

Join the discussion @ p2p.wrox.com

INSIDE: Free 3-Month IntelliJ
IDEA Personal License



Professional Java® for Web Applications

Featuring WebSockets, Spring Framework, JPA Hibernate,
and Spring Security

Nicholas S. Williams

PROFESSIONAL

Java® for Web Applications

Nicholas S. Williams



Professional Java® for Web Applications

Published by

John Wiley & Sons, Inc.
10475 Crosspoint Boulevard
Indianapolis, IN 46256
www.wiley.com

Copyright © 2014 by John Wiley & Sons, Inc., Indianapolis, Indiana

Published simultaneously in Canada

ISBN: 978-1-118-65646-4

ISBN: 978-1-118-65651-8 (ebk)

ISBN: 978-1-118-90931-7 (ebk)

Manufactured in the United States of America

10 9 8 7 6 5 4 3 2 1

No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, except as permitted under Sections 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, 222 Rosewood Drive, Danvers, MA 01923, (978) 750-8400, fax (978) 646-8600. Requests to the Publisher for permission should be addressed to the Permissions Department, John Wiley & Sons, Inc., 111 River Street, Hoboken, NJ 07030, (201) 748-6011, fax (201) 748-6008, or online at <http://www.wiley.com/go/permissions>.

Limit of Liability/Disclaimer of Warranty: The publisher and the author make no representations or warranties with respect to the accuracy or completeness of the contents of this work and specifically disclaim all warranties, including without limitation warranties of fitness for a particular purpose. No warranty may be created or extended by sales or promotional materials. The advice and strategies contained herein may not be suitable for every situation. This work is sold with the understanding that the publisher is not engaged in rendering legal, accounting, or other professional services. If professional assistance is required, the services of a competent professional person should be sought. Neither the publisher nor the author shall be liable for damages arising herefrom. The fact that an organization or Web site is referred to in this work as a citation and/or a potential source of further information does not mean that the author or the publisher endorses the information the organization or Web site may provide or recommendations it may make. Further, readers should be aware that Internet Web sites listed in this work may have changed or disappeared between when this work was written and when it is read.

For general information on our other products and services please contact our Customer Care Department within the United States at (877) 762-2974, outside the United States at (317) 572-3993 or fax (317) 572-4002.

Wiley publishes in a variety of print and electronic formats and by print-on-demand. Some material included with standard print versions of this book may not be included in e-books or in print-on-demand. If this book refers to media such as a CD or DVD that is not included in the version you purchased, you may download this material at <http://booksupport.wiley.com>. For more information about Wiley products, visit www.wiley.com.

Library of Congress Control Number: 2013958292

Trademarks: Wiley, Wrox, the Wrox logo, Programmer to Programmer, and related trade dress are trademarks or registered trademarks of John Wiley & Sons, Inc. and/or its affiliates, in the United States and other countries, and may not be used without written permission. Java is a registered trademark of Oracle America, Inc. All other trademarks are the property of their respective owners. John Wiley & Sons, Inc., is not associated with any product or vendor mentioned in this book.

ABOUT THE AUTHOR



NICK WILLIAMS is a Software Engineer for UL Workplace Health and Safety in Franklin, Tennessee. A computer science graduate from Belmont University, he has been active in commercial and open source software projects for more than 9 years. He is the founder of DNSCrawler.com, a site for free DNS and IP troubleshooting tools, and NWTS Java Code, an open source community that specializes in obscure Java libraries that meet niche needs. In 2010, the Nashville Technology Council named him the Software Engineer of the Year for Middle Tennessee. Nick is a committer for Apache Logging (including Log4j) and Jackson Data Processor JSR 310 Data Types. He has also contributed new features to Apache Tomcat 8.0, Spring Framework 4.0, Spring Security 3.2, Spring Data Commons 1.6, Spring Data JPA 1.4, and JBoss Logging 3.2; serves as a contributor on several other projects, including OpenJDK; and is a member of the Java Community Process (JCP).

Nick currently lives in Tennessee with his wife Allison. You can find him on Twitter @Java_Nick.

ABOUT THE TECHNICAL EDITORS

JAKE RADAKOVICH joined UL Workplace Health and Safety in 2009, and currently serves as Software Developer on the Occupational Health Manager product. Prior to that, he was a research assistant at Middle Tennessee State University working on AlgoTutor, a web-based algorithm development tutoring system. He holds a BS in Computer Science and Mathematics from Middle Tennessee State University. You can follow Jake on Twitter @JakeRadakovich.

MANUEL JORDAN ELERA is an autodidactic developer and researcher who enjoys learning new technologies for his own experiments and creating new integrations. He won the 2010 Springy Award and was a Community Champion and Spring Champion in 2013. In his little free time, he reads the Bible and composes music on his guitar. Manuel is a Senior Member in the Spring Community Forums known as dr_pompeii. You can read about him and contact him through his blog and you can follow him on his Twitter account, @dr_pompeii.

CREDITS

ACQUISITIONS EDITOR
Mary James

PROJECT EDITOR
Maureen Spears Tullis

TECHNICAL EDITORS
Michael Jordan Elera
Jake Radakovich

TECHNICAL PROOFREADER
Jonathan Giles

SENIOR PRODUCTION EDITOR
Kathleen Wisor

COPY EDITOR
Apostrophe Editing Services

EDITORIAL MANAGER
Mary Beth Wakefield

FREELANCER EDITORIAL MANAGER
Rosemarie Graham

ASSOCIATE DIRECTOR OF MARKETING
David Mayhew

MARKETING MANAGER
Ashley Zurcher

BUSINESS MANAGER
Amy Kries

VICE PRESIDENT AND EXECUTIVE GROUP
PUBLISHER
Richard Swadley

ASSOCIATE PUBLISHER
Jim Minatel

PROJECT COORDINATOR, COVER
Todd Klemme

PROOFREADERS
Nancy Carrasco
Josh Chase, Word One

INDEXER
Robert Swanson

COVER DESIGNER
Wiley

COVER IMAGE
iStockphoto.com/ElementalImaging

ACKNOWLEDGMENTS

THANKS TO...

My wife Allison, whose unwavering support and persistent reminders about deadlines during this stressful year made this book possible.

My parents and siblings, who told me that I could do anything I put my mind to.

Drs. Joyce Blair Crowell and William Hooper, whose dedicated instruction and mentoring made my career possible.

Dr. Sarah Ann Stewart, who believed in me when I thought surely calculus and proofs spelled doom for my education.

Mrs. Lockhart, who inspired me to write.

Jay, for introducing me to Mary, and to Mary and Maureen for making this book a reality.

Jake, for being absurd. Oh, and for agreeing to be my technical editor.

CONTENTS

INTRODUCTION

xxiii

PART I: CREATING ENTERPRISE APPLICATIONS

<u>CHAPTER 1: INTRODUCING JAVA PLATFORM, ENTERPRISE EDITION</u>	<u>3</u>
A Timeline of Java Platforms	3
In the Beginning	4
The Birth of Enterprise Java	5
Java SE and Java EE Evolving Together	6
Understanding the Most Recent Platform Features	9
A Continuing Evolution	13
Understanding the Basic Web Application Structure	13
Servlets, Filters, Listeners, and JSPs	13
Directory Structure and WAR Files	14
The Deployment Descriptor	15
Class Loader Architecture	16
Enterprise Archives	17
Summary	18
<u>CHAPTER 2: USING WEB CONTAINERS</u>	<u>19</u>
Choosing a Web Container	19
Apache Tomcat	20
GlassFish	21
JBoss and WildFly	22
Other Containers and Application Servers	22
Why You'll Use Tomcat in This Book	23
Installing Tomcat on Your Machine	23
Installing as a Windows Service	24
Installing as a Command-Line Application	24
Configuring a Custom JSP Compiler	26
Deploying and Undeploying Applications in Tomcat	27
Performing a Manual Deploy and Undeploy	28
Using the Tomcat Manager	28

Debugging Tomcat from Your IDE	30
Using IntelliJ IDEA	30
Using Eclipse	35
Summary	39
CHAPTER 3: WRITING YOUR FIRST SERVLET	41
Creating a Servlet Class	42
What to Extend	42
Using the Initializer and Destroyer	45
Configuring a Servlet for Deployment	46
Adding the Servlet to the Descriptor	46
Mapping the Servlet to a URL	47
Running and Debugging Your Servlet	49
Understanding doGet(), doPost(), and Other Methods	51
What Should Happen during the service Method Execution?	51
Using HttpServletRequest	52
Using HttpServletResponse	55
Using Parameters and Accepting Form Submissions	56
Configuring your Application Using Init Parameters	61
Using Context Init Parameters	61
Using Servlet Init Parameters	62
Uploading Files from a Form	64
Introducing the Customer Support Project	65
Configuring the Servlet for File Uploads	65
Accepting a File Upload	68
Making Your Application Safe for Multithreading	69
Understanding Requests, Threads, and Method Execution	69
Protecting Shared Resources	70
Summary	71
CHAPTER 4: USING JSPS TO DISPLAY CONTENT	73

 Is Easier Than output.println("
")	74
Why JSPs Are Better	75
What Happens to a JSP at Run Time	76
Creating Your First JSP	78
Understanding the File Structure	78
Directives, Declarations, Scriptlets, and Expressions	79
Commenting Your Code	81
Adding Imports to Your JSP	82

Using Directives	83
Using the <jsp> Tag	86
Using Java within a JSP (and Why You Shouldn't!)	88
Using the Implicit Variables in a JSP	88
Why You Shouldn't Use Java in a JSP	93
Combining Servlets and JSPs	94
Configuring JSP Properties in the Deployment Descriptor	94
Forwarding a Request from a Servlet to a JSP	97
A Note about JSP Documents (JSPX)	102
Summary	104
CHAPTER 5: MAINTAINING STATE USING SESSIONS	105
Understanding Why Sessions Are Necessary	106
Maintaining State	106
Remembering Users	107
Enabling Application Workflow	107
Using Session Cookies and URL Rewriting	107
Understanding the Session Cookie	108
Session IDs in the URL	110
Session Vulnerabilities	112
Storing Data in a Session	116
Configuring Sessions in the Deployment Descriptor	116
Storing and Retrieving Data	119
Removing Data	123
Storing More Complex Data in Sessions	125
Applying Sessions Usefully	129
Adding Login to the Customer Support Application	129
Detecting Changes to Sessions Using Listeners	133
Maintaining a List of Active Sessions	135
Clustering an Application That Uses Sessions	139
Using Session IDs in a Cluster	139
Understand Session Replication and Failover	141
Summary	142
CHAPTER 6: USING THE EXPRESSION LANGUAGE IN JSPS	143
Understanding Expression Language	144
What It's For	144
Understanding the Base Syntax	145
Placing EL Expressions	146
Writing with the EL Syntax	147

Reserved Keywords	148
Operator Precedence	148
Object Properties and Methods	154
EL Functions	155
Static Field and Method Access	156
Enums	157
Lambda Expressions	157
Collections	158
Using Scoped Variables in EL Expressions	160
Using the Implicit EL Scope	161
Using the Implicit EL Variables	165
Accessing Collections with the Stream API	167
Understanding Intermediate Operations	168
Using Terminal Operations	170
Putting the Stream API to Use	171
Replacing Java Code with Expression Language	172
Summary	175
CHAPTER 7: USING THE JAVA STANDARD TAG LIBRARY	177
Introducing JSP Tags and the JSTL	178
Working with Tags	178
Using the Core Tag Library (C Namespace)	182
<c:out>	182
<c:url>	183
<c:if>	184
<c:choose>, <c:when>, and <c:otherwise>	185
<c:forEach>	186
<c:forTokens>	187
<c:redirect>	188
<c:import>	188
<c:set> and <c:remove>	189
Putting Core Library Tags to Use	190
Using the Internationalization and Formatting Tag Library (FMT Namespace)	193
Internationalization and Localization Components	193
<fmt:message>	194
<fmt:setLocale>	196
<fmt:bundle> and <fmt:setBundle>	196
<fmt:requestEncoding>	197
<fmt:TimeZone> and <fmt:setTimeZone>	197
<fmt:formatDate> and <fmt:parseDate>	198
<fmt:formatNumber> and <fmt:parseNumber>	199

Putting i18n and Formatting Library Tags to Use	200
Using the Database Access Tag Library (SQL Namespace)	203
Using the XML Processing Tag Library (X Namespace)	205
Replacing Java Code with JSP Tags	205
Summary	208
CHAPTER 8: WRITING CUSTOM TAG AND FUNCTION LIBRARIES	209
Understanding TLDs, Tag Files, and Tag Handlers	210
Reading the Java Standard Tag Library TLD	211
Comparing JSP Directives and Tag File Directives	217
Creating Your First Tag File to Serve as an HTML Template	219
Creating a More Useful Date Formatting Tag Handler	221
Creating an EL Function to Abbreviate Strings	226
Replacing Java Code with Custom JSP Tags	227
Summary	232
CHAPTER 9: IMPROVING YOUR APPLICATION USING FILTERS	233
Understanding the Purpose of Filters	234
Logging Filters	234
Authentication Filters	234
Compression and Encryption Filters	234
Error Handling Filters	235
Creating, Declaring, and Mapping Filters	235
Understanding the Filter Chain	235
Mapping to URL Patterns and Servlet Names	236
Mapping to Different Request Dispatcher Types	236
Using the Deployment Descriptor	237
Using Annotations	238
Using Programmatic Configuration	238
Ordering Your Filters Properly	239
URL Pattern Mapping versus Servlet Name Mapping	239
Exploring Filter Order with a Simple Example	241
Using Filters with Asynchronous Request Handling	243
Investigating Practical Uses for Filters	247
Adding a Simple Logging Filter	248
Compressing Response Content Using a Filter	249
Simplifying Authentication with a Filter	254
Summary	255

CHAPTER 10: MAKING YOUR APPLICATION INTERACTIVE WITH WEB SOCKETS	257
Evolution: From Ajax to WebSockets	258
Problem: Getting New Data from the Server to the Browser	259
Solution 1: Frequent Polling	259
Solution 2: Long Polling	260
Solution 3: Chunked Encoding	262
Solution 4: Applets and Adobe Flash	263
WebSockets: The Solution Nobody Knew Kind of Already Existed	264
Understanding the WebSocket APIs	268
HTML5 (JavaScript) Client API	268
Java WebSocket APIs	270
Creating Multiplayer Games with WebSockets	273
Implementing the Basic Tic-Tac-Toe Algorithm	274
Creating the Server Endpoint	274
Writing the JavaScript Game Console	278
Playing WebSocket Tic-Tac-Toe	283
Using WebSockets to Communicate in a Cluster	284
Simulating a Simple Cluster Using Two Servlet Instances	284
Transmitting and Receiving Binary Messages	286
Testing the Simulated Cluster Application	287
Adding "Chat with Support" to the Customer Support Application	288
Using Encoders and Decoders to Translate Messages	289
Creating the Chat Server Endpoint	291
Writing the JavaScript Chat Application	294
Summary	296
CHAPTER 11: USING LOGGING TO MONITOR YOUR APPLICATION	297
Understanding the Concepts of Logging	298
Why You Should Log	298
What Content You Might Want to See in Logs	299
How Logs Are Written	301
Using Logging Levels and Categories	303
Why Are There Different Logging Levels?	303
Logging Levels Defined	303
How Logging Categories Work	304
How Log Sifting Works	305
Choosing a Logging Framework	305
API versus Implementation	305

Performance	306
A Quick Look at Apache Commons Logging and SLF4J	307
Introducing Log4j 2	307
Integrating Logging into Your Application	312
Creating the Log4j 2 Configuration Files	313
Utilizing Fish Tagging with a Web Filter	316
Writing Logging Statements in Java Code	317
Using the Log Tag Library in JSPs	319
Logging in the Customer Support Application	319
Summary	320
PART II: ADDING SPRING FRAMEWORK INTO THE MIX	
CHAPTER 12: INTRODUCING SPRING FRAMEWORK	323
What Is Spring Framework?	324
Inversion of Control and Dependency Injection	325
Aspect-Oriented Programming	325
Data Access and Transaction Management	325
Application Messaging	326
Model-View-Controller Pattern for Web Applications	326
Why Spring Framework?	326
Logical Code Groupings	326
Multiple User Interfaces Utilizing One Code Base	327
Understanding Application Contexts	327
Bootstrapping Spring Framework	329
Using the Deployment Descriptor to Bootstrap Spring	330
Programmatically Bootstrapping Spring in an Initializer	332
Configuring Spring Framework	336
Creating an XML Configuration	338
Creating a Hybrid Configuration	340
Configuring Spring with Java Using @Configuration	345
Utilizing Bean Definition Profiles	349
Understanding How Profiles Work	350
Considering Antipatterns and Security Concerns	352
Summary	353
CHAPTER 13: REPLACING YOUR SERVLETS WITH CONTROLLERS	355
Understanding @RequestMapping	356
Using @RequestMapping Attributes to Narrow Request Matching	356
Specifying Controller Method Parameters	360
Selecting Valid Return Types for Controller Methods	368

Using Spring Framework's Model and View Pattern	370
Using Explicit Views and View Names	371
Using Implicit Views with Model Attributes	373
Returning Response Entities	375
Making Your Life Easier with Form Objects	380
Adding the Form Object to Your Model	381
Using the Spring Framework <form> Tags	381
Obtaining Submitted Form Data	383
Updating the Customer Support Application	384
Enabling Multipart Support	384
Converting Servlets to Spring MVC Controllers	385
Creating a Custom Downloading View	386
Summary	387
CHAPTER 14: USING SERVICES AND REPOSITORIES TO SUPPORT YOUR CONTROLLERS	389
Understanding Model-View-Controller Plus Controller-Service- Repository	390
Recognizing Different Types of Program Logic	391
Repositories Provide Persistence Logic	392
Services Provide Business Logic	392
Controllers Provide User Interface Logic	393
Using the Root Application Context	394
Instead of a Web Application Context	394
Reusing the Root Application Context for Multiple User Interfaces	394
Moving Your Business Logic from Controllers to Services	396
Using Repositories for Data Storage	399
Improving Services with Asynchronous and Scheduled Execution	404
Understanding Executors and Schedulers	404
Configuring a Scheduler and Asynchronous Support	405
Creating and Using @Async Methods	407
Creating and Using @Scheduled Methods	408
Applying Logic Layer Separation to WebSockets	409
Adding Container-Managed Objects to the Spring Application Context	409
Using the Spring WebSocket Configurator	411
Remember: A WebSocket Is Just Another Interface for Business Logic	412
Summary	416

CHAPTER 15: INTERNATIONALIZING YOUR APPLICATION WITH SPRING FRAMEWORK I18N	417
Why Do You Need Spring Framework i18n?	418
Making Internationalization Easier	418
Localizing Error Messages Directly	418
Using the Basic Internationalization and Localization APIs	419
Understanding Resource Bundles and Message Formats	419
Message Sources to the Rescue	421
Using Message Sources to Internationalize JSPs	422
Configuring Internationalization in Spring Framework	424
Creating a Message Source	424
Understanding Locale Resolvers	425
Using a Handler Interceptor to Change Locales	427
Providing a User Profile Locale Setting	428
Including Time Zone Support	429
Understanding How Themes Can Improve Internationalization	429
Internationalizing Your Code	430
Using the <spring:message> Tag	431
Handling Application Errors Cleanly	433
Updating the Customer Support Application	436
Using the Message Source Directly	437
Summary	440
CHAPTER 16: USING JSR 349, SPRING FRAMEWORK, AND HIBERNATE VALIDATOR FOR BEAN VALIDATION	441
What Is Bean Validation?	442
Why Hibernate Validator?	444
Understanding the Annotation Metadata Model	444
Using Bean Validation with Spring Framework	445
Configuring Validation in the Spring Framework Container	445
Configuring the Spring Validator Bean	446
Setting Up Error Code Localization	448
Using a Method Validation Bean Post-Processor	449
Making Spring MVC Use the Same Validation Beans	450
Adding Constraint Validation Annotations to Your Beans	450

Understanding the Built-in Constraint Annotations	451
Understanding Common Constraint Attributes	452
Putting Constraints to Use	452
Using @Valid for Recursive Validation	454
Using Validation Groups	455
Checking Constraint Legality at Compile-Time	457
Configuring Spring Beans for Method Validation	458
Annotating Interfaces, Not Implementations	458
Using Constraints and Recursive Validation on Method Parameters	459
Validating Method Return Values	459
Indicating That a Class Is Eligible for Method Validation	460
Using Parameter Validation in Spring MVC Controllers	462
Displaying Validation Errors to the User	463
Writing Your Own Validation Constraints	466
Inheriting Other Constraints in a Custom Constraint	466
Creating a Constraint Validator	467
Understanding the Constraint Validator Life Cycle	469
Integrating Validation in the Customer Support Application	470
Summary	472
<hr/> CHAPTER 17: CREATING RESTFUL AND SOAP WEB SERVICES	<hr/> 473
Understanding Web Services	474
In the Beginning There Was SOAP	475
RESTful Web Services Provide a Simpler Approach	476
Configuring RESTful Web Services with Spring MVC	484
Segregating Controllers with Stereotype Annotations	484
Creating Separate Web and REST Application Contexts	485
Handling Error Conditions in RESTful Web Services	488
Mapping RESTful Requests to Controller Methods	491
Improving Discovery with an Index Endpoint	495
Testing Your Web Service Endpoints	496
Choosing a Testing Tool	497
Making Requests to Your Web Service	497
Using Spring Web Services for SOAP	500
Writing Your Contract-First XSD and WSDL	501
Adding the SOAP Dispatcher Servlet Configuration	503
Creating a SOAP Endpoint	504
Summary	508

CHAPTER 18: USING MESSAGING AND CLUSTERING FOR FLEXIBILITY AND RELIABILITY	509
Recognizing When You Need Messaging and Clustering	510
What Is Application Messaging?	510
What Is Clustering?	513
How Do Messaging and Clustering Work Together?	517
Adding Messaging Support to your Application	520
Creating Application Events	520
Subscribing to Application Events	522
Publishing Application Events	523
Making your Messaging Distributable	
Across a Cluster	525
Updating Your Events to Support Distribution	526
Creating and Configuring a Custom Event Multicaster	527
Using WebSockets to Send and Receive Events	529
Discovering Nodes with Multicast Packets	531
Simulating a Cluster with Multiple Deployments	533
Distributing Events with AMQP	534
Configuring an AMQP Broker	536
Creating an AMQP Multicaster	537
Running the AMQP-Enabled Application	539
Summary	540
PART III: PERSISTING DATA WITH JPA AND HIBERNATE ORM	
CHAPTER 19: INTRODUCING JAVA PERSISTENCE API AND HIBERNATE ORM	543
What Is Data Persistence?	543
Flat-File Entity Storage	544
Structured File Storage	544
Relational Database Systems	545
Object-Oriented Databases	546
Schema-less Database Systems	546
What Is an Object-Relational Mapper?	547
Understanding the Problem of Persisting Entities	547
O/RMs Make Entity Persistence Easier	549
JPA Provides a Standard O/RM API	550
Why Hibernate ORM?	552
A Brief Look at Hibernate ORM	552
Using Hibernate Mapping Files	552

Understanding the Session API	554
Getting a Session from the SessionFactory	556
Creating a SessionFactory with Spring Framework	557
Preparing a Relational Database	559
Installing MySQL and MySQL Workbench	559
Installing the MySQL JDBC Driver	562
Creating a Connection Resource in Tomcat	563
A Note About Maven Dependencies	564
Summary	564
CHAPTER 20: MAPPING ENTITIES TO TABLES WITH JPA ANNOTATIONS	565
Getting Started with Simple Entities	566
Marking an Entity and Mapping It to a Table	567
Indicating How JPA Uses Entity Fields	569
Mapping Surrogate Keys	570
Using Basic Data Types	576
Specifying Column Names and Other Details	579
Creating and Using a Persistence Unit	581
Designing the Database Tables	581
Understanding Persistence Unit Scope	583
Creating the Persistence Configuration	584
Using the Persistence API	586
Mapping Complex Data Types	590
Using Enums as Entity Properties	590
Understanding How JPA Handles Dates and Times	592
Mapping Large Properties to CLOBs and BLOBs	594
Summary	596
CHAPTER 21: USING JPA IN SPRING FRAMEWORK REPOSITORIES	597
Using Spring Repositories and Transactions	598
Understanding Transaction Scope	598
Using Threads for Transactions and Entity Managers	599
Taking Advantage of Exception Translation	601
Configuring Persistence in Spring Framework	602
Looking Up a Data Source	602
Creating a Persistence Unit in Code	603
Setting Up Transaction Management	607
Creating and Using JPA Repositories	610
Injecting the Persistence Unit	610
Implementing Standard CRUD Operations	611

Creating a Base Repository for All Your Entities	613
Demarking Transaction Boundaries in Your Services	618
Using the Transactional Service Methods	622
Converting Data with DTOs and Entities	624
Creating Entities for the Customer Support Application	624
Securing User Passwords with BCrypt	628
Transferring Data to Entities in Your Services	630
Summary	632
CHAPTER 22: ELIMINATING BOILERPLATE REPOSITORIES WITH SPRING DATA JPA	633
Understanding Spring Data's	
Unified Data Access	634
Avoiding Duplication of Code	634
Using the Stock Repository Interfaces	638
Creating Query Methods for Finding Entities	639
Providing Custom Method Implementations	642
Configuring and Creating Spring	
Data JPA Repositories	646
Enabling Repository Auto-Generation	646
Writing and Using Spring Data JPA Interfaces	654
Refactoring the Customer Support Application	656
Converting the Existing Repositories	656
Adding Comments to Support Tickets	657
Summary	661
CHAPTER 23: SEARCHING FOR DATA WITH JPA AND HIBERNATE SEARCH	663
An Introduction to Searching	
Understanding the Importance of Indexes	664
Taking Three Different Approaches	666
Using Advanced Criteria to Locate Objects	666
Creating Complex Criteria Queries	667
Using OR in Your Queries	674
Creating Useful Indexes to Improve Performance	676
Taking Advantage of Full-Text Indexes with JPA	676
Creating Full-Text Indexes in MySQL Tables	677
Creating and Using a Searchable Repository	678
Making Full-Text Searching Portable	684
Indexing Any Data with Apache Lucene and Hibernate Search	684

Understanding Lucene Full-Text Indexing	685
Annotating Entities with Indexing Metadata	686
Using Hibernate Search with JPA	688
Summary	692
CHAPTER 24: CREATING ADVANCED MAPPINGS AND CUSTOM DATA TYPES	693
What's Left?	694
Converting Nonstandard Data Types	695
Understanding Attribute Converters	695
Understanding the Conversion Annotations	696
Creating and Using Attribute Converters	698
Embedding POJOs Within Entities	699
Indicating That a Type Is Embeddable	699
Marking a Property as Embedded	700
Overriding Embeddable Column Names	702
Defining Relationships Between Entities	703
Understanding One-to-One Relationships	703
Using One-to-Many and Many-to-One Relationships	705
Creating Many-to-Many Relationships	708
Addressing Other Common Situations	709
Versioning Entities with Revisions and Timestamps	709
Defining Abstract Entities with Common Properties	710
Mapping Basic and Embedded Collections	712
Persisting a Map of Key-Value Pairs	715
Storing an Entity in Multiple Tables	716
Creating Programmatic Triggers	717
Acting before and after CRUD Operations	717
Using Entity Listeners	719
Refining the Customer Support Application	720
Mapping a Collection of Attachments	721
Lazy Loading Simple Properties with Load Time Weaving	723
Summary	725
PART IV: SECURING YOUR APPLICATION WITH SPRING SECURITY	
CHAPTER 25: INTRODUCING SPRING SECURITY	729
What Is Authentication?	729
Integrating Authentication	730
Understanding Authorization	740
Why Spring Security?	743

Understanding the Spring Security Foundation	744
Using Spring Security's Authorization Services	745
Configuring Spring Security	745
Summary	746
CHAPTER 26: AUTHENTICATING USERS WITH SPRING SECURITY	747
Choosing and Configuring an Authentication Provider	748
Configuring a User Details Provider	748
Working with LDAP and Active Directory Providers	759
Authenticating with OpenID	762
Remembering Users	765
Exploring Other Authentication Providers	766
Writing Your Own Authentication Provider	766
Bootstrapping in the Correct Order	767
Creating and Configuring a Provider	769
Mitigating Cross-Site Request Forgery Attacks	775
Summary	778
CHAPTER 27: USING AUTHORIZATION TAGS AND ANNOTATIONS	779
Authorizing by Declaration	780
Checking Permissions in Method Code	780
Employing URL Security	783
Using Annotations to Declare Permissions	786
Defining Method Pointcut Rules	794
Understanding Authorization Decisions	794
Using Access Decision Voters	795
Using Access Decision Managers	796
Creating Access Control Lists for Object Security	798
Understanding Spring Security ACLs	798
Configuring Access Control Lists	800
Populating ACLs for Your Entities	803
Adding Authorization to Customer Support	804
Switching to Custom User Details	804
Securing Your Service Methods	808
Using Spring Security's Tag Library	813
Summary	814
CHAPTER 28: SECURING RESTFUL WEB SERVICES WITH OAUTH	815
Understanding Web Service Security	816
Comparing Web GUI and Web Service Security	816

Choosing an Authentication Mechanism	817
Introducing OAuth	818
Understanding the Key Players	819
The Beginning: OAuth 1.0	819
The Standard: OAuth 1.0a	820
The Evolution: OAuth 2.0	826
Using Spring Security OAuth	833
Creating on OAuth 2.0 Provider	833
Creating an OAuth 2.0 Client	838
Finishing the Customer Support Application	840
Generating Request Nonces and Signatures	840
Implementing Client Services	842
Implementing Nonce Services	845
Implementing Token Services	847
Customizing the Resource Server Filter	850
Reconfiguring Spring Security	852
Creating an OAuth Client Application	856
Customizing the REST Template	857
Configuring the Spring Security OAuth Client	858
Using the REST Template	861
Testing the Provider and Client Together	861
Summary	862
<i>INDEX</i>	865

INTRODUCTION

THOUGH MANY DON'T REALIZE IT, MOST PEOPLE USE JAVA EVERY DAY. It's all around you—it's in your TV, in your Blu-ray player, and on your computer; some popular smart phones run a Java-based operating system; and it powers many of the websites you use every day. When you think of Java, you may naturally picture browser applets or desktop applications with user interfaces that don't match other applications on the operating system. You may even think of that annoying system tray notification that tells you to update Java (seemingly) constantly.

But Java is much more than just these daily, visible reminders you may be exposed to. Java is a powerful language, but much of its capability lies in the power of the platform. Although the Java SE platform provides indispensable tools for creating console, desktop, and browser applications, the Java EE platform extends this platform significantly to help you create rich, powerful web applications. This book covers these tools and shows you how to create modern and useful enterprise Java web applications.

WHO THIS BOOK IS FOR

This book is for software developers and engineers who already have a proficient knowledge in the Java language and the Java Platform, Standard Edition (Java SE). It is a self-guided, self-study book that existing Java developers can use to expand their Java knowledge and grow their skillset from applets or console or desktop applications to enterprise web applications. You can read this book from start to finish to cover all the topics in order, or you can pick and choose topics that interest you and use this book more as a reference. Although some chapters occasionally refer to examples from previous chapters, an effort was made to make each chapter as self-sustaining as possible. The examples are all available for download from wrox.com, which should help you when an example relies on another example from a previous chapter.

