResolver` is asked to resolve a view, it takes the logical view name, prefixes it with `/WEB-INF/jsp/`, and suffixes it with `.jsp` to arrive at the path of the JSP that will render the output. It then hands that path over to a `View` object that dispatches the request to the JSP.

So, when `RantsForVehicleController` returns a `ModelAndView` object with `vehicleRants` as the logical view name, it ends up resolving that view name to the path of a JSP: `/WEB-INF/jsp/vehicleRants.jsp`.

`InternalResourceViewResolver` then loads a `View` object with the path of the JSP. This implies that the rant list template must be named `vehicleRants.jsp`.

By default the `View` object is an `InternalResourceView`, which simply dispatches the request to the JSP to perform the actual rendering. But since `vehicleRants.jsp` uses JSTL tags, you may choose to replace `InternalResourceView` with `JstlView` by setting `InternalResourceViewResolver`'s `viewClass` property as follows:

```
<bean id="viewResolver" class=
    "org.springframework.web.servlet.view.
     ↗ InternalResourceViewResolver">
<property name="viewClass"
    value="org.springframework.web.servlet.view.JstlView" />
<property name="prefix" value="/WEB-INF/jsp/" />
<property name="suffix" value=".jsp" />
</bean>
```

`JstlView` dispatches the request to a JSP just like `InternalResourceView`. However, it also exposes JSTL-specific request attributes so that you can take advantage of JSTL's internationalization support.

Although `InternalResourceViewResolver` is quite easy to use, it may not be the most appropriate view for all circumstances. It assumes that your view is defined in a template file within the web application. That may be the case in most situations, but not always. Let's look at some other ways to resolve views.

14.1.2 Resolving view beans

When you first looked at `RantsForVehicleController` (listing 11.3), you may have assumed that the `vehicleRants` view would be rendered by a JSP. However, nothing about that controller implies that JSP will be used to render the output. What if instead of rendering the list of rants in HTML you want to offer the list as an RSS feed that the user can subscribe to?

Later, in section 14.5.3, you'll learn how to create a custom view that produces RSS feeds. But for now, pretend that you've already written that custom view. Since the rant listing isn't templated by JSP (or any other template resource for

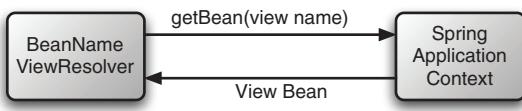


Figure 14.3
A BeanNameViewResolver looks up a view bean from the Spring context that has the same name as the logical view name.

that matter) in the web application, InternalResourceViewResolver isn't going to be of much help. Instead, you're going to have to choose one of Spring's other view resolvers.

BeanNameViewResolver is a view resolver that matches logical view names up with names of beans in the application context. Figure 14.3 shows how this works.

```
<bean id="viewResolver" class=
  ↗ "org.springframework.web.servlet.view.BeanNameViewResolver"/>
```

Now when a controller returns a ModelAndView with a logical view name of rantsRss, BeanNameViewResolver will look for a bean named rantsRss. This means you must register RantRssView in the context configuration file as follows:

```
<bean id="rantsRss" class=
  ↗ "com.roaddrantz.mvc.RantRssView"/>
```

Declaring view beans in a separate XML file

Another way to resolve View objects by their bean name is to use XmlFileViewResolver. XmlFileViewResolver works much like BeanNameViewResolver, but instead of looking for View beans in the main application context, it consults a separate XML file. To use XmlFileViewResolver, add the following XML to your context configuration file:

```
<bean id="viewResolver" class="org.springframework.web.
  ↗ servlet.view.XmlFileViewResolver">
    <property name="location">
      <value>/WEB-INF/roaddrantz-views.xml</value>
    </property>
  </bean>
```

By default, XmlFileViewResolver looks for View definitions in /WEB-INF/views.xml, but here we've set the location property to override the default with /WEB-INF/roaddrantz-views.xml.

XmlFileViewResolver is useful if you end up declaring more than a handful of View beans in DispatcherServlet's context configuration file. To keep the main context configuration file clean and tidy, you may separate the View declarations from the rest of the beans.

Resolving views from resource bundles

Yet another way of resolving Views by name is to use ResourceBundleViewResolver. Unlike BeanNameViewResolver and XmlFileViewResolver, ResourceBundleViewResolver manages view definitions in a properties file instead of XML (as shown in figure 14.4).

To use ResourceBundleViewResolver, configure ResourceBundleViewResolver in roadrantz-servlet.xml as follows:

```
<bean id="viewResolver" class="org.springframework.web.  
    ↪ servlet.view.ResourceBundleViewResolver">  
    <property name="basename" value="views" />  
</bean>
```

The basename property is used to tell ResourceBundleViewResolver how to construct the names of the properties files that contain View definitions. Here it has been set to `views`, which means that the View definitions could be listed in `views.properties`. For example, to add the `RantRssView`, place the following line in `views.properties`:

```
rantsRss.class=com.roadrantz.mvc.RantRssView
```

The name of this property can be broken down into two parts. The first part is `rantsRss`, which is the logical name of the View as returned in `ModelAndView`. The second part, `class`, indicates that you are setting the class name of the View implementation that should render the output for the `rantsRss` view (in this case, `RantRssView`).

By employing properties files, ResourceBundleViewResolver has an advantage over the other view resolvers with regard to internationalization. Whereas the other view resolvers always resolved a logical view name to a single View implementation, ResourceBundleViewResolver could return a different View implementation for the same logical view name, based on the user's Locale.

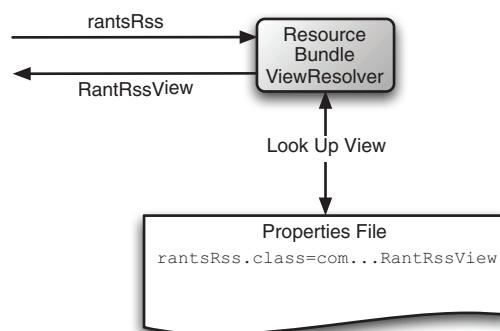


Figure 14.4
ResourceBundleViewResolver resolves views by consulting a properties file.

For example, if the user's browser is configured for English-speaking users in the United States then the views definitions will be retrieved from views_en_US.properties. Alternatively, for French users, the views would be defined in views_fr_FR.properties. The benefit here is that you are able to define a distinct set of views for different locales. Perhaps U.S. users prefer RSS 0.9 for their feeds, but French users prefer Atom feeds. Using ResourceBundleViewResolver, you could satisfy both countries.

Now that you have seen four different view resolvers that come with Spring, which one do you choose? The next section provides some guidelines to help you decide.

14.1.3 Choosing a view resolver

Many projects rely on JSP (or some other template language) to render the view results. Assuming that your application isn't internationalized or that you won't need to display a completely different view based on a user's locale, we recommend InternalResourceViewResolver because it is simply and tersely defined (as opposed to the other view resolvers that require you to explicitly define each view).

If, however, your views will be rendered using a custom View implementation (e.g., RSS, PDF, Excel, images, etc.), you'll need to consider one of the other view resolvers. We favor BeanNameViewResolver and XmlFileViewResolver over ResourceBundleViewResolver because they let you define your View beans in a Spring context configuration XML file. By defining the View in a Spring application context, you're able to configure it using the same syntax as you use for the other components in your application.

Given the choice between BeanNameViewResolver and XmlFileViewResolver, I'd settle on BeanNameViewResolver only when you have a handful of View beans that would not significantly increase the size of DispatcherServlet's context file. If the view resolver is managing a large number of View objects, I'd choose XmlFileViewResolver to separate the View bean definitions into a separate file.

In the rare case that you must render a completely different view depending on a user's locale, you have no choice but to use ResourceBundleViewResolver.

Using multiple view resolvers

Consider the case where most of an application's views are JSP based but a handful require one of the other view resolvers. For example, most of the RoadRantz application will use JSPs to render output, but (as you'll see in section 14.5) some responses will render RSS, PDF, and Excel output. Must you choose a

BeanNameViewResolver or XmlFileViewResolver and explicitly declare all of your views just to handle the special cases of PDF and Excel?

Fortunately, you aren't limited to choosing only one view resolver for your application. To use multiple view resolvers, simply declare all the view resolver beans you will need in your context configuration file. For example, to use InternalResourceViewResolver, BeanNameViewResolver, and XmlFileViewResolver all together, declare them as follows:

```
<bean id="viewResolver" class=
    "org.springframework.web.servlet.view.
     ↗ InternalResourceViewResolver">
    <property name="prefix"><value>/WEB-INF/jsp/</value></property>
    <property name="suffix"><value>.jsp</value></property>
</bean>

<bean id="beanNameViewResolver" class=
    ↗ "org.springframework.web.servlet.view.BeanNameViewResolver">
    <property name="order"><value>1</value></property>
</bean>

<bean id="xmlFileViewResolver" class=
    ↗ "org.springframework.web.servlet.view.XmlFileViewResolver">
    <property name="location">
        <value>/WEB-INF/views.xml</value>
    </property>
    <property name="order"><value>2</value></property>
</bean>
```

Because it's quite possible that more than one view resolver may be able to resolve the same logical view name, you should set the `order` property on each of your view resolvers to help Spring determine which resolver has priority over the others when a logical view name is ambiguous among more than one resolver. The exception to this rule is `InternalResourceViewResolver`, which is always the last view resolver in the chain.¹

View resolvers help `DispatcherServlet` find a view that will render output to the user. Now we must define the views themselves. Spring MVC supports several view layer template technologies, including JSP, Velocity, and FreeMarker. Let's start by looking at JSP. (We'll pick up the discussion on Velocity and FreeMarker in section 14.4.)

¹ In case you're wondering—`InternalResourceViewResolver` is the last in the chain because it will always resolve to a view. While the other view resolvers may not find a view for a given view name, `InternalResourceViewResolver` will always resolve to a view by using the prefix and suffix values (even if the actual view template doesn't exist). Therefore, it is necessary for `InternalResourceViewResolver` to always be last or the other view resolvers won't get a chance to resolve the view.

14.2 Using JSP templates

JavaServer Pages (JSPs) are the most common way of developing dynamic web pages in Java. In their simplest form, JSPs are HTML (or XML) files that are littered with chunks of Java code (known as scriptlets) and custom tags. Although other templating solutions have challenged JSP's position as the standard template for Java-based web applications, JSP remains dominant.

Early versions of JSP relied heavily on scriptlets to render dynamic output. But modern JSP specifications avoid the use of scriptlets in favor of custom JSP tags. Many JSP tag libraries have emerged to perform all sorts of functionality—from simple date formatting tags to complex table rendering tags (see <http://displaytag.sourceforge.net>).

From the beginning, Spring has always included a small set of tags useful for things such as binding form values to fields and rendering externalized messages. Spring 2 introduces a powerful new set of form-binding tags that greatly simplify form binding in JSP pages.

Let's start our exploration of Spring's JSP tag libraries with the new form-binding tags.

14.2.1 Binding form data

As we discussed in chapter 13, Spring's command and form controllers all have a command object associated with them. Request parameters are bound to properties of the command object for processing. And those command objects can be validated to ensure that the data bound to them meets certain criteria.

For example, recall that `AddRantFormController` (listing 13.4) has a `Rant` as its command object. When the form is submitted, the properties in the `Rant` object are set to the values of the corresponding request parameters. Then the `Rant` is validated to ensure that all of the required fields have been entered and that the vehicle's plate number fits the format of a license plate number.

What we didn't show you is where the request parameters come from. As you'll recall, the `formView` property of `AddRantFormController` was set to `addRant` in `roadrantz-servlet.xml`. This means that if we're using `InternalResourceViewResolver`, the view will be resolved to a JSP found in `/WEB-INF/jsp/addRant.jsp`, which is shown in listing 14.1.

Listing 14.1 A JSP form for entering a rant

```
<%@ page contentType="text/html" %>

<html>
  <head>
    <title>Add Rant</title>
  </head>

  <body>
    <h2>Enter a rant...</h2>
    <form method="POST" action="addRant.htm">
      <b>State: </b><input type="text"
        name="rant.vehicle.state"/><br/>
      <b>Plate #: </b>
        <input type="text" name="rant.vehicle.plateNumber"/><br/>
        <textarea name="rant.rantText" rows="5" cols="50"></textarea>
        <input type="submit"/>
    </form>
  </body>
</html>
```

The main purpose of addRant.jsp is to render a form for entering a rant. Here the names of the form fields are used to tell Spring which properties of the command object (`rant`) to populate with the form data when the form is submitted. For example, the first `<input>` tag's value will be given to the `state` property of the `vehicle` property of the command object.

There's only one problem with using plain HTML form tags. If any errors are encountered after submitting the form, the user will be taken back to the form view to correct the errors. But what will become of the values that were submitted? With plain HTML form tags, the binding is one-way from the form fields to the command properties. If the form is redisplayed after an error, there's no way to repopulate the form fields with the command properties. So the data entered will be lost and the user will have to reenter it all from scratch.

To address this problem, Spring 2 introduced a new tag library of form-binding JSP tags.² To start using the new tags, add the following JSP directive at the top of the `addRant.jsp`:

```
<%@ taglib prefix="form"
  uri="http://www.springframework.org/tags/form"%>
```

² In versions of Spring prior to 2.0, you would use the `<spring:bind>` tag to bind command object properties to form fields. But the `<spring:bind>` tag was awkward to use. Since the new `<form:xxx>` tags are so much better, the only mention of `<spring:bind>` will be in this footnote.

This tag library includes several JSP tags that render HTML form elements that are bound to the controller's command object. For example, here's the rant entry form, updated to use Spring's form-binding tags:

```
<form:form method="POST" action="addRant.htm" commandName="rant">
    <b>State: </b><form:input path="vehicle.state" /><br/>
    <b>Plate #: </b><form:input path="vehicle.plateNumber" /><br/>
    <form:textarea path="rantText" rows="5" cols="50" />
    <input type="submit"/>
</form:form>
```

In this new version of the form, we're using three of the form-binding tags. For the state and plate number fields, we use the `<form:input>` tag. For the rant text field we want the user to enter a lot of text, so a `<form:textarea>` is in order. In both cases, the `path` attribute defines the command property that the field is to be bound to. Finally, the `<form:form>` tag sets the command context for the form tags contained within it through the value of the `commandName` attribute.

When rendered, the JSP above is translated into the following HTML:

```
<form method="POST" action="addRant.htm">
    <b>State: </b><input name="vehicle.state" type="text"
        value="" /><br/>
    <b>Plate #: </b><input name="vehicle.plateNumber"
        type="text" value="" /><br/>
    <textarea name="rantText" rows="5" cols="50"></textarea>
    <input type="submit"/>
</form>
```

The nice thing about these form-binding tags is that, unlike regular HTML form tags, the binding goes both ways. Not only is the command populated with the field values, but if the form must be redisplayed after an error, the fields will be automatically populated with the data that caused the error.

This example only showed a few of the form-binding tags that come with Spring. But forms are often more than text fields and Spring provides form-binding tags for all occasions. Refer to appendix B (available at www.manning.com/walls3) for the complete catalog of Spring's JSP tags.

Binding form fields is only one of the many functions offered by Spring's JSP tag libraries. Next up, let's have a look at how to externalize text on JSP pages.

14.2.2 Rendering externalized messages

In listing 14.1, the labels preceding the form fields are hard-coded in the JSP. This presents several problems:

- Any time that the same text appears on multiple pages, there exists a possibility that each occurrence will be inconsistent with the others. For example, what appears as Plate # in addRant.jsp may appear as Plate number on another page.
- If a decision is made to change the text, you must apply the change to every single JSP page where the text appears. If it only appears in one JSP then no problem—but it's a much different story if the text appears in dozens of pages.
- Hard-coded text doesn't lend itself to internationalization. Should you need to expand your application's audience across language boundaries, you'll have a hard time creating custom language versions of your application's templates.

To address these problems, Spring provides the `<spring:message>` tag. As shown in figure 14.5, `<spring:message>` renders a message from an external message properties file to the output. Using `<spring:message>` you can reference an externalized message in your JSP and have the actual message rendered when the view is rendered.

For example, listing 14.2 shows the new version of `addRant.jsp`, which uses the `<spring:message>` tag to render the field labels.

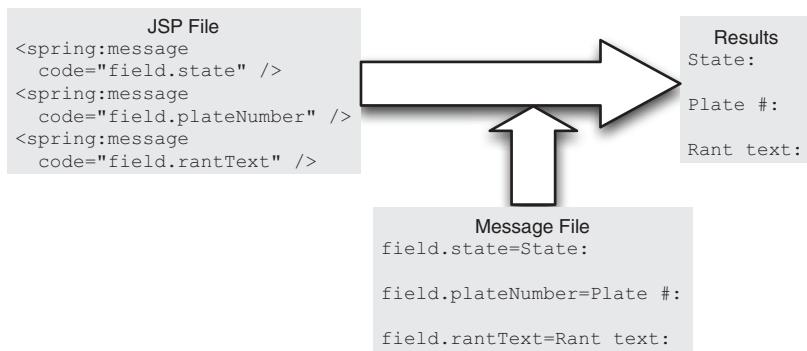


Figure 14.5 The `<spring:message>` tag resolves messages from an external message properties file.

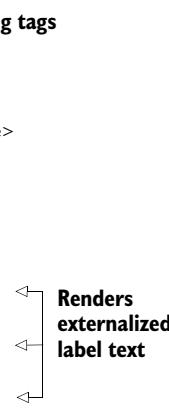
Listing 14.2 Using externalized messages in addRant.jsp

```

<%@page contentType="text/html" %>
<%@taglib prefix="form"
    uri="http://www.springframework.org/tags/form"%>
<%@taglib prefix="spring"
    uri="http://www.springframework.org/tags"%> | Uses Spring tags

<html>
  <head>
    <title><spring:message code="title.addRant" /></title>
  </head>
  <body>
    <h2><spring:message code="title.addRant" /></h2>
    <form:form method="POST" action="addRant.htm"
      commandName="rant">
      <b><spring:message code="field.state" /></b>
      <form:input path="vehicle.state" /><br/>
      <b><spring:message code="field.plateNumber" /></b>
      <form:input path="vehicle.plateNumber" /><br/>
      <b><spring:message code="field.rantText" /></b><br/>
      <form:textarea path="rantText" rows="5" cols="50" />
      <input type="submit" />
    </form:form>
  </body>
</html>

```



To use the `<spring:message>` tag, you must import the `spring` tag library by using the following `<%@taglib %>` directive on every JSP that will use the tag:

```

<%@taglib prefix="spring"
    uri="http://www.springframework.org/tags"%>

```

With this directive in place, you can reference externalized messages by passing the message code to `<spring:message>`'s `code` attribute.

Okay... so, `<spring:message>` renders externalized messages, but where exactly are these messages externalized?

Spring's `ResourceBundleMessageSource` works hand in hand with `<spring:message>` to resolve message codes to actual message values. The following `<bean>` declaration registers a `ResourceBundleMessageSource` in `roadrantz-servlet.xml`:

```

<bean id="messageSource" class=
  "org.springframework.context.support.
   ↗ ResourceBundleMessageSource">
  <property name="basename" value="messages" />
</bean>

```

The first thing to point out about this <bean> is its id. It's important to name the bean messageSource because that's the name Spring will use to look for a message source.

As for the location of the message properties file, this is determined by the basename property. Here the base name is set to messages. This means that, by default, externalized messages will be retrieved from a file called messages.properties in the classpath. The following excerpt from /WEB-INF/classes/messages.properties shows how the messages in addRant.jsp are defined:

```
field.state=State:  
field.plateNumber=Plate #:  
field.rantText=Rant text:  
title.addRant=Add a rant
```

These same properties could also be referenced by using <spring:message> on another JSP. If so then the text rendered by these messages will be consistent across all JSPs in the application. Moreover, changes to the values in messages.properties will be applied universally to all JSPs in the application.

Another benefit of using externalized messages is that it makes it simple to internationalize your application. For example, consider the following excerpt from /WEB-INF/classes/messages_es.properties:

```
field.state=Estado:  
field.plateNumber=Numero del plato:  
field.rantText=Despotrique texto:
```

This version of the properties file provides a Spanish flair to the RoadRantz application. If the user has their locale and language settings set for Spanish, the message properties will be resolved from messages_es.properties instead of the default messages.properties file and our Spanish-speaking users will be able to rant about traffic to *su contenido de corazón*.

14.2.3 Displaying errors

In chapter 13, you saw how to validate command objects using either Spring's Validator interface (listing 13.5) or Commons Validator (listing 13.6). In those examples, a failed validation resulted in a message code being placed in the Errors object. But the actual error message resides in an external properties file.

Where <spring:message> renders general messages from an external properties file, <form:errors> renders externalized error messages based on error codes in the Errors object. The newest form of addRant.jsp in listing 14.3 shows how to use <form:errors> to render validation errors.

Listing 14.3 Using externalized messages in addRant.jsp

```

<%@page contentType="text/html"%
<%@taglib prefix="form"
    uri="http://www.springframework.org/tags/form"%>
<%@taglib prefix="spring"
    uri="http://www.springframework.org/tags"%>

<html>
    <head>
        <title>Add Rant</title>
        <style>
            .error {
                color: #ff0000;
                font-weight: bold;
            }
        </style>
    </head>
    <body>
        <h2>Enter a rant...</h2>
        <form:form method="POST" action="addRant.htm"
            commandName="rant">
            <b><spring:message code="field.state" /></b>
            <form:input path="vehicle.state" />
            <form:errors path="vehicle.state" cssClass="error"/><br/>
            <b><spring:message code="field.plateNumber" /></b>
            <form:input path="vehicle.plateNumber" />
            <form:errors path="vehicle.plateNumber"
                cssClass="error"/><br/>
            <b><spring:message code="field.rantText" /></b>
            <form:errors path="rantText" cssClass="error"/><br/>
            <form:textarea path="rantText" rows="5" cols="50" />
            <input type="submit" />
        </form:form>
    </body>
</html>

```

The diagram shows the CSS rule `.error { color: #ff0000; font-weight: bold; }` annotated with the text "Defines error style". Three arrows point from the word "error" in the rule to three separate occurrences of the `<form:errors>` tag in the JSP code, each annotated with the text "Renders errors for fields".

When a field's value is rejected during validation, an error message code is associated with the field in the `Errors` object. The `<form:errors>` tag looks for any error message codes associated with the field (which is specified with the `path` attribute) and then tries to resolve those messages from an external properties file.

We could put the error messages in the same messages file as the other externalized messages. But let's keep things neat and tidy and place the error messages in their own properties file. To do this, we'll tweak the `messageSource` bean to have multiple base names instead of a single base name:

```

<bean id="messageSource" class=
    "org.springframework.context.support.
     → ResourceBundleMessageSource">
    <property name="basenames">
        <list>
            <value>messages</value>
            <value>errors</value>
        </list>
    </property>
</bean>

```

The following excerpt from /WEB-INF/classes/errors.properties shows the error messages that could result from problems with validation when adding a rant:

```

required.state=State is required.
required.plateNumber=License plate number is required.
required.rantText=Rant text is required.
invalid.plateNumber={0} is an invalid license plate number.

```

Just like other externalized messages, error messages can also be internationalized. For example, here's the Spanish version of the errors file (/WEB-INF/classes/errors_es.properties):

```

required.state=El estado se requiere.
required.plateNumber=El numero de la matricula se requiere.
required.rantText=El texto de lenguaje declamatorio se requiere.
invalid.plateNumber={0} es un numero invalido de matricula.

```

Now we've created a few JSP pages that define the view of the RoadRantz application. Up until now, we've kept the look and feel of the RoadRantz application very generic. We've focused on how to write Spring-enabled web applications with little regard for aesthetics. But how an application looks often dictates its success. To make the RoadRantz application visually appealing, it needs to be placed in a template that frames its generic pages with eye-popping graphics. Let's see how to use Jakarta Tiles, a page layout framework, with Spring MVC to dress up the application's appearance.

14.3 Laying out pages with Tiles

Jakarta Tiles is a framework for laying out pieces of a page in a template. Although originally created as part of Jakarta Struts, Tiles can be used even when the MVC framework isn't Struts. For our purposes, we're going to use Tiles alongside Spring's MVC framework.

Although we'll give a brief overview of working with Tiles, we will not go into too many details of how Tiles works. For more information on Tiles, we recommend that you read *Struts in Action* (Manning, 2002).

14.3.1 Tile views

The template for the RoadRantz application will be kept reasonably simple for the sake of brevity. It will have a header where the company logo and motto will be displayed, a footer where contact and copyright information will be displayed, and a larger area in the middle where the main content will be displayed. Figure 14.6 shows a box diagram of how the template will be laid out.

The template for the RoadRantz application is defined in `rantzTemplate.jsp` (listing 14.4).

Listing 14.4 Using Tiles to put a skin on RoadRantz

```
<%@ taglib prefix="tiles"
   uri="http://jakarta.apache.org/struts/tags-tiles" %>

<html>
  <head>
    <title><tiles:getAsString name="title"/></title>  ← Displays page title
  </head>
  <body>
    <table width="100%" border="0">
      <tr>
        <td><tiles:insert name="header"/></td>           ←
      </tr>
      <tr>
        <td valign="top" align="left">
          <tiles:insert name="content"/>                  ← Displays tile
        </td>                                         components
      </tr>
      <tr>
        <td>
          <tiles:insert name="footer"/>
        </td>
      </tr>
    </table>
  </body>
</html>
```

`rantzTemplate.jsp` uses the Tiles `<tiles:insert>` JSP tag to include content into this template. The details of where the included content originates are specified in the Tiles definition file. A Tiles definition file is XML that describes how to fill

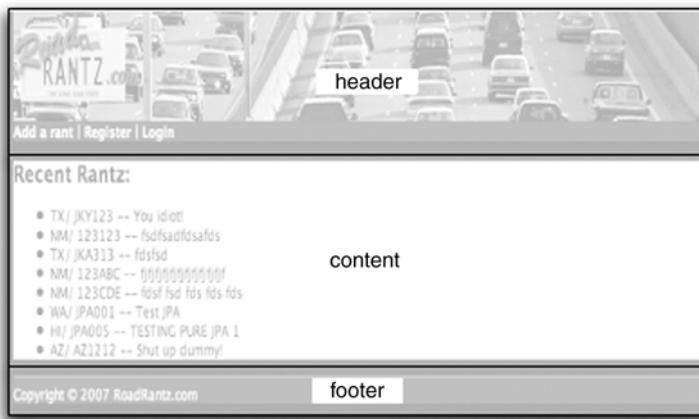


Figure 14.6 The RoadRantz application uses Tiles for page layout. To keep things simple, there are only three tiles: header, content, and footer.

in the template. The file can be named anything you want, but for the purposes of the RoadRantz application, roadrantz-tiles.xml seems appropriate.

The following excerpt from roadrantz-tiles.xml outlines the main template (called template), filling in each of its components with some default values:

```
<tiles-definitions>
<definition name="template"
    page="/WEB-INF/jsp/rantzTemplate.jsp">
    <put name="title" value="RoadRantz"/>
    <put name="header" value="/WEB-INF/jsp/header.jsp"/>
    <put name="content"
        value="/WEB-INF/jsp/defaultContentPage.jsp"/>
    <put name="footer" value="/WEB-INF/jsp/footer.jsp"/>
</definition>
...
</tiles-definitions>
```

Here, the header and footer components are given the path to JSP files that define how the header and footer should look. When Tiles builds a page, it will replace the `<tiles:insert>` tags named header and footer with the output resulting from header.jsp and footer.jsp, respectively.

As for the title and content components, they are just given some dummy values. Because it's just a template, you'll never view the template page directly. Instead, when you view another page that is based on template, the dummy values for title and content will be overridden with real values.

The homepage is a typical example of the pages in the application that will be based on `template`. It is defined in `roadrantz-tiles.xml` like this:

```
<definition name="home" extends="template">
    <put name="title" value="Welcome to RoadRantz" />
    <put name="content" value="/WEB-INF/jsp/home.jsp"/>
</definition>
```

Extending `template` ensures that the homepage will inherit all of its component definitions. However, we want each page to have a unique title and certainly need each page to have unique content. So, we have overridden the template's `title` component with "Welcome to RoadRantz" so that the page will have an appropriate title in the browser's title bar. And the main content for the homepage is defined by `home.jsp`, so the `content` component is overridden to be `/WEB-INF/jsp/home.jsp`.

So far, this is a typical Tiles-based application. You've seen nothing Spring-specific yet. But now we're ready to integrate Tiles into Spring MVC by performing these two steps:

- Configuring a `TilesConfigurer` to load the Tiles definition file
- Declaring a Spring MVC view resolver to resolve logical view names to Tiles definitions

Configuring Tiles

The first step in integrating Tiles into Spring MVC is to tell Spring to load the Tiles configuration file(s). Spring comes with `TilesConfigurer`, a bean that loads Tiles configuration files and makes them available for rendering Tiles views. To load the Tiles configuration into Spring, declare a `TilesConfigurer` instance as follows:

```
<bean id="tilesConfigurer" class="org.springframework.
    ↪ web.servlet.view.tiles.TilesConfigurer">

    <property name="definitions">
        <list>
            <value>/WEB-INF/roadrantz-tiles.xml</value>
        </list>
    </property>
</bean>
```

The `definitions` property is given a list of Tiles definition files to load. But in the case of the RoadRantz application, there's only one definition file: `roadrantz-tiles.xml`.

Resolving Tiles views

The second and final step to integrate Tiles into Spring MVC is to configure a view resolver that will send the user to a page defined by Tiles. `InternalResourceViewResolver` will do the trick:

```
<bean id="viewResolver"
      class="org.springframework.web.servlet.
      ↪ view.InternalResourceViewResolver">
  <property name="viewClass"
    value="org.springframework.web.servlet.view.tiles.
      ↪ TilesJstlView"/>
</bean>
```

Normally, `InternalResourceViewResolver` resolves logical views from resources (typically JSPs) in the web application. But for Tiles, you'll need it to resolve views as definitions in a Tiles definition file. For that, the `viewClass` property has been set to use a `TilesJstlView`.

There are actually two view classes to choose from when working with Tiles: `TilesView` and `TilesJstlView`. The difference is that `TilesJstlView` will place localization information into the request for JSTL pages. Even though we're using JSTL, we're not taking advantage of JSTL's support for internationalization. Nevertheless, we may choose to internationalize RoadRantz in the future, so there's no harm in using `TilesJstlView`. If you're not using JSTL-based views, you should use `TilesView` instead.

With `InternalResourceViewResolver` configured with `TilesJstlView` (or `TilesView`), the rules have changed. Instead of trying to resolve a view by prefixing and suffixing the logical view name, now `InternalResourceViewResolver` will resolve views by looking in the Tiles definition file(s). If a `<definition>` in the Tiles definition file has a same name that matches the logical view name, it will be used to render the page to the user.

For example, consider how the resulting view of `HomeController` is resolved. When finished, `HomeController` returns the following `ModelAndView`:

```
return new ModelAndView("home", "rantz", recentRants);
```

The logical view name is `home`, so `TilesView` will look for the view definition in the Tiles configuration. In this case, it finds the `<definition>` named `home`. Since `home` is based on `template`, the resulting HTML page will be structured like `rantzTemplate.jsp` (listing 14.4), but will have its title set to "Welcome to RoadRantz" and its content will be derived from the JSP in `/WEB-INF/jsp/home.jsp`.

Nothing about the RoadRantz controller classes will need to change to support Tiles. That's because the page definitions in `roaddrantz-tiles.xml` are cleverly named to be the same as the logical view names returned by all the controllers.

14.3.2 Creating Tile controllers

Now let's make the RoadRantz application a bit more personable. As it exists, the header is somewhat plain and boring. To make it more interesting, we'd like to put a message in the header that indicates the number of rants that have been posted for the current day.

One way to accomplish this is to place the following code in each of the controllers:

```
modelMap.add("rantsToday",
    rantService.getRantsForDay(new Date()).size());
```

This would place the number of rants for today into the request so that it can be displayed in `header.jsp` like this:

```
Helping <b>${rantsToday}</b> motorists deal
with road rage today!
```

But for this to work on all pages, you'd need to repeat the rant count code in all of the application's controller classes. There are options to eliminate the redundant code, including creating a common base controller for all other controllers to subclass or putting this functionality in a utility class used by all controllers. But all of these options add complexity that you'd like to avoid.

A unique feature of Tiles is that each component on a page can have its own controller. Be aware that this is a Tiles-specific controller, not to be confused with a Spring MVC controller. Unfortunately, the word "controller" has been overloaded here, which can lead to some confusion. As we saw in chapter 13, Spring MVC controllers process a request on behalf of `DispatcherServlet`. Tiles controllers, on the other hand, can be associated with Tiles components so that each component can perform functionality specific to that component.

To include the rant count message on each page of the RoadRantz application, we will need to build a controller for the header component. `HeaderTileController` (listing 14.5) retrieves the number of rants that have been posted for the current day and places that information into the component context for display in the banner.

Listing 14.5 Counting rants for the day using a Tiles controller

```

package com.roaddrantz.tiles;
import java.util.Date;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.apache.struts.tiles.ComponentContext;
import org.springframework.web.servlet.
    view.tiles.ComponentControllerSupport;
import com.roaddrantz.domain.Motorist;
import com.roaddrantz.service.RantService;

public class HeaderTileController
    extends ComponentControllerSupport {
    public HeaderTileController() {}

    protected void doPerform(ComponentContext componentContext,
        HttpServletRequest request, HttpServletResponse response)
        throws Exception {
        RantService rantService = getRantService();
        int rantsToday = rantService.getRantsForDay(new Date()).size();
        componentContext.putAttribute("rantsToday", rantsToday);
    }

    private RantService getRantService() {
        return (RantService) getApplicationContext().getBean(
            "rantService");
    }
}

```

**Places rant count
in Tiles context**

**Looks up
RantService**

HeaderTileController extends ComponentControllerSupport, a Spring-specific extension of Tiles's ControllerSupport class. ComponentControllerSupport makes the Spring application context available via its getApplicationContext() method. This is perfect, because HeaderTileController is going to need the Spring application context to look up the rantService bean.

HeaderTileController makes a call to getApplicationContext() and uses the application context to look up a reference to the rantService bean so that it can find out how many rants have been posted today. Once it has the information, it places it into the Tiles component context so that the header component can display it.

The only thing left to do is to associate this component controller with the header component. Revisiting the header definition in roadrantz-tiles.xml, extract the header definition and set its controllerClass attribute to point to the HeaderTileController:

```
<definition name=".header" page="/WEB-INF/jsp/header.jsp"
    controllerClass="com.roadrantz.tiles.HeaderTileController" />

<definition name="template" page="/WEB-INF/jsp/rantzTemplate.jsp">
    <put name="title" value="RoadRantz"/>
    <put name="header" value=".header"/>
    <put name="content" value="/WEB-INF/jsp/defaultContentPage.jsp"/>
    <put name="footer" value="/WEB-INF/jsp/footer.jsp"/>
</definition>
```

Now, as the page is constructed, Tiles will use HeaderTileController to set up the component context prior to displaying the header component. In header.jsp, the rant count can be displayed like this:

```
Helping <b><tiles:getAsString name="rantsToday"/></b> motorists deal
with road rage today!
```

At this point, we've developed most of the view layer of the RoadRantz application using JSP and Tiles. Now let's do it again!

JSP is the standard way of building the web layer in Java-based web applications. But, as it turns out, not everybody's a fan of JSP. Many believe that it's bloated and unnecessarily complex. If you're a JSP dissenter then read on... in the next section, we're going to look at how Spring MVC can use JSP alternatives such as Velocity and FreeMarker.

14.4 Working with JSP alternatives

In October 1908, Henry Ford rolled out the “car for the great multitude”: the Model-T Ford. The sticker price: \$950. To speed assembly, all Model-Ts were painted black because black paint dried the fastest. Legend quotes Henry Ford as saying, “Any customer can have a car painted any color that he wants so long as it is black.”

Automobiles have come a long way since 1908. In addition to a dizzying selection of body styles, you also get to choose among several options, including the type of radio/CD player, whether or not you get power windows and door locks, and cloth versus leather upholstery. And nowadays any customer can have any color that they want—including, but not limited to, black.

Up to now we've used JSP to define the view of the RoadRantz application. But unlike Henry Ford's black paint, JSP isn't the only choice when it comes to the view layer of your application. Two other popular templating engines are Velocity and FreeMarker. Let's have a look at how to use these templating engines with Spring MVC.

14.4.1 Using Velocity templates

Velocity is an easy-to-use template language for Java applications. Velocity templates contain no Java code, thus making them easy to understand by nondevelopers and developers alike. From Velocity's user guide: "Velocity separates Java code from the web pages, making the web site more maintainable over the long run and providing a viable alternative to JavaServer Pages."

Aside from JSP, Velocity is probably the most popular template language for web-based applications. So it is highly likely that you may want to develop your Spring-based application using Velocity as the view-layer technology. Fortunately, Spring supports Velocity as a view-layer templating language for Spring MVC.

Let's see how to use Velocity with Spring MVC by reimplementing the view layer of the RoadRantz application so that it's based on Velocity.

Defining the Velocity view

Suppose that you've chosen to use Velocity, instead of JSP, as the view-layer technology for the RoadRantz application. In listing 11.2, we created a JSP version of the RoadRantz homepage. But now let's look at home.vm (listing 14.6), a Velocity template that renders the home page.

Listing 14.6 A Velocity version of the RoadRantz home page

```
<html>
    <head><title>Rantz</title></head>

    <body>
        <h2>Rantz:</h2>

        <a href="addRant.htm">Add rant</a><br/>
        <a href="register.htm">Register new motorist</a><br/>
        <ul>
            #foreach($rant in $rants)
                <li>${rant.vehicle.state} /
                    ${rant.vehicle.plateNumber} --
                    ${rant.rantText}</li>
            #end
        </ul>
    </body>
</html>
```

Iterates over
list of rants

Not much is different between the Velocity template and the original JSP. But one thing you'll notice about this template is that there are no template tags. That's because Velocity isn't tag-based like JSP. Instead, Velocity employs its own lan-

guage—known as Velocity Template Language (VTL)—for control flow and other directives. In `home.vm`, the `#foreach` directive is used to loop through a list of rants, displaying rant details with each iteration.

Despite this basic difference between Velocity and JSP, you'll find that Velocity's expression language resembles that of JSP. In fact, JSP merely followed in Velocity's footsteps when using the `${}` notation in its own expression language.

This template demonstrates only a fraction of what you can do with Velocity. To learn more, visit the Velocity homepage at <http://jakarta.apache.org/velocity>.

Now that the template has been created, you'll need to configure Spring to use Velocity templates for the view in MVC applications.

Configuring the Velocity engine

The first thing to configure is the Velocity engine itself. To do this, declare a `VelocityConfigurer` bean in the Spring configuration file, as follows:

```
<bean id="velocityConfigurer" class=
  ↗ "org.springframework.web.servlet.view.velocity.
    ↗ VelocityConfigurer">
  <property name="resourceLoaderPath" value="WEB-INF/velocity/" />
</bean>
```

`VelocityConfigurer` sets up the Velocity engine in Spring. Here, we've told Velocity where to find its templates by setting the `resourceLoaderPath` property. I recommend placing the templates in a directory underneath the `WEB-INF` directory so that the templates can't be accessed directly.

If you're familiar with Velocity, you already know that you can configure the behavior of Velocity using a `velocity.properties` file. But with `VelocityConfigurer`, you can also set those configuration details by setting the `velocityProperties` property. For example, consider the following declaration of `VelocityConfigurer`:

```
<bean id="velocityConfigurer" class=
  ↗ "org.springframework.web.servlet.view.velocity.
    ↗ VelocityConfigurer">
  <property name="resourceLoaderPath" value="WEB-INF/velocity/" />
  <property name="velocityProperties">
    <props>
      <prop key="directive.foreach.counter.name">loopCounter</prop>
      <prop key="directive.foreach.counter.initial.value">0</prop>
    </props>
  </property>
</bean>
```

Here we've configured the Velocity engine, changing the behavior of the `#foreach` loop. By default, Velocity's `#foreach` loop maintains a counter variable called `$velocityCount` that starts with a value of 1 on the first iteration of the loop. But here we've set the `directive.foreach.counter.name` property to `loopCounter` so that the loop counter can be referred to with `$loopCounter`. We've also made the loop counter zero-based by setting the `directive.foreach.counter.initial.value` property to 0. (For more on Velocity configuration properties, refer to Velocity's developer guide at <http://jakarta.apache.org/velocity/developer-guide.html>.)

Resolving Velocity views

The final thing you must do to use Velocity template views is to configure a view resolver. Specifically, declare a `VelocityViewResolver` bean in the context configuration file as follows:

```
<bean id="viewResolver"
      class="org.springframework.web.servlet.view.
      ↗ velocity.VelocityViewResolver">
    <property name="suffix" value=".vm" />
</bean>
```

`VelocityViewResolver` is to Velocity what `InternalResourceViewResolver` is to JSP. Just like `InternalResourceViewResolver`, it has `prefix` and `suffix` properties that it uses with the view's logical name to construct a path to the template. Here, only the `suffix` property is set with the `.vm` extension. No prefix is required because the path to the template directory has already been set through `VelocityConfigurer`'s `resourceLoaderPath` property.

NOTE

Here the bean's ID is set to `viewResolver`. This is significant when `DispatcherServlet` is not configured to detect all view resolvers. If you are using multiple view resolvers, you'll probably need to change the ID to something more appropriate (and unique), such as `velocityViewResolver`.

At this point, your application is ready to render views based on Velocity templates. All you need to do is return a `ModelAndView` object that references the view by its logical name. In the case of `HomeController`, there's nothing to do because it already returns a `ModelAndView` as follows:

```
return new ModelAndView("home", "rants", recentRants);
```

The view's logical name is `home`. When the view is resolved, `home` will be suffixed with `.vm` to create the template name of `home.vm`. `VelocityViewResolver` will find this template in the `WEB-INF/velocity/` path.

As for the `rants` model object, it will be exposed in the Velocity template as a Velocity property. In listing 14.6, it is the collection that the `#foreach` directive iterates over.

Formatting dates and numbers

Although the application is now set to render Velocity views, we have a few loose ends to tie up. If you compare `home.vm` from listing 14.6 to `home.jsp`, you'll notice that `home.vm` doesn't apply the same formatting to the rant's posted date as in `home.jsp`. In `home.jsp`, the posted date is displayed in "full" format. For `home.vm` to be complete, you'll need to tweak it to properly format the date.

The VTL doesn't directly support date formatting. However, Velocity does have tools for date and number formatting. To enable these tools, you'll need to tell the `VelocityViewResolver` the name of the attributes to expose them through. These attributes are specified through `VelocityViewResolver`'s `dateToolAttribute` and `numberToolAttribute` properties:

```
<bean id="viewResolver"
    class="org.springframework.web.servlet.view.
        velocity.VelocityViewResolver">
<property name="suffix" value=".vm" />
<property name="dateToolAttribute">
    <value>dateTool</value>
</property>
<property name="numberToolAttribute">
    <value>numberTool</value>
</property>
</bean>
```

Here, the date tool is assigned to a `$dateTool` attribute in Velocity. So, to format the rant's posted date, all you need to do is reference the date through the number tool's `format()` function. For example:

```
$dateTool.format("FULL", rant.postedDate)
```

The first parameter is the pattern string. This string adheres to the same syntax as that of `java.text.SimpleDateFormat`. In addition, you can specify one of the standard `java.text.DateFormat` patterns by setting the pattern string to one of `FULL`, `LONG`, `MEDIUM`, `SHORT`, or `DEFAULT`. Here we've set it to `FULL` to indicate the full date format.

As mentioned, the \$numberTool attribute provides a tool for formatting numbers in the Velocity template. Refer to Velocity's documentation for more information on this tool's and the date tool's functions.

Exposing request and session attributes

Although most data that needs to be displayed in a Velocity template can be passed to the view through the model Map given to the ModelAndView object, there are times when you may wish to display attributes that are in the servlet's request or session. For example, if a user is logged into the application, that user's information may be carried in the servlet session.

It would be clumsy to copy attributes out of the request or session into the model Map in each controller. Fortunately, VelocityViewResolver can copy the attributes into the model for you. The exposeRequestAttributes and exposeSessionAttributes properties tell VelocityViewResolver whether or not you want servlet request and session attributes copied into the model. For example:

```
<bean id="viewResolver" class="org.springframework.  
    ↪ web.servlet.view.velocity.VelocityViewResolver">  
...  
    <property name="exposeRequestAttributes">  
        <value>true</value>  
    </property>  
    <property name="exposeSessionAttributes">  
        <value>true</value>  
    </property>  
</bean>
```

By default, both of these properties are false. But here we've set them both to true so that both request and session attributes will be copied into the model and therefore will be visible in the Velocity template.

Binding form fields in Velocity

Earlier in this chapter, you saw how to use Spring's JSP tag libraries to bind form fields to properties of a command object and to display error messages. Although Velocity doesn't have tags, Spring does provide a set of Velocity macros that provide equivalent functionality to Spring's JSP tag libraries. Table 14.2 lists the Velocity macros that come with Spring.

Most of the macros in table 14.2 are form-binding macros. That is, they render HTML form elements whose values are bound to a property of a command object. The specific property that they're bound to is specified in the first parameter (called path in the table). Most of the macros also have a parameter that allows

Table 14.2 Spring MVC provides a handy collection of Velocity macros that bind form fields to a controller's command object.