This book can also be useful for developers with existing Java Platform, Enterprise Edition (Java EE) experience who want to refresh their skills or learn about new features in the latest Java EE version. Software architects might also find this book useful because it covers several web software development concepts and patterns in addition to specific tools and platform components. This book could help architects apply new ideas to their teams' projects and processes.

If you're a manager of a software development team, you may also find this book helpful. Undoubtedly you strive every day to communicate effectively with the developers and engineers that you oversee. By reading this book, you can expand your knowledgebase, understand the tools your developers use to more successfully communicate, and make recommendations to your team to solve certain problems. After reading this book, you may also decide to purchase several copies for your team to improve their skillsets and apply the concepts within your projects.

Finally, teachers and students can apply this book to a classroom environment. Used as a textbook, it can be invaluable for 300 and 400 level courses to instruct students in real-world skills that can help them succeed in the workplace beyond graduation.

WHO THIS BOOK IS NOT FOR

This book is not for readers who have no experience with Java and have never written or compiled Java-based applications. If you have no prior Java experience, you will likely find it difficult to understand the text and examples in this book. This is because this book does not cover the Java language syntax or the specifics of the Java SE platform. It is assumed the reader is comfortable writing, compiling, and debugging Java code and is familiar with the standard platform. Very few explanations are given about standard Java features and tools, except where those features were added in Java SE 8.

In addition, the reader is expected to have a basic understanding of the following technologies and concepts. Although some of them may seem obvious, it's important to note that if you are unfamiliar with one or more of these concepts you may have difficulty with some chapters in the book.

- The Internet and the TCP and HTTP protocols
- HyperText Markup Language (HTML), including HTML 5
- Extensible Markup Language (XML)
- JavaScript or ECMAScript, including jQuery and browser debugging tools
- Cascading Style Sheets (CSS)
- Structured Query Language (SQL) and relational databases, specifically MySQL (If you are familiar with other relational databases, you can adapt to MySQL easily.)
- Transactions and transactional concepts, such as Atomicity, Consistency, Isolation, Durability (ACID)
- Use of an Integrated Development Environment (IDE)
- Execution of simple command-line tasks (You do not need to be a command-line guru.)

WHAT YOU WILL LEARN IN THIS BOOK

In this book, you learn about the Java EE platform version 7 and many of the technologies within it. You'll start with an introduction to what exactly the Java EE platform is and how it evolved, followed by an introduction to application servers and Servlet containers and how they work. You'll then proceed to explore Spring Framework, publish-subscribe, Advanced Message Queuing Protocol (AMQP), object-relational mappers (O/RMs), Hibernate ORM, Spring Data, full-text searching, Apache Lucene, Hibernate Search, Spring Security, and OAuth. Throughout this book you will also explore the following components of Java EE 7:

- Servlets 3.1 – JSR 340
- JavaServer Pages (JSP) 2.3 – JSR 245
- Java Unified Expression Language (JUEL or just EL) 3.0 – JSR 341
- Java API for WebSockets – JSR 356

- > Bean Validation (BV) 1.1 – JSR 349
- > Java Message Service (JMS) 2.0 – JSR 343
- > Java Persistence API (JPA) 2.1 – JSR 338
- > Java Transaction API (JTA) 1.2 —JSR 907

You'll also make extensive use of lambda expressions and the new JSR 310 Java 8 Date and Time API, both additions to Java SE 8.

Part I: Creating Enterprise Applications

Here you explore Servlets, filters, listeners, and JavaServer Pages (JSP). You'll learn about how Servlets respond to HTTP requests and how filters assist them. You'll easily create powerful user interfaces based on JSP. Combining the power of JSP tags and the brand-new Expression Language 3.0, you'll then create Java-free views easily maintained by UI developers who have little or no Java knowledge. You'll learn about HTTP sessions and how they can help you create rich user experiences that span multiple pages in your application. You'll explore the brand-new technology called WebSockets, which helps you create richer, more interactive user interfaces by providing full-duplex, bidirectional communications between your application and the client (such as a browser). As a final note, you'll learn about application logging best practices and technologies, something that will become critical as you create complex applications with lots of code.

Part II: Adding Spring Framework Into the Mix

In this part of the book you start working with Spring Framework and Spring MVC. You'll explore topics such as dependency injection (DI), inversion of control (IoC), and aspect-oriented programming (AOP). You'll configure advanced Spring Framework projects using both XML and annotation-based configuration, and you'll use Spring tools to support your bean validation and internationalization needs. You'll create both RESTful and SOAP web services using Spring MVC controllers and Spring Web Services, and you'll learn how to use the flexible messaging systems built in to Spring Framework. You'll also learn about the Advanced Message Queuing Protocol (AMQP) and configure and use a RabbitMQ installation.

Part III: Persisting Data with JPA and Hibernate ORM

This part focuses on data persistence and different approaches to storing your objects in your databases. After understanding some of the basic issues with using raw JDBC for persisting your entities, you'll learn about object-relational mappers (O/RMs) and explore Hibernate ORM and its API. You'll then take a look at the Java Persistence API, an abstraction that allows you to program to a common API regardless of the O/RM implementation. Next you'll explore Spring Data and how it can help you create persistence applications without writing any persistence code. You'll also learn several methods for searching your persisted data and explore Hibernate Search with Apache Lucene as a potential full-text searching tool.

Part IV: Securing Your Application with Spring Security

The final part of the book introduces you to the concepts of authentication and authorization and shows you several techniques that can be used for both. It then helps you integrate Spring Security into your Spring Framework applications. You'll also learn how to secure your web services using OAuth 1.0a and OAuth 2.0 and create a custom access token type to make your OAuth 2.0 implementation stronger.

WHAT YOU WILL NOT LEARN IN THIS BOOK

This book does not teach you about basic Java syntax or the Java SE platform, though it will briefly explain some new features added in Java SE 7 and 8. It will also not teach you how to write Java-based console or desktop applications or applets. If you are looking for a book on these topics, Wrox has a variety of titles to choose from.

More important, this book **does not teach you how to administer a Java EE application server environment**. There are dozens of different application servers and web containers, and no two are managed identically. Which application server you use strongly depends on the nature of your application, your business requirements, your business practices, and your server environment. It would be impractical to teach you how to administer even a few of the most common application servers. The best way to learn how to deploy and administer your Java EE application server or web container of choice is to consult its documentation and, in some cases, experiment. (Because the use of a web container is necessary to complete the examples in this book, Chapter 2 covers the basic tasks of installing, starting, stopping, and deploying applications to Apache Tomcat.)

Refer back to the introductory section titled "Who This Book Is Not For" —this book does not cover the basics of the technologies and concepts listed in that section. It also does not cover the following Java EE 7 components, which are unsupported by most simple web containers and unnecessary when using Spring Framework and its related projects.

- Java API for RESTful Web Services (JAX-RS) 2.0 – JSR 339
- JavaServer Faces (JSF) 2.2 – JSR 344
- Enterprise JavaBeans (EJB) 3.2 – JSR 345
- Contexts and Dependency Injection (CDI) 1.1 – JSR 346
- JCache – JSR 107
- State Management – JSR 350
- Batch Applications for the Java Platform – JSR 352
- Concurrency Utilities for Java EE – JSR 236
- Java API for JSON Processing – JSR 353

WHAT TOOLS YOU WILL NEED

You'll need several different tools to complete and run the examples in this book. To start, be sure you have the following installed or enabled on your computer:

- Apache Maven version 3.1.1 or newer
- A command line for certain tasks, and an operating system that provides access to the command line (In other words, you cannot compile and run the examples on a smartphone or tablet.)
- A quality text editor useful for tasks such as editing configuration files. You should never use Windows Notepad or Apple TextEdit as a text editor. If you are looking for a quality text editor, consider:
 - **Windows** — Notepad++ or Sublime Text 2.
 - **Mac OS X** — TextWrangler, Sublime Text 2, or Vim.
 - **Linux** — Sublime Text 2 or Vim.

Java Development Kit for Java SE 8

You must have the Java Development Kit (JDK) for Java SE 8 installed on your machine. Java SE 8 is scheduled to release on March 18, 2014. You should be able to download the JDK from Oracle's standard Java SE Downloads site. However, if you purchased this book prior to the release of Java SE 8, you may need to download the Early Access JDK from its Java.net project site (Don't worry, you won't have to compile it.) Always get the latest version of the JDK, and download the version and architecture appropriate for your machine. If your machine contains a 64-bit processor and 64-bit operating system, you should download the 64-bit Java installer.

Integrated Development Environment

You need an integrated development environment, or IDE, for compiling and executing the code samples and general experimentation. An IDE, sometimes also called an interactive development environment, is a software application with coding, building, deploying, and debugging facilities for software developers to use when creating software. There are many different Java IDEs available, and some are better than others. A lot of what makes one IDE better than another is simply perspective and personal practices — an IDE that is perfect for one developer may not be so easy for another developer to use. Generally, however, IDEs that include intelligent code suggestions, code completion, code generation, syntax checking, spell checking, and framework integration (Spring Framework, JPA, Hibernate ORM, and so on) are going to be much more useful and provide you with a much more productive work environment than IDEs without these features.

You may already have an IDE that you use regularly, or you may simply use your favorite text editor and a command line. If you have an IDE, it may or may not be up to the task of running the examples in this book. When choosing an IDE (or evaluating whether your current IDE is sufficient), you should get one with intelligent code completion and suggestions, syntax checking, and integration with Java EE, Spring Framework, Spring Security, Spring Data, JPA, and Hibernate ORM. This means it should have the ability to evaluate your Java EE, Spring, JPA, and Hibernate configurations and tell you whether there are any errors or problems with those configurations. This introduction briefly tells you about three polyglot IDEs and makes a recommendation for this book.

NetBeans IDE 8.0

NetBeans—a free IDE—is the standard, Oracle-sponsored Java IDE, similar to how Microsoft Visual Studio is the standard IDE for .NET development. It is not, however, the most popular Java IDE. Only NetBeans IDE 8.0 has support for Java SE 8 and Java EE 7—previous versions do not. NetBeans provides a strong feature set and built-in support for all Java EE features. It also supports C, C++, and PHP development. You can also extend NetBeans's functionality using plug-ins, and plug-ins are available for Spring Framework and Hibernate ORM. However, the NetBeans feature set is not as rich as other IDEs, so it is not recommended for this book. The code examples in this book are not available as NetBeans downloads, but you should be able to import the samples as Maven projects if you prefer to use NetBeans. You can download NetBeans [here](#).

Eclipse Luna IDE 4.4 for Java EE Developers

Eclipse is another free IDE and the most popular Java IDE worldwide. One of its strengths is its extensibility, which goes beyond its support for plug-ins. Using the Eclipse platform, you can completely customize the IDE for specific tasks and workflows. It already has plug-ins and extensions for Spring Framework, Spring Data, Spring Security, Hibernate ORM, and more. The Spring community offers a customized version of Eclipse—called Spring Tool Suite—that is very well suited for working with Spring-based projects. However, in this author's opinion, Eclipse is a very difficult IDE to use effectively and efficiently. Very simple tasks often require a great amount of effort. Historically, compatible Eclipse releases have trailed Java SE and EE releases considerably. At the time this book was written, the Eclipse community had not yet released an Eclipse IDE version compatible with Java SE 8 and Java EE 7. Therefore, it is not recommended that you choose Eclipse IDE for running the examples in this book. If you do choose to use—or continue to use—Eclipse, you should make sure you get Eclipse Luna IDE 4.4 for Java EE Developers, which is scheduled for release in June 2014. This may require downloading a pre-release edition, and that edition may not support all the topics covered in this book. You can download Eclipse IDE [here](#).

Due to the popularity of Eclipse IDE, the code examples for this book will be available to download as Eclipse projects as soon as Eclipse Luna 4.4 is capable of running them.

IntelliJ IDEA 13 Ultimate Edition

JetBrains's IntelliJ IDEA is a feature-rich Java IDE with both Community (free) and Ultimate (paid) editions. It is, again in this author's opinion, the easiest to use and most powerful Java IDE available. Its code suggestions and completion and framework support are unmatched in any other IDE. In addition, it has historically provided better early support for experimental versions of Java SE and Java EE before they release. IntelliJ IDEA 12, for example, provided Java SE 8 support as early as December 2012—a full 15 months before Java SE 8 was released and 18 months before Eclipse IDE supported it. If you like to test new versions of Java SE and Java EE before they come out, and use them immediately after their release, IntelliJ IDEA is essentially your only option.

This power does come at a cost, however. The Community Edition is useful for many different types of Java SE projects, but not Java EE projects. You need to purchase the Ultimate Edition to realize the full support for Java EE, Spring projects, and Hibernate ORM. The Ultimate Edition is priced reasonably and competitively for companies, individuals, and students, at a fraction of the cost that you would pay for equivalent editions of Microsoft Visual Studio. Educational institutions can get free licenses for official classroom use, and established open source organizations can get free licenses for their projects. You can download a 30-day free trial of IntelliJ IDEA 13 Ultimate Edition [here](#), and you can purchase a license (or obtain a free license if you qualify) for your download at any time.

In addition, the back of this book contains a coupon for a free 90-day personal license of IntelliJ IDEA 13 Ultimate! We recommend you use IntelliJ IDEA Ultimate Edition for all the code examples in this book. Until Eclipse Luna 4.4 is capable of running the examples, the code downloads will initially be available only as IntelliJ IDEA projects.

Be sure to download the latest version of IntelliJ IDEA. Although version 13.0.x is the most current version as of the date this book was published, 13.1.x is scheduled for release sometime in April 2014 with several Spring Framework and Java EE 7 support improvements, and 14.0.x will likely be released in December 2014.

Java EE 7 Web Container

The final tool you'll need while reading this book is a Java EE web container that implements the Servlet, JSP, JUEL, and WebSocket specifications in Java EE 7. This topic is covered more thoroughly in Chapter 2, where you review the most popular web containers and application servers and learn how to download, install, and use Apache Tomcat 8.0.

CONVENTIONS USED IN THIS BOOK

Several conventions are used throughout this book to help draw your attention to certain items or demonstrate something in code. This section covers those conventions by example.

NOTE *Notes indicate notes, tips, hints, tricks, reminders, and other interesting information loosely related to the current discussion. You'll want to pay attention to these boxes.*

WARNING *Warnings hold important information that is directly relevant to the surrounding text and should not be forgotten. Warnings can indicate pitfalls, dangers, and potential for loss or corrupted data. Pay close attention to these boxes.*

You may see several styles in the text:

- New terms and important words are *highlighted* when introduced. This may not be the first time these words appear in the text, but it will be the first time they are explained.
- Keyboard strokes appear as `Ctrl+S`, `Ctrl+Alt+F8`, and so on.
- Filenames, URIs that aren't URLs, class and method names, primitive types, and code within the text appear like this.
- Code variables, method and constructor parameter names, and request parameters look like this.
- Values the user must enter in dialog boxes, prompts, or form fields are **bold** and **monospace**.

Finally, when reading sample code within the text, it may be presented in two different ways:

We use a monofont type with no highlighting for most lines of code.

We use **bold** to emphasize code that's especially important, to show changes from previous examples, or to draw attention to it when mentioning it in the text.

In most cases, code examples are simply written inline, between paragraphs. However, when they are particularly long they will be referenced by number in the text and appear as code listings, as in the sample Listing I-1.

LISTING I-1: A Sample Code Listing

This is what a code listing will look like.



Finally, you will occasionally see an icon in the margin next to a paragraph. This icon will always be referenced in the paragraph it is next to and indicates a toolbar button that you will need to use to perform a task discussed in that paragraph.

CODE EXAMPLES

As with any software development book, this book makes extensive use of code examples to demonstrate the topics explained. For the most part, these examples are full IDE projects that you can just open in your IDE, compile, and execute. All the examples are available for download from the [wrox.com code download site](http://www.wrox.com/go/projavaforwebapps). Just go to <http://www.wrox.com/go/projavaforwebapps> and click the Download Code tab. You can download all the code samples as a single ZIP file or a ZIP file

for each chapter. Within the download for each chapter you'll find two versions of each sample: an IntelliJ IDEA project and an Eclipse project. You should use the version applicable to the IDE you chose. If you are not using one of these two IDEs, your IDE should be able to import the IntelliJ IDEA project as a simple Maven project.

NOTE *Remember, the Eclipse version of the code samples will not be available until Eclipse Luna 4.4 is capable of running them. If you are reading this book before that milestone, you can download the IntelliJ IDEA example projects.*

Near the beginning of the book, you can create the examples from scratch in your IDE without downloading them from the code site (if that's what you want). However, as the examples get more complex this will not be possible. The most critical code is printed in the book, but printing every line of code is not practical — it would make this book considerably longer, and thus make it more expensive for you. In addition, much of the omitted code is repetitive. For example, the Spring Framework configuration is nearly identical for most of the example projects in Parts II through IV. In these cases, it makes much more sense to simply show you how the configuration has changed from previous chapters rather than re-printing the entire configuration. For this reason, you need to download most of the code examples from the wrox.com code download site if you want to execute and test the examples.

On the first page of each chapter, you'll see an area titled "Wrox.com Code Downloads for This Chapter." This section lists the names of all the code examples used in the chapter and reminds you of the link for downloading the code samples. A handful of chapters do not contain code example downloads, but most do.

MAVEN DEPENDENCIES

The code examples in this book make extensive use of third-party dependencies, such as Spring Framework, Hibernate ORM, and Spring Security. Including these dependency JARs in the code downloads on the download site would make these downloads unnecessarily large and cause you to download many hundreds of megabytes over the course of the book. To eliminate this problem, the code samples use Apache Maven and its dependency management capabilities. All the sample projects are Maven projects. When opening each project in your IDE, the IDE should automatically resolve the dependencies in your local Maven repository or, if necessary, download them to your local Maven repository.

On the first page of each chapter you'll see an area titled "New Maven Dependencies for This Chapter." This section lists the Maven dependencies that, in addition to all previous dependencies, you'll use in that chapter. You can also consult the `pom.xml` file in each example project to view its dependencies. Some chapters do not introduce new Maven dependencies, but most do.

Each Maven dependency has a *scope* that defines which classpath that dependency is available on. The most common scope—"compile" scope—indicates that the dependency is available to your project on the compile classpath, the unit test compile and execution classpaths, and the eventual runtime classpath when you execute your application. In a Java EE web application, this means the dependency is copied into your deployed application. "Runtime" scope indicates that the dependency is available to your project on the unit test execution and runtime execution classpaths, but unlike compile scope it is not available when you compile your application or its unit tests. A runtime dependency is copied into your deployed application. Finally, "provided" scope indicates that the container in which your application executes provides the dependency on your behalf. In a Java EE application, this means the dependency is already on the Servlet container's or application server's classpath and is not copied into your deployed application. Maven and your IDE ensures provided dependencies are available when you compile your application and its unit tests. There are other Maven scopes as well, but these are the only scopes you use in this book.

Some of the Maven dependencies you see in the text and the sample projects have exclusions that ignore certain dependencies of those dependencies—these are called *transient dependencies*. To a large extent, these exclusions are usually redundant and are shown only for clarity. When a dependency relies on an older version of a dependency than a version you are already using, the exclusion makes it clear that there is a discrepancy there, and also avoids problems caused by Maven's nearness algorithm. However, some of the exclusions exist because newer versions of Java SE or Java EE provide the dependency already, or because the dependency ID changed. When this is the reason an exclusion exists, it is noted in the text.

WHY SECURITY IS AT THE END OF THE BOOK

Quite frankly, application security gets in the way. The technologies and techniques you must use to add authentication and authorization to your products can clutter your code and make the process of learning more difficult. It's natural to think about security first, and it's never wrong to keep security in mind at all times. However, with the right tools, it's fairly easy to add authentication and authorization to an existing project after it is complete (or nearly so). This book focuses first on creating quality web applications with rich feature sets using industry standard tools. Once you have all the skills you need to create powerful applications, Part IV of this book shows you how to add authentication and authorization to an existing application to secure it from unauthorized and malicious access.

ERRATA

We strove to make this text as thorough and accurate as possible, but nobody is perfect and mistakes do happen. Occasionally this book may contain errors that require correction. If you find factual errors, spelling mistakes, or faulty pieces of code, we want to hear about it! By providing your feedback, you could save other readers' time and effort trying to troubleshoot something that isn't working, and at the same time improve future editions of this book.

To read the discovered errata for this book, go to Wrox's website and use the search box to find this title. Searching for its ISBN is the fastest way to locate it. On this book's page, click the Errata link. Here you can view all the errata that has been submitted by readers and verified by Wrox editors. If you don't spot the errata you found, go to the Wrox technical support page and complete the form there to report the problem. After we verify the error and come up with a correction, we will post it to this book's errata page and fix the problem for future editions.

PART I

Creating Enterprise Applications

- ▶ CHAPTER 1: Introducing Java Platform, Enterprise Edition
- ▶ CHAPTER 2: Using Web Containers
- ▶ CHAPTER 3: Writing Your First Servlet
- ▶ CHAPTER 4: Using JSPs to Display Content
- ▶ CHAPTER 5: Maintaining State Using Sessions
- ▶ CHAPTER 6: Using the Expression Language in JSPs
- ▶ CHAPTER 7: Using the Java Standard Tag Library
- ▶ CHAPTER 8: Writing Custom Tag and Function Libraries
- ▶ CHAPTER 9: Improving Your Application Using Filters
- ▶ CHAPTER 10: Making Your Application Interactive with WebSockets
- ▶ CHAPTER 11: Using Logging to Monitor Your Application

1

Introducing Java Platform, Enterprise Edition

IN THIS CHAPTER

- Java SE and Java EE version timeline
- Introducing Servlets, filters, listeners, and JSPs
- Understanding WAR, and EAR files, and the class loader hierarchy

WROX.COM CODE DOWNLOADS FOR THIS CHAPTER

There are no code downloads for this chapter.

NEW MAVEN DEPENDENCIES FOR THIS CHAPTER

There are no Maven dependencies for this chapter.

A TIMELINE OF JAVA PLATFORMS

The Java language and its platforms have had a long and storied history. From its invention in the mid-'90s to an evolution drought from 2007 to nearly 2012, Java has gone through many changes and encountered its share of controversy. In the earliest days, Java, known as the Java Development Kit or JDK, was a language tightly coupled to a platform composed of a small set of essential *application programming interfaces (APIs)*. Sun Microsystems unveiled the earliest alpha and beta versions in 1995, and although Java was extremely slow and primitive by today's standards, it began a revolution in software development.

In the Beginning

Java's history is summarized in Figure 1-1, a timeline of Java platforms. As of the publication of this book, the Java language and the Java SE platform have always evolved together—new versions of each always release at the same time and are tightly coupled to one another. The platform was called the JDK through version 1.1 in 1997, but by version 1.2 it was clear that the JDK and the platform were not synonymous. Starting with version 1.2 in late 1998, the Java technology stack was divided into the following key components:

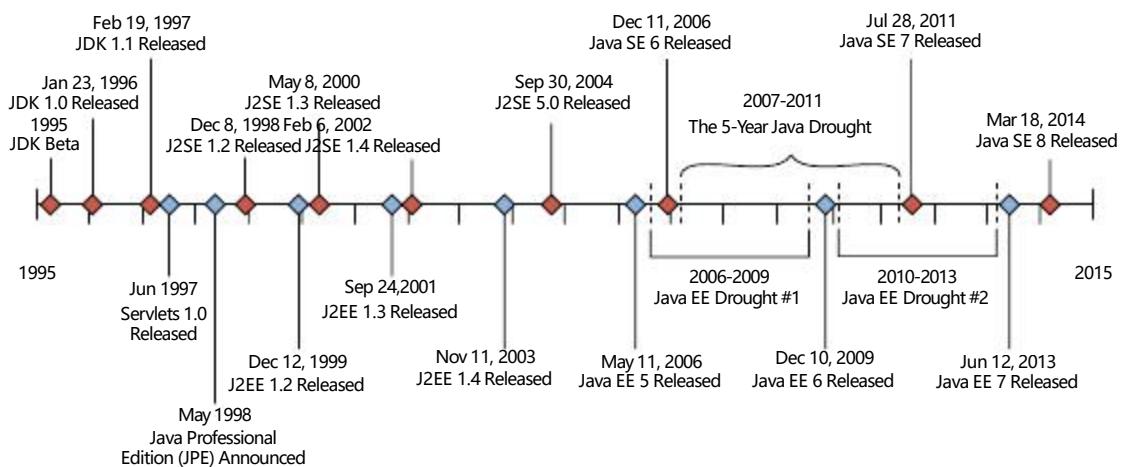


FIGURE 1-1: A timeline showing the correlation of the evolution of Java Platform, Standard Edition and Java Platform, Enterprise Edition. The events on top of the timeline represent Java SE milestones while the events on the bottom represent Java EE milestones.

- *Java* is the language and includes a strict and strongly typed syntax with which you should be very familiar by now.
- *Java 2 Platform, Standard Edition*, also known as *J2SE*, referred to the platform and included the classes in the `java.lang` and `java.io` packages, among others. It was the building block that Java applications were built upon.
- A *Java Virtual Machine*, or *JVM*, is a software virtual machine that runs compiled Java code. Because compiled Java code is merely bytecode, the JVM is responsible for compiling that bytecode to machine code before running it. (This is often called the *Just In Time Compiler* or *JIT Compiler*.) The JVM also takes care of memory management so that application code doesn't have to.
- The *Java Development Kit*, or *JDK*, was and remains the piece of software Java developers use to create Java applications. It contains a Java language compiler, a documentation generator, tools for working with native code, and (typically) the Java source code for the platform to enable debugging platform classes.
- The *Java Runtime Environment*, or *JRE*, was and remains the piece of software end users download to run compiled Java applications. It includes a JVM but does not contain any of the development tools bundled in the JDK. The JDK, however, does contain a JRE.

All five of these components have historically been specifications, not implementations. Any company may create its own implementation of this Java technology stack, and many companies have. Though Sun offered a standard implementation of Java, J2SE, the JVM, the JDK, and the JRE, IBM, Oracle, and Apple also created competing implementations that offered different features.

The IBM implementation was born out of need—Sun didn't offer binaries capable of running on IBM operating systems, so IBM created its own. The situation was similar for the Apple Mac OS operating system, so Apple rolled its own implementation as well. Although the implementations offered by these companies were all free as in beer, they were not free as in freedom, so they were not considered open source software. As such, the open source community quickly formed the OpenJDK project, which provided an open source implementation of the Java stack.

Still more companies created less popular implementations, some of which compiled your application to machine code for a target architecture to improve performance by avoiding JIT compilation. For the vast majority of users and developers, the Sun Java implementation was both sufficient and preferred. After Oracle's purchase of Sun, the Sun and Oracle implementations became one and the same.

Not shown in Figure 1-1 is the development of other languages capable of using the J2SE and running on the JVM. Over the years, dozens of languages appeared that can compile to Java bytecode (or machine code, in some cases) and run on the JVM. The most high-profile of these are Clojure (a Lisp dialect), Groovy, JRuby (a Java-based Ruby implementation), Jython (a Java-based Python implementation), Rhino, and Scala.

The Birth of Enterprise Java

This brief history lesson might seem unnecessary—as an existing Java developer, you have likely heard most of this before. However, it's important to include the context of the history of the Java Platform, Standard Edition, because it is tightly woven into the birth and evolution of the Java Platform, Enterprise Edition. Sun was already aware of the need for more advanced tools for application development, particularly in the arena of the growing Internet and the popularity of web applications. In 1998, shortly before the release of J2SE 1.2, Sun announced it was working on a product called the Java Professional Edition, or JPE. Work had already begun on a technology known as *Servlets*, which are miniature applications capable of responding to HTTP requests. In 1997, Java Servlets 1.0 released alongside the Java Web Server with little fanfare because it lacked many features that the Java community wanted.

After several internal iterations of Servlets and the JPE, Sun released *Java 2 Platform, Enterprise Edition* (or *J2EE*) version 1.2 on December 12, 1999. The version number corresponded with the current Java and J2SE version at the time, and the specification included:

- Servlets 2.2
- JDBC Extension API 2.0
- Java Naming and Directory Interface (JNDI) 1.0
- JavaServer Pages (JSP) 1.2
- Enterprise JavaBeans (EJB) 1.1

- Java Message Service (JMS) 1.0
- Java Transaction API (JTA) 1.0
- JavaMail API 1.1
- JavaBeans Activation Framework (JAF) 1.0.

Like J2SE, J2EE was a mere specification. Sun provided a *reference implementation* of the specification's components, but companies were free to create their own as well. Many implementations evolved, and you learn about some of them in the next chapter. These implementations included and still include open source and commercial solutions. The J2EE quickly became a successful complement to the J2SE, and over the years some components were deemed so indispensable that they have migrated from J2EE to J2SE.

Java SE and Java EE Evolving Together

J2EE 1.3 released in September 2001, a little more than a year after Java and J2SE 1.3 and before Java/J2SE 1.4. Most of its components received minor upgrades, and new features were added into the fold. The following joined the J2EE specification, and the array of implementations expanded and upgraded:

- Java API for XML Processing (JAXP) 1.1
- JavaServer Pages Standard Tag Library (JSTL) 1.0
- J2EE Connector Architecture 1.0
- Java Authentication and Authorization Service (JAAS) 1.0

At this point the technology was maturing considerably, but it still had plenty of room for improvement.

J2EE 1.4 represented a major leap in the evolution of the Java Platform, Enterprise Edition. Released in November 2003 (approximately a year before Java/J2SE 5.0 and 2 years after Java/J2SE 1.4), it included Servlet 2.4 and JSP 2.0. It was in this version that the JDBC Extension API, JNDI, and JAAS specifications were removed because they had been deemed essential to Java and moved to Java/J2SE 1.4. This version also represented the point at which J2EE components were broken up into several higher-level categories:

- **Web Services Technologies:** Included JAXP 1.2 and the new Web Services for J2EE 1.1, Java API for XML-based RPC (JAX-RPC) 1.1, and Java API for XML Registries (JAXR) 1.0
- **Web Application Technologies:** Included the Servlet, JSP, and JSTL 1.1 components, as well as the new Java Server Faces (JSF) 1.1
- **Enterprise Application Technologies:** Included EJB 2.1, Connector Architecture 1.5, JMS 1.1, JTA, JavaMail 1.3, and JAF
- **Management and Security Technologies:** Included Java Authorization Service Provider Contract for Containers (JACC) 1.0, Java Management Extensions (JMX) 1.2, Enterprise Edition Management API 1.0, and Enterprise Edition Deployment API 1.1

The Era of the Name Changes

Enter the era of the name changes, which are often a source of confusion for Java developers. They are highlighted here so that you fully understand the naming conventions used in this book and how they relate to the previous naming conventions you may already be familiar with. Java and J2SE 5.0 were released in September 2004, and included generics, annotations, and enums, three of the most radical language syntax changes in Java history. This version number was a departure from previous patterns, made more confusing by the fact that the J2SE APIs and the `java` command-line tool reported the version number as being 1.5. Sun had made the decision to drop the 1 from the publicized version number and go by the minor version, instead. It quickly recognized that the "dot-oh" on the end of the version number was a source of confusion and quickly began referring to it as simply version 5.