Macro	Purpose
#springFormCheckboxes(path options separator attributes)	Renders a set of check boxes. Checks the box(es) whose value matches that of a command object property.
#springFormHiddenInput(path attributes)	Renders a hidden field bound to a command object property.
#springFormInput(path attributes)	Renders a text field bound to a command object property.
#springFormMultiSelect(path options separator attributes)	Renders a selection list allowing multiple selection. Selected values are bound to a command object property.
#springFormPasswordInput(path attributes)	Renders a password field bound to a command object property.
#springFormRadioButtons(path options separator attributes)	Renders a set of radio buttons where the selected radio button is bound to a command object property.
#springFormSingleSelect(path options attributes)	Renders a selection list, allowing only a single entry to be selected. The selected value is bound to a command object property.
#springFormTextarea(path attributes)	Renders a text area bound to a command object property.
#springMessage(messageCode)	Renders a message externalized in a resource bundle.
#springMessageText(messageCode text)	Renders a message externalized in a resource bundle, with a default value if the message isn't found in the resource bundle.
#springShowErrors(separator class/style)	Renders validation errors.
#springUrl(relativeUrl)	Renders an absolute URL given a relative URL.

you to specify additional attributes to be placed on the rendered HTML elements (e.g., `length='20'`).

As an illustration of how to use Spring's Velocity macros, let's revisit the `addRant` view. Earlier in this chapter we defined that view as a JSP. However, listing 14.7 shows `addRant.vm`, the Velocity version of that view.

Listing 14.7 Adding a rant through a Velocity template

```
<html>
<head>
    <title>#springMessage("title.addRant")</title>
</head>

<body>
<h2>#springMessage("title.addRant")</h2>
<form method="POST" action="addRant.htm">
    <b>#springMessage("field.state")</b>#springFormInput(
        "rant.vehicle.state" "")<br/>
    <b>#springMessage("field.plateNumber")</b>#springFormInput(
        "rant.vehicle.plateNumber" "")<br/>
    #springMessage("field.rantText")
    #springFormTextarea("rant.rantText" "rows='5' cols='50'")
    <input type="submit"/>
</form>
</body>
</html>
```

For the state and plate number fields, we're using `#springFormInput` bound to `rant.vehicle.state` and `rant.vehicle.plateNumber`, respectively. This means that when the form is submitted, the values will be bound to the `state` and `plateNumber` properties of the `vehicle` property of the command object (`rant`). In both cases, there's no need to set additional attributes on the HTML, so the second parameter is empty. The resulting HTML for these two fields looks like this:

```
<b>State: </b><input type="text" id="vehicle.state"
    name="vehicle.state" value="" /> <br/>
<b>Plate #: </b><input type="text" id="vehicle.plateNumber"
    name="vehicle.plateNumber" value="" /> <br/>
```

For the text area where the user enters the rant text, an HTML `<textarea>` is in order. The `#springFormTextarea` macro renders a `<textarea>` that is bound to `rant.rantText`. When the form is submitted, the value will be bound to the `rantText` property of the command object. In this case, however, we need to set the size of the `<textarea>`, so we're passing additional attributes in the second parameter. The HTML for the text area is as follows:

```
<textarea id="rantText" name="rantText"
    rows='5' cols='50'></textarea>
```

To be able to use the Spring macros in your templates, you'll need to enable the macro using the `exposeSpringMacroHelpers` property of `VelocityViewResolver`:

```
<bean id="viewResolver"
      class="org.springframework.web.servlet.view.
      ↗ velocity.VelocityViewResolver">
    <property name="suffix" value=".vm" />
    <property name="exposeSpringMacroHelpers" value="true" />
</bean>
```

By setting the `exposeSpringMacroHelpers` property to true, you'll ensure that your Velocity templates will have access to Spring's macros for Velocity.

Although Velocity is a widely used alternative to JSP, it is not the only alternate templating option available. FreeMarker is another well-known template language that aims to replace JSP in the view layer of MVC applications. Let's see how to plug FreeMarker into your Spring MVC application.

14.4.2 Working with FreeMarker

FreeMarker is slightly more complex than Velocity, but only as the result of being slightly more powerful. FreeMarker comes with built-in support for several useful tasks, such as date and number formatting and white-space removal. These features are only available in Velocity through add-on tools.

You'll soon see how using FreeMarker with Spring MVC isn't much different than using Velocity with Spring MVC. But first things first—let's start by writing a FreeMarker template to be used in the RoadRantz application.

Constructing a FreeMarker view

Suppose that after further consideration, you decide that FreeMarker templates are more suited to your tastes than Velocity. So, instead of developing the view layer of the RoadRantz application using Velocity, you'd like to plug FreeMarker into Spring MVC. Revisiting the home page, you produce `home.ftl` (listing 14.8), the FreeMarker template that renders the home page.

Listing 14.8 A FreeMarker rendition of the RoadRantz homepage

```
<html>
  <head><title>Rantz</title></head>

  <body>
    <h2>Rantz:</h2>

    <a href="addRant.htm">Add rant</a><br/>
    <a href="register.htm">Register new motorist</a><br/>
    <ul>
      <#list rants as rant>
        <li>${rant.vehicle.state} /
          ${rant.vehicle.plateNumber} --
          ${rant.rantText}</li>
```

```
</#list>
</ul>
</body>
</html>
```

You'll notice that the FreeMarker version of the homepage isn't dramatically different from the Velocity version from listing 14.6. Just as with Velocity (and JSP, for that matter), the \${} notation is used as an expression language to display attribute values.

The home.ftl template barely scratches the surface of FreeMarker's capabilities. For more information on FreeMarker, visit the FreeMarker home page at <http://freemarker.sourceforge.net>.

Configuring the FreeMarker engine

Just like Velocity, FreeMarker's engine must be configured in order for Spring's MVC to use FreeMarker templates for rendering views. Declare a `FreeMarkerConfigurer` in the context configuration file like this:

```
<bean id="freemarkerConfig"
      class="org.springframework.web.servlet.view.
      ↗ freemarker.FreeMarkerConfigurer">
    <property name="templateLoaderPath" value="WEB-INF/freemarker/" />
</bean>
```

`FreeMarkerConfigurer` is to FreeMarker as `VelocityConfigurer` is to Velocity. You use it to configure the FreeMarker engine. As a minimum, you must tell FreeMarker where to find the templates. You do this by setting the `templateLoaderPath` property (here set to look for templates in `WEB-INF/freemarker/`).

You can configure additional FreeMarker settings by setting them as properties through the `freemarkerSettings` property. For example, FreeMarker reloads and reparses templates if five seconds (by default) have elapsed since the template was last checked for updates. But checking for template changes can be time consuming. If your application is in production and you don't expect the template to change very often, you may want to stretch the update delay to an hour or more.

To do this, modify FreeMarker's `template_update_delay` setting through the `freemarkerSettings` property. For example:

```
<bean id="freemarkerConfig"
      class="org.springframework.web.servlet.view.
      ↗ freemarker.FreeMarkerConfigurer">
    <property name="templateLoaderPath" value="WEB-INF/freemarker/" />
    <property name="freemarkerSettings">
      <props>
```

```

<prop key="template_update_delay">3600</prop>
</props>
</property>
</bean>
```

Notice that as with VelocityConfigurer's velocityProperties property, the freemarkerSettings property takes a <props> element. In this case, the only <prop> is one to set the template_update_delay setting to 3600 (seconds) so that the template will only be checked for updates after an hour has passed.

Resolving FreeMarker views

The next thing you'll need to do is to declare a view resolver for FreeMarker:

```

<bean id="viewResolver"
      class="org.springframework.web.servlet.view.
      ↗ freemarker.FreeMarkerViewResolver">
    <property name="suffix" value=".ftl" />
</bean>
```

FreeMarkerViewResolver works just like VelocityViewResolver and InternalResourceViewResolver. Template resources are resolved by prefixing a view's logical name with the value of the prefix property and are suffixed with the value of the suffix property. Again, just as with VelocityViewResolver, we've only set the suffix property because the template path is already defined in FreeMarkerConfigurer's templateLoaderPath property.

Exposing request and session attributes

In section 14.4.1, you saw how to tell VelocityViewResolver to copy request and/or session attributes into the model map so that they'll be available as variables in the template. You can do the same thing with FreeMarkerViewResolver to expose request and session attributes as variables in a FreeMarker template. To do so, set either the exposeRequestAttributes or exposeSessionAttributes property (or both) to true:

```

<bean id="viewResolver"
      class="org.springframework.web.servlet.view.
      ↗ freemarker.FreeMarkerViewResolver">
...
<property name="exposeRequestAttributes">
  <value>true</value>
</property>
<property name="exposeSessionAttributes">
  <value>true</value>
</property>
</bean>
```

Here, both properties have been set to true. As a result, both request and session attributes will be copied into the template's set of attributes and will be available to display using FreeMarker's expression language.

Binding form fields in FreeMarker

One last thing you may want to do with FreeMarker is to bind form fields to command properties. You've already seen how to use JSP binding tags and Velocity binding macros in this chapter. Not to be unfair, Spring provides a set of FreeMarker macros that mirror the functionality of the Velocity macros. The FreeMarker macros are listed in table 14.3.

Table 14.3 Spring comes with a set of FreeMarker macros useful for binding form fields to a controller's command object.

Macro	Purpose
<@spring.formCheckboxes path, options, separator, attributes />	Renders a set of check boxes. Checks the box(es) whose value matches that of a com- mand object property.
<@spring.formHiddenInput path, attributes />	Renders a hidden field bound to a command object property.
<@spring.formInput path, attributes, fieldType />	Renders a text field bound to a command object property.
<@spring.formMultiSelect path, options, attributes />	Renders a selection list allowing multiple selection. Selected values are bound to a command object property.
<@spring.formPasswordInput path, attributes />	Renders a password field bound to a com- mand object property.
<@spring.formRadioButtons path, options, separator, attributes />	Renders a set of radio buttons where the selected radio button is bound to a command object property.
<@spring.formSingleSelect path, options, attributes />	Renders a selection list allowing only a single entry to be selected. The selected value is bound to a command object property.
<@spring.formTextarea path, attributes />	Renders a text area bound to a command object property.
<@spring.message messageCode />	Render a message externalized in a resource bundle.
<@spring.messageText messageCode, text />	Renders a message externalized in a resource bundle, with a default value if the message isn't found in the resource bundle.

Table 14.3 Spring comes with a set of FreeMarker macros useful for binding form fields to a controller's command object. (continued)

Macro	Purpose
<@spring.showErrors separator, class/style />	Renders validation errors.
<@spring.url relativeUrl />	Renders an absolute URL given a relative URL.

Except for a few minor syntactical differences, the FreeMarker macros are identical to the Velocity macros. Listing 14.9 shows them in action in addRant.ftl, the FreeMarker version of the addRant view.

Listing 14.9 Entering rants using FreeMarker templates

```

<#import "/spring.ftl" as spring />    ← Imports Spring macros
<html>
  <head>
    <title><@spring.message "title.addRant"/></title>
    <style>
      .error {
        color: #ff0000;
        font-weight: bold;
      }
    </style>
  </head>

  <body>
    <h2><@spring.message "title.addRant"/></h2>
    <form method="POST" action="addRant.htm">
      <b><@spring.message "field.state"/> </b><@spring.formInput
        "rant.vehicle.state", "" /><br/>
      <b><@spring.message "field.plateNumber"/></b><@spring.formInput
        "rant.vehicle.plateNumber", "" /><br/>
      <@spring.message "field.rantText"/>
      <@spring.formTextarea "rant.rantText",
        "rows='5' cols='50'" />
      <input type="submit"/>
    </form>
  </body>
</html>

```

**Imports Spring macros
for FreeMarker**

**Binds form fields
to command**

You may have noticed that listing 14.9 is very similar to the Velocity macro in listing 14.7. But there are two subtle differences. Instead of `#springFormInput`, the FreeMarker version uses `<@spring.formInput>`. And instead of `#springFormTextarea`, `<@spring.formTextarea>` is the way to go in FreeMarker.

Also, unlike Velocity in which the macros were automatically available, FreeMarker macros must be imported. The first line of addRant.ftl imports the Spring form-binding macros.

Just as with Spring's Velocity, macros, in order to use these macros you must enable the FreeMarker macros by setting the `exposeMacroHelpers` property of `FreeMarkerViewResolver` to true:

```
<bean id="viewResolver"
      class="org.springframework.web.servlet.view.
      ➔ freemarker.FreeMarkerViewResolver">
    <property name="suffix" value=".ftl" />
    <property name="exposeSpringMacroHelpers" value="true" />
</bean>
```

Now you have three capable templating options for building web applications in Spring. But all of these options produce HTML. Sometimes HTML isn't enough and you need something a bit richer. Let's switch gears and look at how to generate non-HTML output.

14.5 Generating non-HTML output

Up to now, the output produced by the RoadRantz web layer has been HTML based. Indeed, HTML is the typical way to display information on the Web. But HTML doesn't always lend itself to the information being presented.

For example, if the data you are presenting is in tabular format, it may be preferable to present information in the form of a spreadsheet. Spreadsheets may also be useful if you want to enable the users of your application to manipulate the data being presented.

Or perhaps you'd like precise control over how a document is formatted. Formatting HTML documents precisely is virtually impossible, especially when viewed across various browser implementations. But Adobe's Portable Document Format (PDF) has become the *de facto* standard for producing documents with precise formatting that are viewable on many different platforms.

Spreadsheets and PDF files are commonly static files. But Spring provides view classes that enable you to dynamically create spreadsheets and PDF documents that are based on your application's data.

Let's explore Spring's support for non-HTML views, starting with dynamic generation of Excel spreadsheets.

14.5.1 Producing Excel spreadsheets

If you've been developing software for any considerable length of time, you've probably noticed that Microsoft Excel spreadsheets are the lifeblood of business. For good or bad, businesspeople love their spreadsheets. They communicate, analyze, chart, plan, and budget using spreadsheets. If it weren't for Excel, many businesses would come to a grinding halt.

With such a fondness in the workplace for spreadsheets, it's important to be able to produce spreadsheets from your applications. And there's good news: Spring comes with `AbstractExcelView`, a custom view suitable for producing spreadsheets.

Although the data presented in RoadRantz is hardly mission-critical business data, we can only assume that many motorists on the road are businesspeople. And it's quite possible that while driving they are thinking about facts and figures (and spreadsheets) and aren't driving well. Since these motorists have such a tight connection with spreadsheets, it may prove worthwhile to provide those businesspeople with a list of their vehicle's rants in spreadsheet form.

Spring supports the generation of Excel output through `AbstractExcelView`. As its name implies, this is an abstract class that you must subclass to give the details of the spreadsheet.

As an example of how to subclass `AbstractExcelView`, consider `RantExcelView` (listing 14.10). This view implementation produces a list of rants within an Excel spreadsheet.

Listing 14.10 Listing rants in a spreadsheet

```
package com.roadrantz.mvc;
import java.util.Collection;
import java.util.Iterator;
import java.util.Map;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.apache.poi.hssf.usermodel.HSSFCellStyle;
import org.apache.poi.hssf.usermodel.HSSFDataFormat;
import org.apache.poi.hssf.usermodel.HSSFRow;
import org.apache.poi.hssf.usermodel.HSSFSheet;
import org.apache.poi.hssf.usermodel.HSSFWorkbook;
import org.springframework.web.servlet.view.document.
    ↗ AbstractExcelView;
import com.roadrantz.domain.Rant;
import com.roadrantz.domain.Vehicle;

public class RantExcelView extends AbstractExcelView {
    protected void buildExcelDocument(
```

```

Map model, HSSFWorkbook workbook,
HttpServletRequest request, HttpServletResponse response)
throws Exception {

Collection rants = (Collection) model.get("rants");
Vehicle vehicle = (Vehicle) model.get("vehicle");

HSSFSheet sheet = createSheet(workbook,
    vehicle.getPlateNumber());

HSSFCellStyle cellStyle = workbook.createCellStyle();
cellStyle.setDataFormat(
    HSSFDataFormat.getBuiltinFormat("m/d/yy h:mm"));

int rowNum = 1;
for (Iterator iter = rants.iterator(); iter.hasNext();) {
    Rant rant = (Rant) iter.next();
    rowNum = addRantRow(sheet, cellStyle, rowNum, rant);
}
}

private int addRantRow(HSSFSheet sheet, HSSFCellStyle cellStyle,
    int rowNum, Rant rant) {
    HSSFRow row = sheet.createRow(rowNum++);
    row.createCell((short) 0).setCellValue(rant.getPostedDate());
    row.createCell((short) 1).setCellValue(rant.getRantText());
    row.getCell((short) 1).setCellStyle(cellStyle);
    return rowNum;
}

private HSSFSheet createSheet(HSSFWorkbook workbook,
    String plateNumber) {
    HSSFSheet sheet = workbook.createSheet(
        "Rants for " + plateNumber);

    HSSFRow header = sheet.createRow(0);
    header.createCell((short) 0).setCellValue("Date");
    header.createCell((short) 1).setCellValue("Text");
    return sheet;
}
}

```

Sets date format

Adds rants to spreadsheet

Adds header row

There's a lot going on in `RantExcelView`, but the only method required by `AbstractExcelView` is `buildExcelDocument()`. This method is given a `Map` of model data to use when constructing the spreadsheet. It is also given an `HttpServletRequest` and `HttpServletResponse`, in case information is required that isn't available in the model data. Only the model data is used in `RantExcelView`, however.

`buildExcelDocument()` is also provided with an `HSSFWorkbook`. `HSSFWorkbook` is a component of Jakarta POI that represents an Excel workbook.³ Implementing

a custom Excel view in Spring MVC is simply a matter of using POI's API to render model data in a workbook. In `RantExcelView`, the list of rants is extracted from the model data and is iterated over to add rows to the spreadsheet.

To make `RantExcelView` available to Spring MVC, we need to register it with either `ResourceBundleViewResolver` or `XmlViewResolver`. Since we're using `XmlViewResolver`, the following entry in `roadrantz-views.xml` will do the trick:

```
<bean id="vehicleRants.xls"
      class="com.roadrantz.mvc.RantExcelView" />
```

Now all we need to demonstrate `RantExcelView` is a controller that places a list of rants into the model data. It just so happens that `RantsForVehicleController` already does everything we need. The only problem is that `RantsForVehicleController` is already being used to render HTML output. We'll need to modify it a bit to be more flexible with regard to its view choices.

The first thing to do is to add a URL mapping for the Excel request. Add the following `<prop>` entry to the `mappings` property of the `urlMapping` bean:

```
<prop key="/rantsForVehicle.xls">
    rantsForVehicleController
</prop>
```

So, if `DispatcherServlet` receives a request for `/rantsForVehicle.xls`, it knows to dispatch the request to the `rantsForVehicleController` bean. This is the same controller that's mapped to `/rantsForVehicle.htm`. But how will it know to use the Excel view instead of the JSP view?

The request's URI provides a clue as to the type of view. The request URI already ends with `htm` for HTML requests. For Excel requests, we'll map it to end with `xls`. The following `getViewName()` method extracts the extension of the URI and uses it to derive the view name:

```
private static final String BASE_VIEW_NAME = "vehicleRants";
private String getViewName(HttpServletRequest request) {
    String requestUri = request.getRequestURI();
    String extension = "." +
        requestUri.substring(requestUri.lastIndexOf("."));
    if("htm".equals(extension)) { extension=""; }
    return BASE_VIEW_NAME + extension;
}
```

³ The “HSSF” in `HSSFWorkbook` and other POI classes is an acronym for “Horrible SpreadSheet Format.” Apparently, the POI developers have formed an opinion of the Excel file format.

If the URI ends with `htm` then it's an HTML request and we're going to let `InternalResourceViewResolver` resolve to a JSP view. Otherwise, the view name will be `vehicleRants` followed by the extension of the URI.

Next we need to modify the `handle()` method of `RantsForVehicleController` to use the `getViewName()` method when choosing a view:

```
protected ModelAndView handle(HttpServletRequest request,
    HttpServletResponse response, Object command,
    BindException errors) throws Exception {
    ...
    return new ModelAndView(getViewName(request), model);
}
```

There's just one more thing needed to make this work. `DispatcherServlet` will never receive a request for `/rantsForVehicle.xls` because it is configured in `web.xml` to handle `*.htm` requests. We'll need to configure another `<servlet-mapping>` as follows:

```
<servlet-mapping>
    <servlet-name>roadrantz</servlet-name>
    <url-pattern>*.xls</url-pattern>
</servlet-mapping>
```

Now `DispatcherServlet` will handle both `*.htm` and `*.xls` requests and the `RoadRantz` application is capable of producing Excel lists of rants.

The rant spreadsheet appeals to those businesspeople who are accustomed to working in Excel. But spreadsheets may seem intimidating to some people. For those users, a friendlier output is desirable. To make those users happy, let's see how Spring supports rendering of PDF documents.

14.5.2 Generating PDF documents

PDF documents are commonly used on the Internet to depict information in a format that is both precise in layout and universal. Although Cascading Style Sheets (CSS) go a long way to provide professional layout capabilities to HTML, they have their limitations. Conversely, the contents of a PDF document can be laid out in virtually any arrangement.

Furthermore, CSS implementations vary across different browsers, whereas PDF documents are rendered identically in Adobe's Acrobat Viewer, regardless of the platform.

Suppose that in addition to rendering a rant list in Excel, you'd like to offer a PDF version of the rant list. With PDF, you can dress up the output a bit and be certain that it will appear the same for all users.

Spring's `AbstractPdfView` supports rendering of PDF documents as a view in Spring MVC. Much like `AbstractExcelView`, you'll need to subclass `AbstractPdfView` and implement the `buildPdfDocument()` method to construct a PDF document.

`RantPdfView` (listing 14.11) is an example of a class that extends `AbstractPdfView` to produce a list of rants in PDF form.

Listing 14.11 Generating a PDF report of rants

```
package com.roaddrantz.mvc;
import java.util.Iterator;
import java.util.List;
import java.util.Map;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.springframework.web.servlet.view.document.
    ↗ AbstractPdfView;
import com.lowagie.text.Document;
import com.lowagie.text.Table;
import com.lowagie.text.pdf.PdfWriter;
import com.roaddrantz.domain.Rant;

public class RantPdfView extends AbstractPdfView {
    protected void buildPdfDocument(Map model, Document document,
        PdfWriter pdfWriter, HttpServletRequest request,
        HttpServletResponse response) throws Exception {
        List rants = (List) model.get("rants");
        Table rantTable = new Table(4);   ← Creates table
        rantTable.setWidth(90);
        rantTable.setBorderWidth(1);

        rantTable.addCell("State");
        rantTable.addCell("Plate");
        rantTable.addCell("Date Posted");
        rantTable.addCell("Text");      | Adds
                                       header
                                       row

        for (Iterator iter = rants.iterator(); iter.hasNext();) {
            Rant rant = (Rant) iter.next();

            rantTable.addCell(rant.getVehicle().getState());
            rantTable.addCell(rant.getVehicle().getPlateNumber());
            rantTable.addCell(rant.getPostedDate().toString());
            rantTable.addCell(rant.getRantText());      | Adds row
                                                       for each
                                                       rant

        }
        document.add(rantTable);
    }
}
```

Like `AbstractExcelView`'s `buildExcelDocument()` method, the `buildPdfDocument()` method is provided with a Map of model data, along with an `HttpServletRequest` and `HttpServletResponse`. But it is also provided with a `Document` and a `PdfWriter`. These two classes are part of the iText PDF library (www.lowagie.com/iText) and are used to construct a PDF document. For more information on iText, I recommend *iText in Action* (Manning, 2006).

The `Document` object passed to `buildPdfDocument()` is an empty iText document waiting to be filled with content. In `RantPdfView`, the rant list is pulled from the model and used to construct a table, with one row per rant. Once all rants have been added to the table, the table is added to the `Document`.

To make `RantPdfView` available to Spring MVC, let's add it to `roadrantz-views.xml` alongside `RantExcelView`:

```
<bean id="vehicleRants.pdf"
      class="com.roadrantz.mvc.RantPdfView" />
```

Now any controller that returns a `ModelAndView` whose view name is `vehicleRants.pdf` will have its view rendered by `RantPdfView`.

In section 14.5.1, we altered `RantsForVehicleController` to dynamically choose its view based on the request URI's extension. No further changes need to be made to `RantsForVehicleController` to use `RantPdfView`. We just need to register a URL mapping so that `DispatcherServlet` will dispatch requests for `/rantsForVehicle.pdf` to `RantsForVehicleController`:

```
<prop key="/rantsForVehicle.pdf">
    rantsForVehicleController
</prop>
```

Also, as with Excel views, we need to create a `<servlet-mapping>` in `web.xml` that will direct `*.pdf` requests to `DispatcherServlet`:

```
<servlet-mapping>
    <servlet-name>roadrantz</servlet-name>
    <url-pattern>*.pdf</url-pattern>
</servlet-mapping>
```

Spring's `AbstractExcelView` and `AbstractPdfView` make quick work of producing Excel and PDF documents. But what if you need to produce output that isn't covered by Spring's out-of-the-box solutions? Let's look at how to develop custom view implementations for Spring MVC.

14.5.3 Developing custom views

Excel and PDF documents are great. But syndication is all the rage on the Internet. One of the requirements for the RoadRantz application is to syndicate a vehicle's rants as a Rich Site Summary (RSS) feed.

RSS is an XML dialect that concisely describes website content so that it can be subscribed to. It's often used to subscribe to newsfeeds such as Fox (www.foxnews.com/rss/index.html) and CNN (www.cnn.com/services/rss). And RSS feeds are the best way to stay caught up on your favorite weblogs (such as www.jroller.com/rss/habuma). RSS is a great way to keep RoadRantz users up-to-date on the latest rants that are being written about their vehicle.

Unfortunately, Spring doesn't come prepackaged with RSS-producing view classes. Fear not, however. This presents us with an opportunity to develop a custom RSS view.

RantRssView (listing 14.12) is a view implementation that uses the Rome (<https://rome.dev.java.net>) utility to produce RSS output for a list of rants.

Listing 14.12 Syndicating rants with RSS

```
package com.roaddrantz.mvc;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
import java.util.Map;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.springframework.web.servlet.view.AbstractView;
import com.roaddrantz.domain.Rant;
import com.sun.syndication.feed.synd.SyndContent;
import com.sun.syndication.feed.synd.SyndContentImpl;
import com.sun.syndication.feed.synd.SyndEntry;
import com.sun.syndication.feed.synd.SyndEntryImpl;
import com.sun.syndication.feed.synd.SyndFeed;
import com.sun.syndication.feed.synd.SyndFeedImpl;
import com.sun.syndication.io.SyndFeedOutput;

public class RantRssView extends AbstractView {
    private String author;
    private String title;
    private String description;
    private String link;

    public RantRssView() {}

    protected void renderMergedOutputModel(Map model,
        HttpServletRequest request, HttpServletResponse response)
        throws Exception {
        SyndFeed feed = createFeed(); ← Creates feed
```

```

List rants = (List)model.get("rants");
List entries = new ArrayList();

for (Iterator iter = rants.iterator(); iter.hasNext();) {
    Rant rant = (Rant) iter.next();
    entries.add(createEntry(rant));
}

feed.setEntries(entries);

SyndFeedOutput output = new SyndFeedOutput();
output.output(feed, response.getWriter());
}

private SyndEntry createEntry(Rant rant) {
    SyndEntry entry = new SyndEntryImpl();

    entry.setTitle("Rant entry for " +
        rant.getVehicle().getPlateNumber());
    entry.setLink("http://www.roadrantz.com");
    entry.setPublishedDate(rant.getPostedDate());
    SyndContent content = new SyndContentImpl();
    content.setType("text/html");
    content.setValue(rant.getRantText());
    entry.setDescription(content);

    return entry;
}

private SyndFeed createFeed() {
    SyndFeed feed = new SyndFeedImpl();
    feed.setFeedType("rss_1.0");
    feed.setAuthor(author);
    feed.setTitle(title);
    feed.setDescription(description);
    feed.setLink(link);

    return feed;
}

public void setAuthor(String author) {
    this.author = author;
}

public void setDescription(String description) {
    this.description = description;
}

public void setLink(String link) {
    this.link = link;
}

public void setTitle(String title) {
    this.title = title;
}
}

```

Adds rant entries to feed

Outputs feed

RantRssView extends Spring's AbstractView class. The only method that AbstractView requires RantRssView to implement is renderMergedOutputModel(). This method is given the Model object along with an HttpServletRequest and HttpServletResponse and is expected to render the output to the HttpServletResponse's writer.

Here RantRssView uses Rome's API to create a feed, populate it with rant entries (expected to be passed in the model), and then render the output.

Once again, RantsForVehicleController will be used to produce the list of rants for this view. For the RSS feed, we'll map this controller to /rantsForVehicle.rss:

```
<prop key="/rantsForVehicle.rss">
    rantsForVehicleController
</prop>
```

Since the URI will end in rss, the view name returned from getViewName() will be vehicleRants.rss. The following entry in roadrantz-views.xml will help XmlViewResolver find the custom RSS view:

```
<bean id="vehicleRants.rss"
    class="com.roadrantz.mvc.RantRssView">
    <property name="title" value="RoadRantz" />
    <property name="description" value="RoadRantz.com" />
    <property name="author" value="RoadRantz.com" />
    <property name="link" value="http://www.roadrantz.com" />
</bean>
```

Notice that some of the feed details (title, description, author and link) are configured on the view using dependency injection.

14.6 Summary

A web application isn't a web application unless it interacts with the user. In an MVC web application, the application interacts with the user through the view.

In the previous chapter, you learned how to handle web requests with controllers and to produce model data to be presented to the user. In this chapter you've seen several ways to render model data in the view layer.

HTML is the primary means of rendering information on the web. Views based on JSP, Velocity, or FreeMarker make simple work of dynamically producing HTML output. And Spring provides a rich selection of custom JSP tags and Velocity/FreeMarker macros for binding form input to command objects.

Once you've built a set of basic pages for your application using JSP, Velocity, or FreeMarker, you'll want to adorn them with a visually pleasant template. For that,

we've seen how `TilesView` and `TilesJstlView` help integrate Jakarta Tiles into the view of a Spring MVC application.

When HTML isn't good enough, Spring steps up with built-in support for views that produce Excel spreadsheets and PDF documents. You've also seen that it's possible to create custom views by implementing Spring's `View` interface.

Bridging the controllers in chapter 11 with the views in this chapter are view resolvers. Spring offers several out-of-the-box view controllers from which to choose.

By now you've seen that Spring MVC maintains a loose coupling between request handling, controller implementations, and views. This is a powerful concept, allowing you to mix-'n'-match different Spring MVC parts to build a web layer most appropriate for your application.

Coming up in the next chapter, we're going to build on what we know about Spring MVC by looking at an exciting new extension to Spring MVC called Spring Web Flow. Spring Web Flow provides a new controller class for Spring MVC that enables you to define a navigational flow to your web application, where the application guides the user through a series of steps to achieve a certain goal.

Using Spring Web Flow

15

This chapter covers

- Creating conversational web applications
- Defining flow states and actions
- Integrating Spring Web Flow with Struts and JSF

Have you ever given much thought to what might be the most important force that drives successful software?

There are many opinions out there on this topic. Some say proper methodology leads to successful software projects. Others say that a cleverly schemed architecture sets the foundation for software to flourish. Some might tell you it's the people on the project who determine its outcome. Depending on who you ask, you might be told that certain technology choices lend themselves to triumphant software.

As a seasoned software developer, I have given much thought to this question. And I think I have the answer. Methodology, architecture, people, and technology are certainly important factors that play into a project's success. But I submit that something else is far more critical to whether or not a project is a blockbuster or a dud. And that something is...

Pizza.

That's right... it's pizza. (Go ahead. Get your highlighter out. You'll want to mark this for future reference.)

Every successful project team I've been on has, at one time or another (sometimes frequently), enjoyed a meal together with pizza. Not only is pizza a universal favorite food choice among programmers, but I believe that there's something about a warm slice of melted cheese over crust, adorned with meats and veggies, that inspires a project to greatness. Nothing brings out design ideas and camaraderie like breaking bread over a cardboard box full of pizza.

For a moment, just for fun, imagine that you and your team have just released the latest version of your product and want to treat your team to some pizza to unwind. What would the phone call to the local pizzeria sound like?

Ring... Ring... Ring...

"Hello, this is Spring Pizza, home of the hottest slice in town. May I have your telephone number, please?"

"Sure, it's 972-555-1312."

"Is this for 1414 Oak Drive?"

"Yeah."

"Okay, will this be for carryout or delivery?"

"Delivery."

"Great. What can we get for you?"

"I need two large carnivores"

"Okay, anything else?"

"Yeah, how about two large veggie pizzas..."

"Okay..."

“...and a medium Canadian bacon with pineapple.”

“Anything else?”

“No, that’ll be it.”

“Okay, that’s two large veggies, two large carnivores, and a medium Canadian bacon with pineapple. Is that correct?”

“That’s right.”

“Okay, your total comes to \$44.57. Will that be cash, check, or charge?”

“I’ll pay cash.”

“All right. Your order should be delivered in about 30 minutes. Have a good day and thank you for calling Spring Pizza.”

Okay, I admit it. That was a bit corny. Nevertheless, this conversation (or one similar to it) takes place millions of times every day between pizza lovers and those who bake the cherished Italian pie.

This exchange isn’t much different from how a user might interact with a web application. Some applications seem to follow a script, guiding the user along in a conversation to achieve a certain goal. In the case of the phone call, the goal was to have some pizzas delivered. Similarly, you’ve probably used the shopping cart of an e-commerce site where the goal was to order and pay for a product. A conversation takes place in both scenarios: one between a customer and a pizzeria employee and one between a customer and a web application.

Spring Web Flow is an exciting new web framework based on the Spring Framework that facilitates development of conversational web applications. In this chapter, we’re going to explore Spring Web Flow and see how it fits into the Spring web framework landscape. Along the way, we’ll build a conversational web application that simulates the pizza order phone call.

Before we get too carried away with taking pizza orders, however, there’s a bit of groundwork to be laid. Let’s start by seeing what itch is scratched by Spring Web Flow.

15.1 Getting started with Spring Web Flow

There are two kinds of interaction in web applications. Many websites are based on free-flow navigation. That is, the user is given an array of links and buttons to choose from and they’re in full control of the application’s flow. The conversation between the user and the web application is very one-sided—the user tells the application where to and the application goes there.

But occasionally you come across web applications where the application guides the user from one page to the next. There seems to be a conversation

where the application asks some questions to which the user responds, triggering some functionality in the application. Although the user may have several choices to make, the application follows a predefined flow. The shopping cart of most online shopping sites is a small, but typical, example of conversational web interaction.

Spring Web Flow is an extension to Spring MVC (actually, it's just another controller) that provides for the development of conversation-style navigation in a web application. The key features provided for by Spring Web Flow are:

- The ability to define an application's flow external to the application's logic
- The ability to create reusable flows that can be used across multiple applications

In chapter 13, we saw how to build web applications using Spring MVC. Spring MVC's controllers are great for developing free-flow applications but are ill-suited for conversational web applications.

To understand Spring MVC's shortcomings with regard to conversational applications, let's suppose that the phone call between the pizzeria employee and the customer were to take place in an online pizza order entry application. If we were to build the pizza order entry application using Spring MVC, we'd likely end up coding the application's flow into each controller and JSP.

For example, we might end up with a JSP with a link that looks like this:

```
<a href="addPizza.htm">Add Pizza</a>
```

As for the controller behind that link, it may be a form controller like this one:

```
public class AddPizzaController extends SimpleFormController {  
    public AddPizzaController() {}  
  
    protected ModelAndView onSubmit(Object command,  
        BindException bindException) throws Exception {  
  
        Pizza pizza = (Pizza) command;  
  
        addPizzaToOrder(pizza);  
  
        return new ModelAndView("orderDetail");  
    }  
}
```

While there's nothing inherently wrong with that link or the controller, it isn't ideal when taken in the context of the conversational pizza application. That's because both the link and the controller know too much. The link's URL and the controller's ModelAndView each hold a piece of information about the appli-

cation's flow. But there's no place to go to see the complete picture; instead, the application's flow is embedded and scattered across the application's JSPs and controllers.

Because the flow definition is sprinkled throughout the application's code, it's not easy to understand the overall flow of the application. To do so would require viewing the code for several controllers and JSPs. Moreover, changing the flow would be difficult because it would require changing multiple source files.

What about AbstractWizardFormController?

At this point, you may be wondering why Spring Web Flow is necessary when we have `AbstractWizardFormController` (see section 13.3.4). At first glance, `AbstractWizardFormController` seems to be a way to build conversation into a Spring MVC application. But upon closer inspection you'll realize that `AbstractWizardFormController` is just a form controller where the form is spread across multiple pages. Moreover, the "flow" of a subclass of `AbstractWizardFormController` is still embedded directly within the controller implementation, making it hard to discern flow logic from application logic.

As you'll soon see, Spring Web Flow loosens the coupling between an application's code and its page flow by enabling you to define the flow in a separate, self-contained flow definition. But before we can define an application flow, there's a little bit of groundwork that has to be dealt with. Let's see how to set up the infrastructural pieces of a Spring Web Flow application.

15.1.1 Installing Spring Web Flow

Spring Web Flow, although a subproject of the Spring Framework, isn't part of the Spring Framework proper. Therefore, before we can get started building flow-based applications, we'll need to add Spring Web Flow to our project's classpath.

You can download Spring Web Flow from the Spring Web Flow website (<http://www.springframework.org/webflow>). Be sure to get the latest version (as I write this, Spring Web Flow 1.0.3 has just been released). Once you've downloaded and unzipped the distribution zip file, you'll find the Spring Web Flow JAR files in the root directory. Add these to your application's classpath and you're ready to go.

It's even easier to add Spring Web Flow to your application if you're using Maven 2. In the `pom.xml` file, add the following dependencies:

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webflow</artifactId>
    <version>1.0.3</version>
    <scope>compile</scope>
</dependency>

<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-binding</artifactId>
    <version>1.0.3</version>
    <scope>compile</scope>
</dependency>
```

With Spring Web Flow set up in the application's classpath, we're ready to start configuring it in the Spring application context. We'll start with `FlowController`, the gateway controller to Spring Web Flow.

Setting up the `FlowController`

All interactions with Spring Web Flow go through Spring Web Flow's `FlowController`. `FlowController` is a Spring MVC controller that acts as a front controller for Spring Web Flow applications. However, instead of performing a specific function like most Spring MVC controllers, `FlowController` is responsible for handling all requests pertaining to a flow.

As with any other Spring MVC controller, `FlowController` must be declared in a Spring application context. The following XML shows a typical `FlowController` `<bean>` declaration:

```
<bean id="flowController"
      class="org.springframework.webflow.executor.mvc.
            ↗ FlowController">
    <property name="flowExecutor" ref="flowExecutor" />
</bean>
```

The `flowExecutor` property is the only mandatory property. It must be wired with a flow executor, which ultimately carries out the steps described in a flow. We'll declare a flow executor bean in a moment, but first we have some Spring MVC plumbing to take care of with regard to `FlowController`.

So that we'll have a way of interacting with Spring Web Flow through a URL, we'll also need to configure a mapping to the `FlowController`. This mapping can be created using any of the handler mappings described in chapter 13, but we tend to favor `SimpleUrlHandlerMapping`:

```
<bean id="urlMapping"
      class="org.springframework.web.servlet.
            ↗ handler.SimpleUrlHandlerMapping">
```

```
<property name="mappings">
    <props>
        <prop key="flow.htm">flowController</prop>
    </props>
</property>
</bean>
```

The reason why we chose `SimpleUrlHandlerMapping` here is because we may end up mapping other controllers in the same application. We think that `SimpleUrlHandlerMapping` is the most flexible of Spring MVC's handler mappings, allowing us to map virtually any URL pattern to any controller.

As mapped here, `FlowController` will answer to the URL pattern `flow.htm`. It's worth noting that the same mapping could also be created in a more convention-over-configuration approach using `ControllerClassNameHandlerMapping`:

```
<bean id="urlMapping"
    class="org.springframework.web.servlet.mvc.
        support.ControllerClassNameHandlerMapping" />
```

`ControllerClassNameHandlerMapping` may be an appealing choice if your application will be completely flow-based or if your application's other controllers are named appropriately with respect to the URL patterns they'll be mapped to.

Because `FlowController` is a Spring MVC controller, you'll also need to be sure to configure `DispatcherServlet` in your application's `web.xml` file as you would do for any Spring MVC application. See section 13.1.2 for a refresher on how `DispatcherServlet` should be configured.

Configuring a flow executor

While `FlowController` handles web requests destined for Spring Web Flow, it doesn't execute the flow. It is merely a courier between the user and a *request executor*. A flow executor's job is like that of an air traffic controller. It keeps track of all of the flows that are currently being executed and directs each flow to the state they should go to next.

As you've seen already, the `FlowController` must be wired with a reference to a flow executor. This means that we'll need to configure a flow executor in Spring.

If you're using Spring 2.0, the easiest way to configure a flow executor is to use Spring Web Flow's custom configuration schema. The first thing you must do is add Spring Web Flow's schema to your Spring application context's XML file:

```
<beans xmlns=
    "http://www.springframework.org/schema/beans"
    xmlns:xsi=
        "http://www.w3.org/2001/XMLSchema-instance"
    xmlns:flow=
```

```

"http://www.springframework.org/schema/
    ↗ webflow-config"
xsi:schemaLocation=
    "http://www.springframework.org/schema/beans
     http://www.springframework.org/schema/beans/
         ↗ spring-beans-2.0.xsd
     http://www.springframework.org/schema/webflow-config
     http://www.springframework.org/schema/webflow-config/
         ↗ spring-webflow-config-1.0.xsd">
...
</beans>

```

Once you've added the schema declaration, declaring a flow executor is as simple as using the `<flow:executor>` element like this:

```

<flow:executor id="flowExecutor"
    registry-ref="flowRegistry" />

```

Under the covers, the `<flow:executor>` element will be used to declare a `FlowExecutorFactoryBean` in the Spring application context. If you're using a pre-2.0 version of Spring (or if typing large amounts of XML gives you a warm feeling of satisfaction) then you can manually declare a `FlowExecutorFactoryBean` as follows:

```

<bean id="flowExecutor"
    class="org.springframework.webflow.config.
        ↗ FlowExecutorFactoryBean">
<property name="definitionLocator"
    ref="flowRegistry"/>
<property name="executionAttributes">
    <map>
        <entry key="alwaysRedirectOnPause">
            <value type="java.lang.Boolean">true
                ↗ </value>
        </entry>
    </map>
</property>
<property name="repositoryType"
    value="CONTINUATION"/>
</bean>

```

In Spring Web Flow, flow definitions are defined in separate XML files. `FlowExecutorFactoryBean` uses a flow registry to keep track of all of the flow definitions that it may need. The flow registry is given to the flow executor through `<flow:executor>`'s `registry-ref` attribute or through `FlowExecutorFactoryBean`'s `definitionLocator` property.

Registering flow definitions

The flow registry is effectively a librarian that curates a collection of flow definitions. When the flow executor needs a flow, it will ask the flow registry for it.

A flow registry can be configured in Spring 2.0 using the `<flow:registry>` element, as follows:

```
<flow:registry id="flowRegistry">
    <flow:location
        path="/WEB-INF/flows/**/*-flow.xml" />
</flow:registry>
```

The `<flow:registry>` element must contain one or more `<flow:location>` elements. Each `<flow:location>` element identifies the path to one or more flow definitions that the flow registry should manage. Notice that in the example above, the path is defined using Ant-style wildcards. This indicates that all files ending with `-flow.xml` that are in the `/WEB-INF/flows/` directory (and subdirectories) should be loaded by the flow registry.

When a flow definition is loaded into the flow registry, it is registered with a name that is equal to the filename of the flow definition after chopping off the file extension. For example, if a flow definition file is named `Pizza-flow.xml`, it will be registered in the flow registry with the name `Pizza-flow`. This name will be used to refer to the flow when constructing URLs.

If you're not using Spring 2.0 or would just prefer to configure Spring Web Flow using the traditional `<bean>` element, you can also configure a flow registry as a `<bean>` using the following XML:

```
<bean id="flowRegistry"
    class="org.springframework.webflow.engine.
        ↗ builder.xml.XmlFlowRegistryFactoryBean">
    <property name="flowLocations">
        <list>
            <value>/WEB-INF/flows/**/*-flow.xml </value>
        </list>
    </property>
</bean>
```

As you can see, `XmlFlowRegistryFactoryBean` is the class that hides behind the curtain of the `<flow:registry>` element. And its `flowLocations` property is the pre-2.0 means of itemizing the flow definitions to be loaded into the registry.

With the core Spring Web Flow configuration in place, we're almost ready to build our pizza order flow. But first, let's establish the core concepts of a flow.

15.1.2 Spring Web Flow essentials

In Spring Web Flow, three main elements make up an application flow: *states*, *events*, and *transitions*.

States are points in a flow where some activity takes place. This activity could be the application performing some logic (perhaps saving customer information to a database), or it could be where the user is presented with a page and asked to take some action.

Spring Web Flow defines six different kinds of state, as shown in table 15.1.

Table 15.1 Spring Web Flow's selection of states.

State type	XML element	What it's for
Action	<action-state>	Action states are where the logic of the flow takes place. Action states typically store, retrieve, derive, or otherwise process information to be displayed or processed by subsequent states.
Decision	<decision-state>	Decision states branch the flow in two or more directions. They examine information within flow scope to make flow routing decisions.
End	<end-state>	The end state is the last stop for a flow. Once a flow has reached end state, the flow is terminated and the user is sent to a final view.
Start	<start-state>	The start state is the entry point for a flow. A start state does not do anything itself and effectively serves as a reference point to bootstrap a flow.
Subflow	<subflow-state>	A subflow state starts a new flow within the context of a flow that is already underway. This makes it possible to create flow components that can be composed together to make more complex flows.
View	<view-state>	A view state pauses the flow and invites the user to participate in the flow. View states are used to communicate information to the user or to prompt them to enter data.

The selection of states provided by Spring Web Flow makes it possible to construct virtually any arrangement of functionality into a conversational web application. While not all flows will require all of the states described in table 15.1, you'll probably end up using most of them at one time or another.

Of all the states in table 15.1, you'll probably find that your flow is mostly composed of view states and action states. These two states represent each side of a conversation between the user and the application. View states are the user's side

of the conversation, presenting information to the user and awaiting their response. Action states are the application’s side of the conversation, where the application performs some application logic in response to a user’s input or the results of another action state.

Once a state has completed, it fires an *event*. The event is simply a String value that indicates the outcome of the state. By itself, the event fired by a state has no meaning. For it to serve any purpose, it must be mapped to a *transition*. Whereas a state defines an activity within a flow, transitions define the flow itself. A transition indicates which state the flow should go to next.

To illustrate these concepts, consider the simple flow diagram in figure 15.1.

The flow in figure 15.1 is intended to show a typical “hello world” flow. The flow starts by transitioning to a view state named “hello” that displays the familiar “hello world” greeting. Once the user indicates that they are finished, a “continue” event is fired, triggering a transition to the end state.

This simple flow can be expressed in Spring Web Flow with the following XML:

```
<?xml version="1.0" encoding="UTF-8"?>
<flow xmlns="http://www.springframework.org/schema/webflow"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation=
          "http://www.springframework.org/schema/webflow
          http://www.springframework.org/schema/webflow/
          spring-webflow-1.0.xsd">

    <start-state idref="hello" />

    <view-state id="hello"
                view="hello">
        <transition on="continue" to="finish"/>
    </view-state>

    <end-state id="finish"
                view="flowRedirect:Hello-flow" />
</flow>
```

Observe that all three states are clearly defined in this XML document. Furthermore, the transitions from one state to the next are also easy to spot.

The “hello” flow is a good introduction to Spring Web Flow, but it’s only a small taste of what’s in store. Rather than dwell on this flow too long, let’s see how to put Spring Web Flow to work in a more sizable example. Let’s build a pizza order entry application using Spring Web Flow.



Figure 15.1
A flow is made up of states, events, and transitions. This flow has three states, two transitions, and two events.

15.1.3 Creating a flow

In the next section, we're going to build an application for pizza order entry using Spring Web Flow. In doing so, you'll see that we'll be able to define the application's flow completely external to the application code and views. This will make it possible to rearrange the flow without requiring any changes to the application itself. It will also aid in understanding the overall flow of the application because the flow's definition is contained in a single location.

15.2 Laying the flow groundwork

To define the pizza order flow, we'll start by creating a skeleton flow definition file. It'll start out rather empty, but it will be full of state and transition definitions before this chapter is done.

Flows in Spring Web Flow are defined in XML. Regardless of the specifics of the flow, all flow definition files are rooted with the `<flow>` element:

```
<?xml version="1.0" encoding="UTF-8"?>
<flow xmlns=
       "http://www.springframework.org/schema/webflow"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation=
       "http://www.springframework.org/schema/webflow
       http://www.springframework.org/schema/webflow/
       ↗ spring-webflow-1.0.xsd">

...
</flow>
```

The first thing you should understand about Spring Web Flow definition files is that they are based on a completely different XML schema than the Spring container configuration file. Where the Spring configuration file is used to declare `<bean>`s and how they are related to each other, a Spring Web Flow definition declares flow states and transitions.

15.2.1 Flow variables

The whole purpose of the pizza order flow is to build a pizza order. Therefore, we'll need a place to store the order information. Listing 15.1 shows the `Order` class, a simple domain bean for carrying pizza order information.

Listing 15.1 The Order class represents an order in the pizza flow

```
package com.springinaction.pizza.domain;
import java.io.Serializable;
import java.util.ArrayList;
```

```
import java.util.List;

public class Order implements Serializable {
    private Customer customer;
    private List<Pizza> pizzas;
    private Payment payment;

    public Order() {
        pizzas = new ArrayList<Pizza>();
        customer = new Customer();
    }

    public Customer getCustomer() {
        return customer;
    }

    public void setCustomer(Customer customer) {
        this.customer = customer;
    }

    public List<Pizza> getPizzas() {
        return pizzas;
    }

    public void setPizzas(List<Pizza> pizzas) {
        this.pizzas = pizzas;
    }

    public void addPizza(Pizza pizza) {
        pizzas.add(pizza);
    }

    public Payment getPayment() {
        return payment;
    }

    public void setPayment(Payment payment) {
        this.payment = payment;
    }
}
```

So that the `Order` object is available to all of the states in the flow, we'll need to declare it in the flow definition file with a `<var>` element:

```
<var name="order"
      class="com.springinaction.pizza.domain.Order"
      scope="flow"/>
```

Notice that the `scope` attribute is set to `flow`. Flow variables can be declared to live in one of four different scopes, as listed in table 15.2.

Since we'll need the `Order` object throughout the entire life of the flow, we've set the `scope` attribute to `flow`.

Table 15.2 Scopes that data can live in within a flow.

Scope	Visibility
Request	If an object is created in Request scope, it is only visible within the context of the current request. Request-scoped variables do not survive redirects.
Flash	An object created in Flash scope is visible within the context of the current request and until the next user event is triggered. Flash-scoped variables live beyond redirects.
Flow	If an object is created in Flow scope, it will be visible within the context of the current flow execution, but will not be visible to subflows.
Conversation	Objects created in Conversation scope are visible within the context of the current flow execution as well as in the context of subflow executions.

Now let's add the first states to our flow.

15.2.2 Start and end states

All flows begin with a start state. A start state is to a flow what a `main()` method is to a Java program. That is to say, it exists only as a marker of where the flow should begin. The only thing that occurs within a start state is that a transition is performed to the next state.

In Spring Web Flow, a start state is defined in XML with a `<start-state>` element. All flows must have exactly one `<start-state>` to indicate where the flow should begin. Here's how we've defined the start state for the pizza order flow:

```
<start-state idref="askForPhoneNumber" />
```

This `<start-state>` definition is very typical of any `<start-state>` in any flow definition. In fact, the `idref` attribute, which indicates the beginning state of the flow, is the only attribute available to `<start-state>`. In this case, we're indicating that the flow should begin with a state named `askForPhoneNumber`. (We'll define `askForPhoneNumber` state in a moment.)

Just as all flows must start somewhere, they all eventually must come to an end. Therefore, we must also define an end state in the flow:

```
<end-state id="finish"
    view="orderComplete" />
```

The `<end-state>` element defines the hopping-off point for a flow. When a flow transitions to this state, there's no turning back. An end state terminates the flow and then displays a view specified by the `view` attribute. In this case, we've asked the flow to send the user to the view whose name is `orderComplete`. This logical

view name is ultimately mapped to an actual view implementation using a Spring MVC view resolver.

For example, suppose that we have configured an `InternalResourceViewResolver` in Spring like this:

```
<bean id="viewResolver"
      class="org.springframework.web.servlet.view.
          InternalResourceViewResolver">
    <property name="prefix" value="/WEB-INF/jsp/" />
    <property name="suffix" value=".jsp" />
</bean>
```

When the flow ends, the `orderComplete` view name will be resolved by `InternalResourceViewResolver` to the actual view implementation in `/WEB-INF/jsp/orderComplete.jsp`. (For a review of `InternalResourceViewResolver`, see section 14.1.1.)

As an alternative to jumping out of the flow completely, we could define the `end` state to start over with a brand new flow. For example, consider the following `<end-state>` definition:

```
<end-state id="finish"
           view="flowRedirect:PizzaOrder-flow" />
```

Now, instead of setting the `view` attribute to the logical name of a view, we're using the `flowRedirect:` directive to tell Spring Web Flow to redirect the user to the beginning of a flow. In this case, we're redirecting to the same flow we just finished, so that the user can enter another pizza order.

Although all flows must have exactly one `<start-state>`, a flow can have as many `<end-state>`s as you want to define alternate endings for a flow. Each flow can take the user to a different view after the flow ends. In the case of the pizza order flow, we only need one `<end-state>`. Nevertheless, we thought you might like to know that a flow can have more than one.

The basic shell of the pizza order flow is in place. The state diagram in figure 15.2 illustrates what we've built thus far.

We're now ready to start adding basic functionality in the form of states and transitions. Since the flow's `<start-state>` indicates that the flow should start with a state named `askForPhoneNumber`, that's where we'll also start in defining the part of the flow that collects customer information.

15.2.3 Gathering customer information

A key step in taking a pizza order is to find out who the customer is and where they live. After all, if you don't know where they live, how can you possibly deliver

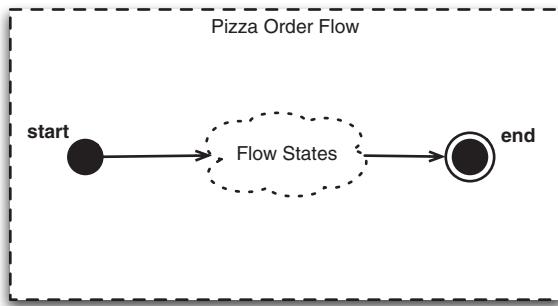


Figure 15.2
The start and end states bookend all Spring Web Flow definitions.

their pizza to them? In figure 15.3, we've fleshed out the flow diagram from figure 15.2 a bit to describe the portion of the flow that collects the customer information.

As is customary in the pizza delivery trade, we'll start by asking the customer for their phone number. If the customer has placed an order with us before, we'll already have the rest of their information and the phone number is all we'll need to look up their address. If the phone number doesn't turn up any customer data, we'll need to ask the user for their address.

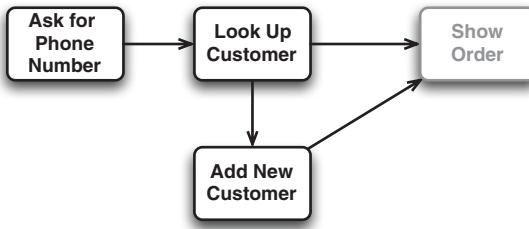


Figure 15.3
Find out where to deliver the pizza by gathering customer information in the flow.

Asking the customer for their phone number

The first step in acquiring customer information is to ask the user for their phone number. This is a simple matter of presenting the user with a web page with a form for entering the phone number.

View states are used to involve a user in a flow by displaying information and possibly asking the user for input. This makes a view state the perfect choice for presenting the phone number form to the user. The following <view-state> element should do the trick:

```
<view-state id="askForPhoneNumber"
    view="phoneNumberForm">
```

```
<transition on="submit" to="lookupCustomer" />
</view-state>
```

The view attribute of `<view-state>` specifies the logical name of a view to be displayed to the user. In this case `phoneNumberForm` is resolved to `/WEB-INF/jsp/phoneNumberForm.jsp` by the `InternalResourceViewResolver` we declared earlier. A simplified form of `phoneNumberForm.jsp` is shown in listing 15.2.

Listing 15.2 A form that asks for a customer's phone number

```
<h2>Customer Lookup</h2>
<form method="post" action="flow.htm"> <-- Submits to flow.htm
    <input type="hidden" name="_flowExecutionKey"
           value="${flowExecutionKey}"> <-- Identifies flow
    <b>Phone number: </b>
    <input type="text" name="phoneNumber"/><br/>
    <input type="submit" class="button"
           name="_eventId_submit" value="Submit"/> <-- Triggers submit event
</form>
```

Listing 15.2 contains a basic HTML form with a text field to prompt for the phone number and a submit button. There are, however, a few details of this form that are specific to Spring Web Flow.

Recall that all links within a Spring Web Flow application should go through `FlowController`'s URL mapping. Likewise, forms should also be submitted to the same URL. Therefore, the `action` parameter of the `<form>` element is set to `flow.htm`, which as we know from section 15.1.1 is the URL pattern mapped to the `FlowController`.

So that `FlowController` will know which flow the request is for, we must also identify the flow by setting the `_flowExecutionKey` parameter. For that reason, we've added a hidden field named `_flowExecutionKey` that holds the flow execution key that will be submitted along with the form data.

The final thing to note is the odd name given to the submit button. Clicking this button triggers an event to Spring Web Flow from a form submission. When the form is submitted, the name of this parameter is split into two parts. The first part, `_eventId`, signals that we're identifying the event. The second part, `submit`, is the name of the event to be triggered when the form is submitted.

Looking back at the `askForPhoneNumber` `<view-state>` declaration, we see that the `submit` event triggers a transition to a state named `lookupCustomer`, where the form will be processed to look up the customer information.

Looking up customer data

When using `<view-state>` to prompt the user for a phone number, we allowed the user to take part in the flow. Now it's the application's turn to perform some work as it attempts to look up the customer data in an action state.

The `lookupCustomer` state is defined as an `<action-state>` with the following excerpt of XML:

```
<action-state id="lookupCustomer">
    <action bean="lookupCustomerAction" />

    <transition on="success"
        to="showOrder" />

    <transition on-exception=
        "com.springinaction.pizza.service.
         ↗ CustomerNotFoundException"
        to="addNewCustomer" />
</action-state>
```

The action implementation is referenced by the `bean` attribute of the `<action>` subelement. Here, we've specified that the functionality behind the `lookupCustomer` action state is defined in a bean named `lookupCustomerAction`. The `lookupCustomerAction` bean is configured in Spring as follows:

```
<bean id="lookupCustomerAction"
    class="com.springinaction.pizza.flow.
        ↗ LookupCustomerAction">
    <property name="customerService"
        ref="customerService" />
</bean>
```

The bean referenced by the `bean` attribute of `<action>` must implement Spring Web Flow's Action interface. As you can see in `LookupCustomerAction` (listing 15.3), the only compulsory method of the Action interface is the `execute()` method.

Listing 15.3 A flow action for looking up customer information

```
package com.springinaction.pizza.flow;
import org.springframework.webflow.execution.Action;
import org.springframework.webflow.execution.Event;
import org.springframework.webflow.execution.RequestContext;
import com.springinaction.pizza.domain.Customer;
import com.springinaction.pizza.domain.Order;
import com.springinaction.pizza.service.CustomerService;

public class LookupCustomerAction implements Action {
    public Event execute(RequestContext context)
        throws Exception {
```

```

String phoneNumber =
    context.getRequestParameters().get("phoneNumber");

Customer customer =
    customerService.lookupCustomer(phoneNumber);           | Looks up
                                                       | customer

Order order =
    (Order) context.getFlowScope().get("order");
order.setCustomer(customer);                         | Sets customer to
                                                       | flow-scoped order

    return new Event(this, "success");      | Returns
}                                              | success event

// injected
private CustomerService customerService;
public void setCustomerService(
    CustomerService customerService) {
    this.customerService = customerService;
}
}

```

`LookupCustomerAction` assumes that it will be processing the submission of the `askForPhoneNumber` view state and retrieves the phone number from the request parameters. It then passes that phone number to the `lookupCustomer()` method of the injected `CustomerService` object to retrieve a `Customer` object. If a `Customer` is found, the pizza order is retrieved from flow scope and its `customer` property is set accordingly.

At this point, we have all of the customer information we need, so we're ready to start adding pizzas to the order. So the `execute()` method concludes by returning a `success` event. Looking back at the definition of the `lookupCustomer <action-state>`, we see that a `success` event results in a transition to the `showOrder` event.

If, however, `lookupCustomer()` can't find a `Customer` based on the given phone number, it will throw a `CustomerNotFoundException`. In that case, we need the user to add a new customer. Therefore, the `lookupCustomer <action-state>` is also declared with an exception transition that sends the flow to the `addNewCustomer` state if a `CustomerNotFoundException` is thrown from `LookupCustomerAction`'s `execute()` method.

Adding a new customer

As for the `addNewCustomer` state itself, it's a view state that prompts the user for customer information. The following `<view-state>` definition specifies that the `newCustomerForm` view should be displayed by this state:

```
<view-state id="addNewCustomer" view="newCustomerForm">
...
</view-state>
```

As configured here, the addNewCustomer view state will display the view defined in /WEB-INF/jsp/newCustomerForm.jsp (listing 15.4).

Listing 15.4 A form for creating a new customer

```
<%@ taglib prefix="form"
    uri="http://www.springframework.org/tags/form" %>

<h2>New customer</h2>
<form:form action="flow.htm" commandName="order.customer">
    <input type="hidden" name="_flowExecutionKey" value="${flowExecutionKey}" />

    <b>Phone: </b> ${requestParameters.phoneNumber} <br/>
    <b>Name: </b> <form:input path="name" /><br/>
    <b>Street address: </b>
        <form:input path="streetAddress" /><br/>
    <b>City: </b> <form:input path="city" /><br/>
    <b>State: </b> <form:input path="state" /><br/>
    <b>Zip: </b> <form:input path="zipCode" /><br/>
    <input type="submit" class="button" name="_eventId_submit" value="Submit" />
    <input type="submit" class="button" name="_eventId_cancel" value="Cancel" />
</form:form>
```

The code is annotated with several callout boxes:

- A box labeled "Submits form to FlowController" points to the opening tag of the `<form:form>` element.
- A box labeled "Identifies current flow execution" points to the `value="${flowExecutionKey}"` attribute of the `<input>` element.
- A box labeled "Uses Spring form-binding JSP tags" points to the `<form:input>` elements.
- A box labeled "Submits form and fires submit event" points to the first `<input type="submit" ...>` element.
- A box labeled "Submits form and fires cancel event" points to the second `<input type="submit" ...>` element.

Although the addNewCustomer state will display the new customer form, it's not able to process the form data. Eventually the user will submit the form and we'll need a way to bind the form data to a back-end Customer object. Fortunately, Spring Web Flow provides `FlowAction`, a special Spring Web Flow Action implementation that knows how to deal with common form-binding logic. To use `FlowAction`, the first thing we'll need to do is configure it as a `<bean>` in the Spring application context:

```
<bean id="customerFormAction"
    class="org.springframework.webflow.action.FormAction">
    <property name="formObjectName" value="customer" />
    <property name="formObjectScope" value="REQUEST" />
    <property name="formObjectClass"
        value="com.springinaction.pizza.domain.Customer" />
</bean>
```

FormAction has three important properties that describe the object that will be bound to the form. The `formObjectName` property specifies the name of the object, `formObjectScope` specifies the scope, and `formObjectClass` specifies the type. In this case we're asking FormAction to work with a `Customer` object in request scope as `customer`.

When the form is first displayed, we'll need FormAction to produce a blank `Customer` object so that we'll have an object to bind the form data to. To make that happen, we'll add FormAction's `setupForm()` method as a render action in the `addNewCustomer` state:

```
<view-state id="addNewCustomer" view="newCustomerForm">
    <render-actions>
        <action bean="customerFormAction"
               method="setupForm"/>
    </render-actions>
    ...
</view-state>
```

Render actions are a way of associating an action with the rendering of a view. In this case, the FormAction's `setupForm()` method will be called just before the customer form is displayed and will place a fresh `Customer` object in request scope. In a sense, FormAction's `setupForm()` is a lot like a Spring MVC form controller's `formBackingObject()` (see section 13.3.3) in that it prepares an object to be bound to a form.

When the new customer form is submitted, we'll need a transition to handle the `submit` event. The following `<transition>` addition to `addNewCustomer` describes what should happen:

```
<view-state id="addNewCustomer" view="newCustomerForm">
    <render-actions>
        <action bean="customerFormAction"
               method="setupForm"/>
    </render-actions>
    <transition on="submit" to="showOrder">
        <action bean="customerFormAction" method="bind" />
        <evaluate-action expression=
            "flowScope.order.setCustomer(requestScope.customer)" />
    </transition>
</view-state>
```

The `<transition>` element itself is mostly straightforward. It simply says that a `submit` event should trigger a transition to the state named `showOrder`. But before we go to the `showOrder` state, we must first bind the form data to the form-backing

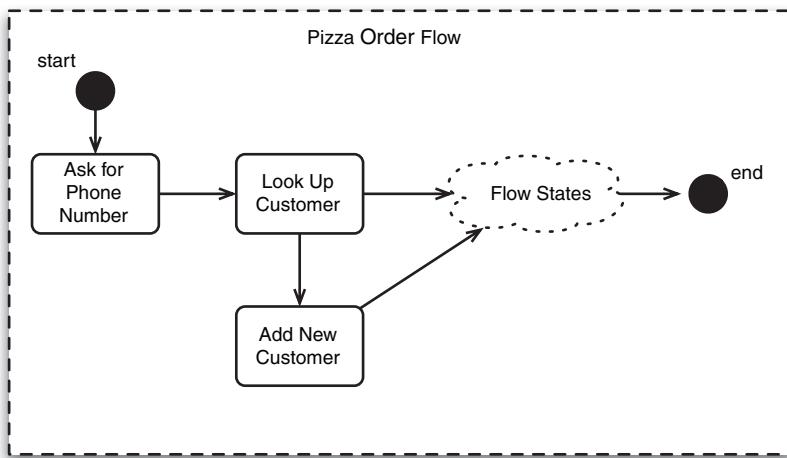


Figure 15.4 Where the customer information states fit into the pizza order flow.

Customer object and set the `customer` property of the flow-scoped `Order`. This is a two-step process:

- 1 First, we use an `<action>` element to ask `FormAction`'s `bind()` method to bind the submitted form data to the form-backing `Customer` object that is in request scope.
- 2 Next, with a fully populated `Customer` object in request scope, we use `<evaluate-action>` to copy the request-scoped `Customer` object to the `Order` object's `customer` property.

Our pizza order flow now has all of the customer information gathering states it needs. At this point, the overall flow looks little like what you see in figure 15.4.

Unfortunately, there's still that fuzzy cloud of yet-to-be-defined "flow states" that needs to be addressed. Let's add a few more states to the flow that enable us to add pizzas to the order.

15.2.4 Building a pizza order

When it comes to defining a flow for creating a pizza order, the most critical part is where the pizzas get added to the order. Without that, we're not really delivering anything to the customer (which in most businesses results in there being no customer).

So, the next couple of states we introduce to the flow will serve to build the pizza order. Figure 15.5 shows the new states and how they'll relate to one another.

In this section, we'll add two states to our flow: one that displays the current order and one to let the user add a pizza to the order.

Displaying the order

The showOrder state is a simple `<view-state>` that renders a view to display the current order information. It is defined as follows:

```
<view-state id="showOrder" view="orderDetails">
    <transition on="addPizza" to="addPizza" />
    <transition on="continue" to="takePayment" />
</view-state>
```

There's nothing particularly special about this `<view-state>` definition. Compared to some `<view-state>`s we've seen already, this one is plain vanilla. It simply renders the view whose name is `orderDetails`. `InternalResourceViewResolver` will resolve this view name to `/WEB-INF/jsp/orderDetails.jsp`, which is the JSP file shown in listing 15.5.

Listing 15.5 `orderDetails.jsp`, which displays the current pizza order to the user

```
<%@ taglib prefix="c"
    uri="http://java.sun.com/jstl/core" %>
<%@ taglib prefix="fmt"
    uri="http://java.sun.com/jstl/fmt" %>

<h2>${order.customer.name}</h2>
<b>${order.customer.streetAddress}</b><br/>
<b>${order.customer.city}, ${order.customer.state}<br/>
    ${order.customer.zipCode}</b><br/>
<b>${order.customer.phoneNumber}</b><br/>
<br/>

<a href="flow.htm?_flowExecutionKey=${flowExecutionKey}
    &_eventId=continue">Place Order</a>
<a href="flow.htm?_flowExecutionKey=${flowExecutionKey}
    &_eventId=cancel">Cancel</a>
<hr/>
<h3>Order total: <fmt:formatNumber type="currency"
    value="${order.total}" /></h3>
```

Displays
customer
info

Links to fire
continue event

Links to fire
cancel event

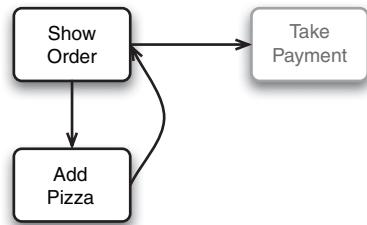


Figure 15.5 Adding states to the flow to build the pizza order.

```

<hr/>
<h3>Pizzas:</h3>
<small>
    <a href="flow.htm?_flowExecutionKey=${flowExecutionKey}
        ↪ &_eventId=addPizza">Add Pizza</a>
</small>
<br/>
<c:forEach items="${order.pizzas}" var="pizza">
<li>${pizza.size} :
    <c:forEach items="${pizza.toppings}" var="topping">
        ${topping},
    </c:forEach>
</li>
</c:forEach>

```

Links to fire addPizza event

Displays pizzas

The main things I'd like to draw your attention to in listing 15.5 are the three links that are being created to fire Spring Web Flow events. One fires a continue event to proceed with checkout. Another fires a cancel event to cancel the order and start over. Another fires an addPizza event so that we can add a new pizza to the order.

Notice that all three of these links are very similar. Consider the addPizza link, for example:

```
<a href="flow.htm?_flowExecutionKey=${flowExecutionKey}
    ↪ &_eventId=addPizza">Add Pizza</a>
```

There are three very important elements of the link's href attribute that guide Spring Web Flow:

- As we've discussed before, all links within a flow must go through the FlowController. Therefore the root of the link is `flow.htm`, the URL pattern mapped to the FlowController.
- To identify the flow execution to Spring Web Flow, we've set the `_flowExecutionKey` parameter to the page-scoped `${flowExecutionKey}` variable. This way FlowController will be able to distinguish one user's flow execution from another.
- Finally, the `_eventId` parameter identifies the event to fire when this link is clicked on. In this case, we're firing the `addPizza` event, which, as defined in the `showOrder` state, should trigger a transition to the `addPizza` state.

Speaking of the `addPizza` state, let's go ahead and add it to the flow.

Adding a pizza to the order

Since we'll be prompting the user to choose a pizza, it makes sense for the addPizza state to be a view state. Here's the <view-state> definition we'll use:

```
<view-state id="addPizza" view="newPizzaForm">
    <render-actions>
        <action bean="pizzaFormAction" method="setupForm" />
    </render-actions>
    <transition on="submit" to="showOrder">
        <action bean="pizzaFormAction" method="bind" />
        <evaluate-action expression=
            "flowScope.order.addPizza(requestScope.pizza)" />
    </transition>
</view-state>
```

If this <view-state> definition looks familiar, it's because we used a similar pattern when adding a new customer. Just as with the customer form, we're using a FormAction to set up and bind the form data. As you can see from the definition of the pizzaFormAction bean, this time the form-backing object is a Pizza object:

```
<bean id="pizzaFormAction"
    class="org.springframework.webflow.action.FormAction">
    <property name="formObjectName" value="pizza" />
    <property name="formObjectClass"
        value="com.springinaction.pizza.domain.Pizza" />
    <property name="formObjectScope" value="REQUEST" />
</bean>
```

When the new pizza form is submitted, the <evaluate-action> copies the request-scoped Pizza into the order by calling the flow-scoped Order object's

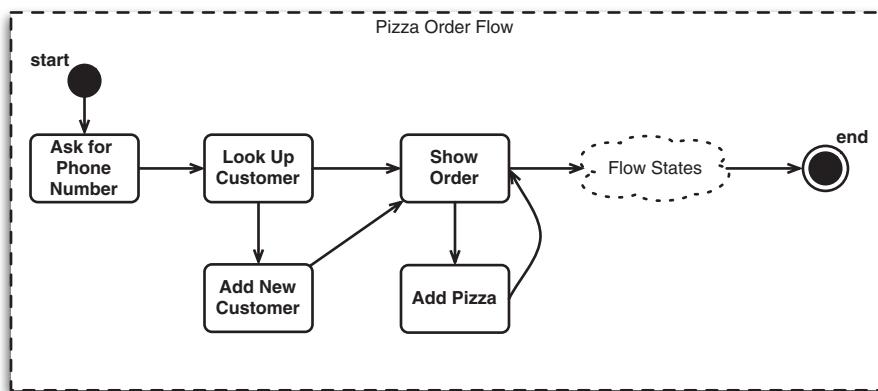


Figure 15.6 The pizza order flow after adding the showOrder and addPizza states.

`addPizza()` method. Once the pizza has been added to the order, the flow transitions back to the `showOrder` view state to display the order's status.

Our flow is really starting to take shape. Figure 15.6 shows how the overall flow looks at this point.

We're almost finished with our pizza order flow definition. But there are still a few more states left to wrap up the order process. Let's complete the flow definition by defining the states that complete the pizza order.

15.2.5 Completing the order

In the last section I stated that building the pizza order is the most critical part of the flow. After further contemplation, however, I think that is the most critical part only from the customer's point of view. From the pizzeria's point of view, the most critical part of the flow is the part where we get paid for making and delivering the pizzas. (You didn't think we were giving pizzas away, did you?)

To complete the pizza order flow (and to take the money out of our customer's pockets), we'll add two more states to the flow, as illustrated in figure 15.7.

Let's start by adding that all-important flow state—the one that sucks money right out of the customer's credit card.

Taking payment

In order to take payment, we must prompt the user for credit card information. Since we're asking the user to get involved again, this means that a view state is in order. The following `<view-state>` displays the payment form to the user and processes the credit card payment:

```
<view-state id="takePayment" view="paymentForm">
    <transition on="submit" to="submitOrder">
        <bean-action bean="paymentProcessor"
            method="approveCreditCard">
            <method-arguments>
                <argument expression=
                    "${requestParameters.creditCardNumber}" />
                <argument expression=
                    "${requestParameters.expirationMonth}" />
                <argument expression=
                    "${requestParameters.expirationYear}" />
                <argument expression=
                    "${flowScope.order.total}" />
            </method-arguments>
        </bean-action>
    </transition>
</view-state>
```

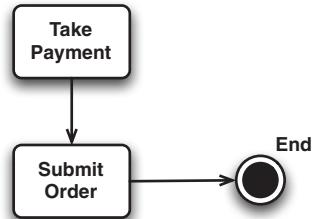


Figure 15.7 The final two states in the flow `takePayment` and `submitOrder`.

```

        </method-arguments>
    </bean-action>
</transition>
<transition on-exception=
            "com.springinaction.pizza.PaymentException"
            to="takePayment" />
</view-state>
```

Once again, we've defined a less-than-simple `<view-state>`. What makes this `<view-state>` interesting is that in addition to displaying a form, it processes the form submission upon a submit event. What makes it *really* interesting is how we process the form data.

In some previously defined `<view-state>`s, we used Spring Web Flow's Form-Action to bind form data to an object, but we never really performed any processing on those objects. This time, we're doing more than just collecting data—we need to approve the credit card transaction by passing the form data to a payment processor.

The payment processor, as referenced by the bean attribute of `<bean-action>`, is just a bean in the Spring application context. It may've been configured in Spring like this:

```

<bean id="paymentProcessor" class=
    "com.springinaction.pizza.service.PaymentProcessor" />
```

The specifics of the `PaymentProcessor` class and its implementation aren't pertinent to this discussion other than to say that `PaymentProcessor` must expose an `approveCreditCard()` method whose signature looks like this:

```

public void approveCreditCard(String creditCardNumber,
    String expMonth, String expYear,
    float amount) throws PaymentException {
    ...
}
```

The reason why this `approveCreditCard()` method is needed is because the method attribute of `<bean-action>` points to an `approveCreditCard()` method that takes four arguments, as enumerated in the `<argument>` elements contained within `<method-arguments>`. The arguments are:

- The first argument is the credit card number. The value passed in to `approveCreditCard()` comes directly from the request parameters, as indicated by the expression `${requestParameters.creditCardNumber}`. This assumes that the `paymentForm` view contains a form with a field named `creditCardNumber`.

- Likewise, the next two arguments are the credit card's expiration month and year and are also pulled from the request parameters. Again, it is assumed that the `paymentForm` view contains fields named `expirationMonth` and `expirationYear`.
- Finally, we must know the amount of the transaction for which we're approving payment. This information is readily available from the flow-scoped `Order` object's `total` property and is specified using the `${flowScope.order.total}` expression.

When the form in the `paymentForm` view is submitted, the credit card number and expiration date fields, along with the order total, will be passed along in a call to `PaymentProcessor`'s `approveCreditCard()` method before transitioning to the `submitOrder` state.

In the event that the payment can't be approved, `approveCreditCard()` will throw a `PaymentException`. For this occasion, we've also included an on-exception transition that will take the user back to the `paymentForm` to remedy the problem (perhaps by trying a different credit card number).

It should be noted that we could have also used an `<action>` definition here instead of a `<bean-action>`. The downside of using `<action>`, however, is that the action class must be an implementation of Spring Web Flow's `Action` interface. It doesn't allow for pure POJO action implementations. `<bean-action>`, on the other hand, provides a nonintrusive alternative, enabling us to invoke any method on any class configured in the Spring application context, without implementing a Spring Web Flow-specific interface.

Submitting the order

Finally, we're ready to submit the order. The user doesn't need to be involved for this part of the flow, so an `<action-state>` is a suitable choice. The following `<action-state>` saves the order and finishes the flow:

```

<action-state id="submitOrder">
    <bean-action bean="orderService" method="saveOrder">
        <method-arguments>
            <argument expression="${flowScope.order}" />
        </method-arguments>
    </bean-action>

    <transition on="success" to="finish" />
</action-state>

```

Notice that we're using a `<bean-action>` to define the logic behind the `<action-state>`. Here, the `saveOrder()` method will be called on the bean whose name is

orderService. The flow-scoped Order object will be passed in as a parameter. This means that the class behind the orderService bean must expose a saveOrder() method whose signature looks like this:

```
public void saveOrder(Order order) {  
    ...  
}
```

As before, the actual implementation of this method isn't relevant to the discussion of Spring Web Flow, so we've purposefully omitted it to avoid confusion.

The flow now appears to be complete. We have all of the states in place and we should be able to start taking pizza orders. But we're missing one small transition that is used throughout the entire flow.

15.2.6 A few finishing touches

On more than one occasion, we've seen links that fire cancel events within the flow, but we've never told you how those cancel events are handled. Up to now, we've been avoiding the issue, but now it's time to meet it head on.

At any view state within the flow, the customer may choose to cancel the order and start over. When that happens, we need the flow to transition to the finish state to close down the flow execution. A naive way of handling cancel events is to place appropriate <transition>s all throughout the flow. For example, consider the cancel transition that *could* have been added to the showOrder state:

```
<view-state id="showOrder" view="orderDetails">  
    <transition on="addPizza" to="addPizza" />  
    <transition on="continue" to="takePayment" />  
    <transition on="cancel" to="finish" />  
</view-state>
```

The problem with doing it this way is that we must copy the exact same <transition> element to all of our flow's <view-state>s. Our flow is simple enough that duplicating the <transition> wouldn't be too painful. Nonetheless, it still results in an undesired duplication of code and should be avoided.

Instead of defining the same <transition> multiple times throughout an entire flow, Spring Web Flow offers the ability to define global transitions. Global transitions are transition definitions that are applicable from any flow state. To add global transitions to a flow, simply add a <global-transitions> element as a child of the <flow> element and place <transition> elements within it. For example:

```
<global-transitions>  
    <transition on="cancel" to="finish" />  
</global-transitions>
```

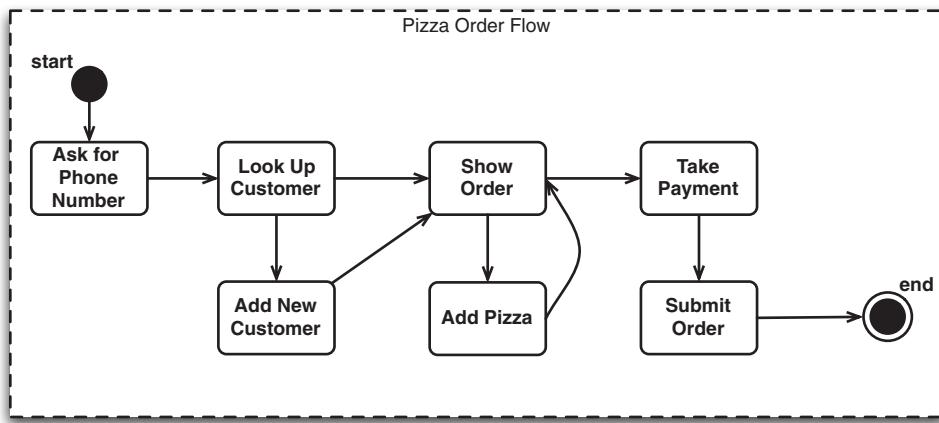


Figure 15.8 The pizza order flow is now complete... or is it?

Here we've defined a global transition to the finish state whenever a cancel event is fired. This makes it possible to handle the cancel event from any state within the flow.

Our pizza order flow is almost complete. We have flow states to gather customer information, to add pizzas to an order, to take payment, and to save the order. The final flow is illustrated in figure 15.8.

After putting all of the pieces together, we get the complete pizza order flow definition file, `PizzaOrder-flow.xml`, as shown in listing 15.6.

Listing 15.6 The complete pizza order flow definition

```
<?xml version="1.0" encoding="UTF-8"?>
<flow xmlns=
    "http://www.springframework.org/schema/webflow"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation=
        "http://www.springframework.org/schema/webflow
        http://www.springframework.org/schema/webflow/
        spring-webflow-1.0.xsd">

    <var name="order"
        class="com.springinaction.pizza.domain.Order"
        scope="flow"/>

    <start-state idref="askForPhoneNumber" />

    <view-state id="askForPhoneNumber"
        view="phoneNumberForm">
```

```
<transition on="submit" to="lookupCustomer" />
</view-state>

<action-state id="lookupCustomer">
    <action bean="lookupCustomerAction" />

    <transition on="success"
        to="checkDeliveryArea" />

    <transition on-exception=
        "com.springinaction.pizza.service.CustomerNotFoundException"
        to="addNewCustomer" />
</action-state>

<view-state id="addNewCustomer" view="newCustomerForm">
    <render-actions>
        <action bean="customerFormAction"
            method="setupForm" />
    </render-actions>
    <transition on="submit" to="checkDeliveryArea">
        <action bean="customerFormAction" method="bind" />
        <evaluate-action expression=
            "flowScope.order.setCustomer(requestScope.customer)" />
    </transition>
</view-state>

<decision-state id="checkDeliveryArea">
    <if test="${flowScope.order.customer.inDeliveryArea}"
        then="showOrder"
        else="warnNoDeliveryAvailable"/>
</decision-state>

<view-state id="warnNoDeliveryAvailable"
    view="deliveryWarning">
    <transition on="continue" to="showOrder" />
</view-state>

<view-state id="showOrder" view="orderDetails">
    <transition on="addPizza" to="addPizza" />
    <transition on="continue" to="takePayment" />
</view-state>

<view-state id="addPizza" view="newPizzaForm">
    <render-actions>
        <action bean="pizzaFormAction" method="setupForm" />
    </render-actions>
    <transition on="submit" to="showOrder">
        <action bean="pizzaFormAction" method="bind" />
        <evaluate-action expression=
            "flowScope.order.addPizza(requestScope.pizza)" />
    </transition>
</view-state>
```

```

<view-state id="takePayment" view="paymentForm">
    <transition on="submit" to="submitOrder">
        <bean-action bean="paymentProcessor"
            method="approveCreditCard">
            <method-arguments>
                <argument expression=
                    "${requestParameters.creditCardNumber}" />
                <argument expression=
                    "${requestParameters.expirationMonth}" />
                <argument expression=
                    "${requestParameters.expirationYear}" />
                <argument expression=
                    "${flowScope.order.total}" />
            </method-arguments>
        </bean-action>
    </transition>

    <transition
        on-exception="com.springinaction.pizza.PaymentException"
        to="takePayment" />
</view-state>

<action-state id="submitOrder">
    <bean-action bean="orderService" method="saveOrder">
        <method-arguments>
            <argument expression="${flowScope.order}" />
        </method-arguments>
    </bean-action>

    <transition on="success" to="finish" />
</action-state>

<end-state id="finish"
    view="flowRedirect:PizzaOrder-flow" />

<global-transitions>
    <transition on="cancel" to="finish" />
</global-transitions>
</flow>

```

We could stop here and move on to another topic. But never content to leave well enough alone, let's look at a few ways that we can improve on the pizza order flow by using some advanced Spring Web Flow techniques.

15.3 Advanced web flow techniques

Although the pizza order flow is now complete and ready to take orders for the delicious Italian pies, there is still some room for improvement. Specifically, we have two improvements in mind:

- Once we have a customer's information on hand, we should determine whether or not they live within our delivery area.
- The customer information portion of the flow may come in handy for other flows. So, it would be nice to extract that portion of the flow into a subflow that can be reused in another flow (perhaps for a flower delivery service).

Coincidentally, these improvements involve the two flow states from table 15.1 that we haven't talked about yet: decision states and subflow states. That makes this an opportune time to give those two flow states a try. Let's start by looking at how to fork the direction of a flow using decision states.

15.3.1 Using decision states

After the `lookupCustomer` and `addNewCustomer` states and just before the `showOrder` state, we have a decision to make: can we deliver the pizza to the customer's given address?

More accurately, our flow has a decision to make. If the customer lives within the delivery area then there's no problem. The flow should proceed normally. But if the customer lives outside of the delivery area, we should transition to a warning page to indicate that we can't deliver pizza to the customer's address but that they are welcome to place the order for carryout and pick it up themselves.

When flow-diverging decisions must be made, a decision state is in order. A decision state is the Spring Web Flow equivalent of an `if/else` statement in Java. It evaluates some Boolean expression and based on the results will send the flow in one of two directions.

Decision states are defined with the `<decision-state>` element. The following `<decision-state>` is what we'll use to decide whether or not to warn the user that they are out of the delivery area:

```
<decision-state id="checkDeliveryArea">
    <if test="${flowScope.order.customer.inDeliveryArea}">
        then="showOrder"
        else="warnNoDeliveryAvailable"/>
    </decision-state>
```

At the heart of `<decision-state>` is the `<if>` element. The `test` attribute specifies some expression to be evaluated. If the expression evaluates to true, the flow will transition to the state specified by the `then` attribute. Otherwise, the flow will transition to the state given in the `else` attribute.

Here the `inDeliveryArea` property of the `Customer` object is evaluated. If it's true then the flow will continue at the `showOrder` state. Otherwise, we'll warn the

user that delivery is not available for them and that they'll have to pick up their pizza.

As for the delivery warning, it's a simple view state, as defined here:

```
<view-state id="warnNoDeliveryAvailable"
    view="deliveryWarning">
    <transition on="continue" to="showOrder" />
</view-state>
```

The user is presented with the view whose name is `deliveryWarning` and is prompted to continue or cancel. If they choose to continue, they will transition to the `showOrder` state. If they decide to cancel the order, they will be transitioned to the `finish` state (per the global `cancel` transition).

Aside from adding these two new states, the only other thing we'll need to do is to rewire the `lookupCustomer` and `addNewCustomer` states to transition to `checkDeliveryArea` instead of `showOrder`:

```
<action-state id="lookupCustomer">
    <action bean="lookupCustomerAction" />
    <transition on="success"
        to="checkDeliveryArea" />
    <transition on-exception=
        "com.springinaction.pizza.service.CustomerNotFoundException"
        to="addNewCustomer" />
</action-state>

<view-state id="addNewCustomer" view="newCustomerForm">
    <render-actions>
        <action bean="customerFormAction"
            method="setupForm"/>
    </render-actions>
    <transition on="submit" to="checkDeliveryArea">
        <action bean="customerFormAction" method="bind" />
        <evaluate-action expression=
            "flowScope.order.setCustomer(requestScope.customer)" />
    </transition>
</view-state>
```

Adding the delivery area `<decision-state>` not only changes the structure of the flow, it also changes how the flow is presented to the user. Before, anyone could place an order for pizza delivery, regardless of where they lived. Now, however, the flow is changed to indicate that delivery isn't available for those outside of the delivery area. This is reflected in the updated flow diagram in figure 15.9.

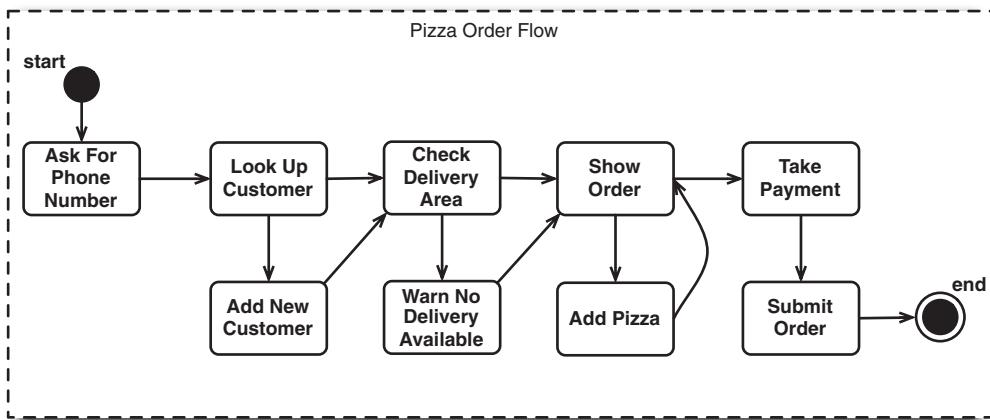


Figure 15.9 The new flow diagram after adding the delivery area check.

The next flow improvement we'll make will alter the structure of the flow, but shouldn't have any impact on how the flow is presented to the user. Let's see how to extract a portion of the pizza order flow into a subflow.

15.3.2 Extracting subflows and using substates

You've probably noticed by now that the flow definition file is getting a bit lengthy. Counting the two flow states that we added in the previous section and the start and end states, our pizza order flow's state count is up to 11. Although this is far from being the most complex flow ever defined, it is starting to get unwieldy.

In Java, when a method gets too big, it is often beneficial to pull out related lines of code and place them into their own method. In his *Refactoring: Improving the Design of Existing Code* (Addison-Wesley, 1999), Martin Fowler refers to this technique as an “extract method.” By performing an extract method refactoring, lengthy methods become shorter and easier to follow.

Fowler's book doesn't cover flow definitions per se. Nevertheless, the idea behind extract methods applies equally well to flow definitions. Flow definitions can get quite lengthy and difficult to follow. It may be advantageous to extract a related set of states into their own flow and to reference that flow from the main flow. As with methods, the reward is smaller and easier-to-understand flow definitions.

Examining the pizza order flow, we find that 5 of the 11 states pertain to gathering customer information. Since these states make up almost half of the whole flow, they make a good candidate to be extracted into their own flow.

Creating a customer data flow

The first step in performing an “extract subflow” refactoring is to copy the five customer-centric flow states out of PizzaOrder-flow.xml into their own flow definition file. Listing 15.7 shows CustomerInfo-flow.xml, the new flow that will contain the customer information states.

Listing 15.7 CustomerInfo-flow.xml—the customer information states extracted into their own flow