About the same time, the decision was made to retire the name Java 2 Platform, Standard Edition in favor of Java Platform, Standard Edition and to abbreviate this new name Java SE. The changes were made formal with Java SE 6, released in December 2006, and to this day the name and version convention has remain unchanged. Java SE 6 is internally 1.6, Java SE 7 is internally 1.7, and Java SE 8 is internally 1.8.

The same name and number change decisions were applied to J2EE, but because J2EE 1.5 was set to release between J2SE 5.0 and Java SE 6, the changes were applied a version early. Java Platform, Enterprise Edition 5, or Java EE 5, was released in May 2006, approximately 18 months after J2SE 5.0 and 7 months before Java SE 6. Internally Java EE 5 is 1.5, Java EE 6 is 1.6, and Java EE 7 is 1.7. Whenever you see the terms J2SE or Java SE, they are interchangeable, and the preferred and accepted name today is Java EE. Likewise, J2EE and Java EE are interchangeable, but Java EE is preferred today. The rest of this book refers to them exclusively as Java SE and Java EE.

Java EE 5 grew and included numerous changes and improvements again, and today it is still one of the most widely deployed Java EE versions. It included the following changes and additions:

- JAXP and JMX moved to J2SE 5.0 and were not included in Java EE 5.
- Java API for XML-based Web Services (JAX-WS) 2.0, Java Architecture for XML Binding (JAXB) 2.0, Web Service Metadata for the Java Platform 2.0, SOAP with Attachments API for Java (SAAJ) 1.2, and Streaming API for XML (StAX) 1.0 were added to Web Services Technology.
- Java Persistence API (JPA) 1.0 and Common Annotations API 1.0 were added to Enterprise Applications Technology.

The Java SE and EE Droughts

The release of Java SE 6 in December 2006, marked the beginning of a drought for Java SE releases that lasted approximately 5 years. This time was a period of frustration and even anger for many in the Java community. Sun continued to promise new language features and APIs for Java SE 7, but the schedule continued to slip year after year with no end in sight. Meanwhile other technologies, such as the C# language and .NET platform, caught up to and surpassed Java in language features and platform APIs, causing some to speculate whether Java had reached the end of its useful life. To make matters worse, Java EE entered its own drought period and by 2009, more than 3 years

had passed since Java EE 5 was released. All was not lost, however. Java EE 6 development picked up in early 2009, and it released in December 2009, 3 years and 7 months after Java EE 5, and 3 years almost to the day after Java SE 6.

By this time, Java Enterprise Edition became enormous:

- SAAJ, StAX, and JAF moved to Java SE 6.
- The Java API for RESTful Web Services (JAX-RS) 1.1 and Java APIs for XML Messaging (JAXM) 1.3 specifications were added to Web Services Technologies.
- The Java Unified Expression Language (JUEL or just EL) 2.0 was added to Web Application Technologies.
- Management and Security Technologies saw the addition of Java Authentication Service Provider Interface for Containers (JASPI) 1.0.
- Enterprise Application Technologies realized the most dramatic increase in features, including Contexts and Dependency Injection for Java (CDI) 1.0, Dependency Injection for Java 1.0, Bean Validation 1.0, Managed Beans 1.0, and Interceptors 1.1, in addition to updates to all its other components.

Java EE 6 also represented a major turning point in the architecture of Java EE on two fronts:

- This version introduced annotation-based and programmatic application configuration to complement the traditional XML configuration used for more than a decade.
- This version marked the introduction of the Java EE Web Profile.

To account for the fact that Java EE had become so large (and maintaining and updating certified implementations was becoming increasingly difficult), the Web Profile certification program offered the opportunity to certify Java EE implementations that included only a subset of the entire Java EE platform. This subset included the features deemed to be most critical to a large number of applications and excluded specifications that are used only by a small minority of applications. As of Java EE 6:

- None of the Web Services or Management and Security components are part of the Java EE Web Profile.
- The Web Profile includes everything from Web Application Technologies and everything from Enterprise Application Technologies except Java EE Connector Architecture, JMS, and JavaMail.

It was during the 5-year Java drought that Oracle Corporation bought Sun Microsystems in January 2010. Coupled with the Java SE drought, this brought a whole new set of concerns for the Java community. Oracle was never known for its agility or willingness to cooperate with open source projects, and many people feared Oracle had bought Sun to shut Java down. However, this turned out not to be the case.

Early on, Oracle began reorganizing the Java team, creating communication pipelines with the open source community, and releasing roadmaps for future Java SE and Java EE versions that were more realistic than anything Sun had promised. Work began anew on Java SE 7, which released on

(Oracle's) schedule in June 2011, almost 5 years after Java SE 6. A second Java EE drought ended with the release of Java EE 7 in June 2013, 3 years and 7 months after Java EE 6. Oracle now says it is on track to begin releasing new versions of both platforms every 2 years, on alternate years. It remains to be seen whether that will come to pass.

Understanding the Most Recent Platform Features

Java SE 7 and 8 and Java EE 7 have brought major changes to the language and supporting APIs and resulted in a rejuvenation of Java technologies. You use these new features throughout this book, so this section provides an overview of them.

Java SE 7

Originally, Java SE 7 had a very ambitious feature list, but after acquiring Sun, Oracle quickly admitted that achieving the goals for Java SE 7 would take many, many years. Every feature was the most important feature to some group of users, so the decision was made to defer some of them to future versions. The alternative was to delay the release of Java SE 7 until 2015 or later — an option that was not acceptable.

Java SE 7 included support for dynamic languages as well as compressed 64-bit pointers (for improved performance on 64-bit JVMs). It also added several language features that made developing Java applications more productive. Perhaps one of the most useful changes was *diamonds*, a shortcut for generic instantiation. Prior to Java 7, both the variable declaration and the variable assignment for generic types had to include the generic type arguments. For example, here is a declaration and assignment for a very complex `java.util.Map` variable:

```
Map<String, Map<String, Map<Integer, List<MyBean>>> map =
    new Hashtable<String, Map<String, Map<Integer, List<MyBean>>>();
```

Of course, this declaration contains a lot of redundant information. Assigning anything other than a `Map<String, Map<String, Map<Integer, List<MyBean>>>` to this variable would be illegal, so why should you have to specify all those type arguments again? Using Java 7 diamonds, this declaration and assignment becomes much simpler. The compiler infers the type arguments for the instantiated `java.util.Hashtable`.

```
Map<String, Map<String, Map<Integer, List<MyBean>>> map = new Hashtable<>();
```

Another common complaint about Java prior to Java 7 is the management of closable resources as it relates to `try-catch-finally` blocks. In particular, consider this nasty bit of JDBC code:

```
Connection connection = null;
PreparedStatement statement = null;
ResultSet resultSet = null;
try
{
    connection = dataSource.getConnection();
    statement = connection.prepareStatement(...);
    // set up statement
    resultSet = statement.executeQuery();
    // do something with result set
}
```

```
        catch(SQLException e)
        {
            // do something with exception
        }
    finally
    {
        if(resultSet != null) {
            try {
                resultSet.close();
            } catch(SQLException ignore) { }
        }

        if(statement != null) {
            try {
                statement.close();
            } catch(SQLException ignore) { }
        }

        if(connection != null && !connection.isClosed()) {
            try {
                connection.close();
            } catch(SQLException ignore) { }
        }
    }
}
```

Java 7's *try-with-resources* has drastically simplified this task. Any class implementing `java.lang.AutoCloseable` is eligible for use in a *try-with-resources* construct. The JDBC `Connection`, `PreparedStatement`, and `ResultSet` interfaces extend this interface. When you use *try-with-resources* as shown in the following example, the resources you declare within the `try` keyword's parentheses are automatically closed in an implicit `finally` block. Any exceptions thrown during this cleanup are added to an existing exception's suppressed exceptions or, if there is no existing exception, are thrown after the resources have all been closed.

```
try(Connection connection = dataSource.getConnection();
    PreparedStatement statement = connection.prepareStatement(...))
{
    // set up statement
    try(ResultSet resultSet = statement.executeQuery())
    {
        // do something with result set
    }
}
catch(SQLException e)
{
    // do something with exception
}
```

Another improvement made to *try-catch-finally* is the addition of *multi-catch*. As of Java 7 you can now catch multiple exceptions within a single `catch` block, separating the exception types with a single pipe. For example:

```
try
{
    // do something
```

```

    }
    catch (MyException | YourException e)
    {
        // handle these exceptions the same way
    }
}

```

One caveat to keep in mind is that you can't multi-catch two or more exceptions such that one inherits from another. For example, the following is prohibited because `FileNotFoundException` extends `IOException`:

```

try {
    // do something
} catch (IOException | FileNotFoundException e) {
    // handle these exceptions the same way
}

```

Of course, this can easily be considered a matter of common sense. In this case, you would simply catch `IOException`, which would catch both types of exceptions.

A few other miscellaneous language features in Java 7 include *binary literals* for bytes and integers (you can write the literal 1928 as `0b11110001000`) and *underscores in numeric literals* (you can write the same literals as `1_928` and `0b111_1000_1000`, if desired). In addition, you can finally use `Strings` as `switch` arguments.

Java EE 7

Java EE 7, released on June 12, 2013, contains a number of changes and new features. You'll cover many of these new features throughout this book, so they are not detailed here. In summary, the changes to Java EE 7 are as follows:

- JAXB was added to Java SE 7 and is no longer included in Java EE.
- Batch Applications for the Java Platform 1.0 and Concurrency Utilities for Java EE 1.0 were added to Enterprise Application Technologies.
- Web Application Technologies picked up Java API for WebSockets 1.0 (which you learn about in Chapter 10) and Java API for JSON Processing 1.0.
- The Java Unified Expression Language has been significantly expanded to include lambda expressions and an analog of the Java SE 8 Collections Stream API. (You learn more about this in Chapter 6.)
- The Web Profile was expanded slightly to include specifications more likely to be required in common web applications: JAX-RS, Java API for WebSockets, and Java API for JSON Processing.

Java SE 8

The new features in Java SE 8 can come in very handy as you work the examples in this book. Perhaps most visible is the addition of *lambda expressions* (unofficially known as *closures*). Lambda expressions are anonymous functions that are defined, and possibly called, without being assigned

a type name or bound to an identifier. Lambda expressions are particularly useful for anonymously implementing those one-method interfaces that are so common in Java applications. For example, a `Thread` that was previously instantiated with an anonymous `Runnable` like this:

```
public String doSomethingInThread(String someArgument)
{
    ...
    Thread thread = new Thread(new Runnable() {
        @Override
        public void run()
        {
            // do something
        }
    });
    ...
}
```

can now be simplified with a lambda expression:

```
public String doSomethingInThread(String someArgument)
{
    ...
    Thread thread = new Thread(() -> {
        // do something
    });
    ...
}
```

Lambda expressions can have arguments, return types, and generics. And where desired, you can use a *method reference* instead of a lambda expression to pass a reference to an interface-matching method. The following code is also equivalent to the previous two instantiations of `Thread`. You can also assign method references and lambda expressions to variables.

```
public String doSomethingInThread(String someArgument)
{
    ...
    Thread thread = new Thread(this::doSomething);
    ...
}

public void doSomething()
{
    // do something
}
```

One of the biggest complaints among Java users since its earliest days is the lack of a decent date and time API. `java.util.Date` has always been rife with problems, and the addition of `java.util.Calendar` just made many problems worse. Java SE 8 finally addresses that with JSR 310, a new date and time API. This API is based largely on Joda Time, but with improvements to the underlying architecture to fix problems in it that the Joda Time inventor pointed out. This API is a revolutionary addition to the Java SE platform APIs and finally brings a powerful and well-designed date and time API to Java.

A Continuing Evolution

As you can tell, the Java SE and EE platforms were born together and have evolved hand-in-hand for nearly two decades. It's probable that they will continue to evolve together for many years or decades to come. You should be fairly familiar with Java SE, but it's possible you know absolutely nothing about using Java EE. It's also possible you're familiar with older Java EE versions but want to learn more about the new features in Java EE.

Part I of this book teaches you about the most important features in Java EE, including:

- Application servers and web containers (Chapter 2)
- Servlets (Chapter 3)
- JSPs (Chapters 4, 6, 7, and 8)
- HTTP sessions (Chapter 5)
- Filters (Chapter 9)
- WebSockets (Chapter 10).

UNDERSTANDING THE BASIC WEB APPLICATION STRUCTURE

A lot of components go into making a Java EE web application. First, you have your code and the third-party libraries it depends on. Then you have the deployment descriptor, which includes instructions for deploying and starting your application. You also have the `ClassLoader`s responsible for isolating your application from other web applications on the same server. Finally, you must package your application somehow, and for that you have WAR and EAR files.

Servlets, Filters, Listeners, and JSPs

Servlets are a key component of any Java EE web application. Servlets, which you learn about in Chapter 3, are Java classes responsible for accepting and responding to HTTP requests. Nearly every request to your application goes through a Servlet of some type, except those requests that are erroneous or intercepted by some other component. A filter is one such component that can intercept requests to your Servlets. You can use filters to meet a variety of needs, from data formatting, to response compression, to authentication and authorization. You explore the various uses of filters in Chapter 9.

As with many other different types of applications, web applications have a life cycle. There are both startup and shutdown processes, and many different things happen during these stages. Java EE web applications support various types of listeners, which you learn about throughout Parts I and II. These listeners can notify your code of multiple events, such as application startup, application shutdown, HTTP session creation, and session destruction.

Perhaps one of the most powerful Java EE tools at your disposal is the JavaServer Pages technology, or JSP. JSPs provide you with the means to easily create dynamic, HTML-based graphical user interfaces for your web applications without having to manually write `String`s of HTML to an `OutputStream` or `PrintWriter`. The topic of JSPs encompasses many different facets, including the

JavaServer Pages Standard Tag Library, the Java Unified Expression Language, custom tags, and internationalization and localization. You will spend significant time on these features in Chapter 4 and Chapters 6 through 9.

Of course, there are many more features in Java EE than just Servlets, filters, listeners, and JSPs. You will cover many of these in this book, but not all of them.

Directory Structure and WAR Files

Standard Java EE web applications are deployed as WAR files or “exploded” (unarchived) web application directories. You should already be familiar with *JAR*, or *Java Archive*, files. Recall that a JAR file is simply a ZIP-formatted archive with a standard directory structure recognized by JVMs. There is nothing proprietary about the JAR file format, and any ZIP archive application can create and read JAR files. A *Web Application Archive*, or *WAR*, file is the equivalent archive file for Java EE web applications.

All Java EE web application servers support WAR file application archives. Most also support exploded application directories. Whether archived or exploded, the directory structure convention, as shown in Figure 1-2, is the same. Like a JAR file, this structure contains classes and other application resources, but those classes are not stored relative to the application root as in a JAR file. Instead, the class files live in `/WEB-INF/classes`. The `WEB-INF` directory stores informational and instructional files that Java EE web application servers use to determine how to deploy and run the application. Its `classes` directory acts as the package root. All your compiled application class files and other resources live within this directory.

Unlike standard JAR files, WAR files can contain bundled JAR files, which live in `/WEB-INF/lib`. All the classes in the JAR files in this directory are also available to the application on the application’s classpath. The `/WEB-INF/tags` and `/WEB-INF/tld` directories are reserved for holding JSP tag files and tag library descriptors, respectively. You’ll explore the topic of tag files and tag libraries thoroughly in Chapter 8. The `i18n` directory is not actually part of the Java EE specifications, but it is a convention that most application developers follow for storing internationalization (i18n) and localization (L10n) files.

You probably also noticed the presence of two different `META-INF` directories. This can be a source of confusion for some developers, but if you remember the simple classpath rules, you can easily differentiate the two. Like JAR file `META-INF` directories, the root-level `/META-INF` directory contains the application manifest file. It can also contain resources for specific web containers or application servers. For example, Apache Tomcat (which you’ll learn about in Chapter 2) looks for and uses a `context.xml` file in this directory to help customize how the application is deployed in Tomcat. None of these files

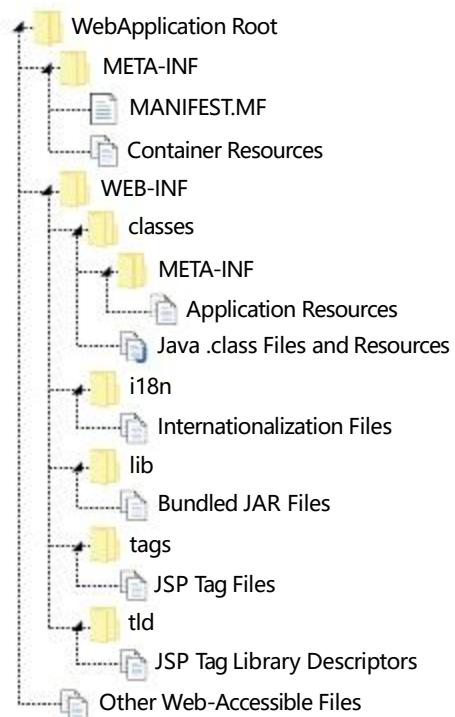


FIGURE 1-2

are part of the Java EE specification, and the supported files can vary from one application server or web container to the next.

Unlike JAR files, the root-level `/META-INF` directory is *not* on the application classpath. You cannot use the `ClassLoader` to obtain resources in this directory. `/WEB-INF/classes/META-INF`, however, is on the classpath. You can place any application resources you desire in this directory, and they become accessible through the `ClassLoader`. Some Java EE components specify files that belong in this directory. For example, the Java Persistence API (which you'll learn about in Part III of this book) specifies two files — one named `persistence.xml` and another `orm.xml` — that live in `/WEB-INF/classes/META-INF`.

Most files contained within a WAR file or exploded web application directory are resources directly accessible through a URL. For example, the file `/bar.html` relative to the root of an application deployed to `http://example.org/foo` is accessible from `http://example.org/foo/bar.html`. In the absence of any filter or security rules to the contrary, this holds true for *all* resources in your application except those resources under the `/WEB-INF` and `/META-INF` directories. The files in these directories are protected resources that are not accessible via URL.

The Deployment Descriptor

The deployment descriptor is the metadata that describes the web application and provides instructions to the Java EE web application server for deploying and running the web application. Traditionally, all this metadata came from the deployment descriptor file, `/WEB-INF/web.xml`. This file contains definitions for Servlets, listeners, and filters, and configuration options for HTTP sessions, JSPs, and the application in general. Servlet 3.0 in Java EE 6 added the ability to configure web applications using annotations and a Java configuration API. It also added the notion of web fragments — JAR files within your application can contain Servlets, filters, and listeners configured in `/META-INF/web-fragment.xml` deployment descriptors within the necessary JAR files. Web fragments can also use annotations and the Java configuration API.

This change to the deployment of web applications in Java EE 6 added significant complexity to the task of organizing this process. To ease this complexity, you can configure the order of your web fragments so that they are scanned and activated in a specific sequence. This happens one of two ways:

- Each web fragment's `web-fragment.xml` file can contain an `<ordering>` element that uses nested `<before>` and `<after>` tags to control whether the web fragment activates before or after other web fragments. These tags contain nested `<name>` elements to specify the name of another fragment relative to which the current fragment should be ordered. `<before>` and `<after>` can alternatively contain nested `<others>` elements to indicate that the fragment should activate before or after any other fragments not specifically named.
- If you didn't create a particular web fragment and don't have control over its contents, you can still control the order of your web fragments within your application's deployment descriptor. The `<absolute-ordering>` element in `/WEB-INF/web.xml`, together with its nested `<name>` and `<others>` elements, configures an absolute order for bundled web fragments that overrides any order instructions that come with the web fragments.

By default, Servlet 3.0 and newer environments scan web applications and web fragments for Java EE web application annotations for configuring Servlets, listeners, filters, and more. You can disable this scanning and disable annotation configuration by adding the attribute `metadata-complete="true"` to the root `<web-app>` or `<web-fragment>` elements as needed. You can also disable all web fragments in your application by adding `<absolute-ordering />` (without any nested elements) to your deployment descriptor.

You learn more about the web application deployment descriptor and annotation configuration throughout Part I of the book. In Part II, you explore the container initializer and programmatic configuration with the Java API, and see how it can make bootstrapping Spring Framework easier and testable.

Class Loader Architecture

When working with Java EE web applications, it's essential to understand the `ClassLoader` architecture because it differs from the architecture to which you are accustomed in standard Java SE applications. In a typical application, the `java.*` classes that come with the Java SE platform are loaded in a special root `ClassLoader` that cannot be overridden. This is a security measure that prevents malicious code from, for example, replacing the `String` class or redefining `Boolean.TRUE` and `Boolean.FALSE`.

After this `ClassLoader` comes the extension `ClassLoader`, which loads classes from the extensions JARs in the JRE installation directory. Finally, the application `ClassLoader` loads all other classes in the application. This forms a hierarchy of `ClassLoaders`, with the root serving as the earliest ancestor for all `ClassLoaders`. When a lower-level `ClassLoader` is asked to load a class, it always delegates to its parent `ClassLoader` first. This continues up until the root `ClassLoader` is checked. With the exception of the root `ClassLoader`, a `ClassLoader` loads a class from its collection of JARs and directories *only* if its parent `ClassLoader` first fails to find the class.

This method of class loading is called the *parent-first class loader delegation model*, and although it works great for many types of applications, it is not ideal for most Java EE web applications. A server that runs Java EE web applications is typically extraordinarily complex and a number of vendors could provide its implementation. The server could use some of the same third-party libraries that your application uses, but they may be of conflicting versions. In addition, different web applications could also provide conflicting versions of the same third-party libraries, leading to even more problems. To solve these problems, you need a *parent-last class loader delegation model*.

In Java EE web application servers, each web application is assigned its own isolated `ClassLoader` that inherits from the common server `ClassLoader`. By isolating the applications from each other, they cannot access each other's classes. This not only eliminates the risk of conflicting classes, but it also serves as a security measure preventing web applications from interfering with or harming other web applications. In addition, a web application `ClassLoader` (typically) asks its parent to load a class only if it can't load the class itself first. In this way, the class loading is delegated to the parent last instead of the parent first, and web application classes and libraries are preferred over those that the server supplies. To maintain the protected status of bundled Java SE classes, web application `ClassLoaders` still check the root `ClassLoader` before attempting to load any classes. Although this delegation model is more preferable for web applications in nearly

all cases, there are still rare circumstances in which it is not appropriate. For this reason, Java EE-compliant servers provide the capability of changing the delegation model from parent-last back to parent-first.

Enterprise Archives

You've learned about WAR files, but there's another type of Java EE archive that you should know about: *EAR* files. An *Enterprise Archive* is a collection of JAR files, WAR files, and configuration files compressed into a single, deployable archive (in ZIP format, just like JARs and WARs).

Figure 1-3 shows a sample EAR file. As with a WAR file, the root `/META-INF` directory contains the archive manifest and is not available to the application classpath. The `/META-INF/application.xml` file is a special deployment descriptor that describes how to deploy the various components included within the EAR file. At the root level of an EAR file are all the web application modules included within it — one WAR file for each module. There is nothing special about these WAR files; they can have all the same contents and features as a normal, standalone WAR file. The EAR file can also contain JAR libraries, which can serve many purposes. The JAR files can contain Enterprise JavaBeans declared in the `/META-INF/application.xml` deployment descriptor, or they can be simple third-party libraries that two or more WAR modules share within the enterprise archive.

As you might have figured, enterprise archives also come with their own `ClassLoader` architecture. Typically, an additional `ClassLoader` is inserted into the hierarchy between the server `ClassLoader` and the web application `ClassLoaders` assigned to each module. This `ClassLoader` isolates the enterprise application from other enterprise applications but enables multiple modules in a single EAR to share common libraries contained within the EAR. This new `ClassLoader` can use either the parent-last (default) or parent-first delegation models. The web application `ClassLoaders` can then either delegate parent-first (enabling EAR library classes to take precedence) or parent-last (enabling WAR classes to take precedence).

Although it is useful to understand enterprise archives, they are a feature of the full Java EE specification, and most web container-only servers (such as Apache Tomcat) do not support them. As such, they are not discussed further in this book.

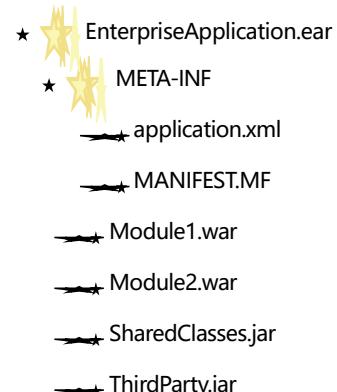


FIGURE 1-3

WARNING *The `ClassLoader` examples described in this section are just that — examples. Though the Java EE specifications do describe parent-first and parent-last class loading, different implementations achieve these models in different ways, and each server could have certain nuances that might cause problems depending on your needs. You should always read the documentation of the server you choose so that you can determine whether the `ClassLoader` architecture of that particular server is appropriate for you.*

SUMMARY

In this chapter you explored the histories of the Java Platform, Standard Edition and Java Platform, Enterprise Edition and learned how the two platforms evolved together over the last 19 years. You were briefly introduced to some of the topics covered in this book—Servlets, filters, listeners, JSPs, and more—and saw how Java EE applications are structured, both internally and on the filesystem. You then learned about web application archives and enterprise archives and how they serve as vessels for transporting and deploying Java EE applications.

The rest of the book explores these topics in much greater detail, answering the many questions that you likely have after reading the last several pages. In Chapter 2 you take a closer look at application servers and web containers, what they are, and how to choose one for your purposes. You also learn how to install and use Tomcat for the examples in this book.

2

Using Web Containers

IN THIS CHAPTER

- Choosing a web container
- Installing Tomcat on your machine
- Deploying and undeploying applications in Tomcat
- Debugging Tomcat from IntelliJ IDEA
- Debugging Tomcat from Eclipse

WROX.COM CODE DOWNLOADS FOR THIS CHAPTER

You can find the `wrox.com` code downloads for this chapter <http://www.wrox.com/go/projavaforwebapps> on the Download Code tab. The code for this chapter is divided into the following major examples:

- sample-deployment WAR Application File
- Sample-Debug-IntelliJ Project
- Sample-Debug-Eclipse Project

NEW MAVEN DEPENDENCIES FOR THIS CHAPTER

There are no Maven dependencies for this chapter.

CHOOSING A WEB CONTAINER

In the previous chapter you were introduced to the Java Platform, Enterprise Edition, and the concepts of Servlets, filters, and other Java EE components. You also learned about some of the new features in Java 7 and 8. Java EE web applications run within Java EE *application servers* and *web containers* (also known as *Servlet containers*, and this book uses the terms interchangeably).

Although the Java EE specification is full of many smaller sub-specifications, most web containers implement only the Servlet, JSP, and JSTL specifications. This is different from full-blown Java EE application servers, which implement the entire Java EE specification. Every application server contains a web container, which is responsible for managing the life cycle of Servlets, mapping request URLs to Servlet code, accepting and responding to HTTP requests, and managing the filter chain, where applicable. However, standalone web containers are often lighter-weight and easier to use when you don't require the entire feature set of Java EE.

Choosing a web container (or an application server, for that matter) is a task that requires careful research and consideration for the requirements of your project. You have many options for choosing a web container, and each has its advantages and challenges. You may use a variety of web containers. For example, you may decide to use Apache Tomcat for local testing on your developers' machines while using GlassFish for your production environment. Or you may write an application that your customers deploy on their own servers, in which case you probably want to test on many different application servers and web containers.

In this section you learn about some common web containers and application servers, and in the remaining sections you take a closer look at the one you use for the rest of this book.

Apache Tomcat

Apache Tomcat is the most common and popular web container available today. Sun Microsystems software engineers originally created this web container as the Sun Java Web Server, and it was the original reference implementation of the Java EE Servlet specification. Sun later donated it to the Apache Software Foundation in 1999, and at that point it became Jakarta Tomcat and eventually Apache Tomcat. It is also interesting to note that Apache's evolution of Tomcat led to the development of the Apache Ant build tool, which thousands of commercial and open source projects use today.

Tomcat's primary advantages are its small footprint, simple configuration, and long history of community involvement. Typically, developers can be up-and-running with a functional Tomcat installation in 5 to 10 minutes, including download time. Tomcat requires very little configuration out-of-the-box to run well on a development machine, but it can also be tuned significantly to perform well in high-load, high-availability production environments. You can create large Tomcat clusters to handle huge volumes of traffic reliably. Tomcat is often used in commercial production environments due to its simplicity and lightweight profile. However, Tomcat lacks the sophisticated web management interface than many of its competitors offer for configuring the server. Instead, Tomcat provides only a simple interface for basic tasks, such as deploying and undeploying applications. For further configuration, administrators must manipulate a collection of XML and Java properties files. In addition, because it is not a full application server, it lacks many Java EE components, such as the Java Persistence API, the Bean Validation API, and the Java Message Service.

As you can imagine, this makes Tomcat great for many tasks but does make deploying more complex enterprise applications challenging and, sometimes, impossible. If you like Tomcat but need a full Java EE application server, you can turn to Apache TomEE, which is built on Tomcat but offers a full implementation of all the Java EE components. Being built on Tomcat, it has the full force of the Tomcat community and more than a decade of testing behind it. Apache also offers Geronimo, another open source full Java EE application server.

NOTE *TomEE and Geronimo are both Oracle-certified Java EE application servers, meaning they have been verified to be in compliance with all aspects of the Java EE specification. Because Tomcat is only a web container, it has no such certification. However, its huge user base and active community ensure that it accurately implements the Java EE components it provides.*

Tomcat provides implementations of the Servlet, Java Server Pages (JSP), Java Unified Expression Language (EL), and WebSocket specifications. Table 2-1 lists several Tomcat versions and the specifications they implement. Only Tomcat 6, 7, and 8 are still supported. Versions 3.3, 4.1, and 5.5 reached end of life years ago. You can read more about Apache Tomcat on the Tomcat website.

TABLE 2-1: Tomcat Versions and Their Specifications

TOMCAT VERSION	JAVA EE*	SERVLET	JSP	EL	WEBSOCKET	MIN. JAVA SE VERSION
3.3.x	1.2	2.2	1.1	—	—	1.1
4.1.x	1.3	2.3	1.2	—	—	1.3
5.5.x	1.4	2.4	2.0	—	—	1.4
6.0.x	5	2.5	2.1	2.1	—	5.0
7.0.x	6	3.0	2.2	2.2	—	6
8.0.x	7	3.1	2.3	3.0	1.0	7

* The Java EE column indicates only the equivalent Java EE version; Tomcat is not an application server and does not implement Java EE. A hyphen in a column indicates that the Tomcat version did not implement that particular specification.

GlassFish

GlassFish Server is an open source and commercial full Java EE application server implementation. It provides all the features in the Java EE specification, including a web container, and is currently the reference implementation for the Java EE specification. Its web container is actually a derivative of Apache Tomcat; however, it has evolved considerably since the Tomcat core was forked to create GlassFish, and the code is hardly recognizable today. The open source edition of GlassFish offers community support, whereas the commercial Oracle GlassFish Server provides paid, commercial support through Oracle Corporation. Oracle is only offering commercial support through Java EE 7. Starting with Java EE 8, GlassFish will not include a commercial support option.