```
<?xml version="1.0" encoding="UTF-8"?>
<flow xmlns="http://www.springframework.org/schema/webflow"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation=
          "http://www.springframework.org/schema/webflow
           http://www.springframework.org/schema/webflow/
           ➔ spring-webflow-1.0.xsd">

    <var name="customer"
         class="com.springinaction.pizza.domain.Customer"
         scope="flow"/>

    <start-state idref="askForPhoneNumber" />

    <view-state id="askForPhoneNumber"
                view="phoneNumberForm">
        <transition on="submit" to="lookupCustomer" />
    </view-state>

    <action-state id="lookupCustomer">
        <action bean="lookupCustomerAction2" />

        <transition on="success"
                   to="checkDeliveryArea" />

        <transition on-exception=
                    "com.springinaction.pizza.service.
                     ➔ CustomerNotFoundException"
                   to="addNewCustomer" />
    </action-state>

    <view-state id="addNewCustomer" view="newCustomerForm">
        <render-actions>
            <action bean="customerFormAction"
                  method="setupForm" />
        </render-actions>
        <transition on="submit" to="checkDeliveryArea">
            <action bean="customerFormAction" method="bind" />
        </transition>
    </view-state>

```

Declares flow-scoped Customer

```

        </view-state>
        <decision-state id="checkDeliveryArea">
            <if test="${flowScope.customer.inDeliveryArea}">
                <then>"finish"
                <else>"warnNoDeliveryAvailable"/>
            </if>
        </decision-state>
        <view-state id="warnNoDeliveryAvailable"
            view="deliveryWarning">
            <transition on="continue" to="finish" />
        </view-state>
        <end-state id="finish">
            <output-mapper>
                <mapping source="flowScope.customer"
                    target="customer"/>
            </output-mapper>
        </end-state>
        <global-transitions>
            <transition on="cancel" to="finish" />
        </global-transitions>
    </flow>

```

The code listing is annotated with three callout boxes:

- A box on the right side, pointing to the `<if>` block, contains the text: "Tests in DeliveryArea on flow-scoped Customer".
- A box below it, pointing to the `<transition>` block, contains the text: "Transitions to finish".
- A box on the right side, pointing to the `<output-mapper>` block, contains the text: "Maps customer to output".

Most of listing 15.7 should look familiar. That's because it was extracted from the original order flow. However, upon closer inspection, you'll find that it's not a direct copy from the original flow. We had to make a few minor changes to accommodate the new flow structure.

Because this new flow knows nothing about the pizza order, we no longer have a place to put the customer data. Therefore, the first change we made was to declare a flow-scoped `Customer` object using the `<var>` element. We'll use this `Customer` object throughout the flow.

For example, the `<decision-state>` element has changed to test the flow-scoped `Customer` object instead of the `customer` property of a flow-scoped `Order` object.

In the original pizza order flow, the `checkDeliveryArea` and `warnNoDeliveryAvailable` states transitioned to the `showOrder` state once finished. But the `showOrder` state isn't in this flow, so we've changed those transitions to go to the `finish` state. As we'll see in a moment, the pizza order flow will transition to the `showOrder` state once the customer info flow completes.

Speaking of the `finish` state, it looks a bit different in the customer info flow than in the pizza order flow. Since this flow will end by transitioning to a state in the order flow, there's no need to specify a `view` attribute.

Also (and even more interesting), notice the `<output-mapper>` element. This element maps the flow-scoped `Customer` to an output variable named `customer`, effectively returning the `Customer` object to the calling flow.

Using a subflow

Back in `PizzaOrder-flow.xml`, we'll need to completely remove all of the customer info states that we extracted into `CustomerInfo-flow.xml`. They'll be replaced with a `<subflow-state>` that references the new flow:

```
<subflow-state id="getCustomerInfo" flow="CustomerInfo-flow" >
    <attribute-mapper>
        <output-mapper>
            <mapping source="customer" target="flowScope.order.customer"/>
        </output-mapper>
    </attribute-mapper>
    <transition on="finish" to="showOrder" />
</subflow-state>
```

According to the `flow` attribute, the `getCustomerInfo` state is a subflow state that references the flow named `CustomerInfo-flow`. A subflow state is a state that references another flow. When a flow enters a subflow state, the current flow is suspended and the referenced flow is executed. This makes subflow states analogous to method calls in Java and the subflows themselves analogous to methods. In the case of the `getCustomerInfo` state, we are calling the `CustomerInfo-flow` flow.

As you'll recall, the last thing that happens in `CustomerInfo-flow` is that the flow-scoped `Customer` object is mapped to an output variable named `customer`. In the `<subflow-state>`, we now map that output variable to the `customer` property of the `order` flow's flow-scoped `Order` object. This makes sure that the `Order` is properly populated with the result of the customer info flow.

There's still one more minor change that will need to be made in the order flow. Previously, the `<start-state>` referred to the `askForPhoneNumber` state. Now, however, the `askForPhoneNumber` state is in a separate flow. Therefore, we'll need to change the `<start-state>` to reference the `getCustomerInfo` state:

```
<start-state idref="getCustomerInfo" />
```

And that's it! Our "extract subflow" refactor is now complete. The new flow (or flows, as the case is now) are depicted in figure 15.10.

Before we move on, we should point out that there's another benefit of the extract subflow refactoring. A close examination of the customer info flow reveals that the customer info flow is focused on one thing and one thing only: gathering customer information. There's no hint of a pizza order to be found. This means

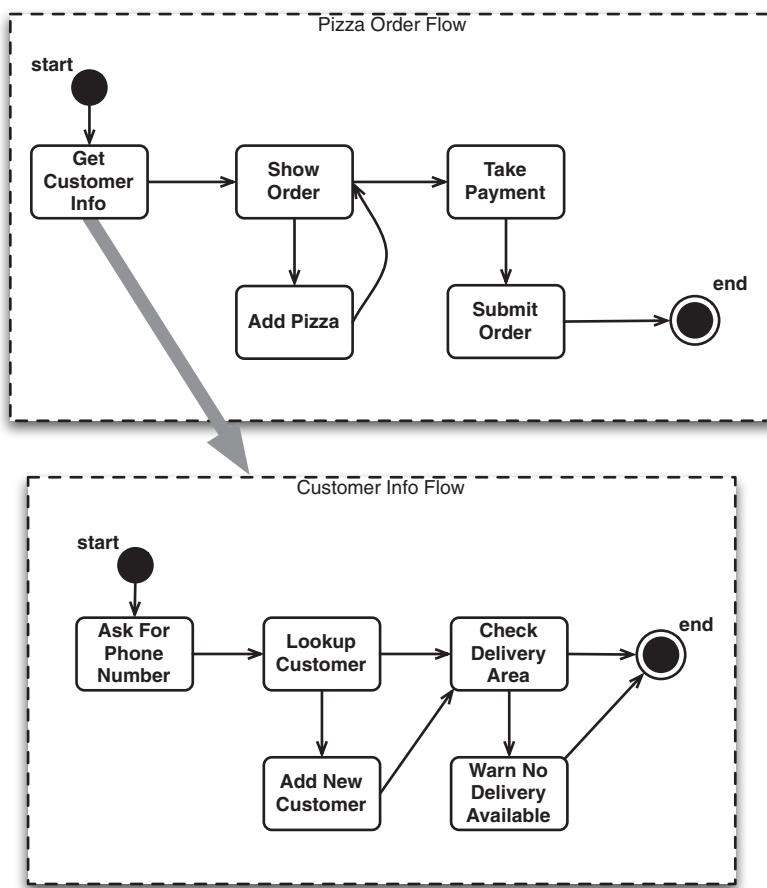


Figure 15.10 The flows after performing an extract subflow on the customer information states.

that the customer info flow could be reused in any flow where we may need to collect customer data. If, for example, we were to open up a flower delivery service, the customer info flow would fit in nicely.

By now you should see the value of building flow-based applications using Spring Web Flow. But what if you want to plug a flow into an existing JSF or Struts application? No problem! Let's look at how Spring Web Flow integrates into other web frameworks.

15.4 Integrating Spring Web Flow with other frameworks

In this chapter, we've focused on building Spring Web Flow applications within Spring MVC. Nevertheless, we thought you might find it interesting to know that Spring Web Flow doesn't have to be used with Spring MVC. In fact, Spring Web Flow comes with out-of-the-box support for use in

- Jakarta Struts
- JavaServer Faces
- Spring Portlet MVC

There's also an open issue in the Spring Web Flow issue list describing how to integrate Spring Web Flow into WebWork 2 (which will presumably work in Struts 2 as well). You can read about the WebWork integration and follow the status of the issue at <http://opensource.atlassian.com/projects/spring/browse/SWF-76>.

Regardless of which web framework you choose to build flows within, you'll find that your flow definitions are portable across all supported frameworks. The only difference between each framework is at the integration points (e.g., `FlowController` for a Spring MVC application versus `FlowAction` in a Struts application).

Before we end our discussion on Spring Web Flow, let's see how to use Spring Web Flow's out-of-the-box support for integration with Jakarta Struts and JSF.

15.4.1 Jakarta Struts

Up to now, the main entry point into a Spring Web Flow application has been through a Spring controller (`FlowController` for Spring). Struts, however, doesn't support Spring controllers. Therefore, we'll need a Struts-specific approach for using Spring Web Flow within a Struts application.

Instead of controllers, Struts is based on Actions. Consequently, Spring Web Flow integrates into Struts through `FlowAction`, a Struts Action implementation that performs the same job as `FlowController` does for Spring MVC. That is, `FlowAction` is a Struts-specific front controller for Spring Web Flow.

To use `FlowAction`, declare it in the `<action-mappings>` section of `struts-config.xml`. The following `<action>` configures Spring Web Flow to respond to requests whose URL ends with `/flow.do`:

```
<action path="/flow"
       name="actionForm"
       scope="request"
       type="org.springframework.webflow.executor.
            struts.FlowAction"/>
```

As with the Spring MVC version of Spring Web Flow, the Struts `Action` uses a handful of parameters to guide the flow (the same as those described in table 15.1). In Struts, these parameters must be bound to an `ActionForm` implementation. More specifically, Spring Web Flow parameters must be bound to `SpringBindingActionForm`. Therefore, we'll also need to configure a Struts `<form-bean>` entry to tell Struts about `SpringBindingActionForm`:

```
<form-bean name="actionForm" type=
    "org.springframework.web.struts.
     ↪ SpringBindingActionForm"/>
```

Notice that the name of the `<form-bean>` is the same as the name of the `<action>`. This is how Struts knows that `SpringBindingActionForm` is to be used to bind parameters for `Action`.

That's all you must do to use Spring Web Flow with Struts. Flows for Struts are defined exactly the same way as flows running under Spring MVC. The only slight difference is that in a Struts-based flow a `<view-state>`'s view refers to a Struts `<forward>`.

If Struts isn't your cup of tea, Spring Web Flow integrates with one more web framework. Let's see how to use Spring Web Flow with JavaServer Faces (JSF).

15.4.2 JavaServer Faces

To use Spring Web Flow with a JSF application, we'll need to configure several custom Spring Web Flow elements in the application's `faces-config.xml` file. The following excerpt from `faces-config.xml` shows the relevant customizations:

```
<application>
    <navigation-handler>
        org.springframework.webflow.executor.jsf.FlowNavigationHandler
    </navigation-handler>
    <property-resolver>
        org.springframework.webflow.executor.jsf.FlowPropertyResolver
    </property-resolver>
    <variable-resolver>
        org.springframework.webflow.executor.jsf.FlowVariableResolver
    </variable-resolver>
    <variable-resolver>
        org.springframework.web.jsf.DelegatingVariableResolver
    </variable-resolver>
    <variable-resolver>
        org.springframework.web.jsf.WebApplicationContextVariableResolver
    </variable-resolver>
</application>
```

In JSF, navigation is usually handled by consulting `<navigation-rule>` entries in `faces-config.xml`. In Spring Web Flow, however, navigation is determined by the flow's `<transition>` definitions. Therefore, for Spring Web Flow to work with JSF, we need to configure `FlowNavigationHandler`, a customer navigation handler that uses a flow definition to guide navigation when running within a flow (falling back to the default `NavigationHandler` when not in a flow).

We're also going to need a way for JSF to resolve variables and their properties from flow scope. Unfortunately, JSF's default variable and property resolution knows nothing about flow-scoped variables. Spring Web Flow's `FlowVariableResolver` and `FlowPropertyResolver` enable JSF to find flow-scoped data when binding expressions that are prefixed with `flowScope` (e.g., `flowScope.order.total`).

We also must configure two other variable resolvers: `DelegatingVariableResolver` and `WebApplicationContextVariableResolver`. These variable resolvers are useful in resolving JSF variables as beans from the Spring application context. We'll talk more about these two variable resolvers in sections 16.4.2 and 16.4.4 when we discuss the integration of JSF and Spring.

Finally, one more tweak must be made to `faces-config.xml` for Spring Web Flow to work with JSF. A flow's execution must be saved between requests and restored at the beginning of each request. Spring Web Flow's `FlowPhaseListener` is a JSF `PhaseListener` implementation that manages the flow execution on behalf of JSF. It is configured in `faces-config.xml` as follows:

```
<lifecycle>
  <phase-listener>
    org.springframework.webflow.executor.jsf.FlowPhaseListener
  </phase-listener>
</lifecycle>
```

`FlowPhaseListener` has three specific responsibilities, each associated with a phase in the JSF lifecycle:

- At `BEFORE_RESTORE_VIEW`, `FlowPhaseListener` restores a flow execution, based on the flow execution identifier in the request arguments (if any).
- At `BEFORE_RENDER_RESPONSE`, `FlowPhaseListener` generates a new identifier for the updated flow execution to be stored as.
- At `AFTER_RENDER_RESPONSE`, `FlowPhaseListener` saves the updated flow execution, using the identifier generated in `BEFORE_RENDER_RESPONSE`.

Once all of the necessary entries have been placed in faces-config.xml, we're ready to launch a flow. The following snippet of JSF creates a link to kick off the flow named `Order-flow`:

```
<h:commandLink  
    value="Order Pizza"  
    action="flowId:Order-flow"/>
```

15.5 Summary

Not all web applications are freely navigable. Sometimes, a user must be guided along, asked appropriate questions and led to specific pages based on their responses. In these situations, an application feels less like a menu of options and more like a conversation between the application and the user.

In this chapter, we've explored Spring Web Flow, a web framework that enables development of conversational applications. Along the way, we built a flow-based application to take pizza orders. We started by defining the overall path that the application should take, starting with gathering customer information and concluding with the order being saved in the system.

A flow is made up of several states and transitions that define how the conversation will traverse from state to state. As for the states themselves, they come in one of several varieties: action states that perform some business logic, view states that involve the user in the flow, decision states that dynamically direct the flow, and start and end states that signify the beginning and end of a flow. In addition, there are subflow states that are, themselves, defined by a flow.

Although much of our discussion of Spring Web Flow assumed that our flow would run as part of a Spring MVC application, we also learned that Spring Web Flow can be used with Jakarta Struts and JavaServer Faces. In fact, the flow definitions themselves are completely transferable from one web framework to another.

Speaking of integration, Spring Web Flow isn't the only part of Spring that plays well with other web frameworks. Coming up in the next chapter, we'll see how to use the rest of Spring with several popular web frameworks, including Tapestry, WebWork, Struts, and JavaServer Faces. We'll also have a look at how to build Ajax functionality into an application by exposing Spring beans as Ajax-remoted objects.

Integrating with other web frameworks

This chapter covers

- Using Spring with Jakarta Struts
- Integrating with WebWork 2 and Struts 2
- Working with Tapestry
- Using JavaServer Faces with Spring
- Ajax-enabling with DWR

Do you always order the same thing when you go out to eat? If you're like a lot of people, you have your favorite dish that you enjoy and rarely, if ever, try anything new. You've heard good things about the pasta frijole, but you *know* that the lasagna's good. So, instead of trying something different, you stick with what you're comfortable with.

Likewise, perhaps you're already quite comfortable with a specific web framework. You have heard good things about Spring MVC, but you're not quite ready to take the leap. You may have legitimate reasons for choosing a different web framework to front your application. Perhaps you're already heavily invested in another MVC framework such as Struts or WebWork and aren't prepared to abandon it for Spring. Nevertheless, you would still like to take advantage of Spring's support for dependency injection, declarative transactions, AOP, and so forth.

No problem. As we've seen throughout this book, the key to the Spring Framework is the freedom to choose what works best for your application. As you'll learn in this chapter, Spring offers you choices when building web applications, too. Although Spring offers its own very capable web framework, you are free to choose another if you'd like and still be able to take advantage of Spring in the other layers of your application.

If you're not quite ready to make the jump to Spring MVC, you certainly have a huge selection of other web frameworks to choose from. In fact, there are hundreds of web frameworks for Java. I have neither the space nor the inclination to show you how to integrate Spring with all of them. But we'll show you how Spring can be used with a few of the more popular MVC frameworks, including Struts, WebWork, Tapestry, and JavaServer Faces (JSF). We'll also see how to use Spring with a popular Ajax toolkit known as DWR (Direct Web Remoting).

Struts has long been the workhorse framework of many Java-based web applications. As it's the most well known among all of the web frameworks that Spring integrates with, let's start this chapter by seeing how to use Spring and Struts together.

16.1 Using Spring with Struts

Despite the seemingly endless barrage of Java-based MVC frameworks, Struts is still the king of them all. It began life in May 2000 when Craig McClanahan launched the project to create a standard MVC framework for the Java community. In July 2001, Struts 1.0 was released and set the stage for Java web development for thousands and thousands of projects.

In this section, we're going to look at how to use Struts in the web layer of a Spring-enabled application. While we will cover just enough Struts basics to accommodate integration with Spring, we won't be going into any great detail on how to develop applications using Struts. If you want more information on Struts, I recommend you check out *Struts in Action* (Manning, 2002) and *Struts Recipes* (Manning, 2004).

Let's turn back the clock and pretend that we had developed the RoadRantz application using Struts instead of Spring MVC. Had that been the case, we would've had written `RantsForVehicleAction` (listing 16.1) instead of `RantsForVehicleController` (see listing 13.3).

Listing 16.1 A Struts-flavored look at the RoadRantz application

```
package com.roaddrantz.struts;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.apache.struts.action.Action;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;
import com.roaddrantz.domain.Vehicle;
import com.roaddrantz.service.RantService;

public class RantsForVehicleAction extends Action {
    public ActionForward execute(
        ActionMapping mapping, ActionForm form,
        HttpServletRequest request, HttpServletResponse response)
        throws Exception {

        VehicleForm vehicleForm = (VehicleForm) form;
        Vehicle vehicle = vehicleForm.getVehicle();

        request.setAttribute("rants",
            rantService.getRantsForVehicle(vehicle)); ← Delegates to
                                                        // rant service
        return mapping.findForward("rantList");
    }
}
```

This Struts action performs the same functionality as the `RantsForVehicleController` that we wrote in chapter 13. Just like that Spring MVC controller, `RantsForVehicleAction` uses a `RantService` to retrieve a list of rants for a particular vehicle. It then places the list into the request so that the view layer can render the list in the browser.

But listing 16.1 isn't complete. What's missing is the part that tells where the RantService comes from.

In the Spring MVC version, RantsForVehicleController is configured in the Spring application context just like any other Spring-managed bean. As a Spring-managed bean, RantsForVehicleController could be given a RantService through dependency injection.

RantsForVehicleAction, on the other hand, is a Struts-managed action class. This means that Spring won't normally have any idea that it exists, much less that it needs to be injected with a RantService.

It would seem that we have a problem. If RantsForVehicleAction isn't a Spring-managed bean then how can we give it a RantService that is managed by Spring?

Fortunately, Spring offers a solution for Struts-integration. Actually, Spring offers two solutions to choose from:

- Write Struts actions to extend a Spring-aware base class.
- Delegate requests to Struts actions configured in the Spring application context.

We'll explore each of these strategies for Struts-Spring integration in the sections that follow. But first, regardless of which approach you take, there's one bit of common configuration that you'll need to take care of: telling Struts about your Spring application context.

16.1.1 Registering the Spring plug-in with Struts

In order for Struts to have access to Spring-managed beans, you'll need to register a Struts plug-in that is aware of the Spring application context. Add the following code to your struts-config.xml to register the plug-in:

```
<plug-in className=
    "org.springframework.web.struts.ContextLoaderPlugIn">
    <set-property property="contextConfigLocation"
        value="/WEB-INF/classes/roaddrantz-servlet.xml,
        /WEB-INF/classes/roaddrantz-services.xml,
        /WEB-INF/classes/roaddrantz-data.xml,
        /WEB-INF/classes/roaddrantz-data-hibernate.xml,
        /WEB-INF/classes/roaddrantz-cache.xml"/>
</plug-in>
```

ContextLoaderPlugIn loads a Spring application context (a WebApplicationContext, to be specific), using the context configuration files listed (comma-separated) in its contextConfigLocation property.

Now that the plug-in is in place, you're ready to choose an integration strategy. Let's first look at how to create Struts actions that are aware of the Spring application context.

16.1.2 Writing Spring-aware Struts actions

One way to integrate Struts and Spring is to write all of your Struts action classes to be aware of the Spring application context. Spring's `WebApplicationContextUtils` class provides some convenient static methods that can be used to retrieve the application context. With the context in hand, you can then use Spring as a factory to retrieve the beans your Struts actions needs. For example, here's how the `execute()` method of `RantsForVehicleAction` could be written:

```
public ActionForward execute(ActionMapping mapping, ActionForm form,
    HttpServletRequest request, HttpServletResponse response)
    throws Exception {

    VehicleForm vehicleForm = (VehicleForm) form;
    Vehicle vehicle = vehicleForm.getVehicle();

    ApplicationContext ctx =
        WebApplicationContextUtils.getRequiredWebApplicationContext(
            getServlet().getServletContext());
    RantService rantService =
        (RantService) ctx.getBean("rantService");

    request.setAttribute("rants",
        rantService.getRantsForVehicle(vehicle));
    return mapping.findForward("rantList");
}
```

The code in bold uses the `getRequiredWebApplicationContext()` method of `WebApplicationContextUtils` to retrieve the Spring application context and ultimately to retrieve the `rantService` bean.

Since a typical Struts application will involve more than one action class, you could end up repeating that same context lookup code in all of them. To avoid duplicating code, it's better to put the Spring context code in a common base class to avoid code duplication. Then each of your application's action classes would subclass the base action class instead of Struts's `Action`.

The good news is that you won't have to write the Spring-aware base action class. That's because Spring comes with `ActionSupport`, an extension of Struts's `Action` that overrides the `setServlet()` method to retrieve the Spring application context from the `ContextLoaderPlugIn`. Any class that extends `ActionSupport` has access to the Spring application context by calling `getWeb-`

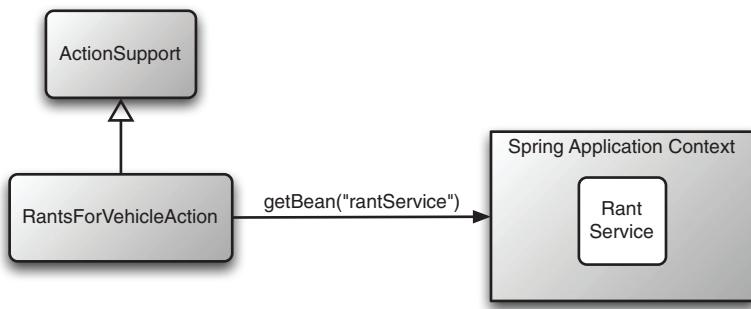


Figure 16.1 Spring's `ActionSupport` is a convenient base class for Action implementations that makes the Spring application context directly available to the action class. From there the action can retrieve its dependencies from the Spring application context.

`ApplicationContext()`. From there, the action class can retrieve beans directly from Spring by calling the `getBean()` method, as shown in figure 16.1.

For example, consider `RantsForVehicleAction` in listing 16.2, which extends `ActionSupport` for access to the Spring application context.

Listing 16.2 A Spring-aware implementation of `RantsForVehicleAction`

```

package com.roaddrantz.struts;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;
import org.springframework.web.struts.ActionSupport;
import com.roaddrantz.domain.Vehicle;
import com.roaddrantz.service.RantService;

public class RantsForVehicleAction extends ActionSupport {           | Extends
    public ActionForward execute(                         ActionSupport
        ActionMapping mapping, ActionForm form,
        HttpServletRequest request, HttpServletResponse response)
        throws Exception {                                |

    VehicleForm vehicleForm = (VehicleForm) form;
    Vehicle vehicle = vehicleForm.getVehicle();
    RantService rantService = (RantService)
        getWebApplicationContext().getBean("rantService");
    request.setAttribute("rants",
  
```

A callout box points to the `extends ActionSupport` line with the text "Extends ActionSupport". Another callout box points to the `getBean("rantService")` line with the text "Looks up rantService bean".

```
    rantService.getRantsForVehicle(vehicle)); <-- Delegates to  
    request.setAttribute("vehicle", vehicle);  
    return mapping.findForward("rantList");  
}  
}
```

When `RantsForDayAction` needs a `RantService`, it starts by calling `getWebApplicationContext()` to retrieve the Spring application context. From there it simply calls `getBean()` to retrieve the `rantService` bean.

The good thing about using this approach to Struts-Spring integration is that it's very intuitive. Aside from extending `ActionSupport` and retrieving beans from the application context, you are able to write and configure your Struts actions in much the same way as you would in a non-Spring Struts application.

But this approach also has its negative side. Most notably, your action classes will directly use Spring-specific classes. This tightly couples your Struts action code with Spring, which may not be desirable. Also, the action class is responsible for looking up references to Spring-managed beans. This is in direct opposition to the notion of dependency injection (DI).

For those reasons, Spring offers another way to integrate Struts and Spring. The other approach lets you write Struts action classes that are completely unaware they are integrated with Spring. And you can use Spring's dependency injection to inject service beans into your actions so that they don't have to look them up for themselves.

16.1.3 Delegating to Spring-configured actions

As you may recall from chapter 8, Acegi security employs several servlet filters to secure web applications. But so that Spring can inject dependencies into those filters, Acegi provides `FilterToBeanProxy`, a servlet filter that doesn't perform any real functionality itself but instead delegates to a Spring-configured filter bean.

As it turns out, the delegation approach used in Acegi can be applied equally well when integrating Struts and Spring. To accommodate Struts action delegation, Spring comes with `DelegatingRequestProcessor`, a replacement for Struts's default `RequestProcessor` that looks up Struts actions from the Spring application context.

The first step in Struts-to-Spring delegation is to tell Struts that we want to use `DelegatingRequestProcessor` instead of its normal request processor. To do this, we add the following XML to the `struts-config.xml`:

```
<controller processorClass=
    "org.springframework.web.struts.DelegatingRequestProcessor" />
```

Optionally, if you are using Tiles in your Struts applications, you'll want to use `DelegatingTilesRequestProcessor` instead:

```
<controller processorClass=
    "org.springframework.web.struts.
    ↗ DelegatingTilesRequestProcessor" />
```

`DelegatingRequestProcessor` (or its Tiles-savvy cohort, `DelegatingTilesRequestProcessor`) tells Struts to automatically send action request to Struts actions that are configured in the Spring application context, as shown in figure 16.2. The way it finds the Spring-configured action depends on how you configure the action in `struts-config.xml`.

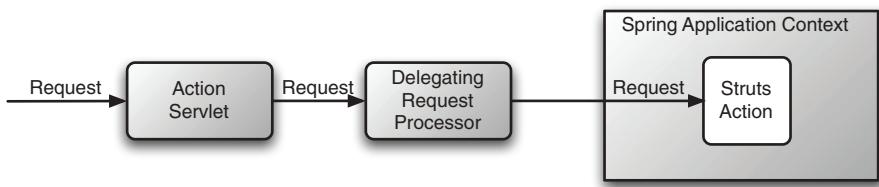


Figure 16.2 `DelegatingRequestProcessor` sends requests from Struts's `ActionServlet` to a Struts Action class configured in the Spring application context.

In its simplest form, you can configure an `<action>` element in `struts-config.xml` like this:

```
<action path="/rantsForVehicle" />
```

When a request comes in for `/rantsForVehicle.do`, `DelegatingRequestProcessor` will automatically refer to the Spring application context, looking for a bean named `/rantsForVehicle`—the same as the action's path. That means we'll need to wire the action as a bean in Spring. Let's do that now.

Wiring actions in Spring

Here's where the real benefit of using `DelegatingRequestProcessor` comes in. Rather than configure the `RantsForVehicleAction` in `struts-config.xml` (where it is outside of Spring's jurisdiction and can't be injected), we will configure it in the Spring application context (`roadrantz-struts.xml`) as follows:

```
<bean name="/rantsForVehicle"
      class="com.roadrantz.struts.RantsForVehicleAction">
```

```
<property name="rantService" ref="rantService" />
</bean>
```

As you can see, `RantsForVehicleAction` is configured in Spring just like any other bean. And, since it needs a `RantService` to do its job, we've obliged by wiring it with a reference to the `rantService` bean.

Take note of how the bean is named. Because `DelegatingRequestProcessor` will look for a bean with the same name as the action's path and because the path contains a slash character, we had to use the `name` attribute to name the bean. According to XML rules, the slash character is not allowed in an XML element's `id` attribute.

If it seems odd to have the Spring bean named with the same name as the Struts action's path (or if that slash vexes you) then you may want to configure the `<action>` element in `struts-config.xml` like this:

```
<action path="/rantsForVehicle"
       type="com.roaddrantz.struts.RantsForVehicleAction" />
```

This `<action>` declaration looks more like conventional Struts in that the action's type is declared in the Struts configuration. The big difference is that instead of Struts taking responsibility for instantiating and managing `RantsForVehicleAction`, `DelegatingRequestProcessor` will rely on Spring to manage the action as a bean. When a request comes in for `/rantsForVehicle`, `DelegatingRequestProcessor` will ask the Spring application context for a bean whose type is `com.roaddrantz.struts.RantsForVehicleAction` and send the request to that bean.

Now we've wired the action in Spring and configured it in `struts-config.xml`. The only thing left to do is to tweak the `RantsForVehicleAction` class so that it may receive the injected `RantService` bean, instead of having to look it up itself.

Implementing the Spring-configured Struts action

The dependency-injected version of `RantsForVehicleAction` isn't much different from what we showed you in listing 16.1. In listing 16.2 `RantsForVehicleAction` had to directly use the application context to retrieve the `RantService` bean. But as you can see in listing 16.3, that's no longer necessary.

Listing 16.3 A Spring-injected RantsForVehicleAction

```
package com.roaddrantz.struts;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.apache.struts.action.Action;
import org.apache.struts.action.ActionForm;
```

```
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;
import com.roadrantz.domain.Vehicle;
import com.roadrantz.service.RantService;

public class RantsForVehicleAction extends Action {
    public ActionForward execute(
        ActionMapping mapping, ActionForm form,
        HttpServletRequest request, HttpServletResponse response)
        throws Exception {
        VehicleForm vehicleForm = (VehicleForm) form;
        Vehicle vehicle = vehicleForm.getVehicle();

        request.setAttribute("rants",
            rantService.getRantsForVehicle(vehicle));

        return mapping.findForward("rantList");
    }

    private RantService rantService;
    public void setRantService(RantService rantService) {
        this.rantService = rantService;
    }
}
```

Injects
RantService

Because `RantsForVehicleAction` is configured in Spring, it is given a `RantService` through the `setRantService()` method.

16.1.4 What about Struts 2?

Struts is dead. Long live Struts 2!

If you're a fan of Struts, you're probably aware that there's a change coming to the Struts framework. Struts 2 (or Struts Action Framework, as it's sometimes referred to) is a completely different approach to building MVC applications from the Struts 1.x programming model. As I'm writing this, Struts 2 has recently released its first general availability release, marking it as a production-ready web framework.

What's interesting about Struts 2 is that it's mostly a rebranding of another popular web framework. According to the Struts 2 website (<http://struts.apache.org/2.x/>): "Apache Struts 2 was originally known as WebWork 2. After working independently for several years, the WebWork and Struts communities joined forces to create Struts 2. This new version of Struts is simpler to use and much closer to how Struts was always meant to be."

Identity crises notwithstanding, how can we integrate Spring with Struts 2? Well, since Struts 2 is effectively the same as WebWork 2, we couldn't ask for a

better segue into the next section. Coming up next, we'll look at how to integrate Spring with WebWork 2 (and Struts 2).

16.2 Working Spring into WebWork 2/Struts 2

WebWork is an open source web framework from Open Symphony that has been popular for quite some time. Despite its name, WebWork is actually a service invocation framework that is not specific to just web applications. In its simplest form, WebWork is based around general-purpose actions. These actions process requests and then return a `String` that indicates the next step in the request chain. This could be another action or a view. However, nothing about this is web specific.

Nevertheless, for our purposes we will be discussing WebWork in the context of web applications. We're going to assume that you are already familiar with WebWork and that you're reading this section to see how you can use Spring with your WebWork applications. Therefore, we'll only cover just enough WebWork to accommodate the integration with Spring. For a more detailed coverage of WebWork, I highly recommend *WebWork in Action* (Manning, 2005).

Let's pretend that we had written the web layer of the RoadRantz application using WebWork instead of Spring MVC. In the Spring MVC version of RoadRantz, we wrote `RantsForDayController` (listing 13.8), which produced a list of rants that were posted on a particular month, day, and year. In the WebWork version, we would instead write `RantsForDayAction`, as shown in listing 16.4.

Listing 16.4 A WebWork implementation of a RoadRantz feature

```
package com.roaddrantz.webwork;
import java.util.Date;
import java.util.List;
import com.opensymphony.xwork.Action;
import com.roaddrantz.domain.Rant;
import com.roaddrantz.service.RantService;

public class RantsForDayAction implements Action {
    private int month;
    private int day;
    private int year;
    private List<Rant> rants;

    public String execute() throws Exception {
        Date date = new Date(month, day, year);
        rants = rantService.getRantsForDay(date);           ← Delegates to
                                                       RantService
        return SUCCESS;
    }
}
```

```
public List<Rant> getRants() {
    return rants;
}

public void setDay(int day) {
    this.day = day;
}

public void setMonth(int month) {
    this.month = month;
}

public void setYear(int year) {
    this.year = year;
}

private RantService rantService;
public void setRantService(RantService rantService) {
    this.rantService = rantService;
}
}
```

Injects
RantService

For the most part, `RantsForDayAction` represents a typical WebWork action implementation. Instead of being given request parameters through a command object as with Spring's command controllers or through an `ActionForm` as in Struts, WebWork actions are given request parameters through properties. In `RantsForDayAction`, the `month`, `day`, and `year` properties are expected to have been populated from request parameters by the time that the `execute()` method is invoked.

The `execute()` method is where all of the action takes place. Here, the `month`, `day`, and `year` properties are pulled together to construct a `Date`, which is then used to retrieve a list of rants from the `rantService` property.

The `setRantService()` method is the clue as to how `rantService` is set. The presence of this method indicates that `RantsForDayAction` expects the `rantService` property to be injected with some implementation of `RantService`. The big question is how we can get Spring to inject into a WebWork action.

Although there are a number of ways to integrate Spring with WebWork, depending on which version of WebWork you're using, we will be focusing our attention on the latest version of WebWork—version 2.2.3.

Originally, the WebWork project maintained its own dependency injection container. But as of WebWork 2.2, the WebWork team has deprecated their own DI container in favor of Spring. Because of the WebWork team's commitment to Spring, integrating Spring with WebWork involves only three very simple steps:

- 1 Add the Spring JARs to the application's classpath.
- 2 Configure a Spring ContextLoaderListener in web.xml.
- 3 Configure WebWork to use Spring as an object factory.

The first step is almost self-explanatory. Just make sure that spring.jar and any other JARs that your Spring beans require are available in the /WEB-INF/lib directory of the deployed web application so that WebWork can use them.

We talked about how to configure a ContextLoaderListener in chapter 13. For a refresher, turn back to section 13.1.2 to see how ContextLoaderListener loads a Spring application context.

The last step is a simple matter of adding the following line to the webwork.properties file:

```
webwork.objectFactory=spring
```

For Struts 2 projects, the configuration is very similar, with only a slight change. The configuration file is struts.properties and the line should read as follows:

```
struts.objectFactory=spring
```

This line of configuration tells WebWork/Struts 2 to try to retrieve objects from the Spring container before creating them itself. This means that whenever WebWork needs an instance of an action class, it will first look in the Spring application context for the action, as shown in figure 16.3. If it's found then it will use the Spring-managed action. Otherwise, WebWork will instantiate the action itself.

In the case of RantsForDayAction, we'll configure it in the Spring application context like this:

```
<bean id="rantsForDayAction"
      class="com.roaddrantz.webwork.RantsForDayAction"
      scope="prototype">
    <property name="rantService" ref="rantService" />
</bean>
```

Notice that the scope attribute has been set to prototype. That's because WebWork expects a fresh instance of the action to handle each request. That makes

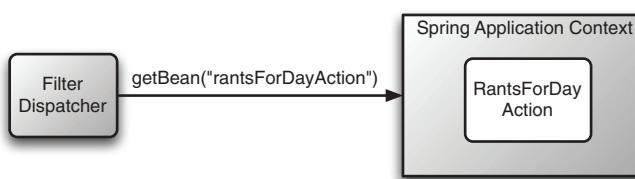


Figure 16.3
By using the `spring` object factory, WebWork/Struts 2 will retrieve its action classes from the Spring application context instead of trying to create them itself.

WebWork actions function very much like Spring MVC's throwaway controllers. In fact, if you compare the code from listing 16.4 with the throwaway controller implementation in listing 13.8, you'll find that they're not very different.

On the WebWork side, we'll need to declare an `<action>` element in `xwork.xml`:

```
<action name="rantsForDay" class="rantsForDayAction">
    <result>dayRants.jsp</result>
</action>
```

Unlike a typical WebWork `<action>` configuration, however, the `class` attribute doesn't contain a class name at all. Instead, it is configured with the name of the Spring-managed bean that holds the action implementation. When WebWork has to handle the `rantsForDay` action, it will ask Spring for an instance of the bean declared with the name `rantsForDayAction`.

Although Struts and WebWork are significant web frameworks, a new breed of component-based web frameworks is capturing the interest of Java web developers. One of the most intriguing of these is Tapestry, which uses plain HTML instead of JSP as its template language. Let's see how to use Spring in a Tapestry application.

16.3 Integrating Spring with Tapestry

Tapestry is another MVC framework for the Java platform that is quite popular. One of the most appealing features of Tapestry is that instead of relying on JSP, Velocity, or some other templating solution, Tapestry uses plain HTML as its template language.

While it may seem peculiar that Tapestry uses a static markup language to drive dynamically created content, it's actually a practical choice. Tapestry components are placed within an HTML page using any HTML tag you want to use (`` is often the tag of choice for Tapestry components). The HTML tag is given a `jwcid` attribute, which references a Tapestry component definition. For example, consider the following simple Tapestry page:

```
<html>
    <head><title>Simple page</title></head>
    <body>
        <h2><span jwcid="simpleHeader">Simple header</span></h2>
    </body>
</html>
```

When Tapestry sees the `jwcid` attribute, it will replace the `` tag (and its content) with the HTML produced by the `simpleHeader` component. The nice thing about this approach is that page designers and Tapestry developers alike can easily understand this HTML template. Even without being processed by the Tapestry engine, Tapestry templates load cleanly into any HTML design tool or browser.

In this section, we're going to look into how to use Spring and Tapestry together so that Tapestry pages and components can use Spring-managed beans. We're going to assume that you are already familiar with Tapestry. If you need to learn more about Tapestry, I recommend *Tapestry in Action* (Manning, 2004).

There are actually two different ways of integrating Spring with Tapestry, depending on whether you are using Tapestry 3 or Tapestry 4. Because there are still a lot of projects being developed on Tapestry 3, we'll start by briefly covering the integration of Spring with Tapestry 3 before looking into the latest integration with Tapestry 4.

16.3.1 Integrating Spring with Tapestry 3

Tapestry's engine maintains an object (known as `global`) that is a simple container for any objects you want shared among all Tapestry sessions. It is a `java.util.HashMap` by default.

The key strategy behind Tapestry-Spring integration is loading a Spring application context into Tapestry's `global` object. Once it's in `global`, all pages can have access to Spring-managed beans by retrieving the context from `global` and calling `getBean()`.

To load a Spring application context into Tapestry's `global` object, you'll need to replace Tapestry's default engine (`org.apache.tapestry.engine.BaseEngine`) with a custom engine. Unfortunately, Spring does not come with such a replacement Tapestry engine. This leaves it up to us to write it for ourselves (even though it's virtually the same for any Spring/Tapestry hybrid application).

`SpringTapestryEngine` (listing 16.5) extends `BaseEngine` to load a Spring application context into the Tapestry `global` property.

Listing 16.5 A replacement Tapestry engine that loads a Spring context into global

```
package com.springinaction.tapestry;
import javax.servlet.ServletContext;
import org.apache.tapestry.engine.BaseEngine;
import org.apache.tapestry.request.RequestContext;
import org.springframework.context.ApplicationContext;
import org.springframework.web.context.support.
    ➔ WebApplicationContextUtils;
```

```

public class SpringTapestryEngine extends BaseEngine {
    private static final String SPRING_CONTEXT_KEY = "springContext";

    protected void setupForRequest(RequestContext context) {
        super.setupForRequest(context);

        Map global = (Map) getGlobal();

        ApplicationContext appContext =
            (ApplicationContext)
            global.get(SPRING_CONTEXT_KEY); <-- Loads context

        if (appContext == null) {
            ServletContext servletContext =
                context.getServlet().getServletContext();
            appContext = WebApplicationContextUtils.
                getWebApplicationContext(servletContext);
            global.put(SPRING_CONTEXT_KEY, appContext);
        }
    }
}

```

Stores context in Tapestry's Global

SpringTapestryEngine (see figure 16.4) first checks `global` to see if the Spring context has already been loaded. If so then there is nothing to do. But if `global` doesn't already have a reference to the Spring application context, it will use `WebApplicationContextUtils` to retrieve a web application context. It then places the application context into `global`, under the name `springContext`, for later use.

Because `SpringTapestryEngine` uses `WebApplicationContextUtils` to look up the application context, you'll need to be sure to load the context into your web application's servlet context using `ContextLoaderListener`. Refer to section 13.1.2 in chapter 13 for details on using `ContextLoaderListener`.

Note that there is one limitation of `SpringTapestryEngine` as it is written. It assumes that the `global` object is a `java.util.Map` object. This is usually not a

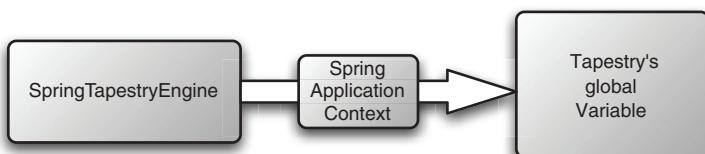


Figure 16.4 `SpringTapestryEngine`, which replaces Tapestry's default engine, places the Spring application context into `global`, thereby making the entire Spring context available to all Tapestry pages and components.

problem as Tapestry defaults `global` to be a `java.util.HashMap`. But if your application has changed this by setting the `org.apache.tapestry.global-class` property, `SpringTapestryEngine` will need to change accordingly.

The last thing to do is to replace the default Tapestry engine with `SpringTapestryEngine`. This is accomplished by configuring the `engine-class` attribute of your Tapestry application:

```
<application name="RoadRantz"
    engine-class="com.springinaction.tapestry.SpringTapestryEngine">
    ...
</application>
```

At this point, the Spring application context is available in Tapestry's `global` object, ready to be used to dispense Spring-managed service beans. Let's have a look at how to wire those service beans into a Tapestry page specification.

Loading Spring beans into Tapestry pages

Suppose that we had implemented the web layer of the RoadRantz application using Tapestry instead of Spring MVC. In a Tapestry application, all pages have a page class that defines the functionality of the page. In a Tapestry implementation of RoadRantz, the homepage may be backed by the `HomePage` class in listing 16.6.

Listing 16.6 A Tapestry 3 version of the RoadRantz homepage

```
package com.roaddrantz.tapestry;
import java.util.List;
import org.apache.tapestry.html.BasePage;
import com.roaddrantz.domain.Rant;
import com.roaddrantz.service.RantService;

public abstract class HomePage extends BasePage {
    public abstract RantService getRantService();           | Getter-injects
                                                               | getRantService()

    public abstract Rant getRant();
    public abstract void setRant(Rant rant);

    public List getRants() {
        return getRantService().getRecentRants();           | Delegates to
                                                               | RantService
    }
}
```

The key thing to note in listing 16.6 is that `HomePage` uses a `RantService` to retrieve the list of recent rant entries. But where does this `RantService` come from? To understand how `HomePage` gets a `RantService`, you first must understand how Tapestry populates page properties.

You've no doubt noticed that the `getRantService()` method is abstract, which implies `HomePage` must be subclassed for `getRantService()` to be given a concrete implementation. But don't worry—you won't have to subclass `HomePage`. Tapestry will handle that for you.

When populating page properties, Tapestry uses a form of getter injection. Much like Spring's getter injection capabilities that we discussed in chapter 3, when Tapestry loads `HomePage`, it will create a subclass of it with a concrete implementation of `getRantService()` that will return a specific value.

The value that will be returned by `getRantService()` will be decided by `HomePage`'s page specification—`home.page`:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE page-specification
  PUBLIC "-//Apache Software Foundation//"
    ↪ Tapestry Specification 3.0//EN"
  "http://jakarta.apache.org/tapestry/dtd/Tapestry_3_0.dtd">

<page-specification
  class="com.roaddrantz.tapestry.HomePage">
  <property-specification name="rantService"
    type="com.roaddrantz.service.RantService">
    global.springContext.getBean("rantService")
  </property-specification>

  <property-specification name="rant"
    type="com.roaddrantz.domain.Rant"/>
</page-specification>
```

The `<property-specification>` entry in `home.page` tells Tapestry to retrieve the value of the `rantService` property by calling `getBean()` on the Spring context that was placed in `global`. Effectively, Tapestry will implement the `getRantService()` method of `HomePage` to return the value in the `<property-specification>` element.

Now that Tapestry knows how to implement the `getRantService()` method, it can be used in the `getRants()` method to retrieve a list of recently posted rants. Consequently, the list of rants can be displayed by placing the following in `home.html`:

```
<ul>
  <span jwcid="@Foreach" source="ognl:rants"
    value="ognl:rant">
    <li><span jwcid="@Insert" value="rant.vehicle.state"/>
      <span jwcid="@Insert" value="rant.vehicle.plateNumber"/> --
      <span jwcid="@Insert" value="rant.rantText "/></li>
    </span>
  </ul>
```

A nice thing about how the Tapestry-Spring integration works is that neither the Tapestry page class nor the HTML template is aware that Spring is involved. The only place where Spring is mentioned is in the page specification file.

Now that you've seen how to integrate Spring into a Tapestry 3 application, let's move forward a bit and see how Spring can work with the latest version of Tapestry, Tapestry 4.

16.3.2 Integrating Spring with Tapestry 4

Tapestry 3 is a fantastic framework for building web applications. But if you've worked with it much, you know that it demands that you write quite a bit of XML in the form of page and component specifications. Among other improvements, Tapestry 4 takes advantage of Java 5 annotations to greatly reduce the amount of XML required in a Tapestry-based application.

Moreover, integrating Spring into a Tapestry 4 application has been greatly simplified, thanks largely to a project called Diaphragma. According to its homepage, Diaphragma's aim is to "provide many excellent components and extensions for [the] Tapestry framework." At the time I'm writing this, however, the only extension offered by Diaphragma is one that simplifies integration with Spring. As fortune would have it, that's exactly what we need!

The first step to using Spring in your Tapestry 4 applications is to download the `tapestry-spring.jar` file from the Diaphragma page on SourceForge: <http://sourceforge.net/projects/diaphragma>. Once you have the JAR file, make sure that it's in your application's classpath by placing it in the `/WEB-INF/lib` directory of the deployed application.

Next you'll need to make sure that you have the Spring application context loaded. Just as with integrating Spring and Tapestry 3, this involves placing a `ContextLoaderListener` in the application's `web.xml` file. Refer to section 13.1.2 in chapter 13 for details on how to configure `ContextLoaderListener`.

Now you're ready to start injecting Spring-configured beans into your Tapestry pages and components. The `HomePage` class in listing 16.7 illustrates the easiest way to do this.

Listing 16.7 Using annotations to inject Spring beans into a Tapestry page

```
package com.roaddrantz.tapestry;
import java.util.List;
import org.apache.tapestry.annotations.InjectObject;
import org.apache.tapestry.html.BasePage;
import com.roaddrantz.domain.Rant;
import com.roaddrantz.service.RantService;
```

```
public abstract class HomePage extends BasePage {  
    @InjectObject("spring:rantService")  
    public abstract RantService getRantService();  
  
    public abstract Rant getRant();  
    public abstract void setRant(Rant rant);  
    public List<Rant> getRants() {  
        return getRantService().getRecentRants();  
    }  
}
```

Retrieves RantService from Spring

You'll notice that there's virtually no difference between listing 16.7 and the Tapestry 3 version in listing 16.6. In fact, Tapestry 3 isn't much different from Tapestry 4 with regard to developing page-backing classes.

However, notice that there's one small addition to listing 16.7. The `getRantService()` method has been annotated with `@InjectObject`. This annotation tells Tapestry that the `getRantService()` should be injected with some object. The value given to the annotation specifies the name of the object to be injected. In this case, the object name is prefixed with the `spring` namespace, indicating that Tapestry should retrieve the `rantService` bean from the Spring application context and use it as the return value of `getRantService()`.

Just because Tapestry 4 encourages the use of Java 5 annotations, that doesn't mean you're out of luck if you've not made the move to Java 5 yet. Instead of using the `@InjectObject` annotation in the page-backing class, you can optionally configure the injection of `getRantService()` in the page specification XML file:

```
<page-specification  
    class="org.apache.tapestry.html.BasePage">  
    ...  
    <inject property="rantService"  
        object="spring:rantService" />  
    ...  
</page-specification>
```

The `<inject>` element performs the same job as the `@InjectObject` annotation. That is, it injects a property with some value. In this case, the `rantService` property (represented by the abstract `getRantService()` method) will be injected with the `rantService` bean from the Spring application context.

Tapestry is just one of several component-based web application frameworks. JavaServer Faces (JSF) is another such framework that carries the distinction of being defined as a Java standard. Coming up in the next section, we're going to see how to use Spring with JSF.

16.4 Putting a face on Spring with JSF

Relatively speaking, JavaServer Faces (JSF) is a newcomer in the space of Java web frameworks. But it has a long history. First announced at JavaOne in 2001, the JSF specification made grand promises of extending the component-driven nature of Swing and AWT user interfaces to web frameworks. The JSF team produced virtually no results for a very long time, leaving some (including me) to believe it was vaporware. Then in 2002, Craig McClanahan (the original creator of Jakarta Struts) joined the JSF team as the specification lead and everything turned around.

After a long wait, the JSF 1.0 specification was released in February 2004 and was quickly followed by maintenance 1.1 specification in May 2004 and another specification update in August 2005. Now JSF is being promoted as the standard Java web development platform and has been adopted by a large number of developers. With so much attention being given to JSF, it would seem appropriate for integration solutions to exist for JSF and Spring.

If you're already familiar with JSF, you know that JSF has built-in support for dependency injection. You may be wondering why you should bother integrating Spring into JSF. It's true that JSF's support for setter injection is not all that different from that of Spring. But remember that Spring offers more than just simple dependency injection. Spring can bring a lot of value-add to JSF. Spring's other features (such as declarative transactions, security, remoting, etc.) could come in handy in a JSF application.

Furthermore, even though JSF is intended as a presentation layer framework, service- and data access-layer components are frequently declared in a JSF configuration file. This seems somewhat inappropriate. Wouldn't it be better to separate the layers, making JSF responsible for presentation stuff and Spring responsible for the rest of the application?

In this section, I'll show you how to expose Spring-managed beans to JSF. I'm assuming that you are already familiar with JSF. If you are new to JSF or just need a refresher, I recommend that you have a look at *JavaServer Faces in Action* (Manning, 2004).

Before I can show you how Spring integrates with JSF, it's important to understand how JSF resolves variables on its pages without Spring. Therefore, let's have a quick look at how JSF works without Spring and then we'll see how Spring can be used to provide beans for use in JSF pages.

16.4.1 Resolving JSF-managed properties

Imagine that before you had ever heard of Spring, you had already developed the RoadRantz application using JSF in the web layer. As part of the application, you have created a form that is used to register new motorists. The following excerpt from the JSF-enabled registerMotorist.jsp file shows how JSF binds the variables of a Motorist object to the fields in the form:

```
<h:form>
    <h2>Register Motorist</h2>
    <h:panelGrid columns="2">
        <f:verbatim><b>E-mail:</b></f:verbatim>
        <h:inputText value="#{motorist.email}" required="true"/>
        <f:verbatim><b>Password:</b></f:verbatim>
        <h:inputText value="#{motorist.password}" required="true"/>

        <f:verbatim><b>First Name:</b></f:verbatim>
        <h:inputText value="#{motorist.firstName}" required="true"/>
        <f:verbatim><b>Last Name:</b></f:verbatim>
        <h:inputText value="#{motorist.lastName}" required="true"/>
    ...
    </h:panelGrid>
    <h:commandButton id="submit" action="#{motorist.register}"
                      value="Register Motorist"/>
</h:form>
```

Notice that the action parameter of the `<h:commandButton>` is set to `#{motorist.register}`. Unlike many other MVC frameworks (including Spring's MVC), JSF doesn't use a separate controller object to process form submissions. Instead, JSF passes control to a method in the model bean. In this case, the model bean is a Motorist. When the form is submitted, JSF will call the `register()` method of the motorist bean to process the form. The `register()` method is defined as follows:

```
public String register() {
    try {
        rantService.addMotorist(this);
    } catch (Exception e) {
        return "error";
    }

    return "success";
}
```

To keep the Motorist bean as simple as possible, the `register()` method simply delegates responsibility to the `addMotorist()` method of a RantService implementation. But where does the `rantService` property get set?

That's a very good question. Internally, JSF uses a variable resolver to locate beans that are managed within the JSF application. The default JSF variable resolver looks up variables declared within the JSF configuration file (`faces-config.xml`). More specifically, consider the following declaration of the `motorist` bean in `faces-config.xml` for the answer to where the `rantService` property comes from:

```
<managed-bean>
  <managed-bean-name>motorist</managed-bean-name>
  <managed-bean-class>
    com.rodrantz.domain.Motorist
  </managed-bean-class>
  <managed-bean-scope>request</managed-bean-scope>
  <managed-property>
    <property-name>rantService</property-name>
    <value>#{rantService}</value>
  </managed-property>
</managed-bean>
```

Here the `motorist` bean is declared as a request-scoped JSF-managed bean. But take note of the `<managed-property>` element. JSF supports a simple implementation of setter injection. `#{rantService}` indicates that the `rantService` property is being wired with a reference to a bean named `rantService`.

In a conventional JSF application (e.g., one where Spring is not involved), the `rantService` bean would be declared in `faces-config.xml` as a JSF-managed bean:

```
<managed-bean>
  <managed-bean-name>rantService</managed-bean-name>
  <managed-bean-class>
    com.rodrantz.service.RantServiceImpl
  </managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
  <managed-property>
    <property-name>rantDao</property-name>
    <value>#{rantDao}</value>
  </managed-property>
</managed-bean>
```

Once again, dependency injection is employed in the `rantService` bean—the `rantDao` property is wired with a reference to a bean named `rantDao`. And if we were to continue this exploration of `faces-config.xml`, we'd find that the `rantDao` bean is injected with a `javax.sql.DataSource`, which itself is also a JSF-managed bean.

Now that you know how JSF resolves variables on its pages without Spring, let's add Spring to the mix so that non-presentation-layer beans can be managed in Spring and take advantage of the full gamut of Spring features.

16.4.2 Resolving Spring beans

For integration with Spring, we would like JSF to resolve its variables from the Spring application context. To do that, we'll need to replace the default JSF variable resolver with a Spring-aware variable resolver.

Spring's `DelegatingVariableResolver` is just such a variable resolver. Rather than resolve variables only from among JSF's managed beans, `DelegatingVariableResolver` also looks in the Spring application context. It is configured in `faces-config.xml` like this:

```
<application>
    <variable-resolver>
        org.springframework.web.jsf.DelegatingVariableResolver
    </variable-resolver>
</application>
```

When JSF needs to resolve a variable, `DelegatingVariableResolver` will first look to the original variable resolver. If a JSF-managed bean can be found that fits the bill then it will be used. If not, however, `DelegatingVariableResolver` will then turn to the Spring application context to see if it can find a bean whose name matches the JSF variable name.

For `DelegatingVariableResolver` to be able to resolve variables as Spring-managed beans, we'll need to make sure that the Spring application context is loaded. To do that we'll need to configure a `ContextLoaderListener` in the application's `web.xml` file. For more information on how to configure `ContextLoaderListener`, please turn to section 13.1.2 in chapter 13.

With `DelegatingVariableResolver` in place and the application context loaded, you are now ready to wire your service and data access layer beans in Spring and access them from JSF.

16.4.3 Using Spring beans in JSF pages

`DelegatingVariableResolver` makes the resolving of Spring-managed beans transparent in JSF. To illustrate, recall that the JSF-managed `motorist` bean is injected with a reference to the `rantService` bean using the following `<managed-property>` declaration in `faces-config.xml`:

```
<managed-property>
    <property-name>rantService</property-name>
    <value>#{rantService}</value>
</managed-property>
```

Even though the `rantService` bean is now going to reside in the Spring context, nothing needs to change about the existing declaration of the `motorist` bean.

When it comes time to inject the `rantService` property of the `motorist` bean, it asks `DelegatingVariableResolver` for the reference to the `rantService` bean. `DelegatingVariableResolver` will first look in the JSF configuration for the bean. When it can't find it, it will then look in the Spring application context, as shown in figure 16.5.

But it will only find the `rantService` bean in the Spring context if you declare it there. So, instead of registering the `rantService` bean as a `<managed-bean>` in `faces-config.xml`, place it in the Spring context definition file as follows:

```
<bean id="rantService"
      class="com.roadrantz.service.RantServiceImpl">
    <property name="rantDao" ref="rantDao" />
</bean>
```

Notice that this declaration of `rantService` is no different from how it would be declared in an application that uses Spring MVC. In fact, from the service layer to the data access layer, you will declare all of your application beans in the Spring application context exactly the same as you would if your application were fronted by Spring MVC. `DelegatingVariableResolver` will find them as though they are part of the JSF configuration.

Resolving Spring beans as JSF variables is the key part of the JSF-Spring integration. But maybe you don't want to pick and choose Spring beans to expose as JSF-managed beans. Instead, it may be more convenient for JSF to have wholesale

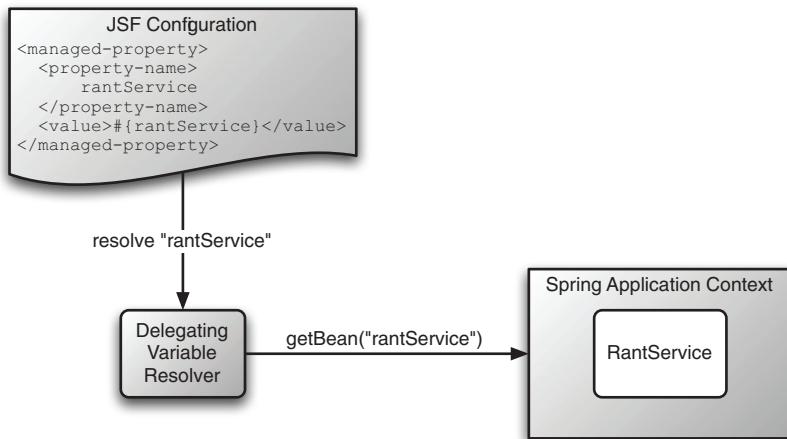


Figure 16.5 After plugging Spring's `DelegatingVariableResolver` into JSF, JSF variables are resolved from Spring beans if they can't be found in the JSF configuration.

access to the Spring application context. For that, let's look at another JSF variable resolver that lets JSF access the Spring application context itself.

16.4.4 Exposing the application context in JSF

It's worth mentioning that Spring offers another option for integration with JSF. Whereas `DelegatingVariableResolver` transparently resolves JSF variables from among all of the beans in a Spring application context, `WebApplicationContextVariableResolver` only resolves one variable.

What good can `WebApplicationContextVariableResolver` possibly be if it only resolves a single variable? Well, it can be very handy if that one variable is the Spring application context itself!

You can configure JSF to use `WebApplicationContextVariableResolver` by placing the following `<variable-resolver>` entry in `faces-config.xml`:

```
<variable-resolver>
    org.springframework.web.jsf.WebApplicationContextVariableResolver
</variable-resolver>
```

With `WebApplicationContextVariableResolver` in place, your JSF pages have direct access to the Spring application context through a variable named `webApplicationContext`.

We've seen how Spring integrates into four of the most popular web frameworks. This opens up our set of options for web development with Spring, making it possible to choose the most appropriate web framework for the project. Now we'll turn our attention to the very exciting topic of Ajax, and how to expose Spring objects.

16.5 Ajax-enabling applications in Spring with DWR

Traditionally, web applications have involved a back-and-forth conversation between the user and the server. The user sends a request to the server by either clicking a link or submitting a form. The server responds in kind by returning a page that represents the state of the application after the request is processed.

Contrasted with rich desktop applications, browser-based applications have been much less dynamic. Where a desktop application can update individual components as needed to communicate the state of the application with the user, web applications have only been able to respond a page at a time. This is a throwback to the browser's original purpose: to display documents.

The web browser has come a long way since the days of the National Center for Supercomputing Applications (NCSA) Mosaic. In addition to displaying simple

documents, modern web browsers are capable of dynamic behavior. For several years, Dynamic HTML (DHTML), the marriage of JavaScript, HTML, and Cascading Style Sheets, has enabled flashier looking web applications that mimic desktop application behavior.

Nevertheless, even though these DHTML applications present themselves as more dynamic than traditional web applications, until recently they were still page based and required a round-trip to the server to set and retrieve application state. While DHTML could dynamically alter the appearance of a web page, the missing piece was still a way to communicate with the server without reloading the web page.

In recent years, however, there has been a near revolution in how browser-based applications are developed. In the 5.0 version of Internet Explorer, Microsoft included an ActiveX component called XMLHttpRequest. XMLHttpRequest can be used with JavaScript to communicate with the server in XML messages. This technique for browser-to-server communication has been named Ajax, or Asynchronous JavaScript and XML.

When Ajax is mixed with DHTML, very dynamic web applications can be built that blur the lines between desktop applications and web applications. A browser-based application can communicate with the server using Ajax and then use DHTML to alter the page's appearance to reflect the current application state. Google's Gmail, Calendar, and Docs applications are prime examples of how Ajax can be used to build extremely dynamic web applications.

Even though XMLHttpRequest got its start in Internet Explorer 5.0, it's now a common implement in all modern web browsers, including Internet Explorer, Mozilla Firefox, and Safari. This means that Ajax-capable web applications can be developed to target virtually every browser on every desktop.

Because they're often used in tandem, many developers have lumped DHTML and Ajax together under the umbrella name of Ajax. But at its core, Ajax is a mechanism for communication between a web page and the server. In this section we're going to focus our attention on the communication aspect of Ajax, looking at how to use an Ajax toolkit known as DWR to expose Spring beans as Ajax-remoted objects.

If you'd like to read more about Ajax, then there's no shortage of books on the topic. I recommend that you have a look at *Ajax in Action* (Manning, 2006), *Prototype and Scriptaculous in Action* (Manning, 2007), *Ajax in Practice* (Manning, 2007), or *Pragmatic Ajax* (Pragmatic Bookshelf, 2006).

16.5.1 Direct web remoting

Almost as quickly as Ajax became theuzziest of buzzwords in software development, several frameworks emerged to simplify Ajax development. Many of them focus on providing DHTML widgets for dressing up web applications. But one framework, known as DWR, has stayed true to the central purpose of Ajax: communication between a web page and the server.

DWR, which is short for Direct Web Remoting, is a Java-based Ajax framework that lets you access virtually any server-side Java object through JavaScript. DWR abstracts XMLHttpRequest away so that invoking methods on a server-side Java object is as simple as invoking methods on a client-side JavaScript object.

In a sense, DWR is more of a remoting technology, akin to the remoting options presented in chapter 8, than it is a web technology. What separates DWR from those other remoting technologies is where its client resides. Whereas the clients of RMI, HTTP invoker, Hessian, and Burlap are other applications, the client of a DWR-exported object is JavaScript within a web page.

You can learn more about DWR from its homepage at <https://dwr.dev.java.net>. From there you can download the latest version of DWR. For the examples in this chapter, I'm using DWR 2.0.¹

We'll look at several ways of using DWR to export Spring beans to be invoked from JavaScript. But before we look at the DWR-Spring connection, let's start with the basics. To set the stage for DWR and Spring integration, let's begin by configuring DWR sans Spring.

Basic DWR configuration

In its most basic configuration, DWR exports remote Java objects to JavaScript through a servlet—DwrServlet to be precise. Therefore, the first thing we'll need to do is add DwrServlet to web.xml using the following <servlet> and <servlet-mapping> entries:

```
<servlet>
    <servlet-name>dwr</servlet-name>
    <servlet-class>
        org.directwebremoting.servlet.DwrServlet
    </servlet-class>
    <init-param>
        <param-name>debug</param-name>
        <param-value>true</param-value>
```

¹ As I write this, DWR 2.0 isn't final, but I'm hopeful that it will be by the time you read this. I'm actually using DWR 2.0 milestone 4d.

```

</init-param>
</servlet>

<servlet-mapping>
  <servlet-name>dwr</servlet-name>
  <url-pattern>/dwr/*</url-pattern>
</servlet-mapping>

```

As configured here, the `DwrServlet` will respond to requests whose URL, relative to the application's context path, is `/dwr/*`. This is important to remember, because it is through this URL that we'll load JavaScript code generated by DWR.

Also, notice that `DwrServlet` is configured with an `<init-param>` named `debug` set to true. This setting turns on a helpful debugging feature that we'll use a little later. For production applications, however, you'll want to remove this parameter or set it to false.

Defining the remote objects

Now that `DwrServlet` is set up, we're ready to export some objects to JavaScript. But what objects should we export?

To demonstrate DWR's capabilities, let's add a traffic report feature to the `RoadRantz` application. After all, cranky motorists might be a bit less cranky if they could only check traffic conditions before they hit the road!

The user interface for the traffic report feature should prompt the user for a zip code and produce a list of traffic incidents in or near the zip code. To further constrain the results, the user should also be able to specify a zoom factor (how large of an area relative to the zip code to include in the search) and a minimum incident severity.

At the heart of the traffic report feature is `TrafficServiceImpl` (listing 16.8), a service class whose `getTrafficInfo()` method accepts a zip code, a zoom factor, and a severity and produces a list of traffic incidents.

Listing 16.8 A traffic service implementation that looks up traffic data from an RSS feed

```

package com.roaddrantz.traffic;
import java.io.InputStreamReader;
import java.net.URL;
import java.util.Iterator;
import com.sun.syndication.feed.synd.SyndEntryImpl;
import com.sun.syndication.feed.synd.SyndFeed;
import com.sun.syndication.io.SyndFeedInput;

public class TrafficServiceImpl
    implements TrafficService {

```

```

private static final String ROOT_TRAFFIC_URL =
    "http://local.yahooapis.com/MapsService/rss/" +
    "trafficData.xml?";
```

Uses Yahoo!'s traffic RSS feed

```

public TrafficServiceImpl() {}
```

```

public TrafficInfo[] getTrafficInfo(
    String zipCode, int zoom, int severity) {
    try {
        URL feedUrl = new URL(ROOT_TRAFFIC_URL +
            "appid=" + appId +
            "&zip=" + zipCode +
            "&zoom=" + zoom +
            "&severity=" + severity +
            "&include_map=0");
```

Sets up URL

```

SyndFeedInput input = new SyndFeedInput();
SyndFeed feed = input.build(
    new InputStreamReader(feedUrl.openStream()));
```

Opens feed

```

TrafficInfo[] trafficInfo =
    new TrafficInfo[feed.getEntries().size()]; ← Gets feed info
```

```

int i=0;
for (Iterator iter = feed.getEntries().iterator();
     iter.hasNext(); i++) {
    SyndEntryImpl entry = (SyndEntryImpl) iter.next();
    trafficInfo[i] = new TrafficInfo();
    trafficInfo[i].setSummary(entry.getTitle());
    trafficInfo[i].setDetails(
        entry.getDescription().getValue());
}
```

Processes entries

```

return trafficInfo;
} catch (Exception e) {
    System.out.println(e.getMessage());
    e.printStackTrace(System.out);
    return new TrafficInfo[] {};
}
}

private String appId = "springinaction";
public void setappId(String appId) {
this.appId = appId;
}
}

```

As you can see, the real magic behind the `getTrafficInfo()` method is that it refers to an RSS feed produced by Yahoo! to get its traffic data. We're using the Rome RSS reader (<http://rome.dev.java.net>) to read and parse the feed. Yahoo!'s traffic data RSS feed is a free service, but it does require that you provide a valid

application ID, configured in `TrafficServiceImpl` through the `appId` property. By default, `TrafficServiceImpl` will use `springinaction` as its application ID (which is a valid ID).

What's most remarkable about `TrafficServiceImpl` is that it has no idea that we're about to use it as a remote object that will be accessed from JavaScript. It's just a POJO. But as you'll soon see, its destiny involves Ajax and will be influenced by DWR.

Get your own ID, please

In my examples, I'm letting you use my Yahoo! application ID (`springinaction`). You're welcome to use it while trying out these examples. But if you intend to use Yahoo!'s services beyond the scope of this book, you should register for your own ID at http://api.search.yahoo.com/webservices/register_application.

The parameters to `getTrafficInfo()` are simple types. The return value, on the other hand, is a bit more complex. It is an array of `TrafficInfo` objects. The `TrafficInfo` class is a simple two-property bean, as shown in listing 16.9.

Listing 16.9 A `TrafficInfo` object contains summary and detail information about a traffic incident

```
package com.roaddrantz.traffic;

public class TrafficInfo {
    private String summary;
    private String details;

    public TrafficInfo() {}

    public String getDetails() { return details; }
    public void setDetails(String details) {
        this.details = details;
    }

    public String getSummary() { return summary; }
    public void setSummary(String summary) {
        this.summary = summary;
    }
}
```

In a traditional web application, built with Spring MVC, the user would enter this information into a form and submit it to a Spring MVC controller. The controller, which would be wired with a reference to `TrafficServiceImpl`, would call the

`getTrafficInfo()` method to get an array of `TrafficInfo` items. Then it would package the array in a `ModelAndView` object and send it to a JSP to render a new page in the browser showing a list of traffic incidents.

But this isn't a traditional web example. We're talking about Ajax and we'd prefer to not load a completely new page in the browser. Instead, we'd like for the list of traffic incidents to appear on the same page where the search criteria are entered. To accomplish that, we'll need DWR to export the `TrafficServiceImpl` class as a JavaScript object.

Exporting the remote objects to JavaScript

The principal means of configuring DWR is by adding a file named `dwr.xml` in the WEB-INF directory of the web application. Within `dwr.xml` we, among other things, tell DWR about the Java classes that we want exported to JavaScript.

Listing 16.10 shows the `dwr.xml` file that we'll use to export the `TrafficServiceImpl` class to JavaScript.

Listing 16.10 Exporting `TrafficServiceImpl` to JavaScript using a new creator in `dwr.xml`

```
<!DOCTYPE dwr PUBLIC
  "-//GetAhead Limited//DTD Direct Web Remoting 1.0//EN"
  "http://www.getahead.ltd.uk/dwr/dwr10.dtd">

<dwr>
  <allow>
    <convert converter="bean"
      match="com.roaddrantz.traffic.TrafficInfo"/> | Converts TrafficInfo
                                                       from a bean

    <create creator="new" javascript="Traffic">
      <param name="class" value=
        "com.roaddrantz.traffic.TrafficServiceImpl" /> | Exports
      <exclude method="setAppId"/>                         TrafficServiceImpl

    </create>
  </allow>
</dwr>
```

The first thing to notice in listing 16.10 is a `<convert>` element. DWR is able to translate simple Java types (such as `String` and `int`) into equivalent types in JavaScript, but complex types such as `TrafficInfo` are a bit more difficult. Therefore, we've configured a “bean” converter to tell DWR to treat `TrafficInfo` as a basic JavaBean when translating it into a JavaScript type. This simply means that the JavaScript version of `TrafficInfo` will have the same properties as the Java version.

The main item of interest in listing 16.10 is the `<create>` element. `<create>` is used to tell DWR to expose a Java class as a remote object in JavaScript. Here we're using a basic new creator, which essentially means that the remote service will be created by instantiating the class specified by the `class` parameter. Because we want to be able to access `TrafficServiceImpl` in JavaScript, we've set the value of the `class` parameter to the fully qualified class name. On the JavaScript side, the remote object will be known as `Traffic`, as configured through the `javascript` attribute.

The only other important item is the `<exclude>` element within `<create>`. While we want the JavaScript client to be able to invoke the `getTrafficInfo()` method on `TrafficServiceImpl`, we do not want the client to be able to set the application ID. Therefore, we've used `<exclude>` to exclude the `setAppId()` method from being exposed to JavaScript.

As configured, the `DwrServlet` will make the `TrafficServiceImpl` available as a remote Ajax object, with the `getTrafficInfo()` method available to JavaScript clients, as shown in figure 16.6.

Speaking of the JavaScript client, we'll need to add a few `<script>` elements to the HTML so that DWR and the exported objects are available to JavaScript. First, we'll need to load the JavaScript that contains the DWR engine:

```
<script type='text/javascript'
       src='dwr/engine.js'></script>
```

Next, we'll need to load our `TrafficServiceImpl` class, exported in JavaScript as `Traffic`:

```
<script type='text/javascript'
       src='dwr/interface/Traffic.js'></script>
```

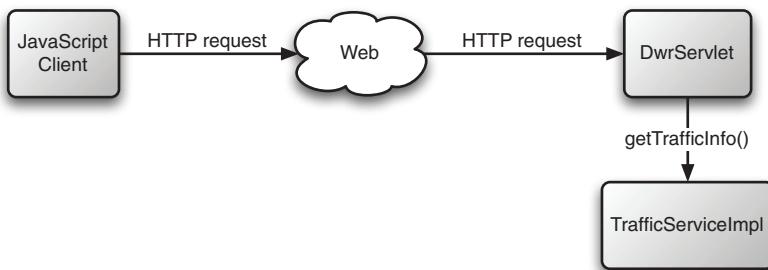


Figure 16.6 `DwrServlet` makes POJOs available as remote Ajax objects that can be invoked from JavaScript clients running in a web browser.

Hold on... We haven't written a JavaScript version of `TrafficServiceImpl`. Where is all of this JavaScript coming from? I'm glad you asked.

Notice that both `engine.js` and `Traffic.js` are loaded with paths that start with `dwr/`. This `dwr/` is the very same `dwr/` that the `DwrServlet` is mapped to in `web.xml`. That means that `DwrServlet` is serving those JavaScript files to the client.

In fact, `Traffic.js` is dynamically generated from the `<create>` element that we configured in `dwr.xml`. It contains code to create a JavaScript object called `Traffic`, which is nothing more than a client stub for the server-side `TrafficServiceImpl` object. In a moment, we'll use the `Traffic` object to retrieve traffic information.

But first, there's one more bit of JavaScript that we'll choose to include:

```
<script type='text/javascript'  
src='dwr/util.js'></script>
```

DWR's `util.js` is optional, but contains some very handy functions for performing DHTML tasks. For the RoadRantz traffic report feature, we'll use a function from `util.js` to dynamically populate an HTML `<table>` with rows of traffic incident data.

Calling remote methods from JavaScript

On the client side, we're going to ask the user to enter a zip code and to optionally choose a zoom factor and a minimum severity. These criteria are entered using an `<input>` and two `<select>`s. The following fragment of HTML shows the relevant portion of the user interface:

```
<input type="text" name="zip" maxlength="5"  
onkeyup="criteriaChanged();"/>  
..  
<select name="zoom" onchange="criteriaChanged();">  
..  
</select>  
..  
<select name="severity" onchange="criteriaChanged();">  
..  
</select>
```

Rather than have the user click a submit button to trigger a traffic incident query, we want the UI to respond immediately to changes in the criteria fields. Therefore, the `<input>` field's `onkeyup` event and both of the `<select>`s' `onchange` event have been set to call a `criteriaChanged()` JavaScript function. Within `criteriaChanged()`, we'll invoke the remote `getTrafficInfo()` method:

```
function criteriaChanged() {  
    var zipCode = document.trafficForm.zip.value;  
    var zoom = document.trafficForm.zoom.value;
```

```

var severity = document.trafficForm.severity.value;
if(zipCode.length == 5) {
    Traffic.getTrafficInfo(zipCode, zoom, severity,
        updateTable);
} else {
    DWRUtil.removeAllRows("trafficTable");
}
}

```

Here's where the proverbial rubber meets the road... or more precisely, where JavaScript meets DWR. If the length of the value in the zip code field is 5 characters (a fully specified zip code) then we use the `Traffic` object created in `Traffic.js` to call the `getTrafficInfo()` method. Under the covers, `getTrafficInfo()` uses the DWR engine to construct an XMLHttpRequest to call the `getTrafficInfo()` on the server-side `TrafficServiceImpl` object.

When we call `getTrafficInfo()`, we pass it the zip code, the zoom factor, the severity, and... wait a minute. There's an extra parameter in there. What's `updateTable` and how did it get into our call to `getTrafficInfo`?

Remember that DWR is an Ajax framework, and that the "A" in Ajax means "asynchronous." In other words, when a call is made to a remote method, the browser doesn't stop everything and wait for a response. This is important, because the remote object could take a moment or two to respond and we don't want the user experience to be ruined because their web browser froze up.

Even though the browser doesn't stop and wait for a response, we still need a way to get the response from the remote call. For that, we need a callback function. In our case, `updateTable` is the name of the callback function. When the server finally responds with a list of `TrafficInfo` objects, the `updateTable()` function will be called to process the results.

Displaying the results

As you may have guessed from its name, the `updateTable()` function is used to populate a table with rows of traffic information. Before we see how `updateTable()` works, however, let's see the table that it will be updating. The following HTML fragment shows the table that we'll use to display traffic incident data:

```

<table width="100%" border="1" style="font-size:8pt;">
    <thead>
        <tr><td width="100">Summary</td><td>Details</td></tr>
    </thead>
    <tbody id="trafficTable"></tbody>
</table>

```

The key thing to pay attention to in the table definition is the `id` of the `<tbody>` element. We'll refer to this `id` when populating the table.

Now here's the `updateTable()` function, the dynamic part of this application:

```
function updateTable(results) {
    DWRUtil.removeAllRows("trafficTable");
    DWRUtil.addRows("trafficTable", results, cellFuncs);
}
```

`updateTable()` uses the `DWRUtil` object (created in `util.js`) to remove and add rows to the table of traffic incidents. The first thing it does is call `removeAllRows()` to clear out the table and prepare it for a fresh set of data. Then it populates the table by calling `addRows()`, passing in the `results` object that was passed to `updateTable()` by the DWR engine. `results` is the data that was returned from the call to `getTrafficInfo()`—specifically an array of JavaScript-ified `TrafficInfo` objects.

The final parameter to `addRows()` is a bit peculiar and demands some explanation. `addRows()` will add one row to the table for each `TrafficInfo` it finds in the `results` array. But how will it know how to populate each column in the table? That's where the `cellFuncs` array comes in:

```
var cellFuncs = [
    function(data) { return data.summary; },
    function(data) { return data.details; }
];
```

`cellFuncs` is an array of functions, one for each column in the table. As `DWRUtil`'s `addRows()` populates the traffic incident table, it will pass in the current `TrafficInfo` object to each of the functions in `cellFuncs` to determine what value goes into each column. In this example, the first function in `cellFuncs` will be used to populate the summary column and the second function will be used to populate the details column.

That's it! We now have a complete, albeit simple, DWR application. To recap, when the user enters data into the zip code, zoom factor, or severity fields, JavaScript will invoke the `getTrafficInfo()` function on the `Traffic` object. The `Traffic` object, being nothing more than a client stub for the remote `TrafficServiceImpl` object, uses DWR to produce an `XMLHttpRequest` to make a call to the server-side `getTrafficInfo()` method. When a response comes back, the `updateTable()` callback function will be called to populate the table with the traffic incident data.

This is fine, but where does Spring fit into the picture? Let's find out.

16.5.2 Accessing Spring-managed beans DWR

So far, we've only used DWR's plain-vanilla new creator to export the server-side `TrafficServiceImpl` class to the JavaScript client. The new creator is fine for simple applications where the exported class doesn't have any dependencies on other objects. But because DWR is responsible for creating the remote object, it rules out any chance for them to be configured using Spring dependency injection or to be advised by Spring AOP. Wouldn't it be better if DWR could export beans that are managed in the Spring application context?

Fortunately, there are a couple of options available for integration between Spring and DWR. These options include

- You can use spring creators in DWR—as an alternative to the new creator we discussed before, DWR also comes with a spring creator that looks up beans that are to be remoted from the Spring application context.
- If you're using Spring 2.0, you can use DWR's Spring 2.0 configuration namespace to declare beans as DWR-remoted.

We're now going to revisit the RoadRantz traffic application, looking at each of these Spring integration options. Don't worry, though... most of the work we've put into the traffic report application is still valid. The client side code will remain unchanged, as will the `TrafficServiceImpl` and `TrafficInfo` classes. We'll only need to tweak the DWR configuration a bit to use Spring.

With that said, let's start by looking at using DWR's spring creator to export Spring-managed beans as remote objects in JavaScript.

Using Spring creators

Unlike the new creator, DWR's spring creator doesn't actually create objects to export as Ajax objects. Instead, the spring creator retrieves objects from the Spring application context. This way, the object can enjoy all of Spring's dependency injection and AOP goodness. Figure 16.7 illustrates how this works.

Using the spring creator isn't much different from using the new creator. In fact, we only need to make a few minor tweaks to the `dwr.xml` file to make the switch from new to spring:

- The `creator` attribute needs to be changed from `new` to `spring`.
- The `class` parameter is no longer appropriate. Therefore, we'll need to change the `<param>` element's `name` attribute from `class` to `beanName`. The `beanName` parameter is used to identify the Spring-managed bean to be exported.

- Since we're identifying a bean by its name, the fully qualified class name in <param>'s value attribute should be replaced with the name of a bean in the Spring application context.

After applying these changes, the <create> element in dwr.xml looks like this:

```
<create creator="spring" javascript="Traffic">
  <param name="beanName" value="traffic" />
  <exclude method="setId" />
</create>
```

Here we're specifying that DWR should export a bean named `traffic`. The bean itself is declared in the Spring application context like this:

```
<bean id="traffic"
      class="com.rodrantz.traffic.TrafficServiceImpl">
  <property name="appId" value="springinaction" />
</bean>
```

Notice that the <bean> element's `id` attribute corresponds to the `beanName` parameter's value in dwr.xml.

Now that `TrafficServiceImpl` is declared as a Spring bean, we are able to configure and wire it using Spring-style dependency injection. In this case, we're merely injecting the `appId` property with our Yahoo! application ID. Nevertheless, by configuring `TrafficServiceImpl` in Spring, it could be subjected to any of Spring's capabilities, including AOP, declarative transactions, security through Acegi, and so forth.

One other benefit of configuring DWR-exposed objects in Spring is that it frees DWR from the knowledge of how the object is implemented. DWR only knows that

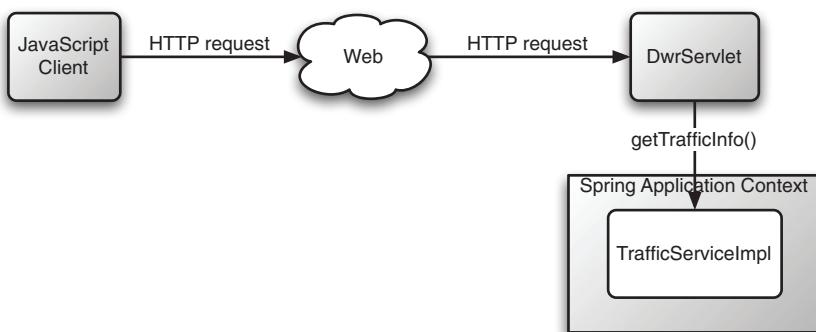


Figure 16.7 Rather than create the remote services itself, DWR's `spring creator` remotes objects loaded from the Spring application object.

it is to expose a bean named `traffic` as an Ajax service. It has no idea what class implements that bean. Although it is declared here as the concrete `TrafficServiceImpl` class, `traffic` could be a reference to a web service, an RMI service, an EJB, or any number of other possibilities.

The question that remains is how the Spring application context gets loaded. By default, the spring creator assumes that a Spring application context has been loaded using `ContextLoaderListener`. For example, the following snippet from the application's `web.xml` loads the Spring context from a file called `traffic.xml` at the root of the application's classpath:

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
        classpath:traffic.xml
    </param-value>
</context-param>
<listener>
    <listener-class>
        org.springframework.web.context.ContextLoaderListener
    </listener-class>
</listener>
```

Optionally, we can explicitly tell the spring creator the name of a Spring application context file by setting the `location` parameter:

```
<create creator="spring" javascript="Traffic">
    <param name="beanName" value="traffic" />
    <param name="location" value="traffic.xml" />
    <exclude method="setappId" />
</create>
```

When `location` is set, DWR will load the Spring application context from the file specified in the `<param>` element's `value` attribute, relative to the application's classpath.

Although the `location` parameter is convenient, it'd be unusual to use DWR's spring creator within an application that hasn't already used `ContextLoaderListener` to load a Spring application context for other purposes. Therefore, in most cases it's best to forego the use of the `location` parameter and rely on `ContextLoaderListener` to load the application context.

DWR's spring creator is great, but it has one small shortcoming: you must configure it in `dwr.xml`. This means that there's one more configuration file for your application. Well, kiss that `dwr.xml` file goodbye, because we're about to see how we can configure DWR completely within Spring.

Using DWR's Spring configuration namespace

One of the most significant new features in Spring 2 is the ability to define custom configuration namespaces. Throughout this book, we've seen several ways that custom namespaces have been used to dramatically cut down on the amount of XML required to configure Spring.

Spring 2's custom namespaces presented an opportunity to the DWR team. Rather than configure DWR in dwr.xml, completely separate from Spring's configuration, why not provide a custom DWR namespace so that DWR can be configured completely within a Spring configuration file? That's precisely what is provided in the latest version of DWR.

To use DWR's configuration in Spring, the first thing we'll need to do is swap out DwrServlet for DwrSpringServlet:

```
<servlet>
  <servlet-name>dwr</servlet-name>
  <servlet-class>
    org.directwebremoting.spring.DwrSpringServlet
  </servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>
```

Where DwrServlet used dwr.xml for determining what objects should be exported as Ajax objects, DwrSpringServlet looks directly at the Spring application context. Specifically, it looks for the elements defined in spring-dwr-2.0.xsd.

So that DWR's configuration elements will be available in the Spring application file, we'll need to add the dwr namespace to the <beans> element:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:dwr=
         "http://www.directwebremoting.org/schema/spring-dwr"
       xsi:schemaLocation=
         "http://www.springframework.org/schema/beans
          http://www.springframework.org/schema/beans/
          spring-beans-2.0.xsd
          http://www.directwebremoting.org/schema/spring-dwr
          http://www.directwebremoting.org/schema/spring-dwr-2.0.xsd">
  ...
</beans>
```

Now we can use <dwr:*> elements to declare certain beans to be exported as Ajax objects. The easiest way to declare a Spring bean as a DWR-exported Ajax object is to include the <dwr:remote> element within the <bean> declaration. For example, here's the TrafficServiceImpl bean, declared to be DWR-remoted:

```
<bean id="traffic"
      class="com.rodrantz.traffic.TrafficServiceImpl">
    <dwr:remote javascript="Traffic">
      <dwr:exclude method="setAppId"/>
    </dwr:remote>
    <property name="appId" value="springinaction" />
</bean>
```

Voilà! Just by adding the `<dwr:remote>` element as a child of the `<bean>` element, we've told DWR to export the bean as a remote Ajax object. The `javascript` attribute specifies a name that the object will be known as in JavaScript.

We still want to hide the `setAppId()` method from the client, so we've also added a `<dwr:exclude>` element as a child of `<dwr:remote>`. By default, all public methods will be exposed, but we can exclude certain methods by listing their names (comma-separated) in `<dwr:exclude>`'s `method` attribute. Here we've specified that the `setAppId()` method should be excluded.