One of GlassFish's strengths is its management interface, which provides a graphical web user interface, a command-line interface, and configuration files to configure anything within the server. Server administrators can even use the management interface to deploy new GlassFish instances within a GlassFish cluster. As the reference implementation, it is also always the first server to roll out a new version whenever the specification is updated. The first version of GlassFish was

released in May 2006, and implemented the Java EE 5 specification. In September 2007, version 2.0 added support for full clustering capabilities. Version 3.0 — the reference implementation for Java EE 6, released in December 2009 — included several enterprise improvements. This version represented a turning point in GlassFish's popularity, and it became extremely simple to manage an enterprise clustered GlassFish environment. In July 2011, version 3.1.1 improved several enterprise features and added support for Java SE 7, though Java SE 6 was still the minimum required version. GlassFish 4.0 released in June 2013 as the reference implementation of Java EE 7 and requires a minimum Java SE 7.

You can read more about GlassFish, and download it if you want, at the GlassFish website.

JBoss and WildFly

Red Hat's JavaBeans Open Source Software Application Server (JBoss AS) was the second-most popular Java EE server, next to Tomcat, as of early 2013. Historically, JBoss AS has been a web container with Enterprise JavaBeans support and some other Java EE features. Eventually it became Web Profile-certified and, in 2012, became certified as a full Java EE application server. Over time, the name JBoss also became synonymous with a development community (like Apache) that provided several products, as well as the commercial JBoss Enterprise Application Platform. The application server retained the name JBoss AS through version 7.1.x, but in 2012, the community decided that the name was the source of too much confusion due to other JBoss projects. The application server was renamed to WildFly as of version 8.0, released in early 2014.

Similar to GlassFish, WildFly is open source with free support provided by the JBoss Community and paid, commercial support provided by Red Hat. It has a comprehensive set of management tools and provides clustering and high-availability capabilities like Tomcat and GlassFish. JBoss AS versions 4.0.x through 4.2.x were built atop Tomcat 5.5 and supported Java EE 1.4 features. Version 5.0 introduced Java EE 5 support and a brand new web container, and 5.1 contained early implementations of some Java EE 6 features (although it was still a Java EE 5 application server). JBoss AS 6.0 implemented the Java EE 6 Web Profile, but it did not seek or obtain a Java EE 6 application server certification. JBoss AS 7.0 represented a complete rewrite of the product to dramatically decrease its footprint and increase its performance, and also supported only the Java EE 6 web profile. It was not until JBoss AS 7.1 that it again became a full application server, achieving Java EE 6 certification more than 2 years after Java EE 6 was released. WildFly 8.0 is a full Java EE 7 application server and requires a minimum of Java SE 7. (Actually, all Java EE 7 application servers and web containers require a minimum of Java SE 7.)

You can learn more about and download JBoss AS 7.1 and earlier at the JBoss website, whereas you can find WildFly 8.0 at the WildFly website.

Other Containers and Application Servers

There are many other web containers, such as Jetty and Tiny, and open source full Java EE application servers, such as JOnAS, Resin, Caucho, and Enhydra. There are also a number of commercial full application servers, of which Oracle WebLogic and IBM WebSphere are the most popular. Table 2-2 shows some of these servers and the versions that supported various Java EE specifications.

TABLE 2-2: Container and Application Server Versions

SERVER	J2EE 1.2	J2EE 1.3	J2EE 1.4	JAVA EE 5	JAVA EE 6	JAVA EE 7
Jetty*	3.x	4.x	5.x	6.x: J2SE 1.4 7.x: Java SE 5.0	8.x: Java SE 6 9.0.x: Java SE 7	9.1.x
WebLogic	6.x	7.x-8.x	9.x	10.x: Java SE 6 11g PS5: Java SE 7	12c	12.1.4**
WebSphere	4.x	5.x	6.x	7.x	8.x: Java SE 6 8.5.x: Java SE 7	9.x**

* Web container only; not a full application server

** These are speculated versions — Oracle and IBM have not officially announced Java EE 7 support yet.

Each web container or application server has its own advantages and disadvantages. The task of picking an application server cannot be covered in a single chapter and is beyond the scope of this book. The needs of your organization's project must be understood, and the right web container or application server that meets those needs should be chosen. Operational budgets must be considered because commercial application servers tend to have an extremely high cost of licensing. All these factors will impact your decision, and you may pick a server that isn't even listed in this book.

Why You'll Use Tomcat in This Book

Many of the advantages of Apache Tomcat (which is referred to simply as Tomcat for the rest of this book) have already been outlined. Perhaps most important for this book is the ease with which developers can start using Tomcat. By far, Tomcat is easier to get running quickly than any other web container, and it provides all the features that you need to complete the examples in this book. In addition, all the major Java IDEs provide tools to run, deploy on, and debug Tomcat, making it easier for you to develop your application.

Although some developers prefer using other web containers — and with the right knowledge nearly any web container can serve you well on a development machine — it's hard to make a case against using Tomcat. By using Tomcat for this book, you can focus on the code and development practices, paying little-to-no attention to the management of your container. The rest of this chapter helps you get Tomcat installed and set up on your machine. It also introduces you to deploying and undeploying applications with the Tomcat manager and debugging Tomcat in your Java IDE.

INSTALLING TOMCAT ON YOUR MACHINE

Before you can install Tomcat on your machine, you need to download it from the Tomcat project site. Go to the Tomcat 8.0 Downloads Page, and scroll down to the "Binary Distributions" section. There are many downloads on this page, and the only ones you need for this book are under the "Core" heading. As a Windows user, the two downloads you are concerned with are the "32-bit/64-bit Windows Service Installer" (works for any system architecture) and the

"32-bit Windows zip" or "64-bit Windows zip" (depending on your machine architecture). If you run on Linux, Mac OS X, or some other operating system, you need the non-Windows zip, which is just called "zip."

Installing as a Windows Service

Many developers want to install Tomcat as a Windows service. This has several advantages, especially in a quality assurance or production environment. It makes management of JVM memory and other resources easier, and it greatly simplifies starting Tomcat automatically when Windows boots. However, in a development environment, installing Tomcat as a service can have some drawbacks. This technique installs only the service and does not install the command-line scripts that run Tomcat from the command line. Most IDEs use these command-line scripts to run and debug Tomcat from within the IDE. You may install Tomcat as a service by downloading the "32-bit/64-bit Windows Service Installer," but you also need to download the "Windows zip" to run Tomcat from your IDE.

This book does not cover installing Tomcat as a windows service because you would usually do this only for production or QA environments. The documentation on the Tomcat website is very helpful if you want to explore this further. Of course, if you are not using Windows, the Windows installer will be of no use to you. There are ways to start Tomcat automatically in other operating systems, but they are also outside the scope of this book.

Installing as a Command-Line Application

Most application developers need to run Tomcat only as a command-line application and usually only from their IDE. To do this, follow these steps:

1. Download the architecture-appropriate Windows zip (if you use Windows) or the non-Windows zip (if you use anything else) from the Tomcat 8.0 download page and unzip the directory.
2. Place the contents of the Tomcat directory in this zip file into the folder `C:\Program Files\Apache Software Foundation\Tomcat 8.0` on your local machine (or into the appropriate directory for a server in your operating system). For example, the `webapps` directory should now be located at `C:\Program Files\Apache Software Foundation\Tomcat 8.0\webapps`.
3. If you use Windows 7 or newer, you need to change some permissions to make Tomcat accessible from your IDE. Right-click the `Apache Software Foundation` directory in `C:\Program Files` and click Properties. On the Security tab, click the Edit button. Add your user or the Users group, and give that entry full control over the directory.
4. To configure Tomcat for its first use, start by opening the file `conf/tomcat-users.xml` in your favorite text editor. Place the following tag between the `<tomcat-users>` and `</tomcat-users>` XML tags:

```
<user username="admin" password="admin" roles="manager-gui,admin-gui" />
```

WARNING *This configures an admin user that you can use to log in to Tomcat's web management interface. Of course, this username and password combination is very insecure and should never be used for production or publicly facing servers. However, for testing on your local machine it is sufficient.*

5. Open the `conf/web.xml` file. Search the file for the text `org.apache.jasper.servlet.JspServlet`. Below the tag that contains this text are two `<init-param>` tags. You learn about Servlet init parameters in the next chapter, but for now add the following init parameters below the existing init parameters:

```
<init-param>
    <param-name>compilerSourceVM</param-name>
    <param-value>1.8</param-value>
</init-param>
<init-param>
    <param-name>compilerTargetVM</param-name>
    <param-value>1.8</param-value>
</init-param>
```

By default, Tomcat 8.0 compiles JavaServer Pages files with Java SE 6 language support even if it runs on Java SE 8. These new Servlet init parameters instruct Tomcat to compile JSP files with Java SE 8 language features, instead.

6. After you make these changes and save these files, you should now be ready to start up Tomcat and make sure that it runs properly. Open up a command prompt and change your directory to the Tomcat home directory (`C:\Program Files\Apache Software Foundation\Tomcat 8.0`).
7. Type the command `echo %JAVA_HOME%` (or `echo $JAVA_HOME` on a non-Windows operating system) and press Enter to check whether the `JAVA_HOME` environmental variable is properly set to your Java Development Kit (JDK) home directory. If it is not, configure the environmental variable, and then log out and back in before proceeding (see the Note that follows). Tomcat cannot run without this variable properly set.
8. Type the command `bin\startup.bat` (or `bin/startup.sh` if you do not use Windows) and press Enter. A Java console window should open showing the output of the running Tomcat process. After a few seconds, you should see the message "INFO [main] org.apache.catalina.startup.Catalina.start Server startup in 1827 ms" or something similar in the console window. This means Tomcat has started properly.

NOTE *When starting, Tomcat initially looks for the `JRE_HOME` environmental variable and uses that if it is set. If it isn't, it next looks for the `JAVA_HOME` variable. If neither is set, Tomcat fails to start. However, to debug Tomcat you must have `JAVA_HOME` set, so it's best to simply go ahead and configure that.*

9. Open your favorite Web browser and navigate to `http://localhost:8080/`. You should see a page that looks like Figure 2-1. This means that Tomcat is running and JSPs are compiling properly with Java SE 8. If this screen does not come up or you observe an error

in the Java console, you need to check the preceding steps and possibly consult the Tomcat documentation.

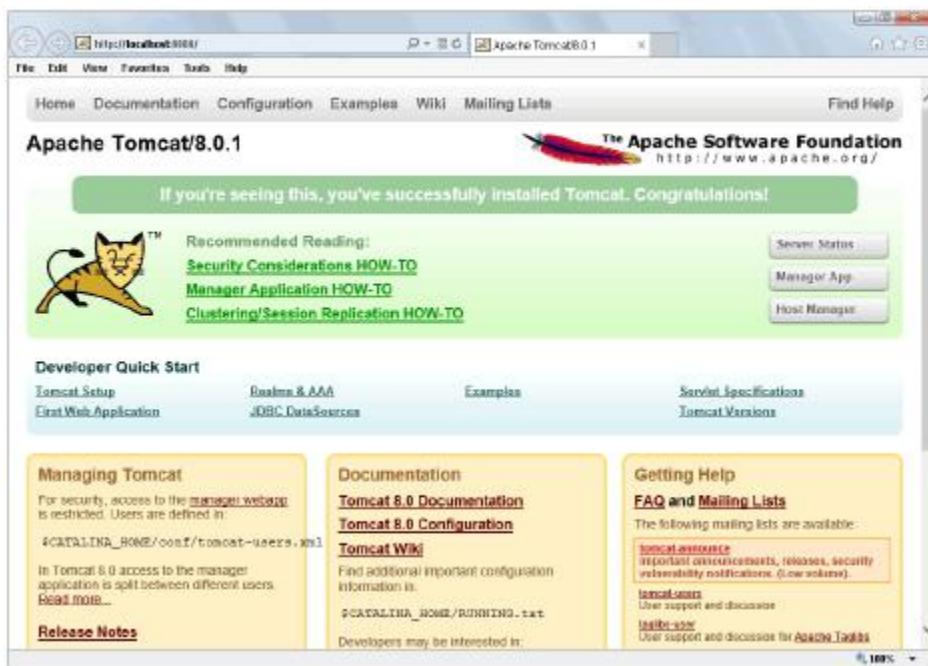


FIGURE 2-1

When you finish using Tomcat, you can stop it by running the command `bin\shutdown.bat` (or `bin/shutdown.sh`) in the command prompt in the Tomcat 8.0 home directory. The Java console window should close, and Tomcat will stop. However, do not do this yet; in the next section, you explore deploying and undeploying applications in Tomcat. (If you have already shut down Tomcat, don't worry about it. It's easy to start it back up again.)

WARNING *The earliest releases of Tomcat 8.0 do not support compiling JSPs for Java 8. You'll know that this is the case for your release if you see "WARNING: Unknown source VM 1.8 ignored" or similar in the Java console. If so, you need to complete the following steps for "Configuring a Custom JSP Compiler."*

Configuring a Custom JSP Compiler

Tomcat ships with and uses the Eclipse JDT compiler for compiling JavaServer Pages files in web applications. (You learn more about JSP files and how they compile in Chapter 4.) This enables Tomcat to run properly without requiring a JDK installation. Using the Eclipse compiler, all you need is a simple Java Runtime Edition (JRE) installation. Because JSPs are usually very simple,

the Eclipse compiler is typically quite adequate for any Tomcat environment. However, there are circumstances for which you don't want to use the Eclipse compiler. Perhaps you find a bug in the Eclipse compiler that prevents one of your JSPs from compiling. Or if a new version of Java comes out with language features you want to use in your JSPs, it could be some time before Eclipse has a compatible compiler. Whatever reason you may have, you can easily configure Tomcat to use the JDK compiler instead of Eclipse.

1. Open Tomcat's `conf/web.xml` file back up and find the `JspServlet` again.
2. Add the following init parameter, which tells the Servlet to use Apache Ant with the JDK compiler to compile JSPs instead of the Eclipse compiler.

```
<init-param>
    <param-name>compiler</param-name>
    <param-value>modern</param-value>
</init-param>
```

3. Tomcat doesn't have a way to use the JDK compiler directly, so you must have the latest version of Ant installed on your system. You also need to add the JDK's `tools.jar` file and Ant's `ant.jar` and `ant-launcher.jar` files to your classpath. The easiest way to do this is to create a `bin\setenv.bat` file and add the following line of code to it (ignore new lines here), replacing the file paths as necessary for your system.

```
set "CLASSPATH=C:\path\to\jdk8\lib\tools.jar;C:\path\to\ant\lib\ant.jar;
C:\path\to\ant\lib\ant-launcher.jar"
```

Of course, this applies only to Windows machines. For non-Windows environments, you should instead create a `bin/setenv.sh` file with the following contents, replacing the file paths as necessary for your system:

```
export CLASSPATH=/path/to/jdk8/lib/tools.jar:/path/to/ant/lib/ant.jar:
/path/to/ant/lib/ant-launcher.jar
```

When running Tomcat with such a custom JSP compilation configuration, be sure to carefully observe the output in the Tomcat logs. If Tomcat cannot find Ant or Ant cannot find the JDK compiler, Tomcat automatically falls back to the Eclipse compiler and outputs only a warning to the logs.

DEPLOYING AND UNDEPLOYING APPLICATIONS IN TOMCAT

In this section you learn how to deploy and undeploy Java EE web applications in Tomcat. You have two options for accomplishing this:

- Manually by placing the application in the `webapps` directory
- Using the Tomcat manager application

If you have not already done so, you should download the `sample-deployment.war` sample application from the Chapter 2 section on the wrox.com download site. This is what you should use to practice deployment and undeployment.

Performing a Manual Deploy and Undeploy

Deploying an application manually on Tomcat is simple—just place the `sample-deployment.war` file in Tomcat's `webapps` directory. If Tomcat is running, within a few moments Tomcat should automatically unpack the application file into a directory with the same name minus the `.war` extension. If Tomcat is not running, you can start it, and the application file will unpack as Tomcat starts. When the application has unpacked, open your browser and navigate to `http://localhost:8080/sample-deployment/`. You should see a page that looks like Figure 2-2. This means that the sample application has successfully deployed.

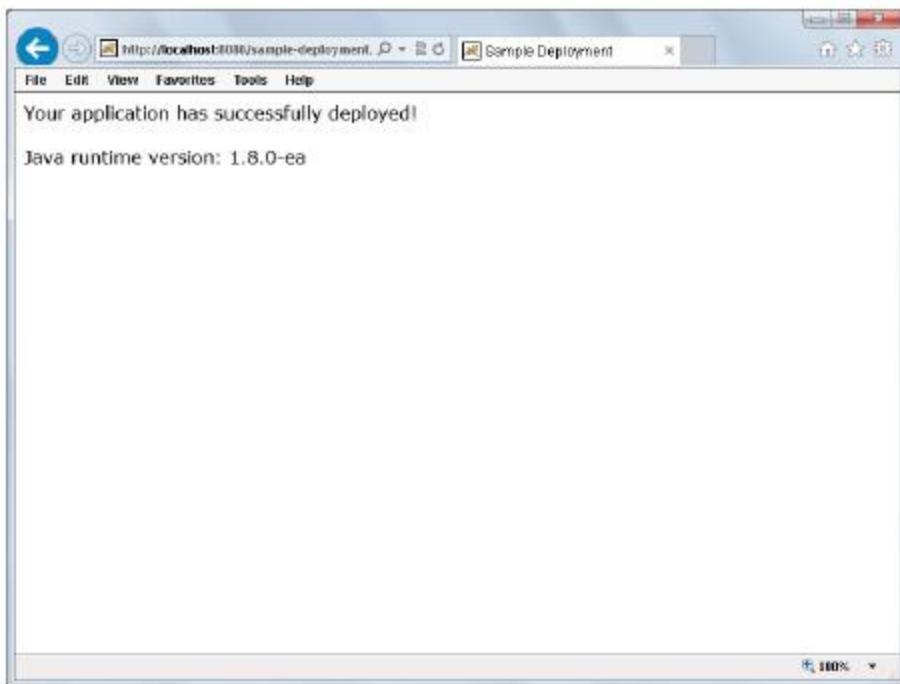


FIGURE 2-2

Undeploying the application is as simple as reversing the process. Delete the `sample-deployment.war` file and wait a few moments. When Tomcat detects that the file was deleted, it undeploys the application and deletes the unpacked directory, and the application will no longer be accessible from your browser. You do not need to shut down Tomcat to perform this task.

Using the Tomcat Manager

You can also deploy a Java EE application using the Tomcat manager web interface. To do so, follow these steps:

1. Open your browser and navigate to `http://localhost:8080/manager/html`.
2. When you are prompted for a username and password, enter **admin** for the username and **admin** for the password (or whatever you configured in `conf/tomcat-users.xml`). The page you are presented with should look like Figure 2-3.

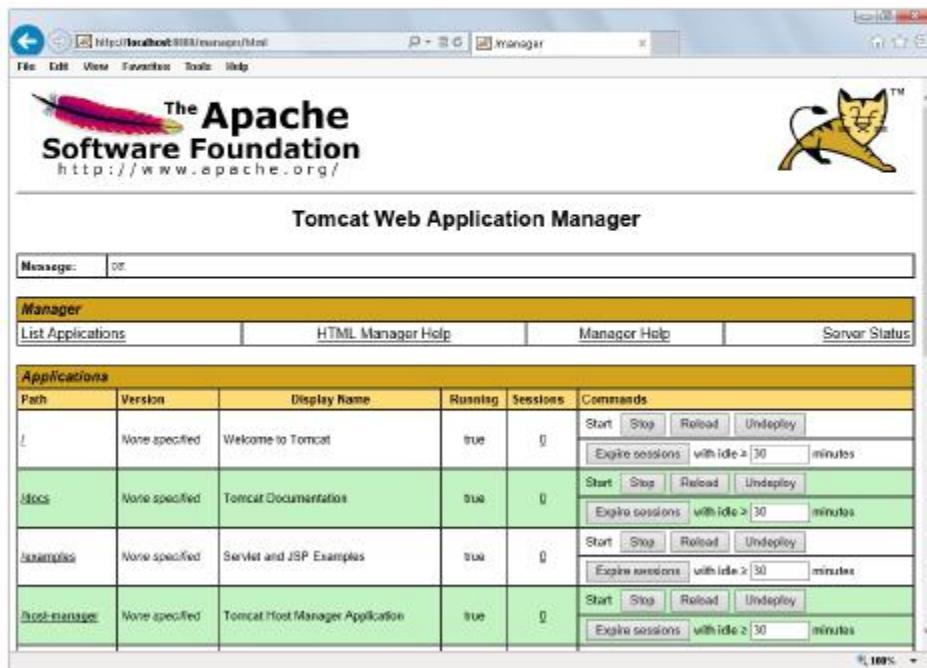


FIGURE 2-3

3. Scroll down to the Deploy section and find the form "WAR file to deploy." In the "Select WAR file to upload" field, choose the `sample-deployment.war` file from your filesystem, as shown in Figure 2-4, and then click the Deploy button. The WAR file uploads to Tomcat, which deploys the application. The `sample-deployment` directory is again created in Tomcat's `webapps` directory. When complete, Tomcat returns you to the list of applications where you can see that the sample application has been deployed, as shown in Figure 2-5.
4. Like before, you can go to `http://localhost:8080/sample-deployment/` and view the sample page in the sample application.

You have now deployed the application using the Tomcat manager.

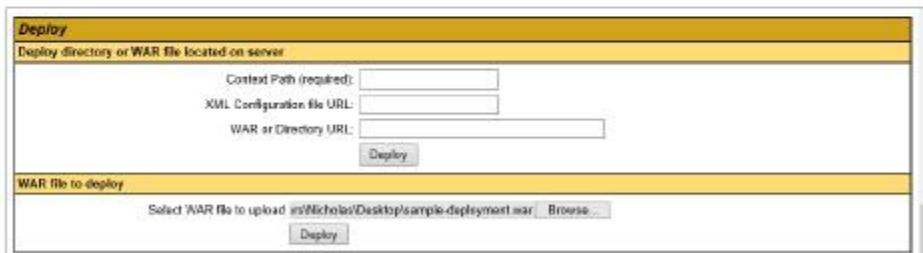


FIGURE 2-4

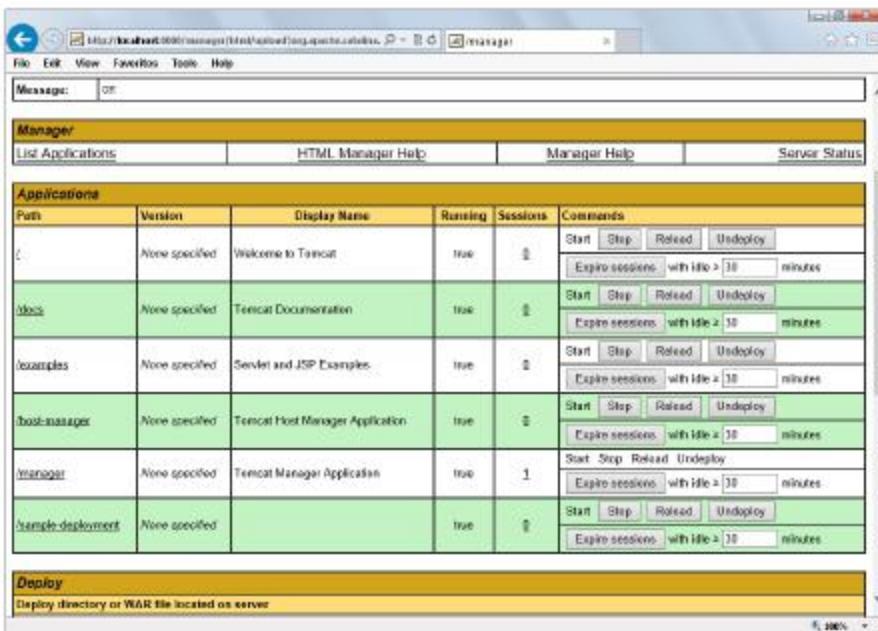


FIGURE 2-5

Undeploying is just as easy to accomplish. On the Tomcat manager page you saw earlier, you should notice an Undeploy button next to the sample application (refer to Figure 2-5). Click this button and the sample application will be undeployed and removed from the `webapps` directory. When complete, you can no longer access the application at `http://localhost:8080/sample-deployment/`.

DEBUGGING TOMCAT FROM YOUR IDE

As a Java EE developer, one of the most important skills you can hold is the ability to deploy and debug applications in Tomcat from your Java IDE. This provides you with immeasurable troubleshooting skills for determining why an application won't run or figuring out why the bug your customer reported occurs. This section covers setting up, running, and debugging web applications in Tomcat using both IntelliJ IDEA and Eclipse. You can read both sets of instructions or just the set that pertains to the IDE you have chosen — that choice is up to you.

Throughout the rest of this book is very little instruction for doing this. This keeps the text decoupled from any particular IDE. You also do not see any IDE-specific screenshots after this chapter. Be sure you are familiar and comfortable with deploying and debugging applications in Tomcat using your IDE before moving on, even if that means going over this section several times.

Using IntelliJ IDEA

If you use IntelliJ IDEA 13 or newer, you have just a few simple steps to take to get up and running with your web applications. The first thing you need to do is set up IntelliJ to recognize

your local Tomcat or other container installation. This is a one-time-only step — you set it up once in your global IDE settings, then you can use the application server for any web application project. Next, set up each web application project to use your configured container. Finally, you just need to start your application from IntelliJ and place breakpoints where you'd like to debug your application.

Setting Up Tomcat 8.0 in IntelliJ

To start, you need to configure Tomcat in IntelliJ's list of application servers.

1. Open up IntelliJ's IDE settings dialog. With a project open you can go to **File \leftrightarrow Settings**, or click the Settings icon in the toolbar (shown here in the margin), or press **Ctrl + Alt + S**. If you don't have a project open, you can click the **Configure** button and then the **Settings** button.
2. In the left pane of the Settings dialog, click **Application Servers** under **IDE Settings**. Initially, you have no application servers configured.
3. Click the green plus icon to add a new application server. Click the **browse** button next to the **Tomcat Home** field to browse for and select the Tomcat home directory (for example, `C:\Program Files\Apache Software Foundation\Tomcat 8.0`). Then click **OK**. IntelliJ should automatically detect your Tomcat version, and the dialog should look like Figure 2-6.

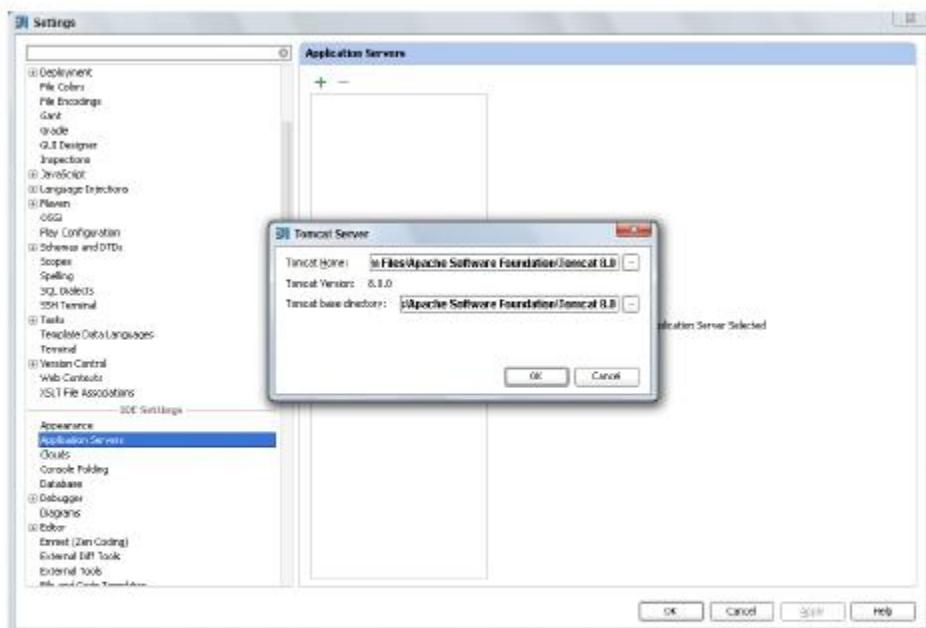


FIGURE 2-6

4. Click OK again to complete adding Tomcat to your list of application servers, and change the name if you want. All the IntelliJ code samples you can download for this book assume an application server name of **Tomcat 8.0**, so for maximum ease you should rename it to Tomcat 8.0 if it is named something else.
5. Click Apply to save the changes and OK to close the Settings dialog.

Adding a Tomcat Configuration to a Project

After you create a project and are ready to deploy it to Tomcat from IntelliJ, you need to add a Tomcat run/debug configuration to your project.

1. Click the run/debug configurations icon (a down arrow) on the toolbar, then click Edit Configurations.



- 2. In the dialog that appears, click the green plus icon, scroll to the bottom of the Add New Configuration menu, hover over Tomcat Server, and click Local. This creates a run/debug configuration for running your project against a local Tomcat, as shown in Figure 2-7.
- 3. If Tomcat 8.0 is the only application server you have added to IntelliJ, it is automatically selected as the application server this run/debug configuration will use. If you have other application servers configured, one of those might be selected, in which case you need to click the "Application server" drop-down and select Tomcat 8.0 instead.
- 4. Name the run configuration something meaningful. In Figure 2-7 and in all the sample IntelliJ projects you download for this book, the run configuration is named Tomcat 8.0 like the application server it uses.
- 5. You'll probably see a warning that no artifacts are marked for deployment. Correcting this is simple. Click the Deployment tab and then the green plus icon under the "Deploy at the server startup" heading. Click Artifact, and then click the exploded war file artifact. Click OK. Change the "Application context" name for the artifact deployment to the server-relative URL you want it deployed to, as shown in Figure 2-8.
- 6. Click Apply and then OK to save the run/debug configuration and dismiss the dialog.

You can download the Sample-Debug-IntelliJ project from the wrox.com code download site to view a sample web application already configured to run on your local Tomcat 8.0 application server. (However, you still need to set up your Tomcat 8.0 installation in IntelliJ's IDE settings.)

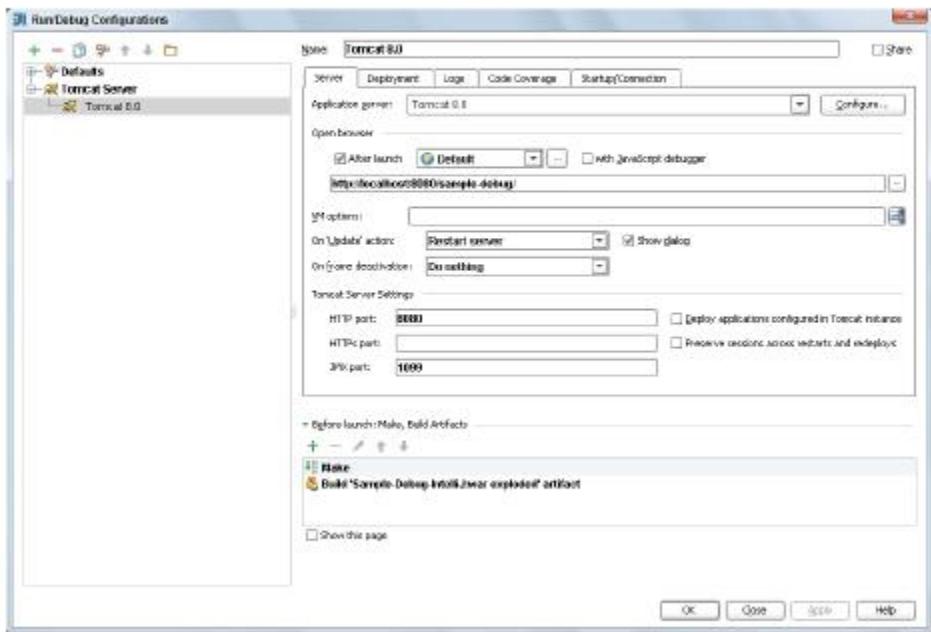


FIGURE 2-7

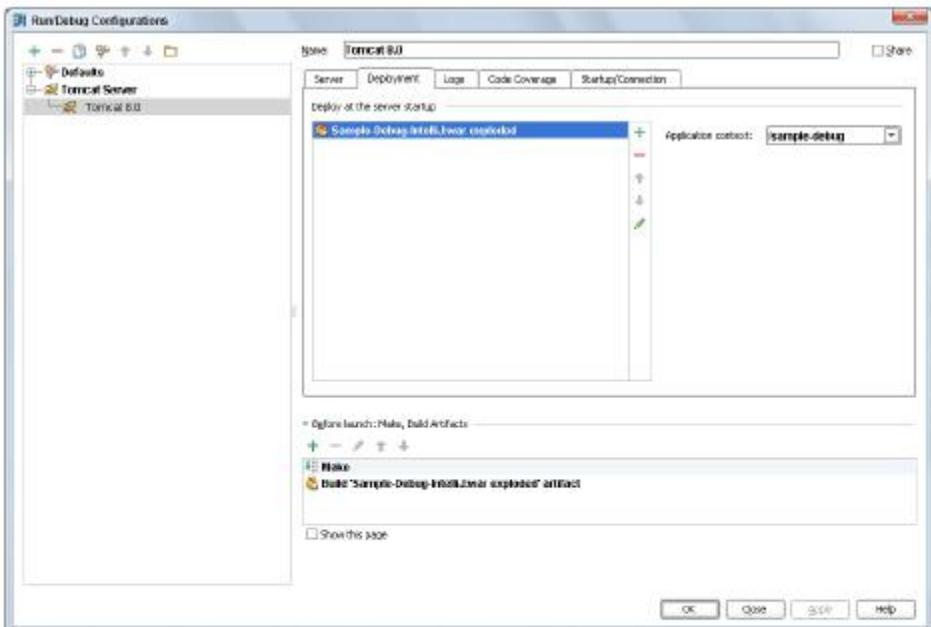


FIGURE 2-8

Starting an Application and Hitting Breakpoints

Now that you have set up Tomcat in IntelliJ and configured an IntelliJ project to run in Tomcat, you're ready to start the application and debug it within your IDE.

1. Download the Sample-Debug-IntelliJ project from the wrox.com code download site, and open it with IntelliJ IDEA.
2. Make sure that its run/debug configuration is properly configured to use your local Tomcat 8.0 application server. You should perform this check for each sample project you download for this book before attempting to start it.
3. When opened, you should see a screen like Figure 2-9, with two breakpoints in place for `index.jsp`.
4. Click the Debug icon on the toolbar (highlighted by the mouse pointer in Figure 2-9) or press Shift + F9 to compile and start your application in debug mode. IntelliJ should launch your default browser, and you should immediately hit the breakpoints in `index.jsp`.

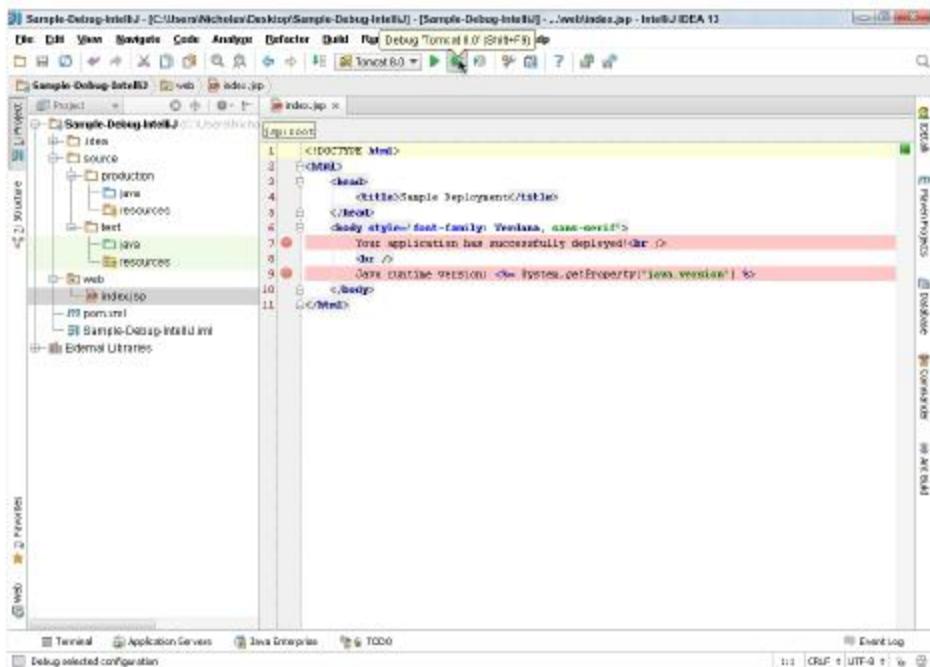


FIGURE 2-9

You should again see the webpage from Figure 2-2 to indicate that your application successfully deployed.

NOTE *IntelliJ may actually access <http://localhost:8080/sample-debug/> before launching your browser. It does this to ensure that your application has been properly deployed. If this is the case, you will hit the breakpoints twice — once when IntelliJ accesses the application and once when your browser opens and accesses the application.*

Using Eclipse

Using Tomcat in Eclipse has some similarities to using Tomcat in IntelliJ IDEA, but it also has many differences, and the screens look very different. The same basic process still applies — you need to set up Tomcat in Eclipse's global settings, configure it for a project, and start and debug the project. In this last part of this section, you learn how to use Tomcat from Eclipse in case you have chosen that as your IDE for this book.

WARNING *As discussed in the introduction, as of the date this book was published, Eclipse does not yet support Java SE 8, Java EE 7, or Tomcat 8.0. You must wait until Eclipse 4.4 Luna is released in June 2014 to realize support for these technologies. As such, the Eclipse instructions and figures in this section may not be completely accurate, and you should respond as needed to changes made to the release version of Eclipse Luna.*

Setting Up Tomcat 8.0 in Eclipse

To begin, you must configure Tomcat 8.0 as a runtime environment in Eclipse's global preferences. To do so, follow these steps:

1. Open your Eclipse IDE for Java EE Developers and go to Windows \Rightarrow Preferences.
2. In the Preferences dialog that appears, expand Server, and then click Runtime Environments. A Server Runtime Environments panel appears where you can manage the application servers and web containers available to all your Eclipse projects.
3. Click the Add button to open the New Server Runtime Environment dialog.
4. Expand the Apache folder and select Apache Tomcat v8.0, making sure you select the "Create a new local server" check box. Then click the Next button.
5. On the next screen, click the Browse button and browse to your Tomcat 8.0 home directory (for example, C:\Program Files\Apache Software Foundation\Tomcat 8.0). Then click OK.
6. In the JRE drop-down, select your local Java SE 8 JRE installation. Name the server whatever you want. The Eclipse sample projects you download throughout this book assume that the server is named Apache Tomcat v8.0, which is the Eclipse default. At this point you should see a screen like Figure 2-10.

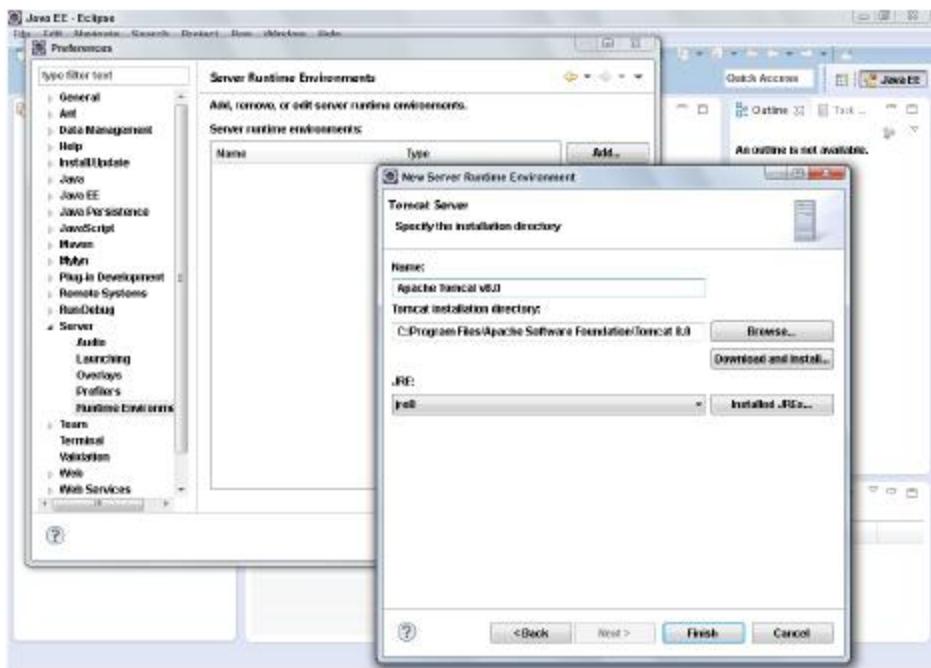


FIGURE 2-10

7. Click the Finish button to complete adding your local Tomcat server to Eclipse, and then click OK to close the preferences dialog.

You are now ready to use Tomcat 8.0 in your Eclipse projects.

One other thing to note is that, by default, Eclipse uses a built-in browser to open your web applications. You should disable this feature and use a mainstream browser, instead, such as Google Chrome, Mozilla Firefox, or Microsoft Internet Explorer. To change this setting, go to the Window \Rightarrow Web Browser menu, and select something other than "0 Internal Web Browser." The option "1 Default System Web Browser" should be sufficient in most cases, but it's easy to change this setting frequently to meet your needs at any given time.

Using the Tomcat Server in a Project

When creating a new project in Eclipse, you have to select the configured runtime server you are going to use for that project on the first dialog, as shown in Figure 2-11. However, this configures only the libraries for your application. It does not select the Tomcat 8.0 server you created. For that, follow these steps:

1. After you create or open the project, go to Project \Rightarrow Properties and click the Server menu item on the left side of the project Properties dialog that appears.
2. By default, the selected server is "<None>", so you should change it to "Tomcat v8.0 Server at localhost" instead, as shown Figure 2-12.

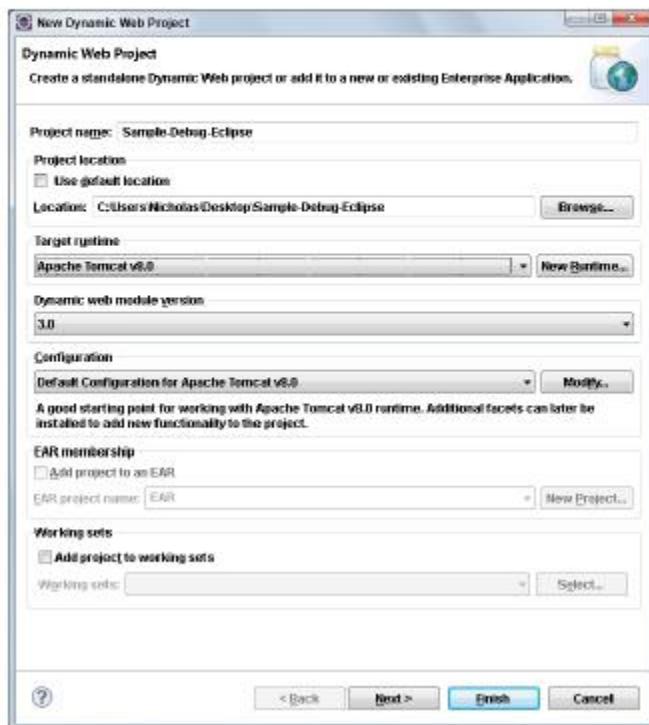


FIGURE 2-11

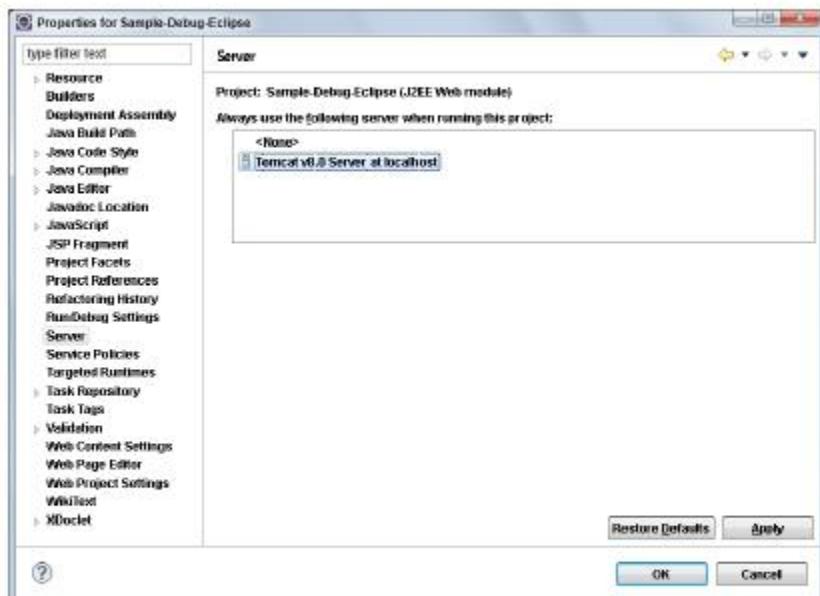


FIGURE 2-12

3. Click Apply to save the changes.
4. Change the application context URL that the application deploys to in Tomcat (assuming you didn't configure it when you created the project). In the project Properties dialog, you can click the Web Project Settings menu item and update the "Context root" field to change this setting.
5. After clicking Apply to save the changes, click OK to dismiss the dialog.

You can download the Sample-Debug-Eclipse project from the wrox.com code download site to view a sample web application already configured to run on your local Tomcat 8.0 application server. (However, you still need to set up your Tomcat 8.0 installation in Eclipse's IDE preferences.)

Starting an Application and Hitting Breakpoints

You're now ready to start your application and debug it from Eclipse.

1. Download the Sample-Debug-Eclipse project from the wrox.com code download site, and open it with Eclipse IDE for Java EE Developers.
2. Make sure that its server settings are properly configured to use your local Tomcat 8.0 application server. You should perform this check for each sample project you download for this book before attempting to start it.
3. When opened you should see a screen like Figure 2-13, with one breakpoint already in place for `index.jsp`.

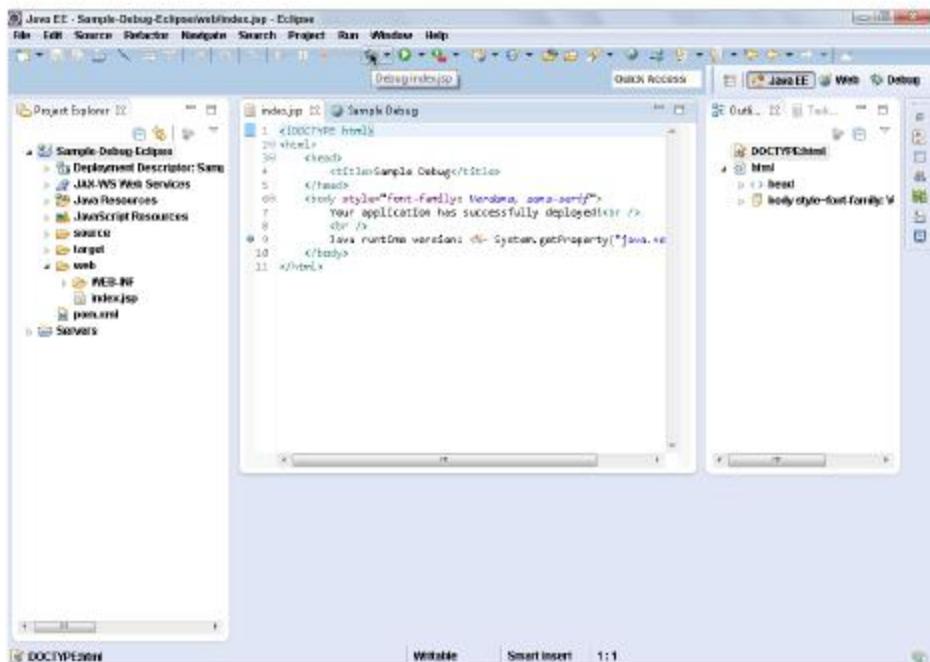


FIGURE 2-13

4. Click the Debug icon in the toolbar (highlighted by the mouse pointer in Figure 2-13) to compile and start your application in debug mode. Eclipse should launch the configured browser, and you should immediately hit the breakpoint in `index.jsp`. You can again see the webpage from Figure 2-2 to indicate that your application successfully deployed.
5.  To continue from the breakpoint, click the continue icon (shown here in the margin) on the Eclipse toolbar.

WARNING *When you run Tomcat from Eclipse, Eclipse overrides any custom `conf\setenv.bat` or `conf\setenv.sh` file that you create to configure advanced JSP compilation. If you do not want to use the Eclipse JDT compiler to compile your JSPs, you need to add the `CLASSPATH` configuration in this file to some other Tomcat configuration file. Consult the Tomcat documentation to determine the appropriate file to place this in.*

NOTE *You likely noticed that the JSP in Eclipse only has one breakpoint, whereas the JSP in IntelliJ IDEA has two breakpoints. The Eclipse JSP debugger is much more limited than the IDEA JSP debugger, so placing a breakpoint on Line 7 in this JSP is not possible in Eclipse.*

SUMMARY

In this chapter, you learned about Java EE application servers and web containers and explored several popular implementations of both. You installed Tomcat 8.0 on your local machine, configured JSP compilation, started it from the command line, and experimented with deploying and undeploying applications in Tomcat. Finally, you learned how to configure and run Tomcat 8.0 and debug your applications using both IntelliJ IDEA and Eclipse IDE for Java EE Developers.

In the next chapter you create Servlets and learn how Java EE web applications work.

3

Writing Your First Servlet

IN THIS CHAPTER

- Creating a Servlet class
- Configuring a Servlet for deployment
- Understanding `doGet()`, `doPost()` and other methods
- Using parameters and accepting form submissions
- Configuring your application using init parameters
- Uploading files from a form
- Making your application safe for multithreading

WROX.COM CODE DOWNLOADS FOR THIS CHAPTER

You can find the [wrox.com](http://www.wrox.com) code downloads for this chapter at www.wrox.com/go/projavaforwebapps on the Download Code tab. The code for this chapter is divided into the following major examples:

- Hello-World Project
- Hello-User Project
- Customer-Support-v1 Project

NEW MAVEN DEPENDENCIES FOR THIS CHAPTER

In this chapter, you'll need your first Maven dependency, shown in the following code. You'll use this dependency for every chapter throughout the rest of the book.

```
<dependency>
  <groupId>javax.servlet</groupId>
  <artifactId>javax.servlet-api</artifactId>
  <version>3.1.0</version>
  <scope>provided</scope>
</dependency>
```

In the last chapter, you became familiar with application servers and web containers and learned how to run, deploy to, and debug Apache Tomcat 8.0 from your Java IDE. In this chapter, you begin building web applications by first exploring the world of Servlets. Throughout this chapter and the rest of the book, you'll continually change and improve these applications, deploying them to Tomcat for testing and debugging.

CREATING A SERVLET CLASS

In the Java Platform, Enterprise Edition, a *Servlet* is what receives and responds to requests from the end user. The Java EE API specification defines a Servlet as follows:

A Servlet is a small Java program that runs within a Web server. Servlets receive and respond to requests from Web clients, usually across HTTP, the HyperText Transfer Protocol.

<http://docs.oracle.com/javaee/7/api/javax/servlet/Servlet.html>

Servlets are the core classes in any web application, the only classes that either perform the work of responding to requests or delegate that work to some other part of the application. Unless some filter prematurely terminates a request (discussed in Chapter 9), every request to your application goes through some Servlet. The web container in which you run your application will have one or more built-in Servlets. These Servlets handle serving JavaServer Pages, displaying directory listings (if you have them enabled) and accessing static resources, such as HTML pages and graphics. You won't need to worry about these Servlets yet (in some cases, ever). In this chapter, you learn how to write and configure the custom Servlets that make up your application.

Every Servlet implements the `javax.servlet.Servlet` interface, but usually not directly. `Servlet` is a simple interface, containing methods for initializing and destroying the Servlet and servicing requests. However, the `service` method will be called for any request of any type, even if it is not an HTTP request (theoretically, assuming your web container supports such a request). As an example, in the future it's possible that new Servlets could be added to Java EE to support File Transfer Protocol (FTP). For that reason, there are various Servlet classes that you can extend instead. As of Java EE 7, the only Servlet protocol currently supported is HTTP.

What to Extend

In almost all cases, Servlets inherit from `javax.servlet.GenericServlet`. `GenericServlet` is still a protocol-independent Servlet with the lone, abstract `service` method, but it contains several helper methods for logging and getting information about the application and Servlet configuration (more on that later in the section "Configuring a Servlet for Deployment").

For responding to HTTP-specific requests, `javax.servlet.http.HttpServlet` extends `GenericServlet` and implements the `service` method to accept only HTTP requests. Then, it provides empty implementations for methods corresponding to each HTTP method type, as illustrated in Table 3-1.

TABLE 3-1: Empty Implementations for HTTP Method Types

METHOD	SERVLET METHOD	PURPOSE
GET	doGet()	Retrieves the resource at the specified URL
HEAD	doHead()	Identical to GET, except only the headers are returned
POST	doPost()	Typically used for web form submission
PUT	doPut()	Stores the supplied entity at the URL
DELETE	doDelete()	Deletes the resource identified by the URL
OPTIONS	doOptions()	Returns which HTTP methods are allowed
TRACE	doTrace()	Used for diagnostic purposes

NOTE *Most web programmers are familiar with the GET and POST methods and use them the majority of the time. If you are not familiar with the various HTTP methods or would like to learn more, now is the time to click <http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html> to see the RFC-2616 specification section on method definitions.*

With no exceptions in this book, your Servlets will always extend `HttpServlet`. It provides all the tools you need to selectively accept and respond to different types of HTTP requests, and its methods accept `javax.servlet.http.HttpServletRequest` and `javax.servlet.http.HttpServletResponse` arguments instead of `javax.servlet.ServletRequest` and `javax.servlet.ServletResponse` so that you have easy access to HTTP-specific attributes of the requests your Servlet services. You should begin by creating a new, empty Servlet that extends `HttpServlet`:

```
package com.wrox;

import javax.servlet.http.HttpServlet;

public class HelloServlet extends HttpServlet
{
}
```

NOTE *In order for this code to compile, you need to have the Java EE Servlet API library on your compile classpath. This is where the Maven artifact listed on the first page of this chapter comes into play. In each chapter you will need the listed Maven artifacts in order to compile any examples in that chapter.*

In this form, your Servlet is already prepared to accept any HTTP request and respond to it with a 405 Method Not Allowed error. This is how you control which HTTP methods your Servlet responds to: Any HTTP Servlet methods you do not override will be responded to with an HTTP status 405. A Servlet that does not handle any requests is, of course, not very useful, so override the `doGet` method to add support for the HTTP method `GET`:

```
package com.wrox;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;

public class HelloServlet extends HttpServlet
{
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException
    {
        response.getWriter().println("Hello, World!");
    }
}
```

Now your Servlet is equipped to respond to `GET` requests and return the plain text response "Hello, World!" in the response body. The code in this example is fairly straightforward. Calling `getWriter` on the `response` parameter returns a `java.io.PrintWriter`, a common Java class used for writing text to an output stream. Next, the code calls the `println` method on the `PrintWriter` to write the text "Hello, World!" to the output stream. Notice that you don't have to worry about any of the details of the raw HTTP request or response. The web container takes care of interpreting the request and reading the headers and parameters from the socket. After your method returns, it takes care of formatting the response headers and body and writing them back to the socket.

NOTE *Notice that you did not call the `close` method on the `PrintWriter` that you obtained from the `response`. Generally speaking, in Java you only need to close resources that you create. The web container created this resource, so it is responsible for closing it. Even if you had assigned the instance to a local variable and called several methods on it, this would still be the case.*

You obviously could do a lot more in this `doGet` method, such as using request parameters, and you haven't taken a look at the other methods yet. Rest assured, you'll get to both soon.

Using the Initializer and Destroyer

While you get your first Servlet up and running, you should probably know about the `init` and `destroy` methods. When a web container first starts a Servlet, it calls that Servlet's `init` method. This is sometimes, though not always, when the application is deployed. (You learn how to control this in the next section.) Later when the web container shuts down the Servlet, it calls the Servlet's `destroy` method. These methods are not the same as the Java constructor and finalizer, and they are not called at the same time as the constructor and finalizer. Normally, these methods do nothing, but you can override them to perform some action:

```
@Override
public void init() throws ServletException
{
    System.out.println("Servlet " + this.getServletName() + " has started.");
}

@Override
public void destroy()
{
    System.out.println("Servlet " + this.getServletName() + " has stopped.");
}
```

NOTE *You should know that another `init` method accepts a single argument of type `javax.servlet.ServletConfig`. This method is specified in the `Servlet` interface, but `GenericServlet` takes care of implementing this method for you and then calls the no-argument overload of `init` overridden in the previous code example. In this way, you do not have to call `super.init(servletConig)` from your own `init` method implementation.*

Although you can override the original method, you shouldn't do so because if you forgot to call the super method, the Servlet might not initialize correctly. If you need to access the `ServletConfig`, it's much easier to just call the `getServletConfig` method. You learn more about the `ServletConfig` class throughout Parts I and II of this book.

You can do many things with these two methods. More important, `init` is called after the Servlet is constructed but before it can respond to the first request. Unlike when the constructor is called, when `init` is called all the properties have been set on the Servlet, giving you access to the `ServletConfig` and `javax.servlet.ServletContext` objects. (You learn what to do with these in the "Configuring your Application Using Init Parameters" section.) So, you may use this method to read a properties file or connect to a database using JDBC, for example. The `init` method is called when the Servlet starts. If the Servlet is configured to start automatically when the web application is deployed and started, that is when it is called. Otherwise, it is not called until the first request for that Servlet is received.

Likewise, `destroy` is called immediately after the Servlet can no longer accept any requests. This typically happens either when the web application is stopped or undeployed or when the web container shuts down. Because it is called immediately upon undeployment or shutdown, you do not have to wait for garbage collection to trigger the finalizer before cleaning up resources such as

temporary files or disconnecting from databases no longer in use. This is particularly important because if your application is undeployed but the server continues running, it may be several minutes or even hours before garbage collection runs. If you clean up your resources in the finalizer instead of the `destroy` method, this could result in your application undeploying partially or failing to undeploy. Thus, you should always use the `destroy` method to clean up resources held by your Servlet between requests.

The previous code example uses the `init` and `destroy` methods to log when the Servlet starts and stops, respectively. When you run your application in the next section, these log messages appear in the output window of your IDE's debugger. Later in this chapter you put these methods to better use.

CONFIGURING A SERVLET FOR DEPLOYMENT

Now that you have created your Servlet, it's time to put it in action. Although you have a working class that can respond to HTTP `GET` requests with a clever greeting, you have not written instructions for the web container to deploy the Servlet with the application. Chapter 1 introduced you to the deployment descriptor (`web.xml`) and the structure of a web application, and in Chapter 2 you learned how to deploy and debug an application using your IDE. In this section, you create the `web.xml` file in your `WEB-INF` directory and configure your Servlet for deployment. You then deploy the application using your IDE and see that greeting in your browser. Finally, you put some breakpoints in your code and examine when certain methods are called.

Adding the Servlet to the Descriptor

As you've learned, the deployment descriptor instructs the web container how the application should be deployed. Specifically, it defines all the listeners, Servlets, and filters that should deploy with the application and the settings the application should use to do this. First, take a look at a (mostly) empty `web.xml` file:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
                             http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd"
          version="3.1">

    <display-name>Hello World Application</display-name>

</web-app>
```

WARNING *If you have worked with deployment descriptors in previous Java EE versions, this might look slightly unfamiliar to you. This is because the XML schema URIs for `web.xml` and other configuration files have changed since Java EE 6. You must use the new URIs for your application to be Java EE 7 compliant.*

In the previous example, the code in bold indicates to the application server what the name of the application is. On the Tomcat manager screen that lists all the installed applications, the name between the `<display-name>` tags appears beside your application. The `version` attribute in the opening `<web-app>` tag indicates which Servlet API version the application is written for—in this case, version 3.1.

Now you need to tell the web container to create an instance of the Servlet you wrote earlier, so you must add a Servlet tag to the descriptor file between the beginning and ending `<web-app>` tags:

```
<servlet>
  <servlet-name>helloServlet</servlet-name>
  <servlet-class>com.wrox.HelloServlet</servlet-class>
</servlet>
```

Earlier in the chapter, you learned about the Servlet `init` method and when it would normally be called. In this example, the `init` method is called when the first request arrives for the Servlet after the web application starts. Normally, this is sufficient for most uses. However, if the `init` method does many things, Servlet startup might become a time-intensive process, and this could make the first request to that Servlet take several seconds or even several minutes! Obviously, this is not desirable. A simple tweak to the servlet configuration can make the servlet start up immediately when the web application starts:

```
<servlet>
  <servlet-name>helloServlet</servlet-name>
  <servlet-class>com.wrox.HelloServlet</servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>
```

The emboldened code instructs the web container to start the Servlet as soon as the web application starts. If multiple Servlet configurations contain this tag, they start up in the order of the values within the tags, with the previously used value “1” coming first and higher numbers later. If two or more Servlets have the same value in the `<load-on-startup>` tag, those conflicting Servlets start in the order they appear in the descriptor file, still after other Servlets with lower numbers and before other Servlets with higher numbers.

Mapping the Servlet to a URL

You have instructed the application server to start the Servlet but have not yet told it what URL requests the Servlet should respond to. This is a simple matter:

```
<servlet-mapping>
  <servlet-name>helloServlet</servlet-name>
  <url-pattern>/greeting</url-pattern>
</servlet-mapping>
```

With this configuration, all requests to the application-relative URL `/greeting` are handled by the `helloServlet`. (Notice that the `<servlet-name>` tags within the `<servlet>` and `<servlet-mapping>` tags match each other. This is how the web container associates the two.) If the application is deployed at `http://www.example.net`, the Servlet responds to requests directed to the URL `http://www.example.net/greeting`. Of course, you are not limited to this one mapping. You could map several URLs to the same Servlet:

```
<servlet-mapping>
    <servlet-name>helloServlet</servlet-name>
    <url-pattern>/greeting</url-pattern>
    <url-pattern>/salutation</url-pattern>
    <url-pattern>/wazzup</url-pattern>
</servlet-mapping>
```

In this case, all three URLs act as aliases for the same logical endpoint: the `helloServlet`. Why, you might ask, do you need to give a Servlet instance a name and then map a request to the name of that instance? Why can't you just map the URL directly to the Servlet class? Well, what if you have two different store Servlets in an online shopping application, for example? Those stores might have identical logic but connect to different databases. This can be achieved simply:

```
<servlet>
    <servlet-name>oddsStore</servlet-name>
    <servlet-class>com.wrox.StoreServlet</servlet-class>
</servlet>
<servlet>
    <servlet-name>endsStore</servlet-name>
    <servlet-class>com.wrox.StoreServlet</servlet-class>
</servlet>

<servlet-mapping>
    <servlet-name>oddsStore</servlet-name>
    <url-pattern>/odds</url-pattern>
</servlet-mapping>
<servlet-mapping>
    <servlet-name>endsStore</servlet-name>
    <url-pattern>/ends</url-pattern>
</servlet-mapping>
```

Now you have two instances of the same Servlet class, but they have different names and are mapped to different URLs. Two examples ago, you had three URLs all pointing to the *same Servlet instance*. However, in this example you have *two different Servlet instances*. You might wonder how the two different instances know which stores they are. A quick call to `this.getServletName()` from anywhere in the servlet code returns either "oddsStore" or "endsStore" depending on which instance it is. Recall that you used this method earlier when you were logging calls to the initializer and the destroyer.

Rewinding a bit, you now have the simple, completed `web.xml` descriptor file:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
        http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd"
    version="3.1">

    <display-name>Hello World Application</display-name>

    <servlet>
        <servlet-name>helloServlet</servlet-name>
        <servlet-class>com.wrox.HelloServlet</servlet-class>
    </servlet>
```

```
<servlet-mapping>
    <servlet-name>helloServlet</servlet-name>
    <url-pattern>/greeting</url-pattern>
</servlet-mapping>

</web-app>
```

Running and Debugging Your Servlet

After it's saved, compile your application and check to make sure you have an IDE run configuration set up to run your project in your local Tomcat 8.0 instance. (If you don't remember how to do this, refer back to Chapter 2). The application should deploy to `/hello-world`. You can also just download the Hello-World IDE project from the wrox.com code download site — it is already configured to deploy properly. When this is done, follow these steps:

1. Click the debug icon in your IDE to start the web container in debug mode. Your IDE deploys your application to the web container after it starts.
2. Open your favorite web browser and navigate to `http://localhost:8080/hello-world/greeting`. You should now see the screen in Figure 3-1.