Optionally, we could've used `<dwr:include>` to explicitly declare the methods that are to be exported:

```
<bean id="traffic"
      class="com.rodrantz.traffic.TrafficServiceImpl">
    <dwr:remote javascript="Traffic">
      <dwr:include method="getTrafficInfo"/>
    </dwr:remote>
    <property name="appId" value="springinaction" />
</bean>
```

When `<dwr:include>` is used, only the methods listed in the `method` attribute (comma-separated) are exported in the remote Ajax object. In this case, we're only interested in including the `getTrafficInfo()` method.

We need to tie up one more loose end. We still have to tell DWR how to translate the `TrafficInfo` type into its JavaScript equivalent. We did this with the `<convert>` element in `dwr.xml`. Since we don't have a `dwr.xml` file anymore, we'll use the `<dwr:convert>` element in the Spring configuration:

```
<dwr:configuration>
  <dwr:convert type="bean"
    class="com.rodrantz.traffic.TrafficInfo" />
</dwr:configuration>
```

Just like before, we're specifying that the `TrafficInfo` object be translated to JavaScript using a bean converter. Notice that the `<dwr:convert>` element must be placed within a `<dwr:configuration>` element.

16.6 Summary

Even though Spring provides its own very capable web framework, you may find another framework more to your liking. Fortunately, choosing to use Spring in your service and data access layers doesn't preclude the use of an MVC framework other than Spring MVC.

As you've seen in this chapter, Spring integrates into several prevalent MVC frameworks, including Struts, WebWork, Tapestry, and JavaServer Faces. As each of these frameworks is unique with respect to the others, the integration strategy is also different for each.

With Struts, you have two integration options. First, you can have your Struts actions become Spring aware, which provides a straightforward solution but couples your actions to Spring. Alternatively, you can have Struts delegate the handling of actions to Spring-managed beans, giving you a more loosely coupled solution, at the cost of a slightly more complex Struts configuration.

Integrating Spring into WebWork 2 couldn't be any easier. WebWork 2 embraces Spring as its dependency injection container, so wiring WebWork actions with Spring beans involves little more than configuring a `ContextLoaderListener` to make the Spring context available to WebWork. And because Struts 2 is really the same as WebWork 2, it's just as easy to integrate Spring with Struts 2.

Tapestry is a great component-based web framework that uses simple HTML templates. Integrating Spring into Tapestry 3 involves replacing Tapestry's base engine with a Spring-aware Tapestry engine. Integration with Tapestry 4, on the other hand, is much simpler—all you must do is add a JAR file from the Diaphragma project to your application's classpath and then refer to Spring beans with a special `spring:` prefix.

Accessing Spring-configured beans in a JSF application involves registering a `DelegatingVariableResolver`, a special JSF variable resolver that is Spring aware. Optionally, you may also expose the entire Spring application context as a JSF variable by using `WebApplicationContextVariableResolver`.

The web development world is all abuzz about Ajax, a means of creating highly dynamic web applications where web pages can dynamically interact with the server without performing a full page refresh. To wrap up this chapter, we looked at Direct Web Remoting (DWR), an Ajax toolkit that makes it possible to transparently invoke methods on Spring-managed beans through JavaScript.

So now you know how to develop web applications using Spring with a variety of web frameworks. You have the option of using Spring's MVC framework, or you can choose from one of several third-party frameworks. And you've also seen how to make your applications more dynamic with JavaScript remoting to Spring beans.

appendix A:
Setting up Spring

The Spring Framework and container is packaged in several JAR files. Spring is a library of classes that will be packaged with and used by your Spring-enabled applications. Installing Spring involves adding one or more JAR files to your application's classpath. It does not have an executable runtime. Therefore, Spring is more akin to a library like Jakarta Commons than an application server like JBoss.

How you make Spring available to your application's classpath will vary depending on how you build and run your application. You may choose to add the Spring JAR files to your system's classpath or to a project classpath in your favorite IDE. If you're building your application using Ant or Maven, be certain to include Spring in your build's dependencies so that it will be included in the project's target deployment.

Even though you happen to be holding one of the best books ever published about Spring (I'm not biased, though), there's plenty of additional materials in Spring's full distribution, including Spring's API documentation, examples, and the full source code for the Spring Framework. Therefore, the first thing you'll want to do is to download the full Spring distribution. Let's see where you can find Spring and what you'll get when you download it.

A.1 **Downloading Spring**

You can download the Spring distribution from Spring's website: <http://www.springframework.org>. Choose the downloads link from the left-hand menu and look for the Spring 2.0 download. As I'm writing this, I'm building the examples against version 2.0.6.

When downloading Spring, you have two choices: you can download a Spring distribution that comes with its own dependencies or you can download a distribution that contains only the Spring JAR files. Even though it's a much larger download, I favor the one that comes with dependencies so that I won't have to hunt down the other JAR files that my application needs.

A.1.1 **Exploring the Spring distribution**

Once you've downloaded the distribution, unzip it to a directory on your local machine. The Spring distribution is organized within the directory structure described in table A.1.

Several of these directories contain the Spring source code. The aspectj/, mock/, src/, and tiger/ directories contain the source code that makes up the Spring Framework itself. Meanwhile, the test/ directory contains the unit tests used to test Spring. Although it's not essential to using Spring, you may want to venture around in these directories to see how Spring does its stuff. The Spring

Table A.1 Spring's full distribution contains a wealth of materials, including API documentation, source code, and examples.

Distribution path	What it contains
/aspectj	Source files for Spring-specific aspects implemented in AspectJ. If you're curious about how the Spring Framework makes use of AspectJ, you'll want to look in here.
/dist	The Spring distribution JAR files. Here you'll find several JAR files, each of which represents a part of the Spring Framework, as described in table A.2 in the next section.
/docs	Spring's API and reference documentation.
/lib	Open source JAR files that Spring depends on, including Log4j, AspectJ, Jakarta Commons, Hibernate, and others. Fear not: Spring doesn't depend on all of these JAR files. Many of them only come into play when you exercise a specific set of Spring functionality.
/mock	The source code for Spring's mock object and unit-testing module.
/samples	Several example Spring applications.
/src	Source code for most of the Spring Framework.
/test	Source code for the unit tests that are used to test Spring itself.
/tiger	Source code for the portion of Spring that is specific to Java 5. This code is kept separate from the core of Spring to ensure that the rest of the framework is compatible with older versions of Java.

developers are extremely talented coders and you'll probably learn a little something by reviewing their code.

The docs/ directory contains two important pieces of documentation. The reference document is an overview of the entire Spring Framework and is a good complement to this book. Also, the JavaDocs for the entire Spring Framework can be found under docs/—you'll probably want to add this as a bookmark in your web browser, because you'll refer to it often.

The samples/ directory contains a handful of sample Spring applications. Of particular note are the petclinic and jpetstore examples. Both of these applications highlight many important elements of the Spring framework.

A.1.2 Building your classpath

The most important directory in the Spring distribution is probably the dist/ directory. That's because this is the directory containing the JAR files that you'll use to build Spring-based applications. Using Spring in your application involves choosing one or more JAR files from this directory and making them available in

your application's classpath. The JAR files and their purposes are described in table A.2.

Table A.2 The JAR files that make up the Spring Framework. The framework is broken into several modules, enabling you to pick and choose the parts of Spring that you need for your application.

JAR file	Directory	Purpose
spring.jar	dist/	Contains most of the modules of the Spring Framework in one convenient JAR file.
spring-aspects.jar	dist/	Spring's AspectJ-specific classes.
spring-mock.jar	dist/	Spring's mock objects and testing classes.
spring-aop.jar	dist/modules/	The Spring AOP module.
spring-beans.jar	dist/modules/	The Spring bean factory module.
spring-context.jar	dist/modules/	The Spring application context module. Includes JNDI, validation, scripting, Velocity, and FreeMarker libraries.
spring-core.jar	dist/modules/	Classes that are core to the Spring Framework.
spring-dao.jar	dist/modules/	The basis for Spring's DAO support.
spring-hibernate2.jar	dist/modules/	Spring's Hibernate 2 support.
spring-hibernate3.jar	dist/modules/	Spring's Hibernate 3 support.
spring-ibatis.jar	dist/modules/	Spring's iBatis support.
spring-jca.jar	dist/modules/	Spring's JCA support.
spring-jdbc.jar	dist/modules/	Spring's JDBC module.
spring-jdo.jar	dist/modules/	Spring's support for Java Data Objects (JDO).
spring-jms.jar	dist/modules/	Spring's support for JMS.
spring-jmx.jar	dist/modules/	Spring's JMX support.
spring-jpa.jar	dist/modules/	Spring's support for the Java Persistence API (JPA).
spring-portlet.jar	dist/modules/	Spring's portlet MVC framework.
spring-remoting.jar	dist/modules/	Spring's remoting module.
spring-struts.jar	dist/modules/	Spring's support for Struts and Tiles.
spring-support.jar	dist/modules/	Support and utility classes.
spring-tplink.jar	dist/modules/	Spring's support for Oracle TopLink.
spring-web.jar	dist/modules/	Spring's web application context and utilities.
spring-webmvc.jar	dist/modules/	Spring MVC.

Although the list of 24 JAR files may seem a bit daunting, sorting them out isn't all that difficult. The JAR files in the dist/modules/ directory allow you to choose which parts of Spring are needed for your application. To make things simple, however, you may want to take the lazy way out and simply add spring.jar (from the dist/ directory) to your application's classpath. The spring.jar file is a convenient, single-JAR equivalent to all of the JARs in the dist/modules/ directory.

You should know that not all Spring-related JAR files are available in the main Spring distribution. Acegi Security and Spring-WS, for example, are subprojects of Spring and have their own distributions available from their own websites. When it comes time to add these modules to your classpath, I'll be sure to tell you where to find them.

If dist/ is the most important directory then lib/ comes in a close second. You will likely need to add other JAR files to your classpath to enable certain Spring features. For example, if you intend to use Spring's support for AspectJ in your application, you'll need to add aspectjrt.jar and aspectjweaver.jar to the classpath. Likewise, Spring's Hibernate modules depend on an appropriate version of Hibernate being available in the classpath. If you downloaded the "with dependencies" distribution of Spring, you'll be able to find most dependencies you need in the lib/ directory.

Although many of the JAR files under the lib/ directory are optional, one is required. Spring relies on Jakarta Commons Logging to log informational and error messages. Therefore, commons-logging.jar (from lib/jakarta-commons) must always be in the classpath of any Spring-enabled application.

Once you've decided which JAR files your application needs, the next step is to add them to your classpath. The naive way of handling this would be to add them to your CLASSPATH system variable:

```
set CLASSPATH=%CLASSPATH%; {SPRING_HOME}/dist/spring.jar
set CLASSPATH=%CLASSPATH%; {SPRING_HOME}/lib/
    ↗ jakarta-commons/commons-logging.jar
```

This may work for quick-and-dirty experimentation. But you're probably going to use Spring to develop full-blown applications more often than one-off trials. You're likely going to let a build system like Maven or Ant construct your application distribution. In the following sections, we'll show you how to add Spring to your Maven and Ant builds.

A.2 Adding Spring as a Maven 2 dependency

If you're using Maven 2 to build and package your Spring-based application, you can let Maven download the Spring JAR files for you. With Maven 2, adding Spring to your build is a simple matter of adding dependency entries to your project's pom.xml file.

Spring and its modules are available in the Maven repository under the group ID org.springframework. Within this group are the modules listed in table A.3.

Table A.3 The Spring modules available in the Maven 2 repository. To include these in your application's build, you'll need to add a <dependency> entry to your pom.xml file.

Maven artifact ID	What it provides
spring	Almost all of the Spring Framework
spring-aop	The Spring AOP framework
spring-aspects	AspectJ aspect library, including the Spring-configured and AspectJ transactional aspects
spring-beans	The Spring bean factory and related classes
spring-context	The Spring application context and related classes
spring-core	The core Spring Framework
spring-dao	Spring's DAO abstraction framework
spring-hibernate2	Spring's support for Hibernate 2
spring-hibernate3	Spring's support for Hibernate 3
spring-ibatis	Spring's support for iBATIS
spring-jca	Spring's support for the Java Connector API
spring-jdbc	Spring's JDBC abstraction framework
spring-jdo	Spring's support for Java Data Objects
spring-jms	Spring's support for the Java Messaging API
spring-jmx	Spring's support for Java management extensions
spring-jpa	Spring's support for the Java Persistence API
spring-mock	Spring's unit-testing and mock-object extensions.
spring-obj	Spring's support for Apache's Object Relational Bridge
spring-portlet	Spring's Portlet MVC framework

Table A.3 The Spring modules available in the Maven 2 repository. To include these in your application's build, you'll need to add a <dependency> entry to your pom.xml file. (continued)

Maven artifact ID	What it provides
spring-remoting	Spring's remoting support
spring-struts	Spring-Struts integration
spring-support	Support and utility classes
spring-tplink	Spring's support for Oracle TopLink
spring-web	Spring's web container and related utility classes
spring-webmvc	Spring's web MVC framework

You may choose to add each module as a dependency as it's needed. For example, if your application depends only on the Spring AOP module, you might add the following <dependency> to your pom.xml file:

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-aop</artifactId>
  <version>2.0.3</version>
</dependency>
```

This dependency declaration will make Spring AOP available for building your application. In addition, the Spring Beans module will be made available thanks to Maven 2's support for transitive dependency resolution.

Although your application may start small and have only a small dependency on Spring, most applications grow to depend on the bulk of the Spring Framework. Therefore, it is often easier to declare a dependency on the entirety of Spring:

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring</artifactId>
  <version>2.0.3</version>
</dependency>
```

This dependency will make the bulk of the Spring Framework available to your application. Even if you're not using it all right away, there's little harm (and plenty of convenience) in using this dependency.

If your application will be using a third-party ORM solution or you will be building a portlet MVC application, you may need to add additional dependencies to

pom.xml. For example, if your application is using Hibernate 3 for persistence, you'll want to add the Spring Hibernate 3 module as follows:

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-hibernate3</artifactId>
  <version>2.0.3</version>
</dependency>
```

Likewise, you may want to take advantage of Spring's mock objects and testing support classes. If that's the case, add the following dependency to pom.xml:

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-mock</artifactId>
  <version>2.0.3</version>
  <scope>test</scope>
</dependency>
```

Notice that the `<scope>` of this dependency is `test`. This ensures that the Spring Mock module will only be available during testing and not packaged for distribution with your application.

If your application is using Spring MVC, you'll need to add Servlet API JAR file as a dependency:

```
<dependency>
  <groupId>javax.servlet</groupId>
  <artifactId> servlet-api</artifactId>
  <version>2.4</version>
  <scope>provided</scope>
</dependency>
```

In this case, the `<scope>` has been set to `provided`. This tells Maven to make the Servlet API available at compile time, but not to include it in a WAR packaging. The Servlet API is usually made available by the application server and should not be included in an application deployment.

Maven 2 has gained a lot of attention in the last year or so. But if you haven't made the switch to Maven 2, odds are that you're an Ant fan. Although I favor Maven 2 for my builds, I won't leave the Ant fanatics hanging. Let's see how to build a Spring-based application using Ant.

What about Maven 1?

Perhaps you're building your application with the original Maven. For the most part, dependency declaration isn't much different between Maven 1 and Maven 2. How Maven resolves those dependencies is slightly different, however. Maven 2 supports transitive dependency resolution, which means that if your application depends on Spring, Maven 2 is smart enough to know that it also depends on Jakarta Commons Logging without having to be told. With Maven 1, you may have to explicitly add Commons Logging as a dependency.

I believe that Maven 2 is far superior to Maven 1 in many ways, including transitive dependency resolution. I therefore recommend that you make the jump to Maven 2. If you need some help getting started with Maven 2, have a look at *Better Builds with Maven 2* (Mergere, 2006).

A.3 Spring and Ant

Although I prefer to use Maven 2 for my builds, many developers will want to use Apache Ant. If you're using Ant to build your Spring project, you'll need to download the Spring Framework for yourself (as described in section A.1) and add the Spring JAR files and their dependencies to the appropriate paths in your project's build.xml file.

I recommend declaring an Ant <path> element that will contain all of your application's dependencies, including the Spring JAR files. Listing A.1 shows a small section of an Ant build file that manages Spring dependencies in this way.

Listing A.1 Building a Spring application with Ant

```
<project name="MyProject" default="war">
  <property name="spring.home"
    location="/opt/spring-framework-2.0"/>  <!-- Defines Spring
                                                distribution location
  <property name="target.dir" location="target"/>
  <property name="classes.dir" location="${target.dir}/classes"/>
  <property name="src.dir" location="src"/>
  <property name="java.src.dir" location="${src.dir}/java"/>
  <property name="webapp.dir" location="${src.dir}/webapp"/>
  <property name="app.lib.dir" location="lib"/>
  <property name="spring.lib.dir"
    location="${spring.home}/dist"/>
  <property name="spring.depends.dir"
    location="${spring.home}/lib"/>

  <path id="dependency.path">
```

```

<fileset dir="${spring.lib.dir}" includes="*.jar"/>
<fileset dir="${spring.depends.dir}" includes="**/*.jar"/>
<fileset dir="${app.lib.dir}" includes="*.jar"/>
</path>

<target name="compile">
    <mkdir dir="${classes.dir}"/>
    <javac destdir="${classes.dir}"
          classpathref="dependency.path">      ←
        <src path="${java.src.dir}"/>
    </javac>
</target>

<target name="war" depends="compile">
    <war destfile="${target.dir}/${ant.project.name}.war"
        webxml="${webapp.dir}/web.xml">
        <lib dir="${spring.lib.dir}"/>      ←
        <lib dir="${app.lib.dir}"/>      ←
        <classes dir="${classes.dir}"/>
    </war>
</target>
...
</project>

```

Regardless of whether you choose Maven 2 or Ant (or some other build mechanism), you'll probably want to configure Log4j to properly handle Spring logging. Let's see how to set up Spring with Log4j.

A.4 Spring and Log4j

With the build file now in place, there is one final thing you will want to do. When you first start using Spring, one feature you'll almost certainly find useful is logging. First, be sure that the Log4j JAR file is in the application's classpath. If you're using Ant to do your build, the build.xml file in listing A.1 already includes Log4j (through the \${spring.lib.dir} path). If you're using Maven 2, simply add the following <dependency> to your pom.xml file:

```

<dependency>
    <groupId>log4j</groupId>
    <artifactId>log4j</artifactId>
    <version>1.2.13</version>
    <scope>compile</scope>
</dependency>

```

With Log4j in the application's classpath, the next thing to do is include a simple Log4j configuration file. Assuming the project structure defined in the build.xml

file in listing A.1, you will need to create a file called log4j.properties in /src/webapp/WEB-INF/classes. The following snippet of log4j.properties configures Log4j to log all of Spring's logging messages to the console:

```
log4j.appenders.stdout=org.apache.log4j.ConsoleAppender
log4j.appenders.stdout.layout=org.apache.log4j.PatternLayout
log4j.appenders.stdout.layout.ConversionPattern=%d %p %c - %m%n
log4j.rootLogger=INFO, stdout
log4j.logger.org.springframework=DEBUG
```

appendix B:
Testing with
(and without)
Spring

Software is precise in its execution. It does exactly what it is written to do—nothing more and nothing less. The software we write would function flawlessly if it weren't for one minor problem: we write it. Even though software perfectly executes the instructions it is given, flawed humans are the ones who write the instructions. Human flaws tend to leak into software instructions, resulting in quirks, snafus, and errors.

When you consider the serious consequences of software bugs, it becomes imperative that we test software as much as possible. Even though it's still possible for bugs to exist in tested code, the odds of a serious bug sneaking into production are greatly diminished by writing comprehensive test suites to exercise every nook and cranny of an application.

In this appendix, we're going to have a look at where Spring fits within the testing picture. You'll see how Spring indirectly supports testing by promoting loose coupling. We'll also explore a handful of Spring classes that you can use to test Spring MVC applications and transactional code.

B.1 ***Introduction to testing***

On the surface, software testing seems to be about ensuring the correctness and quality of software. But as we've already implied, it's ensuring that we as developers are giving the correct instructions to the software. If we're to blame for the mistakes that creep into software, isn't it also our responsibility to write tests that mitigate the chances that someone will see those mistakes?

Not so long ago (for some of you this may sound like an average workday), testing was an activity that was magically crammed into the last few days prior to pushing software into production. And testing was usually done by people who were labeled as “Quality Assurance.” A huge problem with this approach is that people who had no direct control over the instructions given to the software were allowed virtually no time to verify that it was correct. To say that QA is able to assure quality is hoping against hope.

Although developers have long been encouraged to test their own code, many developers dismissed testing as not being their job. Others performed only minimal smoke tests—if the compiler didn't complain about errors, there must not be any!

Agile methodologies such as extreme programming and Scrum have brought developer-written tests to the forefront. And frameworks such as JUnit have put the power to write repeatable and automated tests into the hands of developers. Test-infected developers are now writing better and more correct code than ever before.

That's not to say that QA is obsolete. On the contrary, QA people are trained to think of clever ways to break software, whereas the developer mind-set is focused on clever ways to make software work. I'm only saying that developers should step up to the plate and accept the duty of ensuring the correctness of their code—and for making the lives of QA staff a little easier.

With that said, let's look at different ways that developers can test their code and see how Spring fits into developer-written tests.

B.1.1 **Understanding different types of testing**

Testing comes in various forms. In fact, depending on which testing expert you ask, you may find that there are several dozens of types of tests. Table B.1 lists just a few of the most common types of tests.

Table B.1 Software testing comes in several flavors.

Test type	Purpose
Unit	Tests a logical unit of work (class, method, etc.) in isolation
Integration	Tests two or more units of an application together
Functional	Tests a specific function of an application, end-to-end, involving two or more units of an application
System	Tests an application's functionality through the same interfaces used by the end users of the application
System-Integration	Tests the interactions between two or more collaborating applications
Performance	Tests the performance of an application or system in terms of throughput, load, memory footprint, etc.

There are many more types of tests in addition to those listed in table B.1. But for the purposes of discussing how Spring fits into the testing landscape, we're going to focus on the first two types of tests listed: unit and integration tests.

Both of these types of testing can be automated using JUnit (<http://www.junit.org>), a popular open source testing framework. Before we go into Spring's role in testing, let's set the stage by going through a quick introduction to JUnit.

B.1.2 **Using JUnit**

JUnit is a testing framework created by Kent Beck (of extreme programming fame) and Erich Gamma (coauthor of the “Gang of Four” *Design Patterns: Elements*

of *Reusable Object-Oriented Software* book [Addison-Wesley Professional, 1995]). It is probably the best-known testing framework of all time. Although JUnit was developed with unit testing in mind, several extensions to JUnit have emerged to enable other types of testing.

Creating tests in JUnit is simply a matter of extending the `junit.framework.TestCase` class and implementing one or more test methods. A test method is any method whose name starts with `test`. Within each test method, one or more assertions are made. If any of the assertions fail, the test fails.

The following is a simple example of a JUnit test case that tests certain mathematic functions:

```
import junit.framework.TestCase;  
  
public class MathTest extends TestCase {  
    public MathTest() {}  
  
    public void testAddition() {  
        assertEquals(4, 2+2);  
        assertEquals(99, 33+66);  
    }  
  
    public void testMultiplication() {  
        assertEquals(0, 5*0);  
        assertEquals(-6, 2 * -3);  
        assertEquals(625, 25 * 25);  
    }  
}
```

Although extremely mindless, this is a valid JUnit test case. When run within a JUnit runner, this test can prove (or disprove) the correctness of the mathematic operators. For example, figure B.1 is a screenshot showing the results of running `MathTest` within the JUnit view in Eclipse.

It may be hard to see when printed in grayscale, but there's a green bar in the user interface. That green bar indicates that all tests have passed. The mantra of JUnit testers is, "If the bar's green, the code's clean."

On the other hand, if the bar's red then a test failed. To illustrate, here's a test method that is guaranteed to fail:

```
public void testThatFails() {  
    assertEquals("One", new Integer(1));  
}
```

This test fails because the `String` value of "One" is not equal to the `Integer` value of 1. Figure B.2 shows the results when this test method is run in Eclipse's JUnit view.

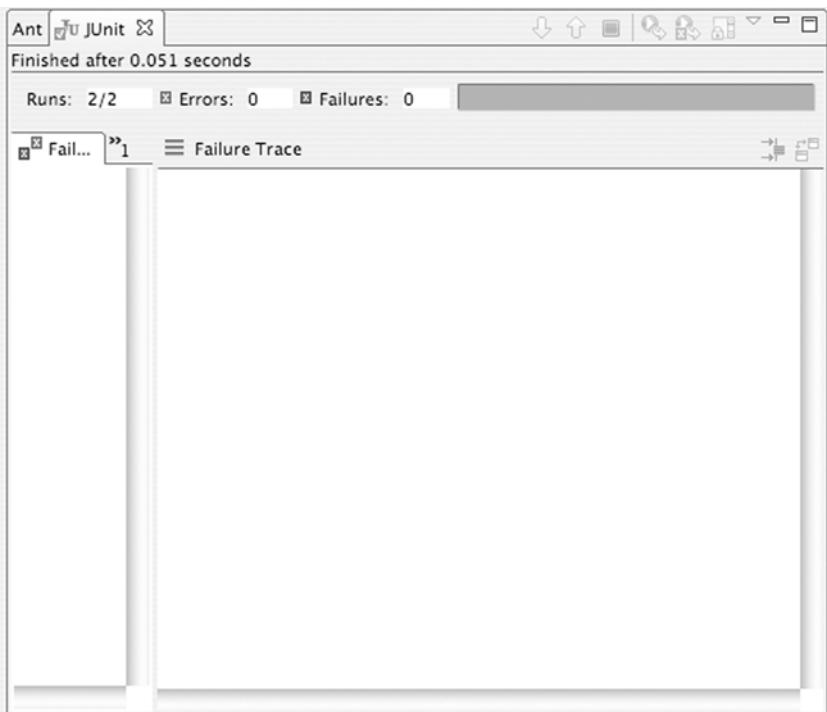


Figure B.1 A green bar indicates that all tests have run successfully in JUnit. The code's clean!

Again, it may not be apparent from the grayscale image in figure B.2, but a failed test results in a red bar. The red bar tells you that something is wrong in the code and that you have a problem to fix. The failure's stack trace hints as to what may have gone wrong.

Setting up tests

It's often important that some test data and collaborators be prepared in advance of the actual test. While it's quite possible to perform such setup at the beginning of each test method, you may find yourself repeating the setup code across multiple test methods in the same test case.

To consolidate test setup into one location, you should override `TestCase`'s `setUp()` method:

```
public void setUp() throws Exception {  
    // perform setup code  
}
```

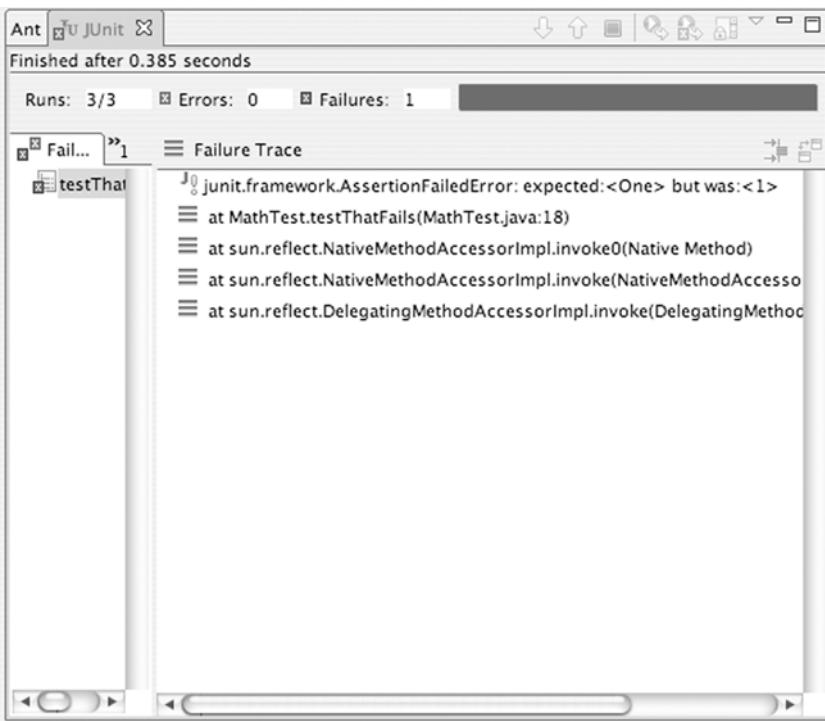


Figure B.2 Oops! A red bar indicates that one or more test methods have failed. In this case "One" is apparently not equal to 1.

It's vital to understand that `setUp()` is called just before each test method is invoked. If your test case has 10 test methods then `setUp()` will be called 10 times in the course of the test case. Depending on what your test setup involves, this could have an impact on how long it takes to run your test case.

Tearing down tests

An important tenet of testing is that each test be run independently of the others. Each test must leave its environment in a consistent state after the test is run. This means that if a test method were to alter some data (a file, database, etc.) that is shared among multiple tests, that data should be returned to its initial state upon completion of a test.

Again, it's possible to have teardown code at the end of each test method. But, as with setup code, doing so will often result in duplication of the teardown code

across multiple test methods. Just as `TestCase` offers `setUp()` for performing test setup, `TestCase` also provides `tearDown()` for performing test cleanup.

To implement test teardown in your test case, override the `tearDown()` method:

```
public void tearDown() throws Exception {  
    // perform teardown code  
}
```

where `setUp()` is called just before each test method and `tearDown()` is invoked just after each test method completes. This guarantees that any cleanup work is done between test methods so that each test method starts in a clean, untainted state.

Now you've been given a crash course in the basics of writing JUnit test cases and should be ready to start unit-testing your Spring applications. Before we move on, I'd like to mention that there's much more to JUnit than has been presented in this section. For more information on JUnit, I highly recommend Manning's *JUnit in Action* (2003) and *JUnit Recipes* (2004).

B.1.3 Spring's role in testing

I'm often asked how Spring is used when testing. The easy answer is that Spring usually isn't directly involved in testing. Applications that are developed to take advantage of Spring's dependency injection are made up of loosely coupled objects, which are, by nature, easier to test. But Spring generally isn't used within tests.

But I did say that was the easy answer. The truth is that there are a few places where Spring can participate directly in tests:

- *Unit-testing Spring MVC*—Spring comes with an extension to JUnit's `TestCase` that exposes a handful of convenient assertion methods for verifying the values returned in a `ModelAndView` object. In addition, Spring provides several out-of-the-box mock implementations such as `MockHttpServletRequest` and `MockHttpServletResponse` that are useful for testing controllers.
- *Integration-testing Spring applications*—Spring also comes with a `TestCase` extension that automatically loads a Spring application context for integration tests. It ensures that the application context is loaded once and only once for all test methods.
- *Transactional testing*—Another `TestCase` extension starts a transaction before each test method and rolls it back after the test is complete. This is

useful when you’re testing code that is writing to a database because it ensures that the database is left in a consistent state between test methods.

The rest of this appendix examines the ways that Spring can help out with testing. Let’s start by writing some unit tests for the RoadRantz controllers.

B.2 Unit-testing Spring MVC controllers

The web layer of an application is often considered one of the most difficult pieces of an application to test. It’s true that determining what goes into a request and what is rendered in a browser presents some interesting challenges in unit testing. JUnit extensions such as Cactus and HttpUnit make this challenge a little less daunting. Nevertheless, the web layer is often left as one of the most untested parts of many applications.

But it doesn’t have to be that way. Take a moment to think about what a Spring MVC controller class does. Forget that controllers have anything to do with the Web. In simple terms, a controller takes input (in the form of an `HttpServletRequest`) and produces output (in the form of a `ModelAndView`). It doesn’t render an HTML page—that’s the job of the view layer. In this light, controllers are no different than any other Java class—and thus shouldn’t be any more difficult to test.

To illustrate, let’s write a basic unit test for `RantsForVehicleController`. The functionality desired from this controller is that it should receive a vehicle’s state and plate number in the request and return a `ModelAndView` populated with a list of rants. Also, if the request URI ends with `.htm` then the view name returned should be `rantsForDay`.

`RantsForVehicleControllerTest` (listing B.1) shows a JUnit test case that might be used to test `RantsForVehicleController`.

Listing B.1 Testing RantsForVehicleController

```
package com.roadrantz.mvc;
import java.util.Iterator;
import java.util.List;
import java.util.Map;
import javax.servlet.http.HttpServletRequest;
import junit.framework.TestCase;
import org.springframework.mock.web.MockHttpServletRequest;
import org.springframework.mock.web.MockHttpServletResponse;
import org.springframework.web.servlet.ModelAndView;
import com.roadrantz.domain.Rant;
import com.roadrantz.domain.Vehicle;

public class RantsForVehicleControllerTest extends TestCase {
```

```

private static final String TEST_STATE = "TX";
private static final String TEST_PLATE_NUMBER = "ABC123";

private RantsForVehicleController controller;

public RantsForVehicleControllerTest() {}

protected void setUp() throws Exception {
    controller = new RantsForVehicleController();
    controller.setCommandClass(Vehicle.class);
} | Sets up controller  
to be tested

protected void tearDown() throws Exception {
    super.tearDown();
}

public void testSimpleCase() throws Exception {
    HttpServletRequest request = ...;
    HttpServletResponse response = ...;

    ModelAndView modelAndView =
        controller.handleRequest(request, response); | Tests handleRequest()

    assertEquals("ModelAndView should not be null", modelAndView);
    assertEquals("View name should be 'vehicleRants'", "vehicleRants", modelAndView.getViewName()); | Views  
assertions

    Map model = modelAndView.getModel();
    assertTrue("Model should contain 'rants' key",
        model.containsKey("rants")); | Models assertions

    List rants = (List) model.get("rants");
    assertNotNull("Model element 'rants' should not be null", rants);

    assertEquals("Model element 'rants' should contain 2 items",
        2, rants.size());

    for (Iterator iter = rants.iterator(); iter.hasNext();) {
        Rant rant = (Rant) iter.next();
        assertEquals("Rant's Vehicle state is incorrect",
            TEST_STATE, rant.getVehicle().getState());
        assertEquals("Rant's Vehicle plateNumber is incorrect",
            TEST_PLATE_NUMBER, rant.getVehicle().getPlateNumber());
    }
}
}

```

In short, the test case in listing B.1 passes in an `HttpServletRequest` and an `HttpServletResponse` to the controller and expects to receive a `ModelAndView` containing certain data. Even though `RantsForVehicleController` is a command controller, we test via the `Controller` interface's `handleRequest()` method so

that we're testing at the basic level of functionality—the same level at which DispatcherServlet will invoke the controller.

But the test case in listing B.1 isn't complete. Some unanswered questions remain about the origin of HttpServletRequest and HttpServletResponse that are passed to handleRequest() (as well as what information is contained in the request). Also, doesn't RantsForVehicleController depend on a RantService to retrieve the rants? Where is that dependency set?

The answer to all of these questions: mock objects.

B.2.1 Mocking objects

When DispatcherServlet invokes the handleRequest() method of RantsForVehicleController, it passes in an instance of HttpServletRequest that is populated with everything that handleRequest() needs to perform its work (or at least as much information as was given it by the servlet container). But when you're unit-testing a controller, the request doesn't come from DispatcherServlet because ultimately the request doesn't originate in a web browser. The request must originate from the test case itself. You need a way to create a request to pass on the call to handleRequest().

We could take the time to write our own implementation of the HttpServletRequest interface. But HttpServletRequest requires so many methods to be implemented, many of which play no purpose in our test. It would be a lot of work to create a suitable HttpServletRequest for our test case.

Fortunately, Spring provides a mock implementation of HttpServletRequest for us to use: org.springframework.mock.web.MockHttpServletRequest. Let's revisit RantsForVehicleController and replace the declaration of the HttpServletRequest object in testSimpleCase() with the following code:

```
MockHttpServletRequest request = new MockHttpServletRequest();
request.setMethod("POST");
request.addParameter("state", TEST_STATE);
request.addParameter("plateNumber", TEST_PLATE_NUMBER);
request.setRequestURI(
    "http://localhost:8080/roadrantz/rantsForVehicle.htm" );
```

Now we have an HttpServletRequest instance to pass to handleRequest(). What's more, we are able to use some of the convenience methods of MockHttpServletRequest to populate the request with the information needed by RantsForVehicleController.

That addresses the request parameter. But what about the response parameter? No worries—Spring also gives us MockHttpServletResponse to accommodate

the other parameter of `handleRequest()`. Replace the `HttpServletResponse` declaration with the following line:

```
HttpServletResponse response = new MockHttpServletResponse();
```

Our test requires no special setup of the response object, so this one line is sufficient to set up the call to `handleRequest()`.

Mocking interfaces with EasyMock

At this point we're able to call `handleRequest()` in our test method. But when we do, a `NullPointerException` will be thrown from `handleRequest()` because internally a `RantService` is used to retrieve the list of rants. When the application is running in the Spring container, Spring will inject the `rantService` property of `RantsForVehicleController` with a `RantService` instance. But our unit test doesn't run in the Spring container. How can we ensure that a `RantService` has been injected into the `rantService` property?

What we need is a mock implementation of the `RantService` interface. Unlike the case of `HttpServletRequest` and `HttpServletResponse`, where Spring provided mock implementations for us, Spring didn't anticipate our need for a `MockRantService` class. So, we must build a mock implementation ourselves.

To make mocking easier, we'll use EasyMock (<http://www.easymock.org>).¹ EasyMock is a tool that is able to generate mock implementations of interfaces on the fly in a unit test. Here's how it works:

- 1 You make EasyMock available by using Java 5's static import to import `org.easymock.EasyMock.*` (i.e., all of the static methods of EasyMock).
- 2 You ask for a mock object. It obliges by returning an object that implements the desired interface (`RantService` in our case).
- 3 You write code to "train" the mock object. This involves making calls to the mock object (as if it were the real object) and telling EasyMock what values to return and what (if any) exceptions to throw.
- 4 You then tell the EasyMock that you're ready to replay the calls and begin your test.

In our test case, we know that `RantsForVehicleController` will call the `getRantsForVehicle()` method on the `RantService` to retrieve a list of rants

¹ Note that we're using EasyMock 2.0, which takes advantage of Java 5's support for static imports. If you're using an older version of Java, you'll need to use an older version of EasyMock that doesn't use static imports.

belonging to that vehicle. Thus, the mock implementation of RantService needs to be able to return a list of rants with the expected values. The `setUp()` method in listing B.2 creates and trains the mock object to do just that.

Listing B.2 Training a mock RantService

```
protected void setUp() throws Exception {
    controller = new RantsForVehicleController();
    controller.setCommandClass(Vehicle.class);

    RantService rantService =           | Creates mock
                                         | object
                                         |
    createMock(RantService.class);      | Sets up
                                         | test data
                                         |
    testVehicle = new Vehicle();
    testVehicle.setState(TEST_STATE);
    testVehicle.setPlateNumber(TEST_PLATE_NUMBER);
    List<Rant> expectedRants = new ArrayList<Rant>();
    Rant rant = new Rant();
    rant.setVehicle(testVehicle);
    rant.setRantText("Rant 1");
    expectedRants.add(rant);
    rant = new Rant();
    rant.setVehicle(testVehicle);
    rant.setRantText("Rant 2");
    expectedRants.add(rant);

    expect(rantService.getRantsForVehicle(testVehicle)).   | Trains mock
        andReturn(expectedRants);                         | object
                                         |
    replay(rantService);                         | Injects mock object
    controller.setRantService(rantService);       | into controller
}
                                         |
                                         |
```

After creating the mock RantService, the `setUp()` method assembles a list of rants that we expect to be returned when `getRantsForVehicle()` is called on the mock object.

Next, the `getRantsForVehicle()` method of the RantService is invoked, passing in the test vehicle. Since this is just a training exercise, the value returned from `getRantsForVehicle()` is unimportant. In fact, the mock object doesn't even know what to return until we tell it by calling the `andReturn()` method.

Once the mock object has been trained, we're ready to start our test. A call to the `replay()` method tells EasyMock to start expecting calls on the mock object and to respond as it was trained.

The last line of `setUp()` injects the mock RantService into the `rantService` property of the controller. `RantsForVehicleController` is now ready to test.

B.2.2 Asserting the contents of ModelAndView

The `testSimpleCase()` method is a good start on testing `RantsForVehicleController`. But it is quite complex with all of the assertions that it makes. It'd be nice if there were a way to streamline those assertions to only a handful that are truly meaningful.

Although the Spring container generally shouldn't be involved in unit testing, Spring does provide some help for unit-testing controllers in its API. The `org.springframework.test.web.AbstractModelAndViewTests` class is an extension of JUnit's `TestCase` class that exposes some convenient assertion methods (table B.2) for dealing with the contents of a `ModelAndView` object.

For illustration, let's adapt `RantsForVehicleControllerTest` to take advantage of `AbstractModelAndViewTests`'s assertion methods. First, our test case should be changed to extend `AbstractModelAndViewTests`:

Table B.2 The assertion methods available in `AbstractModelAndViewTests`.

Assertion method	Purpose
<code>assertAndReturnModelAttributeOfType(ModelAndView mav, Object key, Class type)</code>	Asserts that a model attribute exists at the given key and that it is of a certain type
<code>assertCompareListModelAttribute(ModelAndView mav, Object key, List assertionList)</code>	Asserts that a <code>List</code> model attribute exists at the given key and that it contains the same values (and order) as a given assertion list
<code>assertModelAttributeAvailable(ModelAndView mav, Object key)</code>	Asserts that a model attribute exists at a given key
<code>assertModelAttributeValue(ModelAndView mav, Object key, Object value)</code>	Asserts that the model attribute at a given key is equal to a certain value
<code>assertModelAttributeValues(ModelAndView mav, Map assertionModel)</code>	Asserts that all model attributes match values with a given assertion model <code>Map</code>
<code>assertSortAndCompareListModelAttribute(ModelAndView mav, Object key, List assertionList, Comparator comp)</code>	Same as <code>assertCompareListModelAttribute()</code> except that both lists are sorted before comparing the lists.
<code>assertViewName(ModelAndView mav, name)</code>	Asserts that the given <code>ModelAndView</code> contains a specific view name.

```
public class RantsForVehicleControllerTest
    extends AbstractModelAndViewTests {
    ...
}
```

Now we're ready to rewrite `testSimpleCase()` to use a few of the assertion methods to test the contents of `ModelAndView`:

```
public void testSimpleCase() throws Exception {
    MockHttpServletRequest request = new MockHttpServletRequest();
    request.setMethod("POST");
    request.addParameter("state", TEST_STATE);
    request.addParameter("plateNumber", TEST_PLATE_NUMBER);
    request.setRequestURI(
        "http://localhost:8080/roadrantz/rantsForVehicle.htm");
    HttpServletResponse response = new MockHttpServletResponse();

    ModelAndView modelAndView =
        controller.handleRequest(request, response);

    List<Rant> expectedRants = new ArrayList<Rant>();
    Rant rant = new Rant();
    rant.setId(1);
    Vehicle vehicle = new Vehicle();
    vehicle.setState("TX");
    vehicle.setPlateNumber("ABC123");
    rant.setVehicle(vehicle);
    rant.setRantText("This is a test rant");

    assertNotNull("ModelAndView should not be null", modelAndView);
    assertEquals("vehicleRants", modelAndView.getViewName());
    assertTrue("rants attribute available", modelAndView.getAttribute("rants") != null);
    assertEquals("rants", modelAndView.getAttribute("rants"));
}
```

This new version of `testSimpleCase()` is greatly simplified from the original. Where there were originally seven assertions (two of which occurred in a loop), there are now only four assertions (and no loop). Despite the fact that there are only half as many assertions, the same things are still being tested:

- We still know that the returned `ModelAndView` is not null.
- We still know that the view name is `vehicleRants`.
- We still know that the `rants` attribute is in the model.
- We still know that the `rants` attribute contains a list of the expected rants.

When unit-testing your application objects in isolation, you'll have little need to wire up all of the objects in your application. Although Spring provides some assistance in writing unit tests, the Spring container stays out of the way.

But that's unit testing. Test cases that test all of your application objects in concert are still a valid form of testing. For those kinds of tests, the Spring container should be involved to wire your application objects together. When you're writing those kinds of tests, Spring offers some additional help in the form of JUnit extensions for loading the Spring context. So, let's switch gears from unit testing and look into the support that Spring provides for integration testing.

B.3 **Integration testing with Spring**

Unit-testing the individual pieces of an application is a good start toward verifying the correctness of software. But it's just the beginning. Throughout this book, you've seen how to use dependency injection and Spring to tie objects together to build applications. Once you know that each of these objects is working, it's time to test the aggregation of those objects to be sure that they work together. After unit testing, the next phase of testing is integration testing—testing several units of an application in combination.

To find an opportunity for integration testing, look no further than the RoadRantz application. `RantsForVehicleController` depends on a `RantService` to do its work. In `RantsForVehicleControllerTest`, a mock `RantService` object was injected into the controller in the test's `setUp()` method. But when performing integration tests, it is desirable to wire up application objects just like they'll be wired up in production (or as close as is possible).