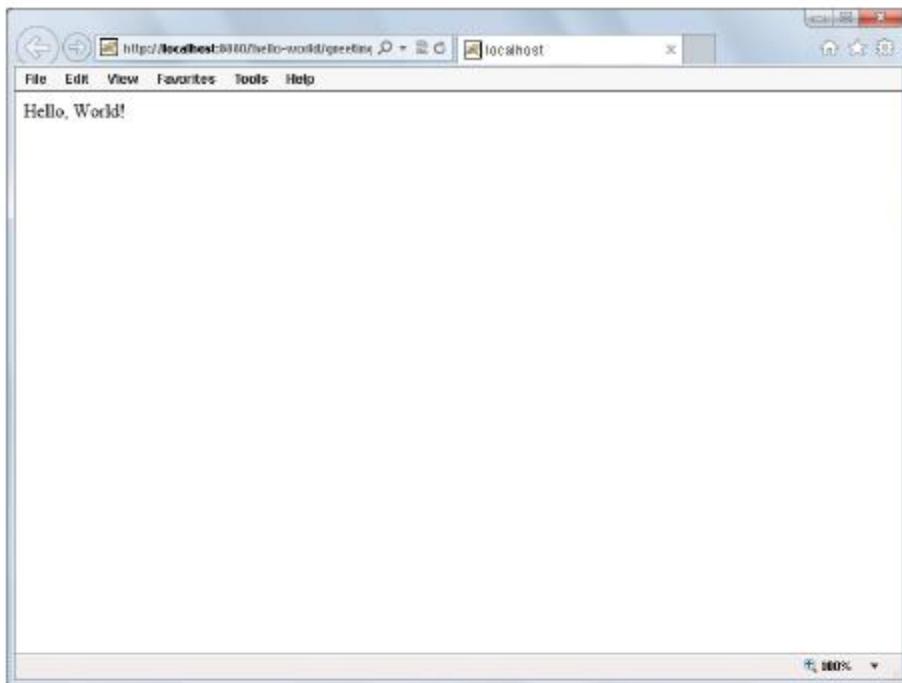


FIGURE 3-1

3. A good way to understand what happened is to place some breakpoints in the `HelloServlet` and run this experiment again. You should stop your debugger (which shuts down Tomcat) so that you can hit a breakpoint in the initializer as well. Place breakpoints

in the single lines of code in the `doGet`, `init`, and `destroy` methods of your Servlet; then restart your debugger. After Tomcat starts and your application deploys, you will notice that you did not hit any breakpoints yet (because `<load-on-startup>` is not present in the deployment descriptor).

4. Refresh the greeting page in your browser and you should hit the breakpoint in the `init` method of your IDE. This means that Tomcat has activated the just-in-time initialization of your Servlet: It was not initialized until the first request came in.
5. Just like it would if the `init` method were taking a long time to complete, the request from your browser remains on hold until you continue your debugger, so do that now. You should immediately hit the breakpoint in the `doGet` method. Now the Servlet services the request, but your browser still waits on a response.
6. Continue your debugger a second time, and now the response is sent to your browser.

At this point, you can press the Refresh button on your browser as many times as you like, and you will hit the breakpoint only in the `doGet` method. The `init` method is not called again until some action destroys the Servlet (for example, Tomcat shutting down) and then it starts again. Up until this point, you have not yet hit the breakpoint in the `destroy` method. You want to do that now, but unfortunately, if you stop Tomcat from your IDE, it detaches the debugger before the breakpoint is hit, so you need to stop Tomcat from the command line. To do this, follow these steps:

1. Open up a command prompt and change your current directory to the Tomcat home directory (`C:\Program Files\Apache Software Foundation\Tomcat 8.0` on a Windows machine, remember).
2. Type the command `bin\shutdown.bat` (or `bin\shutdown.sh` if you are not running Windows) and press Enter.
3. In your IDE window, you should immediately hit the breakpoint in the `destroy` method. Tomcat does not completely shut down until you continue your debugger.

As mentioned earlier, you can change the configuration of your Servlet so that it is initialized when the application starts. Try that now.

1. Update your Servlet declaration in the deployment descriptor to add the code in bold in the following example:

```
<servlet>
  <servlet-name>helloServlet</servlet-name>
  <servlet-class>com.wrox.HelloServlet</servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>
```

2. With the breakpoints still in place in your Servlet, start your debugger again. You should immediately hit the breakpoint in the `init` method before you make the first request to the Servlet.
3. Continue the debugger and then refresh your browser. Now you should hit the breakpoint only in the `doGet` method; the Servlet was initialized on application startup and does not need to be initialized again.

Now that you have created your first Servlet and are familiar with the life cycle of a Servlet, you are encouraged to experiment with different method calls on the Servlet and on the `request` and `response` parameters in the `doGet` method. In the next section, you explore `doGet`, `doPost`, and other methods further to better understand `HttpServletRequest` and `HttpServletResponse`.

NOTE *You should consult (and bookmark) the API documentation for Java EE 7 located at <http://docs.oracle.com/javaee/7/api/> for information on the available methods and their purposes.*

UNDERSTANDING DOGET(), DOPOST(), AND OTHER METHODS

In the previous section you learned about the `doGet` method and other methods that map to the various HTTP request methods. But what exactly can you do in these methods? More important, what *should* you do in these methods? The short answers to these questions are “just about anything” and “not very much,” respectively. This section explores some of the things you can do and how to do them.

What Should Happen during the service Method Execution?

The `Servlet` class’s `service` method, as you learned earlier, services all incoming requests. Ultimately, it must parse and handle the data on the incoming request based on the protocol in use and then return a protocol-acceptable response to the client. If the `service` method returns without sending response data back to the socket, the client will likely observe a network error, such as “connection reset.” In the HTTP protocol specifically, the `service` method should understand the headers and parameters that the client sends and then return a proper HTTP response that at least includes the minimum HTTP headers (even if the response body itself is empty). In reality, the implementation of this is complex (and involves many steps) and may differ from web container to web container.

The beauty of extending `HttpServlet` is that you don’t have to worry about any of these details. Although the reality is that the `service` method must do many things before responding to the user, the developer using `HttpServlet` must do little. Actually, in the Hello-World project you used in the last two sections, if you remove the single line of code from the `doGet` method and run the application, everything still works fine! A properly structured HTTP response with zero-length content returns to the client. The only requirement is that you override the `doGet` method (or `doPost` or `doPut` or whatever you want to support); you don’t need to put anything in it. But how useful is this, really?

The answer to that is “not at all.” Just because you *can* return an empty response doesn’t mean you *should*. This is where `HttpServletRequest` and `HttpServletResponse` come in. These parameters in the various methods defined by `HttpServlet` enable you to read parameters passed from the client, accept files uploaded from a posted form, read the raw data contained in the request body (for doing things such as handling PUT requests or accepting JSON request bodies), read request

headers and manipulate response headers, and write response content back to the client. These are some of the many things that you *can* do when servicing a request, and in reality you usually *should* do one or more of these things.

Using HttpServletRequest

The `HttpServletRequest` interface is an extension of `ServletRequest` that provides additional HTTP protocol-specific information about a received request. It specifies dozens of methods that you can use to obtain details about an HTTP request. It also permits you to set request attributes (different from request parameters).

NOTE *You'll learn about request attributes and the inspection of authentication details in the next chapter. This book does not cover the details of every method (for that, you can consult the API documentation) but covers the most important features.*

Getting Request Parameters

Perhaps the most important capability of `HttpServletRequest`, and one you explore through examples in the next section, is to retrieve request parameters passed by the client. Request parameters come in two different forms: via *query parameters* (also called *URI parameters*), or in an `application/x-www-form-urlencoded` or `multipart/form-data` encoded request body (typically called *post variables* or *form variables*). Query parameters are supported with all request methods and are contained in the first line of data in an HTTP request, as in the following example:

```
GET /index.jsp?productId=9781118656464&category=Books HTTP/1.1
```

NOTE *Technically speaking, the RFC specification for the HTTP protocol does not disallow query parameters in any of the HTTP methods. However, many web servers ignore query parameters passed to `DELETE`, `TRACE`, and `OPTIONS`, and the usefulness of query parameters in such requests is questionable. So, it is best to not rely on query parameters for these types of requests. This book does not cover all the rules and intricacies of the HTTP protocol. That exercise is left up to you.*

In this example, there are two query parameters contained in the request: `productId`, which has this book's ISBN as its value, and `category`, which has the value Books. These same parameters could also be passed in the request body as post variables. Post variables can, as the name implies, be included only in `POST` requests. Consider the following example:

```
POST /index.jsp?returnTo=productPage HTTP/1.1
Host: www.example.com
Content-Length: 48
Content-Type: application/x-www-form-urlencoded

addToCart&productId=9781118656464&category=Books
```

This `POST` request has post variables (instructing the website to add this book to the cart) *and* query parameters (instructing the website to return to the product page when the task is complete). Although there is a difference in the delivery of these two types of parameters, they are essentially the same, and they convey essentially the same information. The Servlet API does not differentiate between the two types of parameters. A call to any of the parameter-related methods on a request object returns parameters whether they were delivered as query parameters or post variables.

The `getParameter` method returns a single value for a parameter. If the parameter has multiple values, `getParameter` returns the first value, whereas `getParameterValues` returns an array of values for a parameter. If the parameter has only one value, this method returns an array with one element in it. The `getParameterMap` method returns a `java.util.Map<String, String[]>` containing all the parameter names mapped to their values, whereas the `getParameterNames` method returns an enumeration of the names of all the available parameters; both are useful for iterating over all the request parameters.

WARNING *The first time you call `getParameter`, `getParameterMap`, `getParameterNames`, or `getParameterValues` on a request object, the web container determines whether the request contains post variables, and if it does it reads and parses those post variables by obtaining the request's `InputStream`. The `InputStream` of a request can be read only once. If you call `getInputStream` or `getReader` on a request containing post variables and then later attempt to retrieve parameters in that request, the attempt to retrieve the parameters results in an `IllegalStateException`. Likewise, if you retrieve parameters on a request containing post variables and then later call `getInputStream` or `getReader`, the call to `getInputStream` or `getReader` fails with an `IllegalStateException`. Simply put, any time you anticipate that a request may contain post variables, it's best to use only the parameter methods and leave `getInputStream` and `getReader` alone.*

Determining Information about the Request Content

Several methods are available to help determine the type, length, and encoding of the content of the HTTP request. The `getContentType` method returns the *MIME content type* of the request, such as `application/x-www-form-urlencoded`, `application/json`, `text/plain`, or `application/zip`, to name a few. A MIME content type describes that the data it marks contains some type. For example, ZIP archives files have a MIME content type of `application/zip` to indicate that they contain ZIP archive data.

The `getContentLength` and `getContentLengthLong` methods both return the number of bytes in the request body (the *content length*), with the latter method being useful for requests whose content might exceed 2 gigabytes (unusual, but not impossible). The `getCharacterEncoding` method returns the *character encoding* (such as `UTF-8` or `ISO-8859-1`) of the request contents whenever the request contains character-type content (`text/plain`, `application/json`, and `application/x-www-form-urlencoded` are some examples of character-type MIME content types.) Although these methods can come in handy in many situations, none of them are necessary if you get post variables from the request body using the parameter methods.

NOTE *The Servlet 3.1 specification in Java EE 7 is the first version that supports the `getContentTypeLong` method. Before this version, you had to call `getHeader("Content-Length")` and convert the returned `String` to a `long` for requests that could be larger than 2,147,483,647 bytes.*

Reading the Contents of a Request

The methods `getInputStream`, which returns a `javax.servlet.ServletInputStream`, and `getReader`, which returns a `java.io.BufferedReader`, can both be used to read the contents of the request. Which one is best completely depends on the context in which the request contents are being read. If the contents are expected to be character-encoded data, such as UTF-8 or ISO-8859-1 text, using the `BufferedReader` is typically the easiest route to take because it lets you easily read `char` data. If, however, the request data is binary in nature, you must use the `ServletInputStream` so that you can access the request content in `byte` format. You should never use them both on the same request. After a call to either method, a call to the other will fail with an `IllegalStateException`. Remember the preceding warning, and do not use these methods on a request with post variables.

Getting Request Characteristics Such as URL, URI, and Headers

There are many request characteristics that you may need to know about, such as the URL or URI the request was made with. These are easy to obtain from the request object:

- `getRequestURL`: Returns the entire URL that the client used to make the request, including protocol (http or https), server name, port number, and server path but not including the query string. So, in a request to `http://www.example.org/application/index.jsp?category=Books`, `getRequestURL` returns `http://www.example.org/application/index.jsp`.
- `getRequestURI`: This is slightly different from `getRequestURL` in that it returns only the server path part of the URL; using the previous example, that would be `/application/index.jsp`.
- `getServletPath`: Similar to `getRequestURI`, this returns even less of the URL. If the request is `/hello-world/greeting?foo=world`, the application is deployed as `/hello-world` on Tomcat, and the servlet-mappings are `/greeting`, `/salutation`, and `/wazzup`, `getServletPath` returns only the part of the URL used to match the servlet mapping: `/greeting`.
- `getHeader`: Returns the value of a header with the given name. The case of the header does not have to match the case of the string passed into the method, so `getHeader("Content-Type")` can match the `Content-Type` header. If there are multiple headers with the same name, this returns only the first value. In such cases, you would want to use the `getHeaders` method to return an enumeration of all the values.
- `getHeaderNames`: Returns an enumeration of the names of all the headers in the request — a great way to iterate over the available headers.

- `getIntHeader`: If you have a particular header that you know is always a number, you can call this to return the value already converted to a number. It throws a `NumberFormatException` if the header cannot be converted to an integer.
- `getDateHeader`: You can call this to return the (millisecond) Unix timestamp-equivalent of a header value that represents a valid timestamp. It throws an `IllegalArgumentException` if the header value is not recognized as a date.

Sessions and Cookies

The `getSession` and `getCookies` methods are mentioned only long enough to tell you that this chapter doesn't cover them, but they are both important citizens in the `HttpServletRequest` realm. You can learn more about these in Chapter 5.

Using `HttpServletResponse`

As the `HttpServletRequest` interface extends `ServletRequest` and provides access to the HTTP protocol-specific properties of a request, the `HttpServletResponse` interface extends `ServletResponse` and provides access to the HTTP protocol-specific properties of a response. You use the response object to do things such as set response headers, write to the response body, redirect the request, set the HTTP status code, and send cookies back to the client. Again, the most common features of this object are covered here.

Writing to the Response Body

The most common thing you'll do with a response object, and something you have already done with a response object, is write content to the response body. This might be HTML to display in a browser, an image that the browser is retrieving, or the contents of a file that the client is downloading. It could be plain text or binary data. It might be just a few bytes long or it could be gigabytes long.

The `getOutputStream` method, which returns a `javax.servlet.ServletOutputStream`, and the `getWriter` method, which returns a `java.io.PrintWriter`, both enable you to write data to the response. Like their counterparts in `HttpServletRequest`, you would probably want to use the `PrintWriter` for returning HTML or some other character-encoded text to the client because this makes it easy to write encoded `strings` and `chars` to the response. However, for sending binary data back, you must use the `ServletOutputStream` to send the response `bytes`. Also, you should never use both `getOutputStream` and `getWriter` in the same response. After a call to one, a call to the other will fail with an `IllegalStateException`.

While you're writing to the response body, it might be necessary to set the content type or encoding. You can do this with `setContentType` and `setCharacterEncoding`. You may call these methods as many times as you like; the last call to the method is the one that matters. However, if you plan to call `setContentType` and `setCharacterEncoding` along with `getWriter`, you must call `setContentType` and `setCharacterEncoding` *before* `getWriter` so that the returned writer is configured for the correct character encoding. Calls made after `getWriter` are ignored. If you do not call `setContentType` and `setCharacterEncoding` *before* calling `getWriter`, the returned writer uses the container's default encoding.

At your disposal, you also have the `setContentLength` and `setContentLengthLong` methods. In almost all cases, these do not need to be called. The web container sets the `Content-Length` header as it finalizes your response, and it is safest to let it do so.

NOTE *The Servlet 3.1 specification in Java EE 7 is the first version that supports the `setContentLengthLong` method. Before this version, you had to call `setHeader("Content-Length", Long.toString(length))` for responses that could be larger than 2,147,483,647 bytes.*

Setting Headers and Other Response Properties

Serving as counterparts to methods in `HttpServletRequest`, you can call `setHeader`, `setIntHeader`, and `setDateHeader` to set nearly any header value you desire. If the existing response headers already include a header with the name you are setting, the value of that header will be overridden. To avoid this, you can instead use `addHeader`, `addIntHeader`, or `addDateHeader`. These versions do not override existing header values, but instead add additional values for the given headers. You can also call `getHeader`, `getHeaders`, `getHeaderNames`, and `containsHeader` to investigate which headers have already been set on the response.

In addition, you can use:

- `setStatus`: To set the HTTP response status code
- `getStatus`: To determine what the current status of the response is
- `sendError`: To set the status code, indicate an optional error message to write to the response data, direct the web container to provide an error page to the client, and clear the buffer
- `sendRedirect`: To redirect the client to a different URL

This section covered most of the things you can do while servicing an HTTP request in your Servlet and noted important details and cautions where necessary. In the past several sections you have used the Hello-World project to demonstrate working with Servlets. In the next section, you move on to a slightly more complex example.

USING PARAMETERS AND ACCEPTING FORM SUBMISSIONS

In this section, you make your Hello-World project a little more dynamic by accepting parameters and form submissions. You also explore annotation configuration and temporarily forego the deployment descriptor. For the examples in this section, you can follow along in the completed Hello-User project, or you can simply incorporate the changes into your existing project as they are covered.

Several changes have been made to the project. The first thing you should notice is that the `doGet` method is much more complex now:

```
private static final String DEFAULT_USER = "Guest";

@Override
protected void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException
{
    String user = request.getParameter("user");
    if(user == null)
        user = HelloServlet.DEFAULT_USER;

    response.setContentType("text/html");
    response.setCharacterEncoding("UTF-8");

    PrintWriter writer = response.getWriter();
    writer.append("<!DOCTYPE html>\r\n");
    writer.append("<html>\r\n");
    writer.append("  <head>\r\n");
    writer.append("    <title>Hello User Application</title>\r\n");
    writer.append("  </head>\r\n");
    writer.append("  <body>\r\n");
    writer.append("    Hello, ".append(user).append("!\r\n"));
    writer.append("    <form action=\"greeting\" method=\"POST\">\r\n");
    writer.append("      Enter your name:<br/>\r\n");
    writer.append("      <input type=\"text\" name=\"user\"/>\r\n");
    writer.append("      <input type=\"submit\" value=\"Submit\"/>\r\n");
    writer.append("    </form>\r\n");
    writer.append("  </body>\r\n");
    writer.append("</html>\r\n");
}
```

The code in bold is new. It is doing a little logic now:

- It tests if the `user` parameter is included in the request and, if it is not, it uses the `DEFAULT_USER` constant instead.
- It sets the content type of the response to `text/html` and the character encoding to `UTF-8`.
- It gets a `PrintWriter` from the response and writes out a compliant HTML5 document (note the HTML5 DOCTYPE), including the greeting (now directed at a particular user) and a form for supplying your username.

You might wonder how the `doGet` method can receive the form submission when the method type for the form is set to `POST`. This is handled with the simple `doPost` implementation, which is also new:

```
@Override
protected void doPost(HttpServletRequest request, HttpServletResponse response)
```

```

        throws ServletException, IOException
    {
        this.doGet(request, response);
    }
}

```

This implementation simply delegates to the `doGet` method. Either a query parameter or a post variable named `user` can trigger the greeting to change.

The last thing you should notice is the annotation just above the Servlet declaration:

```

@WebServlet(
    name = "helloServlet",
    urlPatterns = {"/greeting", "/salutation", "/wazzup"},
    loadOnStartup = 1
)
public class HelloServlet extends HttpServlet
{
    ...
}

```

NOTE *You'll notice that the class imports have been left off of the newest HelloServlet code example. As your code gets more complex, the imports can begin to take up many dozens of lines of code. This is too much to print in this book efficiently. A good IDE, like the one you use for this book, can recognize the class names and suggest the imports for you, taking the hard work out of your hands. With few exceptions, import and package statements are omitted from the rest of the examples in this book. New classes will be in the com.wrox package unless otherwise stated.*

If you also take a look at the deployment descriptor, you'll notice that the Servlet declaration and mapping were removed from the `web.xml` file. (Or if you made these changes to the existing project, you should remove everything in the deployment descriptor except for the `<display-name>` tag.) The annotation in the previous example replaces the XML that you wrote in your previous project and adds a little bit more.

You still get an instance of `HelloServlet` named `helloServlet`; it still starts when the application starts; and it is still mapped to the `/greeting` URL. It is also now mapped to the `/salutation` and `/wazzup` URLs. As you can tell, this is a much more direct and concise approach to instantiating and mapping servlets. However, it has some drawbacks, which are pointed out throughout the rest of the chapter. For now, compile your project and start Tomcat in your debugger; then go to `http://localhost:8080/hello-world/greeting` in your browser. You should see a screen as shown in Figure 3-2.

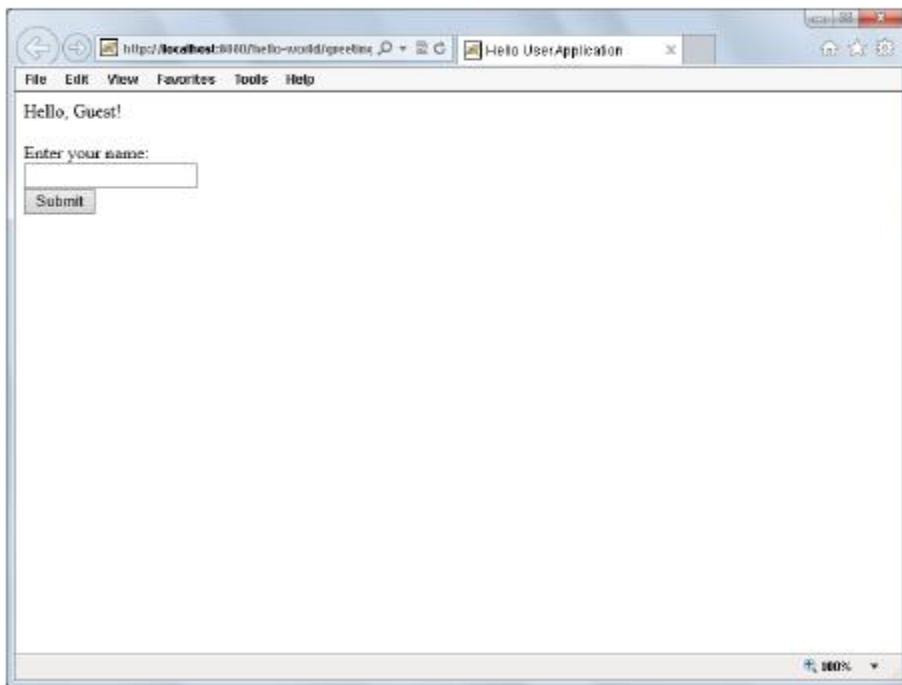


FIGURE 3-2

To understand what this Servlet can do, first add the query string `user=Allison` to the URL so that it is `http://localhost:8080/hello-world/greeting?user=Allison`. The screen should now change and, instead of saying "Hello, Guest!" it should say "Hello, Allison!" In this case the request was serviced by the `doGet` method, which found the `user` query parameter and output it to the screen.

You can confirm this by placing breakpoints in `doGet` and `doPost` and refreshing the page. Now, type your name in the form field on the screen and click the Submit button. If you examine the URL in the address bar, it does not have any query parameters. Instead, your name was included in the request as a post variable, and when the `doPost` method serviced the request and delegated to the `doGet` method, the call to `getParameter` retrieved the post variable, resulting in your name displaying on the screen. Hitting the breakpoints will confirm that this has happened.

Remember from the previous section that single parameter values are not the only thing your Servlets can accept. You can also accept multiple parameter values. The most common example of this is a set of related check boxes, where the user is permitted to check one or more values. Refer to the code Listing 3-1, the `MultiValueParameterServlet`, mapped to `/checkboxes`. Compile and run this code in Tomcat using your debugger and navigate your browser to `http://localhost:8080/hello-world/checkboxes`. The `doGet` method in this Servlet prints out a simple form with five check boxes. The user can select any number of these check boxes and click Submit, which is serviced by the `doPost` method. This method retrieves all the fruit values and lists them on the screen using an unordered list. Try this out by selecting various combinations of check boxes and clicking Submit.

LISTING 3-1: MultiValueParameterServlet.java

```

@WebServlet(
    name = "multiValueParameterServlet",
    urlPatterns = {"/checkboxes"}
)
public class MultiValueParameterServlet extends HttpServlet
{
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException
    {
        response.setContentType("text/html");
        response.setCharacterEncoding("UTF-8");

        PrintWriter writer = response.getWriter();
        writer.append("<!DOCTYPE html>\r\n")
            .append("<html>\r\n")
            .append("      <head>\r\n")
            .append("          <title>Hello User Application</title>\r\n")
            .append("      </head>\r\n")
            .append("      <body>\r\n")
            .append("          <form action=\"checkboxes\" method=\"POST\">\r\n")
            .append("              Select the fruits you like to eat:<br/>\r\n")
            .append("              <input type=\"checkbox\" name=\"fruit\" value=\"Banana\"/>")
            .append("      Banana<br/>\r\n")
            .append("              <input type=\"checkbox\" name=\"fruit\" value=\"Apple\"/>")
            .append("      Apple<br/>\r\n")
            .append("              <input type=\"checkbox\" name=\"fruit\" value=\"Orange\"/>")
            .append("      Orange<br/>\r\n")
            .append("              <input type=\"checkbox\" name=\"fruit\" value=\"Guava\"/>")
            .append("      Guava<br/>\r\n")
            .append("              <input type=\"checkbox\" name=\"fruit\" value=\"Kiwi\"/>")
            .append("      Kiwi<br/>\r\n")
            .append("              <input type=\"submit\" value=\"Submit\"/>\r\n")
            .append("          </form>")
            .append("      </body>\r\n")
            .append("</html>\r\n");
    }

    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException
    {
        String[] fruits = request.getParameterValues("fruit");

        response.setContentType("text/html");
        response.setCharacterEncoding("UTF-8");

        PrintWriter writer = response.getWriter();
        writer.append("<!DOCTYPE html>\r\n")
            .append("<html>\r\n")
            .append("      <head>\r\n")
            .append("          <title>Hello User Application</title>\r\n")

```

```

.append("      </head>\r\n")
.append("      <body>\r\n")
.append("          <h2>Your Selections</h2>\r\n");

if(fruits == null)
    writer.append("          You did not select any fruits.\r\n");
else
{
    writer.append("          <ul>\r\n");
    for(String fruit : fruits)
    {
        writer.append("          <li>").append(fruit).append("</li>\r\n");
    }
    writer.append("          </ul>\r\n");
}

writer.append("      </body>\r\n")
.append("</html>\r\n");
}
}

```

This section has shown you the various ways that you can use request parameters within your Servlet methods. You have explored query parameters and post variables, along with single-value and multivalue parameters. In the next section you learn about various ways to configure your application using init parameters.

CONFIGURING YOUR APPLICATION USING INIT PARAMETERS

When writing a Java web application, the need will inevitably arise to provide ways of configuring your application and the Servlets within it. There are many ways to do that using numerous technologies, and you explore a few of those in this book. The simplest means of configuring your application, through context *initialization parameters* (usually shortened to *init parameters*) and Servlet init parameters, is covered in this section. These parameters can be put to any number of uses, from defining connection information for communicating with a relational database, to providing an e-mail address to send store order alerts to. They are defined at application startup and cannot change without restarting the application.

Using Context Init Parameters

Earlier you emptied the deployment descriptor file and replaced your Servlet declaration and mappings with annotations on the actual classes. Although this is one thing (added in the Servlet 3.0 specification in Java EE 6) that you can do without the deployment descriptor, several things still require the deployment descriptor. Context init parameters are one such feature. You declare context init parameters using the `<context-param>` tag within the `web.xml` file. The following code example shows two context init parameters added to the deployment descriptor:

```

<context-param>
    <param-name>settingOne</param-name>
    <param-value>foo</param-value>

```

```

</context-param>
<context-param>
    <param-name>settingOne</param-name>
    <param-value>foo</param-value>
</context-param>

```

This creates two context init parameters: `settingOne` having a value of `foo` and `settingTwo` having a value of `bar`. You can easily obtain and use these parameter values from anywhere in your Servlet code. The `ContextParameterServlet` demonstrates this ability:

```

@.WebServlet(
    name = "contextParameterServlet",
    urlPatterns = {"/*contextParameters"})
)
public class ContextParameterServlet extends HttpServlet
{
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException
    {
        ServletContext c = this.getServletContext();
        PrintWriter writer = response.getWriter();

        writer.append("settingOne: ").append(c.getInitParameter("settingOne"))
            .append(", settingTwo: ").append(c.getInitParameter("settingTwo"));
    }
}

```

If you compile, debug, and navigate to `http://localhost:8080/hello-world/contextParameters`, you can see these parameters listed on the screen. Every Servlet in your application shares these init parameters, and their values are the same across all servlets. There may be cases, however, in which you need a setting that applies to only a single Servlet. For this purpose you would use Servlet init parameters.