It's quite common to write integration test cases for a Spring application with a `setUp()` method that resembles the following:

```
public void setUp throws Exception {  
    applicationContext = new FileSystemXmlApplicationContext(  
        "roadrantz-service.xml",  
        "roadrantz-data.xml");  
}
```

This loads up the Spring application context and sets it to an instance variable for convenient access. Then, as part of the test methods, the `applicationContext` is used to retrieve one or more beans needed to perform the test. For example, `testGetRantsForVehicle()` uses `applicationContext` to retrieve the `rantService` bean:

```
public void testGetRantsForVehicle() {
    RantService rantService =
        (RantService) applicationContext.getBean("rantService");

    Vehicle testVehicle = new Vehicle();
    testVehicle.setState(TEST_STATE);
    testVehicle.setPlateNumber(TEST_PLATE_NUMBER);

    List<Rant> rants =
        rantService.getRantsForVehicle(testVehicle);

    assertEquals(2, rants.size());

    for(Iterator iter = rants.iterator(); iter.hasNext(); ) {
        Vehicle vehicle = (Vehicle) iter.next()
        assertEquals(TEST_STATE, vehicle.getState());
        assertEquals(TEST_PLATE_NUMBER, vehicle.getPlateNumber());
    }
}
```

That's fine, but it poses a small problem: the nature of `setUp()` and `tearDown()` is that they're respectively called just before and just after each test method. This means that the Spring application context is completely reloaded for every test method in the test case. In a sizable application made up of several objects, this setup could take some time. Over the course of several test methods, it adds up and the test case is slow.

Typically, the beans contained in a Spring application context are stateless. This means that they can't be tainted by running a test method and that they're reusable across multiple test methods. So, there's usually no reason to reload the Spring application context between test methods. In fact, the performance hit usually makes it undesirable to reload the context between test methods.

In support of Spring-based integration testing, Spring provides a handful of test case base classes that handle loading of an application context, ensuring that the context is loaded once and only once:

- `AbstractDependencyInjectionSpringContextTests`
- `AbstractTransactionalSpringContextTests`
- `AbstractTransactionalDataSourceSpringContextTests`

All of these test case classes are found in the `org.springframework.test` package.

Let's get started with Spring-based integration testing by looking at the simplest of these, `AbstractDependencyInjectionSpringContextTests`.

B.3.1 Testing wired objects

Making the job of loading a Spring application context within a test case easier, Spring comes with `AbstractDependencyInjectionSpringContextTests` (see figure B.3). Aside from having one of the longest names of all of the Spring classes,² this class is an extension of JUnit's `TestCase` that manages the loading and reloading of Spring application contexts for a test case. It makes the Spring application context available to your test case classes so that it can retrieve beans that are to be tested.

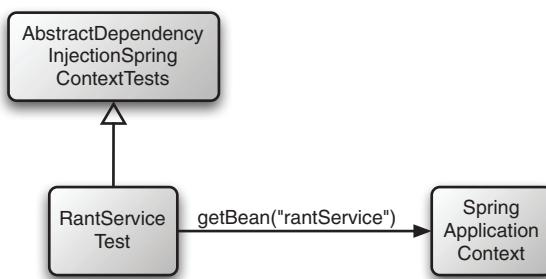


Figure B.3 `AbstractDependencyInjectionSpringContextTests` is a convenient base class for test cases that loads a Spring application context and makes it available to the test case for accessing beans.

To load Spring application contexts in your unit test cases, your test cases should extend `AbstractDependencyInjectionSpringContextTests` instead of `TestCase`. In addition, you should override the `getConfigLocations()` method to define the location(s) of the Spring configuration XML file(s).

For example, consider `AddRantTest` (listing B.3). This test case tests the `addRant()` method of `RantService`. Instead of testing `addRant()` in isolation using a mock DAO, this test allows the Spring application context to be loaded and its beans wired in a way that resembles how the production beans will be wired. In fact, as it is using the real `roadrantz-data.xml`, it will also use a real database.

Listing B.3 Testing `RantService` in the context of the entire `RoadRantz` application

```

package com.roadrantz.service;
import org.springframework.test.*;
import AbstractDependencyInjectionSpringContextTests;
import com.roadrantz.domain.Rant;
  
```

² It's not the longest, though. Before this appendix is finished, you'll see an even longer class name.

```

import com.roaddrantz.domain.Vehicle;
import com.roaddrantz.service.RantService;

public class RantServiceTest
    extends AbstractDependencyInjectionSpringContextTests {
    public RantServiceTest() {}

    protected String[] getConfigLocations() {
        return new String[] {
            "file:src/main/webapp/WEB-INF/roaddrantz-services.xml",
            "file:src/main/webapp/WEB-INF/roaddrantz-data.xml"
        };
    }

    public void testAddRant() throws Exception {
        RantService rantService =
            (RantService) applicationContext.getBean("rantService");
        Rant newRant = new Rant();
        newRant.setRantText("TEST RANT TEXT");
        Vehicle vehicle = new Vehicle();
        vehicle.setPlateNumber("ABC123");
        vehicle.setState("TX");
        newRant.setVehicle(vehicle);
        rantService.addRant(newRant);           ↘ Adds test rant
        List<Rant> rants = rantService.getRantsForVehicle(vehicle);
        assertTrue(rants.contains(newRant));
    }
}

```

Defines Spring context

Gets rantService from Spring

Asserts test rant was added

`AbstractDependencyInjectionSpringContextTests` supports this test case in two ways. First, it automatically loads up the Spring application context and exposes the context through the `applicationContext` instance variable. `testAddRant()` uses the `applicationContext` variable to retrieve the `rantService` bean from the context.

Perhaps more importantly, the application context is loaded once and only once, regardless of how many test methods there are. As shown, `RantServiceTest` only has a single test method. But even if it had 100 test methods, the Spring application context would only be loaded once, improving the overall performance of the test case.

Forcing a context reload

Although it's unusual, you may encounter a situation where your test taints the Spring application context. Since it's a best practice to leave the test in a consistent state between test methods, you'll need a way to force a reload of the Spring context.

Normally, `AbstractDependencyInjectionSpringContextTests` loads the Spring context only once. But you can force it to reload by calling `setDirty()` within your test method. When the test method completes, `AbstractDependencyInjectionSpringContextTests` will see that the Spring context is dirty and will reload a clean context for the next test method to use.

Setting up and tearing down dependency injected tests

You may have noticed that the `setUp()` method is now gone from `RantServiceTest`. That's because there's no longer a need for it, as `AbstractDependencyInjectionSpringContextTests` handles the loading of the Spring context for us. But in the event that you need a `setUp()` or a `tearDown()` method in your test, you should be aware that those methods have been made private so that you will not be able to override them.

The reason why they've been made private is because `AbstractDependencyInjectionSpringContextTests` uses those methods to load and unload the Spring context. If you were to override them, you might accidentally forget to call `super.setUp()` and `super.tearDown()` and the application context would not get loaded.

For your own setup and teardown needs, you must override `onSetUp()` and `onTearDown()` instead:

```
public void onSetUp() throws Exception {  
    // perform setup work  
}  
public void onTearDown() throws Exception {  
    // perform teardown work  
}
```

Aside from the name change, these methods effectively serve the same purpose as `TestCase`'s `setUp()` and `tearDown()` methods.

`AbstractDependencyInjectionSpringContextTests` is great for performing integration tests against objects wired together in a Spring application context. But it only covers simple nontransactional cases. When integration tests involve transactional objects, you'll want to look at Spring's other unit-test case classes such as `AbstractTransactionalSpringContextTests`, which we'll look at next.

B.3.2 Integration-testing transactional objects

As it appears in listing B.3, `RantServiceTest` does a fine job of testing that `RantService`'s `addRant()` actually adds a rant to the database. But it also has one undesirable side effect: it actually adds a rant to the database!

But isn't that what we want `addRant()` to do? Yes, but when testing we don't want that rant to be permanently added to the database. When the test is over, we need the database to return to a known state, ready for the next test. We need the test to roll back the database insert when it's over.

In addition to `AbstractDependencyInjectionSpringContextTests`, Spring also comes with `AbstractTransactionalSpringContextTests` to address the problem of rolling back transactions at the end of a test. If your test cases extend `AbstractTransactionalSpringContextTests`, a transaction will automatically be started at the beginning of every test method and automatically rolled back at the end. That leaves your test methods free to do whatever they want to the database, knowing that the work will be undone when the test method completes.³

To take advantage of `AbstractTransactionalSpringContextTests`, let's change `RantServiceTest` to subclass `AbstractTransactionalSpringContextTests`. Listing B.4 shows the new `RantServiceTest`.

Listing B.4 Testing `addRant()` in a transaction

```
package com.roaddrantz.service;
import java.util.List;
import org.springframework.test.
    ↗ AbstractTransactionalSpringContextTests;
import org.springframework.transaction.PlatformTransactionManager;
import com.roaddrantz.domain.Rant;
import com.roaddrantz.domain.Vehicle;
import com.roaddrantz.service.RantService;

public class RantServiceTest
    extends AbstractTransactionalSpringContextTests { ←
    public RantServiceTest() {}

    protected String[] getConfigLocations() {
        return new String[] {
            "file:src/main/webapp/WEB-INF/roaddrantz-services.xml",
            "file:src/main/webapp/WEB-INF/roaddrantz-data.xml"
        };
    }

    public void testAddRant() throws Exception {
        RantService rantService =
            (RantService) applicationContext.getBean("rantService");

        Rant newRant = new Rant();
```

Performs
transactional
test

³ All of this assumes that the underlying database supports transactions. If you're using MySQL MyISAM tables, for example, your database tables do not support transactions and thus `AbstractTransactionalSpringContextTests` won't be able to apply transactions to the tests.

```
newRant.setRantText("TEST RANT TEXT");
Vehicle vehicle = new Vehicle();
vehicle.setPlateNumber("FOOBAR");
vehicle.setState("TX");
newRant.setVehicle(vehicle);

rantService.addRant(newRant);

List<Rant> rants = rantService.getRantsForVehicle(vehicle);
assertTrue(rants.contains(newRant));
}
}
```

Not much has changed in this version of `RantServiceTest`. In fact, the only difference that you'll find is that `RantServiceTest` now extends `AbstractTransactionalSpringContextTests`. A lot changes, however, when you run the test. Now no rows are permanently added to the database. Within the context of the test, a row is added for the rant and another row is added for the vehicle (if the vehicle didn't already exist). But once the test is finished, the changes are rolled back and there will be no trace of this test having been run.

To accommodate the `AbstractTransactionalSpringContextTests` use of transactions, the application context will need to have a transaction manager bean configured (see chapter 6, section 6.2, for information on Spring's transaction managers). `AbstractTransactionalSpringContextTests` will look for a transaction manager in the Spring context and use it to create and roll back transactions. The `RoadRantz` application already has a transaction manager configured in `roadrantz-data.xml`, so `RantServiceTest` is ready to go.

Committing the test's changes

Although it's unusual, you may want to commit the changes done in a test. If so then you may simply call the `setComplete()` method within your test method. When the test method completes, the transaction will be committed instead of rolled back. Also, if you'd like to commit the changes before the test method completes, you may call `endTransaction()` after calling `setComplete()` to end the transaction and commit the changes immediately.

Be aware of the consequences of `setComplete()`. That is, any changes made within the test method will be committed after the test completes (or after `endTransaction()` is called). This means that you'll need to perform some extra work to clean up the data later or your database will be cluttered with remnants of old tests. This leftover data may get in the way of future tests and make it difficult for your tests to be repeatable.

Setting up and tearing down transactional tests

When working with transactional tests, the notions of setting up before a test and tearing down after a test take on a slight twist. Do you want your setup to occur before or after the transaction starts? Should teardown occur inside the transaction or after the transaction has ended? A single pair of setup and teardown methods is no longer flexible enough.

Therefore, `AbstractTransactionalSpringContextTests` provides two setup methods and two teardown methods:

- `onSetUpBeforeTransaction()`—Called before the transaction starts
- `onSetUpInTransaction()`—Called after the transaction starts
- `onTearDownInTransaction()`—Called just before the transaction is rolled back (or committed)
- `onTearDownAfterTransaction()`—Called after the transaction has been rolled back (or committed)

Just like `setUp()` and `tearDown()` in JUnit’s `TestCase` class or `onSetUp()` and `onTearDown()` in Spring’s `AbstractDependencyInjectionSpringContextTests`, you can define the setup and teardown behavior of your test cases by overriding one or more of these methods. A little later in this appendix, we’ll override the `onTearDownAfterTransaction()` method to perform some database cleanup.

`RantServiceTest` is almost done. But there are still a few loose ends to tie up. Let’s see how to give `RantServiceTest` access to the database itself to assert that the data is being persisted as we expect.

B.3.3 Testing against the database

There’s a small problem with `RantServiceTest` as it appears in listing B.4. When testing that the rant has been added, we don’t know with any certainty that the rant has been added to the database. The only thing that we can say for sure is that `RantService` says that a rant has been added to the database.

In a unit test, it’s okay to trust that `RantService` is telling us the truth because when we’re unit-testing `RantService`, we’re only concerned that `RantService` is doing what we expect—not that any database has been updated. But in an integration test, we want to be certain that the database is being updated with the values we expect. Therefore, it’s not good enough to trust `RantService`. We must go to the database to be certain.

Querying a database with JDBC is not all that complex, but Spring’s `JdbcTemplate` makes it even easier (as discussed in chapter 5). And `Abstract-`

`TransactionalDataSourceSpringContextTests`⁴ makes working with `JdbcTemplate` even easier when in an integration test.

- `AbstractTransactionalDataSourceSpringContextTests` is a transactional test case class like `AbstractTransactionalSpringContextTests`. But to support database queries, it also provides a `jdbcTemplate` instance variable. Test methods can use this instance variable to query the database directly.

For example, `RantServiceTest` has been changed again in listing B.5 to use `jdbcTemplate` to ensure that the rant has been added to the database.

Listing B.5 `RantServiceTest` modified to query the database to ensure that a rant has been added

```
package com.roadrantz.service;
import java.sql.ResultSet;
import java.sql.SQLException;
import org.springframework.jdbc.core.RowMapper;
import org.springframework.test.
    ↗ AbstractTransactionalDataSourceSpringContextTests;
import com.roadrantz.domain.Rant;
import com.roadrantz.domain.Vehicle;
import com.roadrantz.service.RantService;

public class RantServiceTest
    extends AbstractTransactionalDataSourceSpringContextTests {
    public RantServiceTest() {}

    protected String[] getConfigLocations() {
        return new String[] {
            "file:src/main/webapp/WEB-INF/roadrantz-services.xml",
            "file:src/main/webapp/WEB-INF/roadrantz-data.xml"
        };
    }

    public void testAddRant() throws Exception {
        RantService rantService =
            (RantService) applicationContext.getBean("rantService");

        Rant newRant = new Rant();
        newRant.setRantText("TEST RANT TEXT");
        Vehicle vehicle = new Vehicle();
        vehicle.setPlateNumber("FOOBAR");
```

Performs
database-enabled
test

⁴ Believe it or not, `AbstractTransactionalDataSourceSpringContextTests` is still not the class with the longest name in the Spring API. A quick scan of Spring's JavaDoc reveals that `LazySingletonMetadataAwareAspectInstanceFactoryDecorator` takes top prize for having the longest class name. What does that class do? Well... isn't it obvious from the name?

```

vehicle.setState("TX");
newRant.setVehicle(vehicle);

int before = jdbcTemplate.queryForInt(
    "select count(*) from rant");
rantService.addRant(newRant);

int after = jdbcTemplate.queryForInt(
    "select count(*) from rant");

assertEquals("There should be one more row in rant table.",
    after, before+1);

String testRantText = (String) jdbcTemplate.queryForObject(
    "select rantText from rant where id=?",
    new Object[] { newRant.getId() }, String.class);
assertEquals("newRant.getRantText()", testRantText);
}
}

```

**Asserts exactly
one row was
added**

**Asserts
rant text is
correct**

Rather than take the word of RantService, RantServiceTest now performs some simple queries against the database. The first assertion made is that the number of rows in the rant table has grown by exactly 1 after the addRant() method is called. The actual rant text is also checked in another assertion.

AbstractTransactionalDataSourceSpringContextTests expects that a DataSource be declared in the Spring application context. That's not a problem, as we have already defined one in roadrantz-data.xml.

Cleaning up

Although AbstractTransactionalDataSourceSpringContextTests will roll back any changes made in the test methods, it's still possible for you to force a commit by calling setComplete(). Even if you have good reasons for committing the data in the course of your tests, you'll probably still want to clean up the mess when you're done.

To accommodate test data cleanup, AbstractTransactionalDataSourceSpringContextTests provides a convenience method for deleting all data in one or more tables. The deleteFromTables() method deletes all rows from the tables specified in a String array.

To illustrate, here's an onTearDownAfterTransaction() method that calls deleteFromTables() to delete everything in the rant and vehicle tables:

```

protected void onTearDownAfterTransaction() throws Exception {
    deleteFromTables(new String[] {"rant", "vehicle"});
}

```

It's important to understand the consequences of calling `deleteFromTables()`. All data will be deleted from the tables specified—not just the data that was added by the test. Therefore, it's important that you only use `deleteFromTables()` when testing against a database where you can afford to wipe the tables clean. (In other words, *please* don't run these kinds of tests against your production database!)

B.3.4 Testing in JUnit 4 with Gienah Testing

All of the examples in this appendix so far have been based on JUnit 3.8.1, as that's the version of JUnit that Spring's abstract test case classes are based on. Even so, as I write this, JUnit 4.3.1 is the latest version of JUnit available, and JUnit 4.4 is expected to be released very soon. JUnit 4 takes advantage of Java 5 annotations in defining test cases, resulting in slightly cleaner testing code.

In case you'd like to move up to JUnit 4, I thought I should inform you of gienah-testing (<http://code.google.com/p/gienah-testing/>), a new JUnit 4 extension that enables dependency injection of JUnit 4 test classes. It uses a custom JUnit 4 runner to load a Spring application context and inject values into class-scoped variables in the test case class.

To demonstrate gienah-testing, let's rewrite the `RantServiceTest` test class, basing it on JUnit 4 and using gienah-testing to inject the `RantService` directly into the test class's `rantService` variable. Listing B.6 shows the new version of `RantServiceTest`.

Listing B.6 The RantServiceTest, rewritten as a JUnit 4 test case, using gienah-testing

```
package com.roaddrantz.service;
import java.util.List;
import junit.framework.Assert;
import org.gienah.testing.junit.Configuration;
import org.gienah.testing.junit.Dependency;
import org.gienah.testing.junit.SpringRunner;
import org.gienah.testing.junit.Transactional;
import org.junit.Test;
import org.junit.runner.RunWith;
import com.roaddrantz.domain.Rant;
import com.roaddrantz.domain.Vehicle;
@RunWith(value = SpringRunner.class)    <-- Uses gienah's  
SpringRunner
@Configuration(locations = {
    "src/main/webapp/WEB-INF/roaddrantz-services.xml",
    "src/main/webapp/WEB-INF/roaddrantz-data.xml"})
public class RantServiceTest {
    @Dependency
    private RantService rantService;    | Injects  
rantService
}                                     | Specifies  
context  
definition files
```

```
@Transactional    ← Tests in a transaction
@Test
public void testAddRant() throws Exception {
    Rant newRant = new Rant();
    newRant.setRantText("TEST RANT TEXT");
    Vehicle vehicle = new Vehicle();
    vehicle.setPlateNumber("ABC123");
    vehicle.setState("TX");
    newRant.setVehicle(vehicle);
    rantService.addRant(newRant);

    List<Rant> rants = rantService.getRantsForVehicle(vehicle);
    Assert.assertTrue(rants.contains(newRant));
}
```

The first thing to notice about this new JUnit 4/gienah-based test case is that the class is annotated with `@RunWith` to use `SpringRunner`. JUnit 4 uses test runners to know how to execute tests. JUnit 4's default test runner knows nothing about Spring, so here `@RunWith` specifies that gienah's Spring-aware test runner be used instead.

Now that JUnit 4 knows to use the Spring-aware test runner, it will need to know which Spring context definition files to load. For that, I'm using gienah's `@Configuration` annotation to load `roadrantz-services.xml` and `roadrantz-data.xml`.

In the previous versions of `RantServiceTest`, the `RantService` instance was retrieved directly from the Spring application context. But with gienah, it can be injected from the Spring context. The `@Dependency` annotation tells gienah to inject the `rantService` variable with a bean from the Spring context whose name is `rantService` (the same name as the variable). Figure B.4 illustrates how this works.

If for some reason the `RantService` was named something else (perhaps `roadRantService`), I could specify a bean name using the `bean` attribute:

```
@Dependency(bean="roadRantService")
private RantService rantService;
```

Finally, the `@Transactional` annotation placed on the test method indicates that any work done within the test method should occur within a transaction and be rolled back after the test completes. This keeps the test method from leaving a mess behind when it's over.

gienah-testing brings much of Spring's context-aware testing into the JUnit 4 world. But you should be aware that as I write this, gienah-testing is at version E0.31, where "E" stands for "experimental," and E0.4 is to be released at the same time as JUnit 4.4. As it is experimental, you should expect some quirks. Here are a couple that I've encountered:

- The `@Transactional` annotation depends on a transaction manager being configured in the Spring application context with a bean ID of `transactionManager`. It will not work if the transaction manager is configured with any other ID.
- Oddly, the E0.31 JAR file is compiled using a Java 6 compiler. If you want to use gienah-testing with Java 5, you're out of luck. This is unfortunate as it shuts out anyone who can't use Java 6 yet (such as Mac users).

Both of these issues are expected to be fixed with E0.4. Even so, E0.4 is still an experimental release—so proceed with caution.

One other annoyance is that gienah-testing is not currently available in any of the known Maven 2 repositories. Therefore, if you're using Maven 2 for your build (as I am), you'll need to download gienah-testing and add it to your local repository using the `install:install-file` goal:

```
% mvn install:install-file
  -DgroupId=gienah
  -DartifactId=gienah
  -Dversion=E0.31-patched
  -Dpackaging=jar
  -Dfile=gienah-testing-bin-E0.31-patched.jar
```

Despite its quirkiness, gienah-testing shows a lot of promise. Keep an eye on its progress at its homepage: <http://code.google.com/p/gienah-testing>.

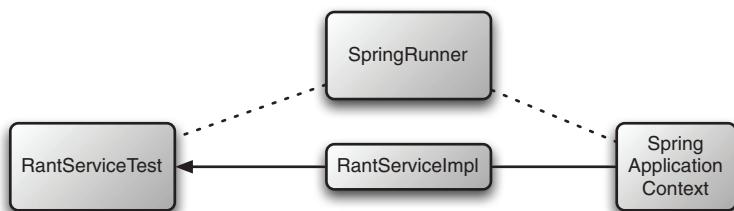


Figure B.4 gienah-testing's `SpringRunner` automatically injects dependency variables in a JUnit 4 test case with beans from the Spring application context.

B.4 Summary

Test-driven development is a practical way of ensuring the correctness of the code you write. Testing frameworks like JUnit are indispensable in writing and running automated test cases.

Although Spring supports unit testing by encouraging decoupled code, Spring isn't often directly involved in the activity of testing. Nevertheless, there are some circumstances where Spring can pitch in and make testing a bit easier.

For unit testing, Spring's `AbstractModelAndViewTests` supplies developers with some handy assertion methods that make short work of verifying the data returned from a Spring MVC controller in a `ModelAndView` object. And Spring also comes with several mock implementations of common interfaces that are perfect for testing controller classes.

When it comes to integration testing, `AbstractDependencyInjectionSpringContextTests` can handle loading and unloading of the Spring application context. `AbstractTransactionalSpringContextTests`, along with `AbstractTransactionalDataSourceSpringContextTests`, can also ensure that any database work is rolled back between tests.

index

Symbols

#springFormInput macro 563
#springFormTextarea macro 563
<@spring.formInput> macro 568
<@spring.formTextArea> macro 568
<[CDATA...]> construct 113
@AspectJ annotation 141, 144
 @Around annotation 144
@Cacheable annotation 217
@CacheFlush annotation 217
@Configurable annotation 87
@Pointcut annotation 142
@Transactional annotation 244

A

AbstractCommandController 508–510
AbstractController 508–509
AbstractDependencyInjectionSpringContext Tests 694, 696, 705
AbstractExcelView 570–571
AbstractFormController 508–509
abstracting base bean type 74, 76
 common properties 76, 78
 overriding inherited properties 76
 ProxyFactoryBean 138–139
AbstractJDomPayloadEndpoint 355
 invokeInternal() method 357
AbstractJmsMessageDrivenBean 431
AbstractMarshallingPayloadEndpoint 358
 invokeInternal() 360
AbstractMessageDrivenBean 431
AbstractModelAndViewTests class 690, 705
 assertion methods 690

AbstractPdfView 574–575
AbstractStatefulSessionBean 431
AbstractStatelessSessionBean 431–432
AbstractTransactionalDataSourceSpringContextTests 700–701, 705
AbstractTransactionalSpringContextTests 697, 699
AbstractTransactionSpringContextTests 705
AbstractWizardFormController 508, 521, 524, 584
 getTargetPage() 524–525
 processCancel() 526
 processFinish() 525–526
access decisions managers 251, 271–275
 voter abstinence 275
 voting 272–274
AccessDecisionManager 271–272
 allowIfAllAbstain property 275
 decide() 272
 supports() 272
AccessDecisionVoter interface 272–273
 See also RoleVoter
AccessDeniedHandlerImpl 292
Acegi Security System 248–249
 See also Spring Security
<action> 607
<action-state> 597, 607
ActionSupport 627, 629
ActiveMQ message broker
 setting up 392–393
 creating connection factories 392
 declaring destinations 392–393
ActiveMQConnectionFactory 392–393
 brokerURL property 392
Adobe’s Portable Document Format (PDF) 569–575

- advice 119–120, 127, 132
 AfterReturningAdvice 129–130
 MethodBeforeAdvice interface 129
 MethodInterceptor interface 131–132
 Spring AOP 127
 ThrowsAdvice interface 130–131
advisors 132–136
 combining pointcuts with 134
after-invocation managers 251–252
AfterReturningAdvice 129–130
Ajax 663–664
 accessing Spring-managed beans DWR 659, 661–663
 Direct Web Remoting (DWR) 648, 650–651, 654, 656–658
allowIfAllAbstain property 275
annotated domain objects 187–188
AnnotationAwareAspectJAutoProxyCreator 143
annotation-driven caching 217–218
annotation-driven transactions 243, 245
AnnotationJmxAttributeSource 474
annotations
 declaring interceptors using Pitchfork 438–439
 injecting resources by Pitchfork 437–438
 supported by Pitchfork 435
AnnotationSessionFactoryBean 187–188
Ant 675–676
aop namespace 243
AOP. *See* aspect-oriented programming (AOP)
<aop:after-throwing> 147
<aop:aspect> 28
<aop:aspectj-autoproxy> 143
<aop:before> 29, 147
<aop:config> 28, 146
<aop:pointcut> 29, 148
<aop:spring-configured> 87
Apache Ant 675–676
application context
 exposing in JSF 648
 splitting up 493–494
application context module 8
application events 101–103
application lifecycle events 8
application objects, associations between 32
ApplicationContext interface 35, 40
 lifecycle 37, 40
 publishEvent() 102
ApplicationContextAware interface 104
 setApplicationContext() 105
ApplicationContextAwareProcessor 95
ApplicationContexts 33
 compared to BeanFactories 36
 loading 36–37
 retrieving beans from 37
application-managed Java Persistence API (JPA) 198–199
approveCreditCard() 606
AspectJ 122
AspectJ 5 141
AspectJ aspects 88
AspectJ pointcuts 135–136
aspectj/ directory 668
AspectJExpressionPointcut 135
AspectJExpressionPointcutAdvisor 135
aspect-oriented programming (AOP) 8, 24, 30, 118, 125
 autoproxying 139–144
 compared to dependency injection (DI) 117
 configuration elements 145
 creating classic aspects 125–139
 declaring pure-POJO aspects 145, 149
 description 24–25
 example 26–27, 30
 module 8
 overview 4, 6, 118–119
 Spring support for 122, 125
 terminology 119–120
aspects 120, 125, 139
 @AspectJ 141, 144
 advice, creating 127–132
 advisors 132–136
 AspectJ 149, 152
 autoproxies, creating 140–141
 compared to inheritance and delegation 119
 implementing 27, 30
 pointcuts, defining 132, 136
 AspectJ 135–136
 combining with advisors 134
 regular expression 133–134
ProxyFactoryBean 136, 139
 abstracting 138–139
 pure-POJO 145, 149
asynchronous messaging 385–386, 422
Java Message Service (JMS) 386, 393, 407
 architecture 387, 389
 benefits 390–391
 converting messages 402, 405
 gateway support classes 405, 407
 runaway code 393, 395
 setting up ActiveMQ message broker 392–393

- asynchronous messaging, Java Message Service (JMS) (*continued*)
 templates 395, 402
 message-based RPC 416, 422
 exporting services 418–419
 Lingo 417–418
 proxying Java Message Service (JMS) 420, 422
 message-driven POJOs (MDPs) 407, 416
 creating message listeners 408, 412
 writing 412, 416
atomic, consistent, isolated, durable (ACID) 223–224
attributes property,
 MethodDefinitionAttributes 303
attributes, MBean
 transactions 233, 237
 isolation levels 235–236
 propagation behavior 233, 235
 read-only 236–237
 rollback rules 237
 transaction timeout 237
 using interfaces to define 472–473
AttributesJmxAttributeSource 474
AuthByAuthenticationProvider 254
 See also ProviderManager
authenticating users 252, 271
 against databases 256, 264
 caching user information 263–264
 encrypted passwords 260, 262
 InMemoryDaoImpl 257, 259
 JdbcDaoImpl 259–260
 against LDAP repositories 264, 271
 BindAuthenticator 265, 267
 comparing passwords 267, 269
 DefaultLdapAuthoritiesPopulator 269, 271
 ProviderManager 253, 256
authentication 287–291
authentication manager 250
authentication providers 254
AuthenticationDao 257
 See also DaoAuthenticationProvider
authenticationFailureUrl property 291
AuthenticationManager 252
 authenticate() 253
 interface 252
authenticationManager property 291
 See also ProviderManager
authentication-processing filters 277
AuthenticationProcessingFilter 290–291
AuthenticationProcessingFilterEntryPoint 289
AuthenticationProvider interface 256
authoritiesByUsernameQuery property 260
authorization exceptions 292–293
<authz:authentication> 300
<authz:authorize> 298
<authz:ifAllGranted> 298
<authz:ifAnyGranted> 298
<authz:ifNotGranted> 299
<authz:operation> 300
autoboxing 182
autodetect autowiring 62
autowire attribute 62
autowire property 58
autowiring 58–64
 mixing with explicit wiring 63
 shortcomings 63–64
 types 59, 62
awareness 103–106
 BeanNameAware interface 104–105
AxisBeanMappingServicePostProcessor 337
-
- B**
- base bean types, abstracting 74, 76
BaseCommandController 508–509
BasicDataSource 167–168
BasicProcessingFilter 288
BasicProcessingFilterEntryPoint bean 286
<bean> 41
 abstract attribute 74–75
 factory-method attribute 67
 parent attribute 74
BeanCounter 96
BeanDeserializer class 337
BeanFactories 33
 retrieving beans from 35
BeanFactory class 13
BeanFactory container 7
BeanFactory interface 34–35
BeanFactoryAware interface 104–105
BeanFactoryPostProcessor interface 95–96
BeanNameAutoProxyCreator 301
BeanNameAware interface 104–105
BeanNameUrlHandlerMapping 499, 501, 503
BeanNameViewResolver 538, 540–541
BeanPostProcessor interface 93, 95
<beans> 12
beans
 controller 498–499
 exposing as HTTP services 324, 326

- beans (*continued*)
- exposing functionality with Burlap
 - exporting services 321–322
 - exposing functionality with Burlap and Hessian 318, 322
 - exposing functionality with Hessian controllers 320–321
 - HessianServiceExporter 319–320
 - lifecycle 34
 - steps 37
 - proxied inner 216–217
 - proxying for caching 215, 217
 - flushing 215–216
 - retrieving from ApplicationContexts 37
- <beans> element 12
- BeanSerializer class 337
- BeanShell, scripting beans in 110–111
- BindAuthenticator 265–267
- boilerplate code, JDBC 172
- brokerURL property 392
- buildExcelDocument() 571
- building
- classpaths 669, 671
 - controllers 496–497
 - homepages 495–502
 - orders 601, 605
 - wizard controllers 521, 524
- buildPdfDocument() 575
- Burlap 316, 322
- accessing services 317–318
 - exposing bean functionality 318, 322
 - exporting services 321–322
- BurlapProxyFactoryBean 317
- BurlapServiceExporter 321
-
- C**
- <cache> 213
- cache property, JndiObjectFactoryBean 448
- caching data 208, 218
- annotation-driven 217–218
 - caching solutions 210, 215
 - EHCache 213, 215
 - proxying beans for 215, 217
 - flushing 215–216
 - proxied inner beans 216–217
- callback classes 162
- CasAuthenticationProvider 254
- See also* ProviderManager
- Cascading Style Sheets (CSS) 573
- case normalization 270
- CastorMarshaller 366
- catch blocks 159–161
- Caucho Technology 316
- cellFuncs 658
- ChannelDecisionManagerImpl 297
- ChannelProcessingFilter 295
- filterInvocationDefinitionSource 296
- channels 294–297
- child beans
- declaring 73, 78
 - abstracting base bean type 74, 76
 - abstracting common properties 76, 78
- class attribute 41
- class inheritance 75
- ClassEditor 89
- classload time 121
- classpaths, building 669, 671
- ClassPathXmlApplicationContext 36–37
- collaboration 19–20
- collection configuration elements 52–53, 58
- arrays 53–54
 - lists 53–54
 - maps 55–56
 - properties 56, 58
 - sets 54–55
- commands, processing 509, 511
- Commons Attributes compiler 505
- Commons Logging 671
- Commons Validator 517, 520
- validation rules 519
- CommonsHttpMessageSender 376
- CommonsPathMapHandlerMapping 503, 505
- compile time 121
- ComponentControllerSupport 555
- conditionally rendering content 298–299
- configLocation property 205
- configuration attributes, EHCache 214
- configuration elements
- aspect-oriented programming (AOP) 145
 - Spring Modules 212
- configuring
- context loaders 494–495
 - controller beans 498–499
 - data sources 96, 165, 169
 - Direct Web Remoting (DWR) 650–651
 - DispatcherServlet 492–495
 - engines
 - FreeMarker 565–566
 - Velocity 558–559
 - flow executors 586–587

configuring (*continued*)
 JDBC driver-based 168–169
 JNDI 165, 167
 Spring 2.0 166–167
 mail senders 451, 453
 Java Naming and Directory Interface (JNDI) 452
 wiring into service beans 453
 pooled 167–168
Remote Method Invocation (RMI) services 313, 316
Tiles 552
connection factories, Java Message Service (JMS) 392
ConnectorServerFactoryBean 478
constructor autowiring 61–62
constructor injection versus setter injection 44
<constructor-arg> 13, 42
 autowiring 63
 ref attribute 44
 value attribute 44
constructors, injecting through 42, 45
 object references 43, 45
container 6
container-managed Java Persistence API (JPA) 200, 202
containers 6, 33
 Bean factories 33
containing beans 33–40
 ApplicationContext interface 35, 40
 lifecycle 37, 40
 BeanFactory interface 34–35
context loaders, configuring 494–495
context reload, forcing 695–696
contextConfigLocation parameter 495
ContextLoaderListener 494, 661
ContextLoaderServlet 495
contextual sessions, Hibernate 3 192, 194
contract-first web services 344–347, 373, 382–383
 creating sample XML messages 348, 353
 messages with service endpoints 353, 360
 JDOM-based message endpoints 355, 357
 marshaling message payloads 358, 360
 web service gateway support 381–382
 web service templates 374, 380
 marshallers on client side 379–380
 sending messages 377, 379
 wiring 361, 373
 configuration 361, 363
 deploying 373
 endpoint exceptions 367, 369
 mapping messages to endpoints 363–364
 message marshalers 364, 367
 service endpoints 364
 WSDL files 369, 373
contract-last web services 345
Controller 508
controller beans, configuring 498–499
Controller interface 507
ControllerClassNameHandlerMapping 503–505, 586
controllers
 building 496–497
 classes 508
 creating Tile 554, 556
 form 512
 Hessian 320–321
 hierarchy 507
 mapping requests to 502–506
 Spring MVC 491–492, 685, 692
 throwaway 528, 531
 wizard 521, 524
conversation-style navigation 583
convertAndSend() 404
converted messages 402–405
convertToUpperCase 270
core container 7–8
coupling 17, 19
creating beans 40, 45
 controlling 64, 71
 creating from factory methods 66, 68
 destroying 68, 71
 initializing 68, 71
 scoping 66
 declaring 40–41
 injecting through constructors 42, 45
 object references 43, 45
cron expressions 464
cron jobs, scheduling 462–463
CronTriggerBean 104, 462–463
cross-cutting concerns 24, 117–118
custom editors 89–92
CustomDateEditor 89
CustomEditorConfigurer 91
customer information 594, 601
 adding new customers 598, 601
 creating data flow 615, 617
 looking up data 597–598
 phone numbers 595–596

D

- DAO. *See* data access objects (DAOs)
- DaoAuthenticationProvider 256–260
authenticationDao 257
diagram of 256
MD5 encoding 261
userDetailsService property 257
wiring 257
See also ProviderManager
- data access 156, 219
caching 208, 218
annotation-driven 217–218
caching solutions 210, 215
proxying beans for 215, 217
- data sources 165, 169
JDBC driver-based 168–169
JNDI 165, 167
pooled 167–168
- Hibernate 183, 194
data access objects (DAOs) 190–191
Hibernate 3 contextual sessions 192, 194
templates 186, 190
versions 185–186
- iBATIS 203, 208
data access objects (DAOs) 207–208
templates 204, 207
- Java Persistence API (JPA) 194, 203
data access objects (DAOs) 202–203
entity manager factories 197, 202
templates 194, 197
- JDBC 170, 183
data access object (DAO) support classes 180, 183
runaway code 170, 173
templates 173, 180
- overview 156–157
- tiers 157, 165
data access object (DAO) support classes 163, 165
exceptions 158, 161
templating 161, 163
- data access exceptions 160
- data access objects (DAOs) 157
Hibernate 190–191
iBATIS 207–208
Java Persistence API (JPA) 202–203
JDBC 180, 183
named parameters 181–182
simplified in Java 5 182–183
module 8
- service objects access 157
support classes 163, 165
- data access templates 163
HibernateTemplate 162
JdbcTemplate 162
JpaTemplate 162
- data contracts 349, 353
- data sources 165, 169
configuring 96
JDBC driver-based 168–169
JNDI 165, 167
Spring 2.0 166–167
pooled 167–168
- Database Connection Pools (DBCP) 167
- databases
authenticating users against 256, 264
caching user information 263–264
encrypted passwords 260, 262
InMemoryDaoImpl 257, 259
JdbcDaoImpl 259–260
integration-testing against 699, 701–702
- datacentric messaging 390
- DataSource 443
JdbcTemplate class 174
retrieving from Java Naming and Directory Interface (JNDI) 443, 445
- dataSource property
AnnotationSessionFactoryBean 188
LocalSessionFactoryBean 187
- DataSourceTransactionManager 226
- dates, formatting in Velocity 560–561
- dateToolAttribute property
VelocityViewResolver bean 560
- DBCP. *See* Database Connection Pools (DBCP)
- decide() 272
- decision states 612, 614
<decision-state> 612–613, 616
- declarative transaction management 224
- decoupling 101–103
- default destinations, setting 399–400
- DefaultAdvisorAutoProxyCreator 95, 140
- DefaultBeanValidator 518
- <defaultCache> 213
- default-destroy-method 69
- DefaultInitialDirContextFactory 266, 269
- default-init-method attribute 69
- DefaultLdapAuthoritiesPopulator 269, 271
constructor arguments 269
convertUpperCase property 270
groupRoleAttributes property 269
groupSearchFilter property 271

- DefaultMessageListenerContainer 411
 transactionManager property 412
defaultObject property,
 JndiObjectFactoryBean 449
DefaultPointcutAdvisor class 134
defaultTargetUrl property 291
DefaultValidatorFactory 518
DelegatingRequestProcessor 629, 631
DelegatingTilesRequestProcessor 630
DelegatingVariableResolver 621, 646, 648, 664
delegation, compared to aspects 119
dependency injected tests
 setting up 696
 tearing down 696
dependency injection (DI) 14–23
 compared to aspect-oriented programming (AOP) 117
 enterprise applications 21, 23
 example 15, 21
 coordinating collaboration 19–20
 coupling 17, 19
 unit testing 16–17
 wiring 20–21
 method injection 79, 85
 getter injection 83, 85
 method replacement 80, 83
 overview 4, 6, 14
destinations
 default Java Message Service (JMS) 399–400
 defined 388
destinations, ActiveMQ declaring 392–393
destroy() 70
destroy-method attribute 68
DHTML. *See* Dynamic HTML (DHTML)
DI. *See* dependency injection (DI)
Diaphragma 641
Direct Web Remoting (DWR) 648, 650, 658, 664
 calling remote methods from JavaScript 656–657
 configuration 650–651
 defining remote objects 651, 654
 displaying results 657–658
 exporting remote objects to JavaScript 654, 656
Spring configuration namespace 662–663
 Spring creators 659, 661
dirty reads 235
DispatcherServlet 329, 491–492, 495, 501
 configuring context loaders 494–495
 splitting up application context 493–494
DisposableBean interface 69–71
docs/ directory 669
- doInTransaction() 232
domain objects 86
driverClassName property 167
DriverManagerDataSource class 169
DWR. *See* Direct Web Remoting (DWR)
dwr.xml file 654
DwrServlet 650, 656
DwrSpringServlet 662
DWRUtil object 658
Dynamic HTML (DHTML) 649
dynamic MBeans 467
DynamicWSDL11Definition 369
-
- E**
- EasyMock 688–689
EHCache 213–217
 <ehcache:annotations> 217
 <ehcache:caching> 215
 <ehcache:flushing> 215
 <ehcache:proxy> 215
 EhCacheBasedUserCache 263
 EhCacheFactoryBean 264
 <ehcache:when> 218
EJB. *See* Enterprise JavaBeans (EJBs)
EJB 2.x
 compared to EJB 3 425
 complexities 434
 message-driven beans (MDBs) 408
 session beans, proxying 426, 430
 Spring-enabled Enterprise JavaBeans 431, 434
EJB 3 434, 439
 compared to EJB 2.x 425
 message-driven beans (MDBs) 408
 Pitchfork 435–437
 declaring interceptors using
 annotations 438–439
 injecting resources by annotation 437–438
 session beans, declaring 429–430
 specification 194
EJB specification 194
email
 constructing 453, 456
 sending 450, 456–458
 configuring mail senders 451, 453
embark() 17–18
embarkOnQuest() 29
encrypted passwords 260, 262
end states 593–594
endpoint exceptions 367, 369
endpoints, mapping messages to 363–364

- <end-state> 593–594
engines, configuring
 FreeMarker 565–566
 Velocity 558–559
enterprise applications 21, 23
Enterprise JavaBeans (EJB) 424, 440
 complexities 434
 specification 4–5
enterprise services 442, 485
 Java Management Extensions (JMX) 466, 484
 exporting Spring beans as MBeans 467, 477
 handling notifications 482, 484
 remoting MBeans 477, 482
 Java Naming and Directory Interface (JNDI)
 conventional 443, 446
 in Spring 2 449–450
 injecting objects 446, 449
 wiring objects from 442, 450
scheduling tasks 456, 466
 invoking methods on schedule 464, 466
 Java Timer class 457, 460
 Quartz scheduler 460, 464
sending email 450, 456
 configuring mail senders 451, 453
 constructing email 453, 456
entity beans 194
entity manager factories
 Java Persistence API (JPA) 197, 202
 application-managed 198–199
 container-managed 200, 202
EntityManager interface 194–197
entityManagerFactory property 195
<entry> 56
errors, displaying in JavaServer Pages (JSPs)
 547, 549
events 590
 listening for 102–103
 publishing 101–102
Excel spreadsheets 570, 573
exception translation filters 277
exceptionMappings property 368
exceptions
 authorization 292–293
 handling 531–532
 security 291, 293
ExceptionTranslationFilter 292
execute() 231
explicit wiring, mixing with autowiring 63
exporting
 Burlap services 321–322
 Lingo services 418–419
methods by name 471–472
remote MBeans 478
Remote Method Invocation (RMI) services
 312, 316
Spring beans as MBeans 467–477
exposeRequestAttributes property
 FreeMarkerViewResolver 566
 VelocityViewResolver bean 561
exposeSessionAttributes property
 FreeMarkerViewResolver 566
 VelocityViewResolver bean 561
exposeSpringMacroHelpers property
 VelocityViewResolver bean 563
exposing bean functionality
 Burlap 318, 322
 exporting services 321–322
 Hessian 318, 322
 controllers 320–321
 HessianServiceExporter 319–320
 HTTP services 324, 326
externalized messages, rendering in JavaServer
 Pages (JSPs) 544, 547
externalizing configuration properties 96, 99
extract methods 614
-
- F**
- factory methods, creating beans from 66, 68
factory-method attribute element 67
fallback objects, Java Naming and Directory
 Interface (JNDI) 448–449
FileEditor 89
FileSystemXmlApplicationContext 36
<filter> 278
filter security interceptors 278
FilterChainProxy 281, 283
filterInvocationDefinitionSource property
 282, 296
<filter-mapping> 278
filterProcessesUrl property 290
filters, proxying 279–281
FilterSecurityInterceptor 293
 objectDefinitionSource property 294
FilterToBeanProxy 279, 629
 web.xml files 283
final methods 124
find() 197
flow definitions, registering 588
flow executors, configuring 586–587
flow variables 591, 593
 scopes 593