NOTE *It should be noted that as of Servlet 3.0 you can call the `setInitParameter` method on the `ServletContext` as an alternative to defining context init parameters using `<context-param>`. However, this method can only be called within the `contextInitialized` method of a `javax.servlet.ServletContextListener` (which you learn about in Chapter 9) or the `onStartup` method of a `javax.servlet.ServletContainerInitializer` (which you learn about in Chapter 12). Even so, changing the values would require recompiling your application, so XML is usually the best option for context init parameters.*

Using Servlet Init Parameters

Consider the code for the `ServletParameterServlet` class. You may immediately notice that it is not annotated with `@WebServlet`. Don't worry; you learn why in a minute. The code is otherwise nearly identical to the `ContextParameterServlet`. Instead of getting your init parameters from the `ServletContext` object, you obtain them from the `ServletConfig` object:

```

public class ServletParameterServlet extends HttpServlet
{
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException
    {
        ServletConfig c = this.getServletConfig();
        PrintWriter writer = response.getWriter();

        writer.append("database: ").append(c.getInitParameter("database"))
            .append(", server: ").append(c.getInitParameter("server"));
    }
}

```

Of course, just having the Servlet code isn't enough. The following XML added to the deployment descriptor declares and maps the servlet and also does a little bit more:

```

<servlet>
    <servlet-name> servletParameterServlet </servlet-name>
    <servlet-class> com.wrox.ServletParameterServlet </servlet-class>
    <init-param>
        <param-name> database </param-name>
        <param-value> CustomerSupport </param-value>
    </init-param>
    <init-param>
        <param-name> server </param-name>
        <param-value> 10.0.12.5 </param-value>
    </init-param>
</servlet>
<servlet-mapping>
    <servlet-name> servletParameterServlet </servlet-name>
    <url-pattern> /servletParameters </url-pattern>
</servlet-mapping>

```

The `<init-param>` tag, like the `<context-param>` tag for the Servlet context, creates an init parameter specific to this Servlet. If you compile, debug, and navigate to `http://localhost:8080/hello-world/servletParameters`, you can see the `database` and `server` parameters specified in the deployment descriptor. So why, you might ask, can't you use annotations for this like you can for the rest of the Servlet mapping? Well, technically you can. You can achieve the same result as in the previous code by removing the initialization and mapping from the deployment descriptor and adding this annotation to the Servlet declaration:

```

@WebServlet(
    name = "servletParameterServlet",
    urlPatterns = {"/servletParameters"},
    initParams = {
        @WebInitParam(name = "database", value = "CustomerSupport"),
        @WebInitParam(name = "server", value = "10.0.12.5")
    }
)
public class ServletParameterServlet extends HttpServlet
{
    ...
}

```

The drawback to doing this, however, is that the values of the Servlet init parameters can no longer be changed without recompiling the application. Sure, there may be settings that you wouldn't want to change without recompiling the application, but at that point why not just make them class constants? The advantage of putting Servlet init parameters in the deployment descriptor is that a server administrator needs to change only a few lines of XML and restart the deployed application to effect the change. If such settings contain connection information for a relational database, the last thing you want to do is to recompile the application to change the IP address of the database server!

The next section introduces a new feature of `HttpServletRequest`s added in the Servlet 3.0 specification and a new example application that you improve upon throughout the rest of the book.

THE DRAWBACKS OF @CONFIG

As mentioned earlier there are advantages and disadvantages to using annotation-based configuration (often simply called @Config) in your web application. The primary advantage is the lack of XML and the direct, concise annotation language used to configure your application. However, there are numerous drawbacks to this approach as well.

One example of this is the inability to create multiple instances of a single Servlet class. You saw earlier in the chapter how such a pattern might be used. This is impossible using annotations and can be accomplished only using XML configuration or programmatic Java configuration.

In Chapter 9, you learn about filters and why it's important to carefully construct the order the filters execute in. You can make filters execute in a specific order when declaring them using XML configuration or programmatic Java configuration. If you declare your filters using `@javax.servlet.annotation.WebFilter`, however, it is impossible to make them execute in a specific order (something many feel is a glaring oversight in the Servlet 3.0 and 3.1 specifications). Unless your application has only one filter, `@WebFilter` is virtually useless.

There are many smaller things that still require the XML deployment descriptor to accomplish, such as defining error-handling pages, configuring JSP settings, and providing a list of welcome pages. Thankfully, you can mix-and-match XML, annotation, and programmatic Java, and configuration, so you can use each when it's most convenient. Throughout this book, you use all three techniques.

UPLOADING FILES FROM A FORM

Uploading files to Java EE Servlets has nearly always been possible, but it used to require considerable effort. The task was so complex that Apache Commons made an entire project, called Commons FileUpload, to handle all the work. Thus, what seemed to be the simple requirement of accepting file upload submissions required introducing a third-party dependency in your application. Servlet 3.0 in Java EE 6 changed all that when it introduced the multipart configuration options for Servlets and the `getPart` and `getParts` methods in `HttpServletRequest`.

You can use this feature as a launching point for your interchapter example application: the Customer Support project. Although each chapter has smaller examples to demonstrate specific points, each chapter also includes a new version of the Customer Support project that incorporates the new topics learned in that chapter.

Introducing the Customer Support Project

The setup is a global website serving customers around the world for Multinational Widget Corporation. Your product managers have been tasked with adding an interactive customer support application to the company's website. It should enable users to post questions or support tickets and enable employees to respond to those inquiries. Support tickets and comments alike should contain file attachments. For urgent matters, customers should enter a chat window with a dedicated support representative. And, to top it all off, because this is Multinational Widget Corporation, the entire application should be localizable in as many languages as the company decides to translate. That's not asking much, right?

Oh, yea. It needs to be really secure, too.

Obviously you can't tackle this all at once, especially with how little you've learned so far, so for each chapter you either tackle a small feature or improve upon code written in the chapter before. For the rest of this chapter, refer to the Customer-Support-v1 project. The project is relatively simple right now. It consists of three pages, handled by `doGet`: a list of tickets, a page to create tickets, and a page to view a ticket. It also has the capability of downloading a file attached to a ticket and of accepting a `POST` request to create a new ticket. Although the code is not complex and consists largely of concepts you have already covered in this chapter, there is too much to print it all here. You need to follow along in the code downloaded from the website.

Configuring the Servlet for File Uploads

In the project you can find a `Ticket` class, an `Attachment` class, and the `TicketServlet` class. The `Ticket` and `Attachment` classes are simple *POJOs*—plain old Java objects. The `TicketServlet` does all the hard work at this time, so start by looking at its declaration and fields:

```
@WebServlet(  
    name = "ticketServlet",  
    urlPatterns = {"tickets"},  
    loadOnStartup = 1  
)  
@MultipartConfig(  
    fileSizeThreshold = 5_242_880, //5MB  
    maxFileSize = 20_971_520L, //20MB  
    maxRequestSize = 41_943_040L //40MB  
)  
public class TicketServlet extends HttpServlet  
{  
    private volatile int TICKET_ID_SEQUENCE = 1;  
    private Map<Integer, Ticket> ticketDatabase = new LinkedHashMap<>();  
    ...  
}
```

Already you should see some things you recognize and some things you don't. The `@MultipartConfig` annotation instructs the web container to provide file upload support for this servlet. It has several important attributes you should look at. The first, which is not shown here, is `location`. This instructs the web container in which directory to store temporary files if it needs to. In most cases, however, it is sufficient to omit this field and let the application server use its default temporary directory. The `fileSizeThreshold` tells the web container how big the file has to be before it is written to the temporary directory.

In this example, uploaded files smaller than 5 megabytes are kept in memory until the request completes and then they become eligible for garbage. After a file exceeds 5 megabytes, the container instead stores it in `location` (or default) until the request completes, after which it deletes the file from disk. The last two parameters, `maxFileSize` and `maxRequestSize`, place limits on uploaded files: `maxFileSize` in this example prohibits an uploaded file from exceeding 20 megabytes, whereas `maxRequestSize` prohibits the total size of a request from exceeding 40 megabytes, regardless of the number of file uploads it contains. That's really all there is to it. The Servlet is now configured to accept file uploads.

NOTE *As with configuring Servlet init parameters using annotations, the multipart configuration parameters in the previous example cannot be changed without recompiling the application. If you anticipate server administrators' needing to customize these settings without recompiling the application, you need to use the deployment descriptor instead of @WebServlet and @MultipartConfig. Within the <servlet> tag you can place a <multipart-config> tag, and within that you can use the <location>, <file-size-threshold>, <max-file-size>, and <max-request-size> tags.*

You may also notice that the "ticket database" isn't a database at all (Or is it? It's a medium for storing data, no?), but rather a simple hash map. Eventually in Part III of this book you back your application with a relational database. For now, however, you want to get the user interface right and understand the business requirements so that product management at Multinational Widget Corporation is happy. After that, you can worry about persisting your data.

Now that you understand what you've seen so far, take a look at the `doGet` implementation:

```

@Override
protected void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException
{
    String action = request.getParameter("action");
    if(action == null)
        action = "list";
    switch(action)
    {
        case "create":
            this.showTicketForm(response);
            break;
        case "view":
            this.viewTicket(request, response);
            break;
    }
}

```

```

        case "download":
            this.downloadAttachment(request, response);
            break;
        case "download":
        default:
            this.listTickets(response);
            break;
    }
}

```

There's too much to do to put everything in the `doGet` method; before long, you could have a method that spans hundreds of lines. In this example, the `doGet` method uses a primitive action/executor pattern: The action is passed in through a request parameter, and the `doGet` method sends the request to an executor (method) based on that action. The `doPost` method is similar:

```

@Override
protected void doPost(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException
{
    String action = request.getParameter("action");
    if(action == null)
        action = "list";
    switch(action)
    {
        case "create":
            this.createTicket(request, response);
            break;
        case "download":
        default:
            response.sendRedirect("tickets");
            break;
    }
}

```

One new thing you can notice in `doPost` is the use of the `redirect` method. You learned about this method a few sections ago. In this case, if the client performs a `POST` with a missing or invalid `action` parameter, his browser is redirected to the page that lists tickets. Most of the methods in this class are nothing new: use of parameters, use of the `PrintWriter` to output content to the client's browser, and so on. Not all the code can fit in this book, but there are some new features used here that you should look at. The following example is a snippet of the `downloadAttachment` method, only the part that contains something new you haven't seen yet:

```

response.setHeader("Content-Disposition",
    "attachment; filename=" + attachment.getName());
response.setContentType("application/octet-stream");

ServletOutputStream stream = response.getOutputStream();
stream.write(attachment.getContents());

```

This simple bit of code is responsible for handing off the file download to the client's browser. The `Content-Disposition` header, as set, forces the browser to ask the client to save or download the file instead of just opening the file inline in the browser. The content type is a generic, binary content type that keeps the data from having some kind of character encoding applied to it. (A more correct implementation would know the attachment's actual MIME content type and use that value, but that task is outside the scope of this book.) Finally, the `ServletOutputStream` is used to write

the file contents to the response. This may not be the most efficient way to write the file contents to the response because it may suffer memory issues for large files. If you anticipate permitting large file downloads, you shouldn't store files in-memory, and you should copy the bytes from a file's `InputStream` to the `ResponseOutputStream`. You should then flush the `ResponseOutputStream` frequently so that bytes are continuously streaming back to the user's browser instead of buffering in memory. The exercise of improving this code is left up to you.

Accepting a File Upload

Lastly, take a look at the `createTicket` method and the method that it uses, `processAttachment`, in Listing 3-2. These methods are particularly important because they deal with handling a file upload — something you have not done yet. The `processAttachment` method gets the `InputStream` from the multipart request and copies it to the `Attachment` object. It uses the `getSubmittedFileName` method added in Servlet 3.1 to identify the original file name before it was uploaded. The `createTicket` method uses this method and other request parameters to populate the `Ticket` object and add it to the database.

LISTING 3-2: Part of TicketServlet.java

```

private void createTicket(HttpServletRequest request,
                         HttpServletResponse response)
    throws ServletException, IOException
{
    Ticket ticket = new Ticket();
    ticket.setCustomerName(request.getParameter("customerName"));
    ticket.setSubject(request.getParameter("subject"));
    ticket.setBody(request.getParameter("body"));

    Part filePart = request.getPart("file1");
    if(filePart != null)
    {
        Attachment attachment = this.processAttachment(filePart);
        if(attachment != null)
            ticket.addAttachment(attachment);
    }

    int id;
    synchronized(this)
    {
        id = this.TICKET_ID_SEQUENCE++;
        this.ticketDatabase.put(id, ticket);
    }

    response.sendRedirect("tickets?action=view&ticketId=" + id);
}

private Attachment processAttachment(Part filePart)
    throws IOException
{
    InputStream inputStream = filePart.getInputStream();
    ByteArrayOutputStream outputStream = new ByteArrayOutputStream();

```

```
int read;
final byte[] bytes = new byte[1024];

while((read = inputStream.read(bytes)) != -1)
{
    outputStream.write(bytes, 0, read);
}

Attachment attachment = new Attachment();

attachment.setName(filePart.getSubmittedFileName());
attachment.setContents(outputStream.toByteArray());

return attachment;
}
```

One thing you may notice in the `createTicket` method is the use of a `synchronized` block to lock access to the ticket database. You explore this a little more in the next and final section of the chapter.

MAKING YOUR APPLICATION SAFE FOR MULTITHREADING

Web applications are, by nature, multithreaded applications. At any given time, zero, one, or a thousand people may be using your web application simultaneously, and your code must anticipate and account for this. There are dozens of different facets to this topic, and entire books have been written about multithreading and managing concurrency in applications. Obviously, this book cannot possibly cover all the important multithreading discussions. However, you should know two things above all else when considering concurrency in your web applications.

Understanding Requests, Threads, and Method Execution

Every web container is, of course, slightly different. But in the Java EE world, generally speaking, a web container contains some type of thread pool, possibly called the *connector pool* or *executor pool*.

When the container receives a request, it looks for an available thread in the pool. If it does not find an available thread and the thread pool has already reached its maximum size, the request enters a queue—first in first out—and waits for an available thread. (Typically, there is also a higher limit, called the `acceptCount` setting in Tomcat, which defines the maximum number of connections that can be queued before the container starts rejecting connections.) Once a thread is available, the container borrows the thread from the pool and hands the request off to be handled by the thread. At this point, the thread is no longer available for any other incoming requests. In a normal request, the thread and request will be linked throughout the life of the request. As long as the request is processed by your code, that thread will be dedicated to the request. Only when the request has completed and the content of your response has been written back to the client will the thread be free from the request and return to the pool to service another request.

Creating and destroying threads includes a lot of overhead that can slow an application down, so employing a pool of reusable threads in this manner eliminates this overhead and improves performance.

The thread pool has a configurable size that determines how many connections can be serviced at once. Although this is not a discussion of the techniques and practices of managing application servers, hardware limitations place a practical limit on the size of this pool, after which increasing the pool size achieves no performance gains (and often can hurt performance). The default maximum pool size in Tomcat is 200 threads, and this number can be increased or decreased. You must understand this because it means that, in a worst-case scenario, 200 different threads (or more, if you increase the number) could be executing the same method in your code on the same instance of that code simultaneously. Therefore, you should consider the way that code functions so that simultaneous executions of the code in multiple threads do not result in exceptional behavior.

NOTE *On the subject of requests and threads, there are circumstances during which a thread may not be devoted to a request for the entire life of the request. Servlet 3.0 in Java EE 6 added the concept of asynchronous request contexts. Essentially, when your Servlet services a request, it can call ServletRequest's startAsync method. This returns a javax.servlet.AsyncContext object in which that request resides. Your Servlet can then return from the Servlet's service method without responding to the request, and the thread will be returned to the pool. The request does not close, but instead stays open, unanswered. Later, when some event occurs, your application can retrieve the response object from the AsyncContext and use it to send a response to the client. You learn more about using asynchronous request contexts in Chapter 9. This approach is often employed for a technique called long polling, something that Chapter 10 discusses.*

Protecting Shared Resources

The most typical complication when coding for a multithreaded application is the access of shared resources. Objects and variables created during the execution of a method are safe as long as that method is executing — other threads do not have access to them. However, static and instance variables in a Servlet, for example, could be accessed by multiple threads simultaneously (remember: in the worst case, even 200 threads simultaneously). It's important to synchronize access to these shared resources to keep their contents from becoming corrupt and possibly causing errors in your application.

You can employ a few techniques to protect shared resources from these problems. Consider the first line of code in the `TicketServlet`:

```
private volatile int TICKET_ID_SEQUENCE = 1;
```

In Java, it is sometimes possible for one thread to read the previous value of a variable even after the value has been changed in another thread. This can cause consistency issues in some circumstances. The `volatile` keyword in this case establishes a happens-before relationship for all future reads of the variable and guarantees that other threads will always see the latest value of the variable.

Next, recall the synchronized block of code in the `createTicket` method from Listing 3-2:

```
        synchronized(this)
        {
            id = this.TICKET_ID_SEQUENCE++;
            this.ticketDatabase.put(id, ticket);
        }
```

Two things are happening in this block of code: the `TICKET_ID_SEQUENCE` is incremented and its value retrieved, and the `Ticket` is inserted into the hash map of tickets. Both of these variables are instance variables of the Servlet, meaning multiple threads may have access to them simultaneously. Putting these actions within the synchronized block guarantees that no other thread can execute these two lines of code at the same time. The thread currently executing this block of code has exclusive access to execute the block until it completes. Of course, care should always be taken when using synchronized code blocks or methods because incorrect application of synchronization can result in a deadlock, a problem beyond the scope of this book.

WARNING *One final thing to keep in mind when writing your Servlet methods: **Never** store request or response objects in static or instance variables. Just don't do it. There is no maybe — it **will** cause problems for you. Any objects and resources that belong to a request should exist only as local variables and method arguments.*

SUMMARY

In this chapter, you were introduced to the `Servlet` interface and `GenericServlet` and `HttpServlet` abstract classes, along with the `HttpServletRequest` and `HttpServletResponse` interfaces. You learned how to service incoming requests and respond to them appropriately using the request and response objects. You experimented with the deployment descriptor and explored how to configure Servlets using `web.xml` and annotations. You also discovered one of the most important tasks when dealing with HTTP requests: handling request parameters, including query parameters and post variables, and accepting file uploads through form submissions. You were introduced to context and Servlet init parameters and how to use them to configure your application. Finally, you learned about request threads and thread pools and why multithreading considerations are so important in web application programming.

At this point, you should have a firm grasp on the basics of creating and using Servlets in your web application. One of the major inconveniences you may have noticed during this chapter is the complexity and cumbersomeness of writing simple HTML to the response. In the next chapter you explore the answer to this problem and how it makes life much easier in the Java EE world: JavaServer Pages.

4

Using JSPs to Display Content

IN THIS CHAPTER

- Using `
` is easier than `output.println("
")`
- Creating your first JSP
- Using Java within a JSP (and why you shouldn't)
- Combining Servlets and JSPs
- A note about JSP Documents (JSPX)

WROX.COM CODE DOWNLOADS FOR THIS CHAPTER

You can find the [wrox.com](http://www.wrox.com/go/projavaforwebapps) code downloads for this chapter at www.wrox.com/go/projavaforwebapps on the Download Code tab. The code for this chapter is divided into the following major examples:

- Hello-World-JSP Project
- Hello-User-JSP Project
- Customer-Support-v2 Project

NEW MAVEN DEPENDENCIES FOR THIS CHAPTER

In addition to the Maven dependency introduced in the previous chapter, you will also need the following Maven dependencies. The exclusions are necessary because the JSTL implementation defines transient dependencies on older versions of the JSP and Servlet specifications that have different Maven artifact IDs than the current versions.

```
<dependency>
  <groupId>javax.servlet.jsp</groupId>
  <artifactId>javax.servlet.jsp-api</artifactId>
  <version>2.3.1</version>
  <scope>provided</scope>
```

```
</dependency>

<dependency>
    <groupId>javax.servlet.jsp.jstl</groupId>
    <artifactId>javax.servlet.jsp.jstl-api</artifactId>
    <version>1.2.1</version>
    <scope>compile</scope>
</dependency>

<dependency>
    <groupId>org.glassfish.web</groupId>
    <artifactId>javax.servlet.jsp.jstl</artifactId>
    <version>1.2.2</version>
    <scope>compile</scope>
    <exclusions>
        <exclusion>
            <groupId>javax.servlet</groupId>
            <artifactId> servlet-api</artifactId>
        </exclusion>
        <exclusion>
            <groupId>javax.servlet.jsp</groupId>
            <artifactId> jsp-api</artifactId>
        </exclusion>
        <exclusion>
            <groupId>javax.servlet.jsp.jstl</groupId>
            <artifactId> jstl-api</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

In the last chapter you learned about Servlets and handling requests, responses, request parameters, file uploads, Servlet configuration, and more. However, you may have noticed a serious inconvenience when writing the Servlet code to output HTML content to the response: Repeatedly calling methods on the `ServletOutputStream` or `PrintWriter` classes to output the content and having to put HTML content within Java strings, requiring escaping of quotation marks, is a real pain. In this chapter, you explore JavaServer Pages and how they can make your life a whole lot easier.

 IS EASIER THAN OUTPUT.PRINTLN("
")

Java is a powerful language. It has many capabilities and features that make it useful, flexible, and easy to use. Chances are, you are reading this book because you like Java and want to learn how to do more with it. So what's up with this?

```
PrintWriter writer = response.getWriter();
writer.append("<!DOCTYPE html>\r\n")
    .append("<html>\r\n")
    .append("    <head>\r\n")
    .append("        <title>Hello World Application</title>\r\n")
    .append("    </head>\r\n")
    .append("    <body>\r\n")
    .append("        Nick says, \"Hello, World!\"\r\n")
    .append("    </body>\r\n");
writer.append("</html>\r\n");
```

The number of ways this is inconvenient and cumbersome is rather long. Significantly more code must be written to achieve this. More file space is needed to store the code. Time is wasted writing and testing the code. Verbosity with line endings (`\r\n`) is necessary to make HTML source that is readable in the browser's View Source feature. Any quotation marks that appear in the HTML must be escaped so that they do not prematurely terminate the `String` literal. And — perhaps one of the worst problems — code editors cannot easily (in most cases, at all) recognize and validate HTML code within `Strings` to tell you if you're doing something wrong. Surely there is a better way. After all, it's just text. If you wrote the previous example in a plain HTML file, it would be simple:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Hello World Application</title>
  </head>
  <body>
    Hello, World!
  </body>
</html>
```

Fortunately, the creators of the Java EE specification realized that this system would quickly become unwieldy and designed *JavaServer Pages*, also known as *JSPs*, to answer the need.

Why JSPs Are Better

The problem with the most-recent code example is that it's a static HTML document. It may have been easier to write and it will likely be infinitely easier to maintain than the example written in Java, but there's nothing dynamic about it. JSPs are essentially a hybrid solution, combining Java code and HTML tags. JSPs can contain any HTML tag in addition to Java code, built-in JSP tags (Chapter 7), custom JSP tags (Chapter 8), and something called the Expression Language (Chapter 5). Many of these features you learn about in later chapters.

In this chapter, you explore the basic rules of JSPs and learn about the syntax, directives, declarations, scriptlets, and expressions of the JSP technology. You also learn about the life cycle of a JSP and how it is ultimately used to send a response back to the user.

There are alternatives to JSPs. Perhaps the most common alternative is Facelets, part of the broader JavaServer Faces technology (or JSF for short, making it easy to confuse with JSP). There are also templating frameworks, such as Velocity, Freemarker, SiteMesh, and Tiles, that all, in some fashion, supplement or replace the features provided by JSPs. This book cannot possibly cover all the options and variations of presentation technologies that work with the Servlet 3.1 specification. It will, therefore, focus on the most popular and widely used technology.

The following example, which you can find in the `index.jsp` file of the Hello-World-JSP project on the wrox.com downloads page, re-creates the Hello-World project from Chapter 2, but uses a JSP instead of a Servlet to display the greeting to the user.

```
<%@ page contentType="text/html; charset=UTF-8" language="java" %>
<!DOCTYPE html>
<html>
  <head>
    <title>Hello World Application</title>
```

```
</head>
<body>
    Hello, World!
</body>
</html>
```

This example is nearly identical to the original HTML-only example earlier in the section. The only new code is the first line, highlighted in bold. This is one of several JSP *directives* that you will examine in more detail in the section “Creating Your First JSP.” This particular directive sets the content type and character encoding of the page, something you previously did with the `setContentType` and `setCharacterEncoding` method calls on the `HttpServletResponse`. Everything else in this JSP is plain HTML, transmitted back to the client as-is in the response. The interesting question is, “What’s actually happening behind the scenes?”

What Happens to a JSP at Run Time

A JSP is really just a fancy Servlet. Perhaps you have heard the phrase “syntactic sugar.” Ultimately, in one way or another, all popular languages programmers use today on a regular basis are syntactic sugar. Take Java, for example, as a code you write with. When you compile Java code, it is turned into *bytecode*. The bytecode is what matters — not the Java code. In fact, many different statements in Java can turn into identical bytecode. But to take it a step further, bytecode is not the final rendering of a Java program. This bytecode is still platform-independent, but that is not sufficient to run on varying operating systems.

When Java runs in the JRE, the Just In Time compiler compiles it into *machine code*, which is specific to the platform it runs on. Ultimately, it’s this machine code that is executed. Even lower-level languages, such as C, are simply syntactic sugar for the machine code they actually get compiled to. JSPs are another form of syntactic sugar. At run time, the JSP code is interpreted by the JSP compiler, which parses out all the special features in the JSP code and translates them to Java code. The Java class created from each JSP implements `Servlet`. Then, the Java code goes through the same cycle it normally does. Still at run time, it is compiled into bytecode and then into machine code. Finally, the JSP-turned-Servlet responds to requests like any other Servlet.

To investigate this, for these steps:

1. Compile the Hello-World-JSP project in your IDE, start your debugger, and open your browser to `http://localhost:8080/hello-world/`. You should see the all-too-familiar greeting on your screen.
2. Browse your file system to the Tomcat 8.0 home directory (`C:\Program Files\Apache Software Foundation\Tomcat 8.0 on Windows`) and go into the directory `work\Catalina\localhost\hello-world`. Tomcat puts all compiled JSPs for the application in this directory, but it also leaves behind the intermediate Java files it generates so that you can inspect and troubleshoot with them.
3. Continue going down further directories until you come across the `index_jsp.java` file. Open it (not the `index_jsp.class` file) in your favorite text editor.

What you should find is a class that extends `org.apache.jasper.runtime.HttpJspBase`. This abstract class extends — you may have guessed — `HttpServlet`. `HttpJspBase` provides some base

functionality that will be used by all JSPs that Tomcat compiles, and when your JSP is executed, ultimately the `service` method on that Servlet is executed, which eventually executes the `_jspService` method.

If you inspect the `_jspService` method, you'll find a series of method calls writing your HTML to the output stream. This code should look very familiar to you because it's not that different from the Java code that you replaced with this JSP. Of course, the JSP Servlet class does not look the same on every web container. The `org.apache.jasper` classes, for example, are Tomcat-specific classes. Your JSP compiles differently on each different web container you run it on. The important point is that there is a standard specification for the behavior and syntax of JSPs, and as long as the web containers you use are compliant with the specification, your JSPs should run the same on all of them, even if the Java code they get translated into looks completely different.

JSPs, just like your normal Servlets, can also be debugged at run time. To demonstrate this, place a breakpoint on the line of your JSP that contains "Hello, World," and then refresh your browser. At this point you should hit the breakpoint in the JSP, and you should notice a few things. First, you can hit breakpoints directly within the JSP code! You don't have to place breakpoints in the translated JSP Servlet class; Java, Tomcat, and your IDE can match the breakpoint in the JSP to code executing in the run time. You should also notice that, although the breakpoint might be in the JSP code, the debugger clearly is not. The stack will show that your run time has paused within the `_jspService` method, and the variables window shows you all the instance and local variables defined within that scope in the `index_jsp` class.

WARNING *IntelliJ IDEA has much better JSP debugging facilities than does Eclipse IDE. If you are using Eclipse, it's possible that you may not be able to place a breakpoint in this JSP at all. As of now, Eclipse only lets you place breakpoints in the Java code embedded in your JSPs, while IntelliJ allows breakpoints in any JSP code.*

Like all other Servlets running in your web container, JSPs have a life cycle. In some web containers, such as Tomcat, the JSP is translated and compiled just in time when the first request to that JSP arrives. For future requests, the JSP is already compiled and ready to use. This, as you can imagine, introduces some performance impacts. Although the performance hit generally comes only on the first request, leaving all subsequent requests to run at a decent speed, this is still unwanted in some production environments. Because of this, many web containers give you the option of precompiling all of an application's JSPs as it deploys. This, of course, significantly slows down deployment for large applications. If you have many thousands of JSPs, your application could conceivably take 10 minutes to deploy instead of just 1. It's up to the organization to decide which configuration meets its needs best. Regardless of the time of compilation, after the first request arrives, the JSP Servlet will be instantiated and initialized, and then the first request can be serviced.

By this point you should realize that the code you write in your JSP ultimately is translated into some version of the code you would have had to write anyway if you didn't have JSPs. So why, you might ask, should one even bother with JSPs? The fact remains that the JSP is a much easier file format for producing markup for display in a web browser than writing straight Java code. If this can improve the speed, efficiency, and accuracy of your development process, the question actually is, "Why *wouldn't* you use JSPs?"

CREATING YOUR FIRST JSP

You've explored a JSP that was already written for you, so now work on creating your own JSP. You need to know some things about how JSPs are structured and what you can put in JSPs. You go over some of the basic need-to-knows in this section and then delve a little further in the next.

Understanding the File Structure

In the previous chapter you explored Servlets and answered the question, "What must you do in the `service` method?" The answer was that you *must* appropriately respond to the HTTP request with a valid HTTP response, but because `HttpServlet` takes care of all of that for you, your `doGet` and `doPost` methods could literally be empty methods (as useless as that was). As it turns out, the question in this case is still the same. There are many things that must happen when a JSP is executed, but all those "musts" are handled for you.