<flow:executor> 587
<flow:location> 588
<flow:registry> 588
FlowAction 599–601
FlowController 585–586, 596
FlowExecutorFactoryBean 587
FlowNavigationHandler 621
FlowPhaseListener 621
flows, creating 591, 611
 building orders 601, 605
 completing orders 605, 608
 flow variables 591, 593
 gathering customer information 594, 601
 start states 593–594
flushing caches 215–216
forceHttps property 289
foreign languages 540, 547, 549
form controllers 510
form data binding in JavaServer Pages (JSPs)
 542, 544
form fields, binding
 in FreeMarker 567, 569
 in Velocity 561, 564
<form:errors> 547–548
<form:form> 544
<form:input> 544
<form:textarea> 544
formatting, in Velocity 560–561
form-based authentication 289, 291
form-binding JavaServer Page (JSP) tags 543
formObjectClass property, FlowAction 600
formObjectName property, FlowAction 600
formObjectScope property, FlowAction 600
forms
 pages 524–525
 processing with wizards 520–521, 524–526, 528
 submissions 512, 515, 517, 520
 validating input 515, 517, 520
Fowler, Martin 14
free-flow navigation 582
FreeMarker 564, 569
 binding form fields 567, 569
 configuring engines 565–566
 defining views 564–565
 exposing attributes 566–567
 macros 566
 resolving views 566
FreeMarkerConfigurer 565–566
FreeMarkerViewResolver 566–567
front controllers 491

G

gateway support classes for Java Message Service (JMS) 405, 407
 web services 381–382
getAsText() 89
getBean() 13, 35, 37
getConnection() 164
getContents() 81
getHibernateTemplate() 190
getInstance() 67
getJdbcTemplate() 181
getNamedParameterJdbcTemplate() 182
getTargetPage() 524–525
getter injection 83, 85
gienah-testing 702, 704
 SpringRunner 704
global object 637, 639
global-session scoping option 65
greeting property 12–13
GreetingService interface 11
GreetingServiceImpl class 11–12
Groovy, scripting beans in 110
groupRoleAttributes property 269
groupSearchFilter property 271

H

handler mappings 505–506
HandlerMapping
 implementations 503
 order property 506
HeaderTileController 554, 556
hello.xml file 12
Hessian 316, 322
 accessing services 317–318
 controllers 320–321
 exposing bean functionality 318, 322
HessianProxyFactoryBean 317
HessianServiceExporter 319–320
Hibernate 86, 159, 183, 194
 catch blocks 159
 contextual sessions
 advantages/disadvantages 193
 data access objects (DAOs) 190–191
 exception hierarchy 159
 Hibernate 3 contextual sessions 192, 194
 package structure 185
 Session interface 186
 SessionFactory interface 186
 templates 186, 190

- Hibernate, templates (*continued*)
 - annotated domain objects 187–188
 - classic mapping files 186–187
 - transaction managers 227
 - versions 185–186
 Hibernate 3 192–194
 - HibernateDaoSupport class 190
 - HibernateException class 186
 - hibernateProperties property 187–188
 - HibernateTemplate class 162, 186, 192
 - load() 190
 - saveOrUpdate() 189
 - HibernateTemplate-based DAO 189
 - HibernateTransactionManager 227
 - HolyGrailQuest class 16–18
 - HomePageController 496, 501
 - name attribute 498
 - homepages 495, 502
 - building controllers 496–497
 - configuring controller beans 498–499
 - creating JSP 500–501
 - declaring view resolvers 499–500
 - ModelAndView 497–498
 - HTML content 493
 - HttpInvoker 322, 326
 - accessing services 323–324
 - exposing bean functionality 324, 326
 - HttpInvokerProxyFactoryBean 323
 - HttpInvokerServiceExporter 324
 - HttpSessionContextIntegrationFilter 286
 - HttpURLConnectionMessageSender 376
-
- iBATIS 86, 203, 208
 - data access objects (DAOs) 207–208
 - templates 204, 207
 - SQL maps 205–206
 - SqlMapClientTemplate 204–205
- id attribute 41
- ifAllGranted 298
- ifAnyGranted 298
- ifNotGranted 299
- implementing aspects 27, 30
- indirection 387
- inheritance 75
 - compared to aspects 119
- initialization on demand holder 67
- initializing beans 68, 71
 - default-init-method attribute 69
- init-method attribute 68
- <inject> 642
- InjectObject annotation 642
- InMemoryDaoImpl 257–259
 - See also* AuthenticationDao
- inner beans 51–52
 - proxied 216–217
- integration filters 277
- integration-testing 684–704
 - AbstractDependencyInjectionSpringContextTests 705
 - AbstractTransactionalDataSourceSpringContextTests 705
 - AbstractTransactionSpringContextTests 705
 - against database 699, 701–702
 - in JUnit 4 with gienah-testing 702, 704
 - transactional objects 696, 698–699
 - wired objects 694–696
- interceptors, in Pitchfork 438–439
- InterfaceBasedMBeanInfoAssembler 472–473
- interfaces
 - defining MBean operations/attributes 472–473
 - mocking with EasyMock 688–689
 - template methods 161
- InternalResourceViewResolver 499–502, 536–537, 540–541, 553
 - viewClass property 537
- internationalization 8, 99, 547–549
- introductions 121
- invokeInternal()
 - AbstractJDomPayloadEndpoint 357
 - AbstractMarshallingPayloadEndpoint 360
- IoC. *See* inversion of control (IoC)
- isolation levels 235–236
-
- J**
- JaasAuthenticationProvider 254
 - See also* ProviderManager
- Jakarta Commons Database Connection Pools (DBCP) 167
- Jakarta Commons Logging 671
- Jakarta Commons Validator 517–520
- Jakarta Struts, integrating Web Flow with 619–620
- Jakarta Tiles 549, 556
 - creating controllers 554, 555
 - views 550–554
- JAR files 670
- Java
 - aspect-oriented programming (AOP) 123
 - SimpleJdbcTemplate class 174
 - Timer class 457–460

- Java (*continued*)
 creating timer tasks 458
 starting timers 459
- Java 5
 autoboxing 182
 runtime targeting 187
 simplified data access objects (DAOs) 182–183
 simplifying JDBC 178, 180
 varargs 182
- Java Data Objects (JDOs) 228
- Java EE Connector API (JCA) 9
- Java Management Extensions (JMX) 9
 managing Spring beans with 466, 484
 exporting as MBeans 467, 477
 handling notifications 482, 484
 remoting MBeans 477, 482
- Java Message Service (JMS) 10–11, 386–393, 407
 ActiveMQ message broker 392–393
 architecture 387–389
 conventional (non-Spring)
 sending/receiving 393–394
 converting messages 402, 405
 messageConverter property 405
 sending/receiving 404–405
 gateway support classes 405, 407
 proxying 420–422
 receiving messages 400, 402
 runaway code 393, 395
 sending messages 397, 399
 setting default destinations 399–400
 templates 395, 402
 JmsTemplate 397
 receiving messages 400, 402
 sending messages 397, 399
 setting default destinations 399–400
- Java Naming and Directory Interface (JNDI)
 caching objects 447–448
 conventional 443, 446
 fallback objects 448–449
 injecting objects 446, 449
 lazily loading objects 448
 mail sessions 452
 retrieving DataSource from 443, 445
 wiring objects from 442, 450
 in Spring 2 449–450
- Java Persistence API (JPA) 194, 203
 data access objects (DAOs) 202–203
 entity manager factories 197–202
 templates 194, 197
 transaction managers 227–228
- Java Transaction API (JTA)
 transaction managers 229
- java.beans.PropertyEditor interface 89
- java.util.Map collection 53
- java.util.Properties collection 53
- JavaBeans 1.00-A specification 4
- JavaDocs 669
- JavaMailSenderImpl 451–452
- JavaScript 652–655
- JavaServer Faces (JSF) 643, 648
 exposing application context in 648
 integrating Web Flow with 620, 622
 resolving JSF-managed properties 644–645
 resolving Spring beans 646
 using Spring beans in pages 646, 648
- JavaServer Page (JSP) 500–501
 templates 540, 542, 547, 549
- JaxRpcPortProxyFactoryBean 334, 337
- JaxRpcServicePostProcessor 338
- JBoss 165
- JBoss AOP 122
- JDBC 170–183
 boilerplate code 172
 compared to object-relational mapping (ORM) 203
 data access object (DAO) support classes 180–183
 exception hierarchy 159
 inserting rows into databases 170
 querying rows from databases 172
 runaway code 170, 173
 templates 173–180
 JdbcTemplate 174, 176
 transaction managers 226–227
 updating rows in databases 171
- JDBC abstraction 8
- JDBC driver-based data source 168–169
- JdbcDaoImpl 259–260
 authoritiesByUsernameQuery property 260
 usersByUserNameQuery property 260
 wiring 259
 See also AuthenticationDao
- JdbcDaoSupport base class 181
- JdbcDaoSupport class
 getConnection() 164
- JdbcTemplate 174, 176
- JdbcTemplate class 162, 173
 DataSource 174
 named parameters 177
 query() 176
 querying data using 175
 SQLException 175
 update() 175

jdbcTemplate instance variable 700
 jdbcTemplate property 174
 JDO. *See* Java Data Objects (JDOs)
 JDOM-based message endpoints 355, 357
 JdoTransactionManager 228
 JEE namespace 166, 428–429
 JEE specification 442
 <jee:jndi-lookup> 166, 449–450, 452
 <jee:local-slsb> 428
 <jee:remote-slsb> 428
 JeeBeanFactoryPostProcessor 436
 JeeEjbBeanFactoryPostProcessor 436
 JMS. *See* Java Message Service (JMS)
 JMSEception subclasses 395
 JmsGatewaySupport class 405, 407
 JmsProxyFactoryBean 420–421
 JmsServiceExporter 418–419
 JmsTemplate 395, 397, 402
 convertAndSend() 404
 receive() 401, 408
 receiveAndConvert() 404
 receiving messages 400, 402
 send() 398
 sending messages 397, 399
 wiring 397
 wiring default destinations 399–400
 JmsTemplate102 396
 JMX. *See* Java Management Extensions (JMX)
 JMXConnectorServer 478
 JNDI data sources 165, 167
 JNDI. *See* Java Naming and Directory Interface (JNDI)
 jndiName property 166–167
 LocalStatelessSessionProxyFactoryBean 427
 JndiObjectFactoryBean 166–167, 429–430, 445,
 448–452
 cache property 448
 defaultObject property 449
 JndiProxyFactoryBean 429
 Johnson, Rod 5
 joinpoints 120
 method 124
 JPA. *See* Java Persistence API (JPA)
 JpaDaoSupport class
 getJpaTemplate() 203
 JpaDialect 228
 JpaTemplate class 162, 194
 entityManagerFactory property 195
 find() 197
 JpaTransactionManager 228
 jpaVendorAdapter property 201
 JPS. *See* Java Persistence API (JPA)

JSF. *See* JavaServer Faces (JSF)
 JSP tags 86, 298–300
 JSP. *See* JavaServer Page (JSP)
 JSR-181 annotations 330–333
 Jsr181HandlerMapping 332
 Jsr181WebAnnotations 333
 JTA. *See* Java Transaction API (JTA)
 JtaTransactionManager 229
 JUnit 680–684
 setting up tests in 682–683
 tearing down tests in 683–684
 JUnit 4
 gienah-testing 702, 704
 junit.framework.TestCase class 681

K

key attribute
 element 57
 Knight interface 18
 Knight object 21
 KnightOfTheRoundTable class 19–20, 26, 29

L

<lang:bsh> 111
 <lang:groovy> 110
 <lang:inline-script> 113
 <lang:jruby> 109
 <lang:property> 112
 lazy loading 201
 objects 448
 lazy-init attribute 427
 LDAP repositories 264–271
 LdapAuthenticationProvider 264–269
 LdapShaPasswordEncoder 268
 lib/ directory 671
 Lingo 417–418
 exporting services 418–419
 <list> 52
 values 54
 load() 190
 LocalContainerEntityManagerFactoryBean 198, 200
 loadTimeWeaver property 201
 LocaleEditor 89
 LocalEntityManagerFactoryBean 198–199
 LocalSessionFactoryBean 186
 dataSource property 187
 hibernateProperties property 187
 mappingResources property 187

LocalStatelessSessionProxyFactoryBean 426, 429
 jndiName property 427
location property 97
Log4j 676–677
loginFormUrl property 289
<lookup-method> 84–85

M

macros
 FreeMarker 566
 Velocity 562
mail senders
 configuring 451, 453
 Java Naming and Directory Interface (JNDI)
 mail sessions 452
 wiring into service beans 453
mail sessions, Java Naming and Directory Interface
 (JNDI) 452
MailSender interface 450, 452
main() 13
<map> 53, 56
mapping
 complex types 337–338
 Hibernate files 186–187
 messages to endpoints 363–364
 requests to controllers 502–506
 SQL queries 205
mapping arrays 338, 340
mapping requests
 to JSR-181 annotated beans 332–333
 to XFireExporter 329–330
mapRow() 176, 179
MarketingMdb class 409
MarketingMdp class 409–410, 413–414, 416
 onMessage() 412
marshalers
 message 364, 367
 web service templates 379–380
Marshaller interface 364
marshalSendAndReceive() 379
Maven 1 675
Maven 2
 adding Spring as dependency 672, 674
 adding Spring Web Flow 584
 Spring modules available in repository 672
MBean servers (MBean agents) 467
MBeanExporter 467, 470
 registrationBehaviorName property 476–477
MBeanProxyFactoryBean 480–481
MBeans 472–473, 477
 dynamic 467

exporting Spring beans as 467, 477
 defining operations/attributes 472–473
 exposing methods by name 471–472
 metadata-driven 474, 476
object name collisions 476–477
open 467
proxying 480, 482
remoting 477–482
 accessing 479–480
 exposing 478
 standard 467
MBeanServerConnectionFactoryBean 479–480
MD5 encoding 261
MDBs. *See* message-driven beans (MDBs)
MDPs. *See* message-driven POJOs (MDPs)
message brokers 387
 ActiveMQ 392–393
message endpoints 353
 options 354
message listeners
 containing 410–411
 creating 408, 412
 transactional message-driven POJOs
 (MDPs) 411–412
message payloads, marshaling 358, 360
MessageConverter interface 402, 404
messageConverter property 405
MessageDispatcherServlet 361, 363
message-driven beans (MDBs) 408
message-driven POJOs (MDPs) 407, 416
 converting messages 415–416
 creating message listeners 408, 412
 containing message listeners 410–411
 transactional (MDPs) 411–412
 writing 412, 416
MessageListenerAdapter 412–414
messages
 internationalization (I18N) of 99
 mapping to endpoints 363–364
 rendering externalized
 in JSPs 544, 547
 sample XML 348, 353
 forging data contracts 349, 353
 sending with web service templates 377, 379
 with service endpoints 353, 360
 JDOM-based message endpoints 355, 357
 marshaling message payloads 358, 360
MessageSource interface 99
messaging 385–386, 422
 asynchronous 385–386
 datacentric 390

- messaging (*continued*)
 destinations defined 388
 indirection 387
 Java Message Service (JMS) 386, 393, 407
 architecture 387, 389
 benefits 390–391
 conventional (non-Spring) 393–394
 converting messages 402, 405
 gateway support classes 405, 407
 runaway code 393, 395
 setting up ActiveMQ message broker 392–393
 templates 395, 402
 message brokers 387
 message-based RPC 416, 422
 exporting services 418–419
 Lingo 417–418
 proxying JMS 420, 422
 message-driven POJOs (MDPs) 407, 416
 creating message listeners 408, 412
 writing 412, 416
 models 388–389
 point-to-point model
 location independence 391
 publish-subscribe model
 location independence 391
 queues 388–389
 synchronous 386, 390
 topics 389
 metadata 303–304
 using to map controllers 505
 MetadataMBeanInfoAssembler 474
 method injection 79, 85
 getter injection 80–85
 method replacement 80–83
 method invocations 300, 304
 metadata 303–304
 security aspects 301–302
 method joinpoints 124
 method replacement 80, 83
 MethodBeforeAdvice interface 129
 MethodDefinitionAttributes 303
 MethodInterceptor interface 131–132
 MethodInvocationJobDetailFactoryBean 465
 MethodInvocationTimerTaskFactoryBean 464–465
 MethodNameBasedMBeanInfoAssembler
 471–472
 MethodReplacer interface 83
 methods
 exposing by name 471–472
 invoking on schedule 464, 466
 MethodSecurityInterceptor 302
 Minstrel class 26–29
 mock/ directory 668
 MockHttpServletRequest 687
 MockHttpServletResponse 687
 model MBeans 467
 Model/View/Controller (MVC) framework 9
 ModelAndView 497–498
 asserting contents of 690, 692
 ModelAndView object 492, 501, 534
 modules 6–11
 MultiActionController 508
 MVC. *See* Model/View/Controller (MVC) framework
-
- N**
- name attribute, HomePageController 498
 named parameters
 DAO support for 181–182
 JDBC templates 176–178
 NamedParameterJdbcDaoSupport 182
 NamedParameterJdbcTemplate class 173,
 177, 181
 namingStrategy property 476
 navigation 582–583
 new creator 655
 noncolonized names 351
 nonrepeatable reads 235
 non-Spring beans, injecting 85, 88
 NotificationListener interface 484
 NotificationPublisherAware interface 482–483
 notifications
 handling 482, 484
 listening for 484
 <null/> 58, 63
 NullUserCache 263
 numberToolAttribute property
 VelocityViewResolver bean 560
-
- O**
- object name collisions
 MBean 476–477
 Object parameter 176
 object references, injecting with constructors
 43, 45
 objectDefinitionSource property 294
 object-relational mapping (ORM)
 compared to JDBC 203
 frameworks 184
 integration module 9

- objects
 caching 447–448
 fallback 448–449
 injecting 446, 449
 lazily loading 448
 mocking 687, 689
 wiring 442, 450
 wiring in Spring 2 449–450
onApplicationEvent() 102
open MBeans 467
operation attribute 300
operational contracts 349
operations, MBean, using interfaces to
 define 472–473
orders
 adding to 604–605
 building 601, 605
 completing 605, 608
 taking payments 605, 607
 displaying 602–603
 submitting 607–608
OrderService interface 23
OrderServiceImpl class 22
org.springframework.beans.factory.BeanFactory
 33–34
org.springframework.beans.propertyeditors.URLEditor 89
org.springframework.context.ApplicationContext
 interface 33
org.springframework.context.ApplicationEvent
 abstract class 102
org.springframework.context.ApplicationListener
 interface 103
ORM. *See* object-relational mapping (ORM)
<output-mapper> 617
-
- P**
- PagingNotificationListener 484
ParameterizableViewController 508
ParameterizedRowMapper class 179
parent attribute 74
parent beans 73–78
password encoders 261
password property 168
PasswordComparisonAuthenticator 267
PasswordDaoAuthenticationProvider 254
 See also ProviderManager
passwords
 comparing 267, 269
 encrypted 260, 262
PaymentProcessor class 606
payments 605, 607
persistence frameworks, Hibernate 159
persistence platform agnostic exceptions 159–160
persistence.xml file 198
persistence-layer code 494
PersistenceProvider interface
 createContainerEntityManagerFactory() 197
 createEntityManagerFactory() 197
persistenceUnitName property 199
persisting data 156, 219
 caching 208, 218
 annotation-driven 217–218
 caching solutions 210, 215
 proxying beans for 215, 217
data sources 165, 169
 JDBC driver-based 168–169
 JNDI 165, 167
 pooled 167–168
Hibernate 183, 194
 data access objects (DAOs) 190–191
 Hibernate 3 contextual sessions 192, 194
 templates 186, 190
 versions 185–186
iBATIS 203, 208
 data access objects (DAOs) 207–208
 templates 204, 207
Java Persistence API (JPA) 194, 203
 data access objects (DAOs) 202–203
 entity manager factories 197, 202
 templates 194, 197
JDBC 170, 183
 data access object (DAO) support classes
 180, 183
 runaway code 170, 173
 templates 173, 180
overview 156–157
tiers 157, 165
 data access object (DAO) support classes
 163, 165
 exceptions 158, 161
 templating 161, 163
phantom reads 235
phone numbers 595–596
Pitchfork 435–437
 annotations supported by 435
 declaring interceptors using annotations
 438–439
 injecting resources by annotation 437–438
plain-old Java objects (POJOs) 4, 34
 creating message listeners 408, 412
 message-driven (MDPs) 407, 416
 writing 412, 416

PlatformTransactionManager 232
 pointcuts 120, 132, 136
 combining with advisors 134
 declaring regular expression 133–134
 defining AspectJ 135–136
 point-to-point messaging model 388–389
 location independence 391
 POJOs. *See* plain-old Java objects (POJOs)
 pool-configuration properties 168
 pooled data sources 167–168
 Portlet Model/View/Controller (MVC) 10
 postProcessAfterInitialization() 93
 postProcessBeanFactory() 95
 postProcessBeforeInitialization() 93
 postprocessing beans 93, 95
 writing 93–94
 processCancel() 526
 processFinish() 525–526
 processing
 form submissions 517, 520
 forms with wizards 524–526, 528
 programmatic transaction management 224
 programming transactions 229, 232
 prompting users to log in 286, 291
<prop> 57
 propagation behavior 233, 235
 properties
 abstracting common 76, 78
 externalizing configuration 96, 99
 injecting 46, 58
 referencing other beans 48, 52
 values 47–48
 wiring nothing (null) 58
 overriding inherited 76
 Properties class 56
<property> 12, 47
 autowiring 63
 value attribute 48
 property editors, registering custom 88, 92
 property inheritance 75
 PropertyEditorSupport class 89
 PropertyPlaceholderConfigurer 96–97
<props> 53
 prototype scoping option 65
 ProviderManager 253–256
 proxied classes 124
 proxied inner beans 216–217
 proxies 121
 ProxyFactoryBean 136–139
 proxying
 beans for caching 215, 217
 filters 278–284

Java Message Service (JMS) 420–422
 MBeans 480, 482
 session beans (EJB 2.x) 426–430
 transactions 238–241
 publish-subscribe messaging model 389
 location independence 391
 pure-POJO aspects 145, 149

Q

QA. *See* Quality Assurance (QA)
 Quality Assurance (QA) 679
 Quartz scheduler 460, 464
 creating jobs 460–461
 scheduling cron jobs 462–463
 scheduling jobs 461–462
 starting jobs 464
 QuartzJobBean 460–461
 query() 176
 varargs 180
 querying data
 JdbcTemplate class 175, 179
 SimpleJdbcTemplate class 179
 querying rows from databases
 JDBC 172
 Quest interface 17–19
 queues 388–389

R

read-only transactions 236–237
 reads 235
 receiveAndConvert() 404
 receiving messages 404–405
 conventional (non-Spring) Java Message Service (JMS) 394
 JmsTemplate 400, 402
 ref attribute element 44
 reference document 669
 referencing other beans 48, 52
 injecting inner beans 51–52
 refresh-check-delay attribute 113
 refreshing scripted beans 112–113
 RegexpMethodPointcutAdvisor class 134
 registerCustomEditor() method 91
 registering
 BeanPostProcessor interface 94–95
 custom editors 92
 custom property editors 88, 92
 flow definitions 588
 Spring plug-in with Struts 626–627

- registrationBehaviorName property 476–477
regular expression pointcuts 133–134
 Perl5RegexpMethodPointcut class 133
reimplement() 83
remote exceptions 308
Remote Method Invocation (RMI) 309, 316
 configuring 313, 316
 exporting 312, 316
 wiring 310, 312
remote methods, JavaScript and 654–657
remote objects, defining 651, 654
remote procedure calls (RPCs) 307–308
 Lingo 418
 message-based 416, 422
 Lingo 417, 419
 proxying Java Message Service (JMS) 420, 422
remote services 306–309, 342
 Burlap 316–322
 Hessian 316–322
 HttpInvoker 322–326
Remote Method Invocation (RMI) 309–316
web services 326, 333, 340–341
 JaxRpcPortProxyFactoryBean 334, 337
 JSR-181 annotations 330, 333
 mapping arrays 338, 340
 mapping complex types 337–338
 proxying with XFire client 340–341
 XFire 326, 330
RemoteAuthenticationProvider 254
 See also ProviderManager
remoting 10
 MBeans 477, 482
 accessing 479–480
 exposing 478
 proxying 480, 482
 <replaced-method> 82
request attributes, exposing
 FreeMarker 566–567
 Velocity 561
requests
 handling with controllers 506, 509, 511–512,
 520, 528, 531
 lifecycles 491–492
 mapping to controllers 502–506
resource bundles, resolving views from 539–540
Resource objects 34
 ResourceBundleMessageSource 100, 546
 ResourceBundleViewResolver 539–540
resource-ref attribute 167
resourceRef property 166–167
Rich Site Summary (RSS) 576, 578
RMI. *See* Remote Method Invocation (RMI)
RmiProxyFactoryBean 311
RmiServiceExporter 314
RoleVoter 274
rollback rules 237
RowMapper object 176
 mapRow() 176, 179
RPC. *See* remote procedure calls (RPCs)
RSS. *See* Rich Site Summary (RSS)
Ruby
 adding new methods 79
 scripting beans in 109
run-as managers 251
RunAsImplAuthenticationProvider 254
 See also ProviderManager
runaway code 170, 173, 393–395
runtime 121
-
- S**
- salt sources 261
saveOrUpdate() 189
sayGreeting() 12, 14
ScheduledTimerTask 458–459
SchedulerFactoryBean 464
scheduling tasks 456, 466
 invoking methods on schedule 464, 466
 Java Timer class 457, 460
 creating timer tasks 458
 delaying start of timers 459–460
 scheduling timer tasks 458–459
 starting timers 459
 Quartz scheduler 460–464
 scheduling cron jobs 462–463
 scheduling jobs 461–462
 starting jobs 464
 timer tasks 458–459
schema declarations, Spring Modules 212
scope attribute 65
scopes 593
scoping beans 66
scoping options 65
scripting beans 106–114
 BeanShell 110–111
 Groovy 110
 injecting properties 111–112
 refreshing 112–113
 Ruby 109
 writing inline 113–114
secure channels 297
security context 285–286

security exceptions 250, 291, 293
 authorization exceptions 292–293
 security-layer code 494
`send()` 398
`sendAndReceive()` 377
 service beans, wiring mail senders into 453
 service endpoints 364
 messages with 353, 360
 JDOM-based message endpoints 355, 357
 marshaling message payloads 358, 360
 service objects, accessing of DAOs 157
 service-layer code 494
 service-oriented architecture (SOA) 326
 servlet filters 276
 authentication-processing 277
 exception translation 277
 filter security interceptors 278
 integration 277
 proxying 279, 281
 with Acegi 275
 servlet listeners 494
`<servlet-name>` 493
 session attributes 561–567
 session beans
 EJB 2.x 426, 430
 EJB 3 429–430
 session scoping option 65
 SessionFactory interface 192
 sessionFactory property 186
`<set>` 52, 54
`setApplicationContext()` 40, 105
`setAsText(String value)` 89
`setBeanFactory()` 105
`setBeanName()` 104
`setComplete()` 698
`setGreeting()` 12–13
`setOrderService()` 23
`setQuest()` 20
`setRollbackOnly()` 231
 setter injection
 compared to constructor injection 45, 253
 versus constructor injection 44
`setUp()` 683
 SimpleFormController 508–509, 514, 520
 SimpleJdbcDaoSupport 182
 SimpleJdbcTemplate 174, 178
 SimpleJdbcTemplate class
 Java 174
 querying data using 179
 RowMapper object
 `mapRow()` 179
 SimpleMappingExceptionResolver 531–532

SimpleMessageListenerContainer 411
 SimpleRemoteStatelessSessionProxyFactoryBean
 426–427, 429
 SimpleTriggerBean 461–462
 SimpleUrlHandlerMapping 329, 503–504, 585
 SimpleWsdl11Definition 371
 Since 448
`singAfter()` 29
`singBefore()` 29
 SingleConnectionDataSource 169
 singleton beans, loading 37
 singleton scoping option 65
 SOA. *See* service-oriented architecture (SOA)
 SoapFaultMappingExceptionResolver 367
 exceptionMappings property 368
 SoapServiceExporter 326
 software testing, Quality Assurance (QA) 679
 special beans 92, 106
 awareness 103, 106
 BeanNameAware interface 104–106
 BeanFactoryPostProcessor interface 95–96
 decoupling with application events 101–103
 externalizing configuration properties 96, 99
 postprocessing beans 93–95
 resolving text messages 99, 101
 Spring 4–6, 30
 basic example 11, 14
 data access exceptions 158, 161
 catch blocks 160–161
 persistence platform agnostic 159–160
 data access object (DAO) support classes 163, 165
 data access tiers 157, 165
 DAO support classes 163, 165
 exceptions 158, 161
 templating 161, 163
 directory structure 669
 downloading 668–669, 671
 integrating with 624, 632–633, 636, 642–643,
 648, 663, 665
 JavaDocs 669
 mixing with Enterprise JavaBeans (EJBs) 424
 modules available in Maven 2 repository 672
 participation in testing 684–685
 reference document 669
 templating 161, 163
 transaction management support 224–225
 website 668
 Spring 2 449–450
 Spring 2.0
 declaring transactions 241, 243
 JNDI data sources 166–167

- Spring AOP 122, 125
 advice 127
- Spring aspects 140–141
- Spring BeanDoc, autowiring 64
- Spring beans
- exporting as MBeans 467, 477
 - defining operations/attributes 472–473
 - exposing methods by name 471–472
 - metadata-driven 474, 476
 - object name collisions 476–477
 - loading into Tapestry pages 639, 641
 - managing with Java Management Extensions (JMX) 466, 484
 - exporting as MBeans 467, 477
 - handling notifications 482, 484
 - remoting MBeans 477, 482
 - resolving 646
 - using in JavaServer Faces (JSF) pages 646, 648
- wiring Enterprise JavaBeans (EJBs) into 430–431
- Spring IDE, autowiring 64
- Spring Model-View-Controller (MVC) 490, 502
 - building homepages 495–502
 - configuring DispatcherServlet 492–495
 - request lifecycles 491–492
- Spring Modules 210–212
- Spring MVC controllers 491–492
 - unit-testing 685, 687, 689–690, 692
- Spring MVC conversational applications 583
- spring object factory 635
- Spring Security 248, 304
 - access decisions managers 271–272, 275
 - voter abstinence 275
 - voting 272–274
 - authenticating users 252, 271
 - against databases 256, 264
 - against LDAP repositories 264, 271
 - ProviderManager 253, 256
 - authentication entry points 287
 - authentication providers 254
 - authentication-processing filters 287
 - credentials 250
 - elements 249
 - JSP tags 298
 - method invocations 300, 304
 - metadata 303–304
 - security aspects 301–302
 - password encoders 261
 - principals 250
 - servlet filters 276–278
 - order of configuration 284
 - view-layer security 297–300
- web security 275, 297
 - enforcing 293–294
 - prompting users to log in 286, 291
 - proxying filters 278, 284
 - secure channels 294, 297
 - security context 285–286
 - security exceptions 291, 293
- Spring Web Flow. *See* Web Flow
- spring.jar file 671
- <spring:message> 545, 547
- SpringBindingActionForm 620
- Spring-enabled Enterprise JavaBeans (EJB 2.x) 431, 434
- Spring-managed beans
 - accessing 659, 661–663
- Spring-OXM
 - marshaling options 365
- SpringRunner 704
- SpringTapestryEngine 637, 639
- Spring-WS 344–348, 353, 373, 382–383
 - message factory implementations 375
 - message senders 375
 - messages with service endpoints 353, 360
 - JDOM-based message endpoints 355, 357
 - marshaling message payloads 358, 360
 - service configuration 362
 - web service gateway support 381–382
 - web service templates 374, 380
 - marshallers on client side 379–380
 - sending messages 377, 379
 - wiring 361, 373
 - configuration 361, 363
 - deploying 373
 - endpoint exceptions 367, 369
 - mapping messages to endpoints 363–364
 - message marshalers 364, 367
 - service endpoints 364
 - WSDL files 369, 373
- SQL queries, mapping 205
- SQLException 175
- SqlMapClient interface 204
- SqlMapClientDaoSupport 207
- SqlMapClientFactoryBean 205
- SqlMapClientTemplate 204–205
- src/ directory 668
- standard MBeans 467
- start states 593–594
- starting
 - Quartz jobs 464
 - timers 459
- <start-state> 593–594

states 589–590
String parameter 176
StringArrayPropertyEditor 89
StringTrimmerEditor 89
Struts 624, 633
 actions 627, 629–632
 integrating Web Flow with 619–620
 registering Spring plug-in with 626–627
Struts 2 632–633, 636
subbeanning 74, 78
subflows 614–618
<subflow-state> 617
substates 614, 618
supports() 272
synchronous messaging 386, 390

T

tags, form-binding JavaServer Page (JSP) 543
Tapestry 636, 642
 loading Spring beans into Tapestry pages 639, 641
Tapestry 3 637, 641
Tapestry 4 641–642
targets 121
tasks
 scheduling 456, 466
 invoking methods on schedule 464, 466
 Java Timer class 457, 460
 Quartz scheduler 460, 464
timer
 creating 458
 scheduling 458–459
tearDown() 684
tearing down tests in JUnit 684
template classes 162
 HibernateTemplate 162
 JdbcTemplate 162, 173
 JpaTemplate class 162
 NamedParameterJdbcTemplate 173
 relationship to DAO support classes 164
 SimpleJdbcTemplate 174
template methods 161
template views, resolving 534–535, 537
templates
 Hibernate 186, 190
 annotated domain objects 187–188
 classic XML mapping files 186–187
iBATIS 204, 207
 SQL maps 205–206
 SqlMapClientTemplate 204–205

Java Message Service (JMS) 395, 402
 JmsTemplate 397
 receiving messages 400, 402
 sending messages 397, 399
 setting default destinations 399–400
Java Persistence API (JPA) 194, 197
JavaServer Page (JSP) 542, 549
 binding form data 542, 544
 displaying errors 547, 549
 rendering externalized messages 544, 547
JDBC 173, 180
 JdbcTemplate 174, 176
 named parameters 176, 178
 simplifying in Java 5 178, 180
transaction proxy 240–241
Velocity 557, 564
 binding form fields 561, 564
 configuring engines 558–559
 defining views 557–558
 exposing request/session attributes 561
 formatting dates/numbers 560–561
 resolving views 559–560
web services 374, 380
 marshalers on client side 379–380
 sending messages 377, 379
templating data access 161, 163
test methods 681
 setUp() 683
 tearDown() 684
testing 679, 705
 integration-testing 692–694, 696, 699, 702, 704
 JUnit 680, 682–684
 Spring participation in 684–685
 types of 680
 unit-testing Spring MVC controllers 685, 687, 689–690, 692
TestingAuthenticationProvider 254
 See also ProviderManager
text messages, resolving 99, 101
ThrowawayController 508, 528, 531
ThrowawayControllerHandlerAdapter 530
ThrowsAdvice interface 130–131
 afterThrowing() 130
tiger/ directory 668
<tile:insert> 550
Tiles 549, 556
 creating controllers 554, 556
 views 550, 554
 configuring 552
 resolving 553–554
TilesConfigurer 552
TilesJstlView 553

timeouts 237
Timer class 457, 460
 creating timer tasks 458
 delaying start of timers 459–460
 scheduling timer tasks 458–459
 starting timers 459
timer tasks, creating 458
TimerFactoryBean 457, 459
timers, starting 459
Tomcat 165
TopLinkJpaVendorAdapter property 201
Trang 350
transaction managers 225, 229
 Hibernate 227
 Java Data Objects (JDOs) 228–229
 Java Persistence API (JPA) 227–228
 Java Transaction API (JTA) 229
 JDBC 226–227
transactional message-driven POJOs (MDPs)
 411–412
transactional objects, integration-testing 696,
 698–699
transactional testing 684
transactionAttributes property 239
TransactionCallback interface 231
 doInTransaction() 232
 execute() 231
 setRollbackOnly() 231
TransactionDefinition 233
transactionManager property 412
TransactionProxyFactoryBean 238
transactions 221–225, 246
 atomicity 223
 consistency 223
 declaring 232, 245
 annotation-driven transactions 243, 245
 attributes 233, 237
 proxying 238, 241
 Spring 2.0 241, 243
 declarative 224
 described 221
 durability 223
 isolation 223
 managers 225
 programmatically adding 231
 programming 229, 232
 propagation behaviors 234
 read-only, propagation behaviors 237
 timeouts, propagation behaviors 237
transaction managers 225, 229
 Hibernate 227
 Java Data Objects (JDOs) 228–229

Java Persistence API (JPA) 227–228
Java Transaction API (JTA) 229
 JDBC 226–227
<transition> 608
transitions 590
Trigger class 461
tx namespace 241
<tx:advice> 242
<tx:annotation-driven> 244
<tx:attribute> 242
<tx:method> 242
txProxyTemplate 240

U

UnanimousBased wiring 272
unit testing 16–17, 445
Unmarshaller interface 364
update() 175
updateTable 657
updating rows in databases, JDBC 171
url property 167
URLEditor 89
UrlFilenameViewController 508
UserCache interface 263
UserDetailsService interface 257
userDetailsService property 257
userDnPatterns property 266
username property 168
usersByUserNameQuery property 260

V

validate() 528
validatePage() 527
validating form input 517
validation 8
Validator interface 515, 517–518
Validator object
 validate() 528
 validatePage() 527
validatorFactory bean 518
value attribute element 44
values, injecting 47–48
varargs 178, 182
 query() 180
Velocity macros 562
Velocity templates 557, 564
 binding form fields 561, 564
 configuring engines 558–559
 defining views 557–558

Velocity templates (*continued*)

- exposing
 - request attributes 561
 - session attributes 561
- formatting
 - dates 560–561
 - numbers 560–561
- resolving views 559–560

VelocityConfigurer bean 558–559

VelocityViewResolver bean 559, 561

- dateToolAttribute 560
- exposeRequestAttributes 561
- exposeSessionAttributes 561
- exposeSpringMacroHelpers 563
- numberToolAttribute 560

view beans

- resolving 537, 540
 - declaring in separate XML files 538
 - from resource bundles 539–540

view resolvers

- choosing 540–541
- declaring 499–500
- using multiple 540–541

viewClass property 537

view-layer security 297, 300

- conditionally rendering content 298–299
- displaying user authentication information 299–300

ViewResolver implementations 535

views 534, 556, 579

- developing custom 576, 578

FreeMarker 564–569

- binding form fields 567, 569
- configuring engines 565–566
- defining views 564–565
- exposing request/session attributes 566–567
- resolving views 566

JavaServer Page (JSP) templates 542, 549

- binding form data 542, 544

displaying errors 547, 549

rendering externalized messages 544, 547

non-HTML output 569, 578

- developing custom views 576, 578

Excel spreadsheets 570, 573

PDF documents 573, 575

resolving 534, 541

- choosing view resolvers 540–541

FreeMarker 566

resolving view beans 537, 540

template views 535, 537

Velocity 559–560

Tiles 549–550, 554, 556

- creating controllers 554, 556

Velocity templates 557, 564

- binding form fields 561, 564

configuring engines 558–559

defining views 557–558

exposing request/session attributes 561

formatting dates/numbers 560–561

resolving views 559–560

<view-state> 595, 598, 602, 604–605

voting access decisions 272–274

W

weaving 121–122

web containers, initializing servlet listeners before servlets 494

Web Flow 581–622

advanced techniques 611, 618

decision states 612, 614

extracting subflows 614, 618

substates 614, 618

creating flows 591, 611

building orders 601, 605

completing orders 605, 608

end states 593–594

flow variables 591, 593

gathering customer information 594, 601

start states 593–594

installing 584, 588

configuring flow executors 586–587

FlowController 585–586

registering flow definitions 588

integrating with other frameworks 619, 622

Jakarta Struts 619–620

JavaServer Faces 620, 622

main elements 589–590

web module 10

web requests 490, 532

handling exceptions 531–532

handling requests with controllers 506, 509,

511–512, 520, 528, 531

mapping requests to controllers 502–506

Spring Model-View-Controller (MVC) 490–492, 495, 502

web security 275, 297

enforcing 293–294

prompting users to log in 286, 291

basic authentication 287, 289

form-based authentication 289, 291

- web security (*continued*)
 proxying filters 278, 284
 configuring proxies 283–284
 multiple filters 281–282
 servlet filters 279, 281
 secure channels 294, 297
 channel decisions 297
 security context 285–286
 security exceptions 291–293
web service templates 374, 380
 marshalers on client side 379–380
 sending messages 377, 379
web services 326, 333, 340–341
 contract-first 344, 373, 382–383
 defined 347, 353
 messages with service endpoints 353, 360
 overview 344–345, 347
 web service gateway support 381–382
 web service templates 374, 380
 wiring 361, 373
JaxRpcPortProxyFactoryBean 334, 337
JSR-181 annotations 330, 333
 mapping requests to JSR-181 annotated beans 332–333
mapping arrays 338, 340
mapping complex types 337–338
proxying with XFire client 340–341
XFire 326, 330
 DispatcherServlet 329
 mapping requests to XFireExporter 329–330
 XFireExporter 328–329
Web Services Metadata for the Java Platform (JSR-181), annotations 330
web views 534, 556, 579
 FreeMarker 564, 569
 binding form fields 567, 569
 configuring engines 565–566
 defining views 564–565
 exposing request/session attributes 566–567
 resolving views 566
JavaServer Page (JSP) templates 542, 549
 binding form data 542, 544
 displaying errors 547, 549
 rendering externalized messages 544, 547
non-HTML output 569, 578
 developing custom views 576, 578
Excel spreadsheets 570, 573
PDF documents 573, 575
resolving 534, 541
 choosing view resolvers 540–541
 resolving view beans 537, 540
 template views 535, 537
Tiles 549–550, 554, 556
 creating controllers 554, 556
Velocity templates 557, 564
 binding form fields 561, 564
 configuring engines 558–559
 defining views 557–558
 exposing request/session attributes 561
 formatting dates/numbers 560–561
 resolving views 559–560
web.xml files, FilterToBeanProxy 283
webApplicationContext 648
WebApplicationContextUtils 627
WebApplicationContextVariableResolver 621, 648, 664
web-layer code 494
WebServiceTemplate 373
 marshalSendAndReceive() 379
 sendAndReceive() 377
WebSphere 165
WebWork 2 633, 636
wired objects
 integration-testing 694–696
wiring 20–21
 default destinations into JmsTemplate 399–400
 defined 32
Enterprise JavaBeans 425, 431
 into Spring beans 430–431
 proxying session beans (EJB 2.x) 426, 430
Java Naming and Directory Interface (JNDI)
 objects 442, 450
 in Spring 2 449–450
JmsTemplate 397
mail senders into service beans 453
nothing (null) 58
Remote Method Invocation (RMI) services 310, 312
Spring-WS services 361, 373
 configuration 361, 363
 deploying 373
 endpoint exceptions 367, 369
 mapping messages to endpoints 363–364
 message marshalers 364, 367
 service endpoints 364
 WSDL files 369, 373
Struts actions in Spring 630–631
wiring beans 32, 71, 73, 115
 auto-wiring 58, 64
 mixing with explicit wiring 63
 shortcomings 63–64
 types 59, 62

- wiring beans (*continued*)
 containing 33, 40
 ApplicationContext interface 35, 40
 BeanFactory interface 34–35
 controlling creation 64, 71
 creating from factory methods 66, 68
 destroying 68, 71
 initializing 68, 71
 scoping 66
 creating 40, 45
 declaring 40–41
 injecting through constructors 42, 45
 declaring child beans 73, 78
 abstracting base bean type 74, 76
 abstracting common properties 76, 78
 declaring parent beans 73, 78
 abstracting base bean type 74, 76
 abstracting common properties 76, 78
injecting non-Spring beans 85, 88
injecting properties 46, 58
 injecting simple values 47–48
 referencing other beans 48, 52
 wiring nothing (null) 58
method injection 79, 85
 getter injection 83, 85
 method replacement 80, 83
registering custom property editors 88, 92
scripting beans 106, 114
 example 107–108
 injecting properties 111–112
 refreshing 112–113
 scripting bean 108, 111
 writing inline 113–114
special beans 92, 106
 awareness 103, 106
- BeanFactoryPostProcessor interface 95–96
BeanPostProcessor interface 93, 95
decoupling with application events 101, 103
externalizing configuration properties 96, 99
 resolving text messages 99, 101
wizard controllers, building 521, 524
wizards
 canceling 526
 finishing 525–526
 processing forms with 520–521, 524–526, 528
 validating forms 526, 528
WSDL files 369, 373
 predefined WSDL 371, 373
-
- X**
- XFire 326, 330
 DispatcherServlet 329
XFireClientFactoryBean 334, 340
XFireExporter 327–329
 mapping requests to 329–330
XML files
 declaring view beans in separate 538
 splitting application context across multiple 493
XML mapping files, Hibernate 186–187
XML messages 348, 353
 forging data contracts 349, 353
XML Schema (XSD) 349
XmlBeanFactory 21
 creating 34
XmlFileViewResolver 538, 540–541
XmlFlowRegistryFactoryBean 588
XMLHttpRequest 649
XmlWebApplicationContext 36