To demonstrate this, create a file named `blank.jsp` in the web root of an empty project; delete all its contents (your IDE might put some code in there for you — delete it all); and redeploy your project. Alternatively, just use the Hello-World-JSP you downloaded from wrox.com, which already contains a `blank.jsp`. When you go to `http://localhost:8080/hello-world/blank.jsp`, you don't get any errors. Everything works fine; you just get a useless blank page back. Now put the following code in it, redeploy, and reload:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Hello World Application</title>
  </head>
  <body>
    Hello, World!
  </body>
</html>
```

There's just a slight difference between `blank.jsp` and `index.jsp` now, that being the missing special tag that's on the first line of `index.jsp`. And yet, the content still displays the same. This is because JSPs by default have a content type of `text/html` and a character encoding of ISO-8859-1. However, this default character encoding is incompatible with many special characters like those in non-English languages, which can interfere with efforts to localize your application. So, at a minimum, your JSP needs to contain HTML to display to the user. However, to ensure that HTML displays correctly in all browsers on all systems in many languages, you'll want to include certain JSP tags to control the data sent to the client, such as setting the character encoding to the localization-friendly UTF-8.

Several different types of tags can be used in JSPs, and you explore more of them in the next section. Of the directive tag type, there is one that you have already seen:

```
<%@ page ... %>
```

This directive tag provides you with some controls over how the JSP is translated, rendered, and transmitted back to the client. In the `index.jsp` example, the `page` directive looks like this:

```
<%@ page contentType="text/html; charset=UTF-8" language="java" %>
```

The `language` attribute tells the container which *JSP scripting language* this JSP uses. A JSP scripting language (not to be confused with interpreted scripting languages) is a language that can be embedded in a JSP for scripting certain actions. Currently, Java is the only supported scripting language for JSPs, but this attribute leaves that open for extension someday.

Technically, you can omit this attribute. Because Java is the only supported JSP scripting language, and in addition Java is the default in the specification, Java is implied when this attribute is missing. The `contentType` attribute tells the container the value of the `Content-Type` header that should be sent back along with the response. The `Content-Type` header contains both the content type and the character encoding, separated with a semicolon. If you recall reading the `index.jsp.java` file, it contained the Java that this attribute was translated to:

```
response.setContentType("text/html; charset=UTF-8");
```

It should be noted that the previous code snippet is the equivalent of the following two lines of code, which you saw in your Hello-User project from Chapter 3:

```
response.setContentType("text/html");
response.setCharacterEncoding("UTF-8");
```

And furthermore, these are both equivalent to the following line of code:

```
response.setHeader("Content-Type", "text/html; charset=UTF-8");
```

As you can see, there are several ways to accomplish the same task. The `setContentType` and `setCharacterEncoding` methods are convenience methods. Which method you use is up to you; although, you should generally pick one and stick to it to avoid confusion. However, as most of your content code from here on will be JSP-based, you'll mostly just be concerned with the `contentType` attribute of the page directive.

Directives, Declarations, Scriptlets, and Expressions

In addition to the various HTML and JSP tags you can use within a JSP, there are several unique structures that define a sort of JSP language. They are *directives*, *declarations*, *scriptlets*, and *expressions*. In the simplest terms, they look like this:

```
<%@ this is a directive %>
<%! This is a declaration %>
<% this is a scriptlet %>
<%= this is an expression %>
```

Using Directives

Directives are used to direct the JSP interpreter to perform an action (such as setting the content type) or make an assumption about the file (such as which scripting language it uses), to import a class, to include some other JSP at translation time, or to include a JSP tag library.

Using Declarations

You use declarations to declare something within the scope of your JSP Servlet class. For example, you could define instance variables, methods, or classes within a declaration tag. You need to remember that these declarations all are made within the generated JSP Servlet class, so any classes you define are actually inner classes of the JSP Servlet class.

Using Scriptlets

Like a declaration, a scriptlet also contains Java code. However, scriptlets have a different scope. Although code within a declaration is copied to the JSP Servlet class body at translation time and must therefore be used to *declare* some field, type, or method, scriptlets are copied to the body of the `_jspService` method you looked at earlier. Any local variables that are in scope within this method execution will be in scope within your scriptlets, and any code that is legal within a method body is legal within a scriptlet. So, you can define local variables, but not instance fields. You can use conditional statements, manipulate objects, and perform arithmetic, all things you cannot do within a declaration. You can even define classes (as odd as that may sound, but it is legal in Java to have class definitions within a method), but the classes do not have scope outside the `_jspService` method. A class, method, or variable defined within a declaration can be used within a scriptlet, but a class or variable defined within a scriptlet cannot be used within a declaration.

Using Expressions

Expressions contain simple Java code that returns something that can be written to the client output, and expressions output the return variable of that code to the client. So, you could have an arithmetic calculation within an expression because that results in a numeric value that can be displayed. You could call some method that returns a `String` or number or other primitive because that results in a displayable returned value. Essentially, any code that can legally be the entire right side of an assignment statement can be placed within an expression. Expressions execute within the same method scope as scriptlets; that is, expressions get copied into the `_jspService` method just like scriptlets do.

Take a look at the following example code. It doesn't actually do anything useful, but it demonstrates the variety of things you can do within directives, declarations, scriptlets, and expressions.

```
<%@ page contentType="text/html; charset=UTF-8" language="java" %>
<%
    private final int five = 0;

    protected String cowboy = "rodeo";

    //The assignment below is not declarative and is a syntax error if uncommented
    //cowboy = "test";

    public long addFive(long number)
    {
        return number + 5L;
    }

    public class MyInnerClass
    {

    }
    MyInnerClass instanceVariable = new MyInnerClass();

    //WeirdClassWithinMethod is in method scope, so the declaration below is
    // a syntax error if uncommented
```

```

    //WeirdClassWithinMethod bad = new WeirdClassWithinMethod();
%>
<%
    class WeirdClassWithinMethod
    {
    }

    WeirdClassWithinMethod weirdClass = new WeirdClassWithinMethod();
    MyInnerClass innerClass = new MyInnerClass();
    int seven;
    seven = 7;
%>
<%= "Hello, World" %><br />
<%= addFive(12L) %>

```

Putting It All Together

Create a JSP file named `gibberish.jsp` in the web root of an empty project, and put the preceding gibberish code in there (or just use the JSP in the Hello-World-JSP project). Compile and run your application and go to `http://localhost:8080/hello-world/gibberish.jsp`. Obviously, this page isn't useful in the browser; the point that you should understand is in the source file. Go back into the Tomcat work directory and find the `gibberish_jsp.java` file. Examine how the code in your JSP got translated into Java code in the JSP Servlet class to gain a better understanding of the differing purposes of directives, declarations, scriptlets, and expressions.

Commenting Your Code

Like nearly every other language or markup in existence, JSP also has a method for commenting code. There are four different ways that you can comment code within a JSP:

- XML comments
- Traditional Java line comments
- Traditional Java block comments
- JSP comments.

The XML comment (also known as the HTML comment) is syntax you are most likely already familiar with:

```
<!-- This is an HTML/XML comment -->
```

This type of comment is passed through to the client because it is standard XML and HTML markup. The browser ignores it, but it appears in the source of the response. *More important, any JSP tags within this comment will still be evaluated.* This is essential to remember because commenting out code with this style of comment does not prevent Java code within from executing. To demonstrate this, consider the following example:

```
<!-- This is an HTML/XML comment: <%= someObject.dumpInfo() %> -->
```

If `someObject.dumpInfo()` returns "connections=5, errors=12, successes=3847," the response sent back to the client's browser will contain the following HTML comment in it:

```
<!-- This is an HTML/XML comment: connections=5, errors=12, successes=3847 -->
```

You can use any legal Java comment within declarations and scriptlets in JSPs. This includes, as mentioned previously, line comments and block comments. In the following example, all the code in bold is commented out and will not be evaluated:

```
<%
  String hello = "Hello, World!"; // this is a comment
  //long test = 12L;
  /*int i = 0;
  int j = 12;*/
  String goodbye = "Goodbye, World!";
%>
```

The new type of comment that you have not used yet is the JSP comment. The syntax of the JSP comment closely resembles an XML/HTML comment, with the only difference being the percent sign instead of the exclamation point at the beginning, and the percent sign at the end:

```
<%-- This is a JSP comment --%>
```

Just as with the XML/HTML comment, everything between the `<%--` and the `--%>` is considered commented. Not only is it not sent to the browser, it isn't even interpreted/translated by the JSP compiler. Whereas all three of the previously covered comment types appear in the JSP Servlet.java file, this last comment type does not. To the translator, it does not even exist. This is especially useful for commenting out some range of code that includes JSP scriptlets, expressions, declarations, directives, and markup that you do not want to be evaluated or sent to the browser.

Adding Imports to Your JSP

In Java when you use a class directly, you must either reference it using its fully qualified class name, or you must include an import statement at the top of the Java code file. The rules are the same in JSPs. Any time a JSP contains Java code that uses a class directly, it must either use the fully qualified class name or include an import directive in the JSP file. And just as every class in the `java.lang` package is imported implicitly in Java files, similarly every class in the `java.lang` package is implicitly imported in JSP files.

Importing Java classes in JSPs is different but just as easy as importing Java classes in a Java code file. Importing one or more classes is as simple as adding an import attribute to the page directive you learned about earlier:

```
<%@ page import="java.util.* , java.io.IOException" %>
```

In this example, you use a comma to separate multiple imports, and the result is that the `java.io.IOException` class and all the members of the `java.util` package are imported. Of course, you do not have to use a separate directive to import classes. You could combine this with the example seen earlier:

```
<%@ page contentType="text/html; charset=UTF-8" language="java"
  import="java.util.* , java.io.IOException" %>
```

You also don't have to combine multiple imports into a single directive using a comma separator. You could use multiple directives to accomplish this task:

```
<%@ page import="java.util.Map" %>
<%@ page import="java.util.List" %>
<%@ page import="java.io.IOException" %>
```

Something to consider when doing this is that every JSP tag that results in no output, and also every directive, declaration, and scriptlet, results in an empty line being output to the client. So, if you have many page directives for imports followed by various declarations and scriptlets, you could end up with dozens of blank lines in your output. To compensate for this, JSP developers often chain the end of one tag to the beginning of the next:

```
<%@ page import="java.util.Map"
%><%@ page import="java.util.List"
%><%@ page import="java.io.IOException" %>
```

This code example has the exact same logical outcome as the previous example, but it results in only one blank line at the top of the output instead of three. In the section “Combining Servlets and JSPs,” you will learn about a deployment descriptor setting that trims this white space entirely.

Using Directives

Earlier you were introduced to the directive, a JSP feature denoted with a beginning `<%@` and an ending `%>`. There are three different types of directives, which are discussed at this time.

Changing Page Properties

You have already explored some features of the `page` directive, such as the `contentType`, `language`, and `import` attributes. There are also many more features of the `page` directive. As explained earlier, the `page` directive provides you with some controls over how the JSP is translated, rendered, and transmitted back to the client. Here are some of the other attributes that may be included in this directive:

pageEncoding

Specifies the character encoding used by your JSP and is equivalent to `setCharacterEncoding` on `HttpServletResponse`. Instead of `contentType="text/html; charset=UTF-8"`, you could write `contentType="text/html" pageEncoding="UTF-8"`.

session

This must either be `true` or `false`, and indicates whether the JSP participates in HTTP sessions. By default it is `true`, giving you access to the implicit `session` variable in the JSP (covered in the section “Using Java within a JSP (and Why You Shouldn’t)”). If you set it to `false`, you cannot use the implicit `session` variable. If your application does not use sessions and you want to improve performance, setting this to `false` might be a good idea. You learn more about HTTP sessions in Chapter 5.

isELIgnored

This attribute specifies whether expression language (EL) is parsed and translated for this JSP. You learn more about EL in Chapter 6. Prior to the JSP 2.0 specification, the default value was `true`, meaning you had to set it to `false` for every JSP in which you wanted to use EL. As of JSP 2.0 (you use JSP 2.3 in this book) the default value is `false`, so you should never need to worry about this setting.

buffer and autoFlush

These attributes are closely related, and their defaults are “8kb” and `true`, respectively. They control whether the output of the JSP is sent immediately to the browser as it is generated or

buffered and sent in batches. The `buffer` attribute specifies the size of the JSP buffer or "none" (the output will not be buffered), whereas `autoFlush` indicates whether the buffer will be flushed automatically after it reaches its size limit. If `buffer` is set to "none" and `autoFlush` is set to `false`, an exception occurs when the JSP is translated to Java. If `autoFlush` is set to `false` and the buffer becomes full, an exception occurs. This is a handy way to ensure that the content a JSP generates does not exceed a certain length.

With `autoFlush` set to `true` (the default), the smaller your buffer, the more often data will be flushed to the client, and the larger the buffer, the less often data will be flushed to the client. Disabling the buffer entirely with `buffer="none"` can improve the performance of your JSPs because it decreases memory consumption and CPU overhead. However, this is not without its setbacks. Using no buffer can result in sending more packets to the browser, which can increase bandwidth consumption marginally. Also, when the first character of the response begins flowing to the client, the HTTP response headers must be committed and sent before the response. Because of this, you cannot set response headers (`response.setHeader(...)`) or forward the JSP (`<jsp:forward />`) after the buffer has flushed, and you cannot set response headers or forward the JSP *at all* in a JSP where the buffer has been disabled. This may be an acceptable sacrifice to improve server-side performance in certain circumstances.

errorPage

If an error occurs during the execution of the JSP, this attribute instructs the container what JSP to forward the request to.

isErrorPage

This attribute indicates that this JSP is serving as an error page (by default, it is `false`). If set to `true`, this enables the implicit `exception` variable on the page. You would do this on JSPs that you forward to when errors occur, or that you have defined in the container as error-handling JSPs.

isThreadSafe

`true` by default, this tells the container that the JSP can safely serve multiple requests simultaneously. If changed to `false`, the container only serves requests to this JSP one-by-one. A good rule of thumb is to *never* change this. Remember, "If your JSP isn't thread safe, you're doing it wrong."

extends

This attribute specifies which class your JSP Servlet should inherit from. Using this is not portable from one web container to another, and it should never be necessary. Just don't do it.

Other Attributes

In most of your JSPs, `contentType` (and optionally `pageEncoding`) are the only attributes of the `page` directive that you will ever change from the default values. The `session` and `isErrorPage` attributes are probably the two most common of the other attributes. Occasionally, you may need to disable buffering. With each JSP, you should evaluate your options and decide which attributes should be changed to suit your application's needs.

Including Other JSPs

Including other JSPs in a JSP is easy, but there are some interesting rules and options to keep in mind. The first tool that you can use to include another JSP in your JSP is the `include` directive. It is straightforward:

```
<%@ include file="/path/to/some/file.jsp" %>
```

The `file` attribute provides the container with the path to the JSP file that should be included. If it is absolute, the path resolves from the web root of the application, so a file named `included.jsp` in the `WEB-INF` directory could be included with path `/WEB-INF/included.jsp`. If the path is relative, it resolves from the same directory the including JSP exists in. The `include` directive is evaluated at translation time. Before the JSP is translated to Java, the `include` directive is replaced (virtually) with the contents of the included JSP file. After this happens, the combined contents are then translated to Java and compiled. Thus, as you can see, this process is static and only occurs once.

To demonstrate this, follow these steps:

1. Create a JSP called `includer.jsp` in the web root of your Hello-World-JSP project and place the following line of code in it (deleting any code your IDE generated). Alternatively, just use the Hello-World-JSP project.

```
<%@ include file="index.jsp" %>
```

2. Compile and debug your application and navigate to `http://localhost:8080/hello-world/includer.jsp` in your favorite browser. You should see the familiar page, which means your include has worked.
3. Now go into the Tomcat work directory and open the `includer_jsp.java` file that Tomcat created. You should immediately notice that, other than the class name, it is identical to `index.jsp.java`. This is because the JSP was included statically at translation time.

There is a different way to include other JSPs that results in a dynamic (run time) inclusion instead of a static (translation time) inclusion. You use the `<jsp:include>` tag to achieve this:

```
<jsp:include page="/path/to/some/page.jsp" />
```

The `<jsp:include>` tag doesn't have a `file` attribute; it has a `page` attribute. The path is still relative to the current file or absolute from the web root, just like with the `include` directive. But it is not included at translation time. Instead, the included file is compiled separately. At run time, the request is temporarily forwarded to the included JSP, the resulting output of that JSP is written to the response, and then the control returns back to the including JSP. This can easily be seen by creating a file named `dynamicIncluder.jsp` in your project's web root with the following line of code (or use the Hello-World-JSP project):

```
<jsp:include page="index.jsp" />
```

Compile and debug again and navigate to `http://localhost:8080/hello-world/dynamicIncluder.jsp`, then open the `dynamicIncluder_jsp.java` file that Tomcat created. You can see now that the content of this Java file is quite different. The most interesting line in the file is:

```
org.apache.jasper.runtime.JspRuntimeLibrary.include(request, response, "index.jsp",
out, false);
```

This sends the request and response down into another method, which runs the included JSP, writes its contents to the response, and returns.

Both of these methods of inclusion have their strengths and weaknesses. The `include` directive is fast because it is evaluated only once, and all variables defined in the including JSP are in scope and can be referenced by the included JSP. But this method makes your JSP (and the `_jspService` method, as a result) longer, which is important to keep in mind because the bytecode of compiled Java methods can't be longer than 65,534 bytes. The `<jsp:include>` tag does not cause this problem, but it also does not perform as well because it must be evaluated every page load, and variables defined in the including JSP are out of scope and cannot be used in the included JSP. Ultimately, you must decide which is appropriate each time you need to include a file, but in most cases, the `include` directive is a good choice.

NOTE *By default, web containers translate and compile files ending in `.jsp` and `.jspx` (which you learn about later) as JSPs. You may have also seen the extension `.jspf`. JSPF files are generally called JSP Fragments and are not compiled by the web container. Although there are no hard-and-fast rules governing JSPF files (you can technically configure most web containers to compile them if you want), there are some agreed-upon best practices. JSPF files represent fragments of JSPs that cannot stand alone and should always be included, not accessed directly. This is why web containers do not normally compile them. Actually, in many cases a JSPF file references variables that can exist only if it is included in another JSP file. For this reason, JSPF files should be included only using the `include` directive because variables defined in the including JSP must be in scope in the included JSP.*

Including Tag Libraries

Chapters 7 and 8 talk more about tag libraries, but they are mentioned now because of how they are included. You use the `taglib` directive to reference a tag library so that you can use the tags defined by that tag library in your JSP. Like the `include` directive, the `taglib` directive is quite simple:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
```

The `uri` attribute specifies the URI namespace that the tag library is defined under, and the `prefix` attribute defines the alias with which you reference the tags in that library. You learn more about what this means in Chapter 7.

Using the `<jsp>` Tag

All JSPs support a special kind of tag with an XMLNS prefix of `jsp`. This tag has many uses and features. Most of the features are used in JSP Documents (XML versions of JSPs that you learn about in the last section of this chapter) or relics from older versions of JSP in which some things were much harder to do than they are now (and so are not covered here). However, you should learn about a few useful features of this tag.

You have already learned about `<jsp:include>` and how it differs from the `include` directive. A similar tag is the `<jsp:forward>` tag. This enables you to forward a request from the JSP it is currently executing in to some other JSP. Unlike a `<jsp:include>`, the request does not return to the original JSP. This is not a redirect; the client's browser does not see the change. Also, anything the JSP writes to the response stays in the response when the forward occurs; it is not erased, like it would be with a redirect. Using the `<jsp:forward>` tag is simple:

```
<jsp:forward page="/some/other/page.jsp" />
```

In this example the request is internally forwarded to `/some/other/page.jsp`. Any response content generated before the tag still goes to the client's browser. Any code that comes after the tag is ignored and not evaluated. This is how this tag differs from the `<jsp:include>` tag. If the code after the `<jsp:forward>` tag were not ignored, this tag would behave just like the `<jsp:include>` tag.

Three other related tags are `<jsp:useBean>`, `<jsp:getProperty>`, and `<jsp:setProperty>`. The `<jsp:useBean>` tag declares the presence of a JavaBean on the page, whereas `<jsp:getProperty>` retrieves properties (using getter methods) from beans declared with `<jsp:useBean>`. Similarly, `<jsp:setProperty>` sets properties (using setter methods). A Java bean in this case is any instantiated object. `<jsp:useBean>` instantiates a class to create a bean, and this bean can then be accessed using the other two bean tags, custom tags, and JSP scriptlets and expressions. The advantage to declaring a bean in this way is that it makes the bean available to other JSP tags; if you simply declared the bean in a scriptlet, it would only be available to scriptlets and expressions.

Finally, there is the `<jsp:plugin>` tag, which is a handy tool for embedding Java Applets in the rendered HTML. This tag removes the risk of messing up the careful structure of `<object>` and `<embed>` tags necessary to get Java Applets to work in all browsers. It handles creating these HTML tags for you so that the Applet should work in all mainstream browsers that support the Java plug-in. Here is an example of using the `<jsp:plugin>` tag:

```
<jsp:plugin type="applet" code="MyApplet.class" jreversion="1.8">
  <jsp:params>
    <jsp:param name="appletParam1" value="paramValue1"/>
  </jsp:params>
  <jsp:fallback>
    The browser you are using does not support Java Applets. You might
    consider switching browsers.
  </jsp:fallback>
</jsp:plugin>
```

Note that `<jsp:plugin>` can also contain standard `object/embed` HTML attributes such as `name`, `align`, `height`, `width`, `hspace`, and `vspace`. These attributes are copied to the HTML markup.

NOTE Java Applets are a completely different subject from web applications and are outside the scope of this book. If you want to learn more about Java Applets, most beginner Java books cover the topic.

USING JAVA WITHIN A JSP (AND WHY YOU SHOULDN'T!)

In this section you explore using Java within a JSP a little more by replacing the Servlet in the Hello-User project (from the previous chapter) with just a JSP. Then you briefly consider why using Java in a JSP is discouraged (and why there's actually a deployment descriptor setting to disable it). For the rest of this section you use the Hello-User-JSP project on the wrox.com download site.

Using the Implicit Variables in a JSP

JSP files have several *implicit variables* (objects) available for use within scriptlets and expressions in the JSP. They are considered implicit because you do not have to define or declare them anywhere in your code. The JSP specification requires that the translator and compiler of the JSP provide these variables, with the exact names specified. The variables have *method scope*. They are defined at the beginning of the Servlet method that the JSP executes in (in Tomcat 8.0, the `_jspService` method). This means you cannot use them within any code you place inside JSP declarations. Declarations have *class scope*. Because the implicit variables are in scope only within the method that the JSP executes in, code inside declarations cannot use them. You can see an example of how the implicit variables are defined by looking at the `_jspService` method of any of the previously compiled JSPs you examined in the last section:

```
public void _jspService(final javax.servlet.http.HttpServletRequest request,
                      final javax.servlet.http.HttpServletResponse response)
    throws java.io.IOException, javax.servlet.ServletException
{
    final javax.servlet.jsp.PageContext pageContext;
    javax.servlet.http.HttpSession session = null;
    final javax.servlet.ServletContext application;
    final javax.servlet.ServletConfig config;
    javax.servlet.jsp.JspWriter out = null;
    final java.lang.Object page = this;
    javax.servlet.jsp.JspWriter _jspx_out = null;
    javax.servlet.jsp.PageContext _jspx_page_context = null;

    try {
        response.setContentType("text/html;charset=UTF-8");
        pageContext = _jspxFactory.getPageContext(this, request, response,
            null, true, 8192, true);
        _jspx_page_context = pageContext;
        application = pageContext.getServletContext();
        config = pageContext.getServletConfig();
        session = pageContext.getSession();
        out = pageContext.getOut();
        _jspx_out = out;
        ...
    }
    ...
}
```

The code isn't exactly the picture of clean code, but the important parts of the code are bold so that you can understand what's going on. The bold code emphasizes the declaration or assignment (or both) of implicit variables required by the JSP specification. Variables that are not bold (such as `_jspx_out` or `_jspx_page_context`) are Tomcat-specific variables that are not guaranteed to

exist and should never be used in your JSP. Eight implicit variables are in this code, but the JSP specification defines nine implicit variables. Now take a look at each of these implicit variables, and then you'll understand why one is missing.

request and response

The `request` variable is an instance of `HttpServletRequest` and the `response` variable is an instance of `HttpServletResponse`, both of which you learned about in detail in Chapter 3.

Anything you can do with a request in a Servlet you can also do in a JSP, including getting request parameters, getting and setting attributes, and even reading from the response body. The same rules you learned about in the last chapter apply here. However, there are some restrictions on what you can do with the response object in a JSP. These restrictions are not contract restrictions, so they are not enforced at compile time. Instead, they are enforced at run time because violating them could cause unexpected behavior or even errors. For example, you should not call `getWriter` or `getOutputStream` because the JSP is already writing to the response output. You also should not set the content type or character encoding, flush or reset the buffer, or change the buffer size. These are all things that the JSP does, and if your code does them, too, it can cause problems.

session

This variable is an instance of `HttpSession`. You learn more about sessions in the next chapter. Remember from the previous section that the `page` directive has a `session` attribute that defaults to `true`. This is why the `session` variable is available in the previous code example and will be available by default in all of your JSPs. If you set the `page` directive's `session` attribute to `false`, the `session` variable in the JSP is not defined and cannot be used.

out

The `JspWriter` instance `out` is available for you to use in all your JSPs. It is a `Writer`, just like what you get from calling the `getWriter` method on `HttpServletResponse`. If for some reason you need to write directly to the response, you should use the `out` variable. However, in most cases you can simply use an expression or write text or HTML content in the JSP.

application

This is an instance of the `ServletContext` interface. Recall from Chapter 3 that this interface gives you access to the configuration of the web application as a whole, including all the context init parameters. Why this variable was named `application` instead of `context` or `servletContext` is a mystery.

config

The `config` variable is an instance of the `ServletConfig` interface. Unlike the `application` variable, its name actually reflects what it is. As you learned in Chapter 3, you can use this object to access the configuration of the JSP Servlet, such as the Servlet init parameters.

pageContext

This object, an instance of the `PageContext` class, provides several convenience methods for getting request attributes and session attributes, accessing the request and response, including other files,

and forwarding the request. You will probably never need to use this class within a JSP. It will, however, come in handy when you write custom JSP tags in Chapter 8.

page

The `page` variable is an interesting object to examine. It is an instance of `java.lang.Object`, which initially makes it seem unuseful. However, it essentially is the `this` variable from the JSP Servlet object. So, you could cast it to `Servlet` and use methods defined on the `Servlet` interface. It is also a `javax.servlet.jsp.JspPage` (which extends `Servlet`) and a `javax.servlet.jsp.HttpJspPage` (which extends `JspPage`), so you could cast it to either of those and use methods defined on those interfaces. In reality, you will probably never have a reason to use this variable. It may be useful if other JSP scripting languages are ever supported. However, the JSP 2.3 specification, section 1.8.3 note "a," says that `page` is always a synonym for `this` when the scripting language is Java. Thus, anything you can do with `page` (such as get the Servlet name or access methods or instance variables you defined in a JSP declaration) you can also do with `this`.

exception

This is the variable that was missing from the previous code example. Recall from the previous section that you can specify as `true` the `isErrorPage` attribute on the `page` directive to indicate that the JSP's purpose is to handle errors. Doing so makes the `exception` variable available for use within the JSP. Because the default value for `isErrorPage` is `false` and you have not used it anywhere, the `exception` variable has not been defined in any JSPs you created. If you create a JSP with `isErrorPage` set to `true`, the implicit `exception` variable, a `Throwable`, is defined automatically.

NOTE You can read the JavaServer Pages 2.3 specification document on the JSP specification page.

Trying Out the Implicit Variables

Now that you understand the available implicit variables and their purposes, you should explore this more by writing some JSP code that uses the implicit variables. In your project, create a `greeting.jsp` file in the web root, and place the following code in it (or just use the Hello-User-JSP project):

```
<%@ page contentType="text/html; charset=UTF-8" language="java" %>
<%!
    private static final String DEFAULT_USER = "Guest";
%>
<%
    String user = request.getParameter("user");
    if(user == null)
        user = DEFAULT_USER;
%>
<!DOCTYPE html>
<html>
<head>
```

```

<title>Hello User Application</title>
</head>
<body>
    Hello, <%= user %>!<br /><br />
    <form action="greeting.jsp" method="POST">
        Enter your name:<br />
        <input type="text" name="user" /><br />
        <input type="submit" value="Submit" />
    </form>
</body>
</html>

```

Compare this to the code you wrote in `HelloServlet.java` for the Hello-User project in the previous chapter. There's much less to it, but it accomplishes the same thing. Notice the use of a declaration to define the `DEFAULT_USER` variable, a scriptlet to look for the `user` request parameter and default it if it is not set, and an *expression* to output the value of the `user` variable. Now compile and debug this code and go to `http://localhost:8080/hello-world/greeting.jsp` in your browser. Try entering a name in the input field and clicking the Submit button — the post variable is detected and used. Now try going to `http://localhost:8080/hello-world/greeting.jsp?user>Allison`, and you should see that the query parameter is also detected and used. You are encouraged to explore the Java code that Tomcat translated your JSP into.

Another thing you did in the Hello-User project was create a Servlet to demonstrate using multiple-value parameters. This, too, can be replicated using JSPs. Create a file in your project web root named `checkboxes.jsp` (or use the Hello-User-JSP project):

```

<%@ page contentType="text/html; charset=UTF-8" language="java" %>
<!DOCTYPE html>
<html>
    <head>
        <title>Hello User Application</title>
    </head>
    <body>
        <form action="checkboxesSubmit.jsp" method="POST">
            Select the fruits you like to eat:<br />
            <input type="checkbox" name="fruit" value="Banana" /> Banana<br />
            <input type="checkbox" name="fruit" value="Apple" /> Apple<br />
            <input type="checkbox" name="fruit" value="Orange" /> Orange<br />
            <input type="checkbox" name="fruit" value="Guava" /> Guava<br />
            <input type="checkbox" name="fruit" value="Kiwi" /> Kiwi<br />
            <input type="submit" value="Submit" />
        </form>
    </body>
</html>

```

This file replicates the output of the `doGet` method in the `MultiValueParameterServlet.java` file from the Hello-User project. Next, create `checkboxesSubmit.jsp` (also in the Hello-User-JSP project):

```

<%@ page contentType="text/html; charset=UTF-8" language="java" %>
<%
    String[] fruits = request.getParameterValues("fruit");
%>
<!DOCTYPE html>
<html>

```

```
<head>
    <title>Hello User Application</title>
</head>
<body>
    <h2>Your Selections</h2>
    <%
        if(fruits == null)
        {
            %>You did not select any fruits.<%
        }
        else
        {
            %><ul><%
                for(String fruit : fruits)
                {
                    out.println("<li>" + fruit + "</li>");
                }
            %></ul><%
        }
    %>
</body>
</html>
```

This file replicates the logic and output of the `doPost` method from the `MultiValueParameterServlet` class. Notice how the bold code jumps in and out of scriptlets, using Java only where the logic requirements demand and leaving the scriptlets to use straight output instead of writing with the implicit `out` variable. The exception is inside the `for` loop, which demonstrates one use case for the `out` variable. This could have just as easily been replaced with `%><%= fruit %><%` to accomplish the same thing. Now compile and debug the project and go to `http://localhost:8080/hello-world/checkboxes.jsp` in your browser. You should see a page like that in Figure 4-1. Experiment with different combinations of the check boxes, and verify that it behaves identically to the Hello-User project in Chapter 3. Try replacing the use of `out` in the `for` loop with `%><%= fruit %><%`. When you recompile and run the project again, the output should not change.

Finally, create a file named `contextParameters.jsp` to explore the use of the `application` implicit variable and the retrieval of context init parameters. Alternatively, use the file already in the Hello-User-JSP project.

```
<%@ page contentType="text/html; charset=UTF-8" language="java" %>
<!DOCTYPE html>
<html>
    <head>
        <title>Hello User Application</title>
    </head>
    <body>
        settingOne: <%= application.getInitParameter("settingOne") %>,
        settingTwo: <%= application.getInitParameter("settingTwo") %>
    </body>
</html>
```

Also, you need to have some context init parameters defined in your deployment descriptor, just like in Chapter 3:

```
<context-param>
    <param-name>settingOne</param-name>
    <param-value>foo</param-value>
</context-param>
<context-param>
    <param-name>settingTwo</param-name>
    <param-value>bar</param-value>
</context-param>
```

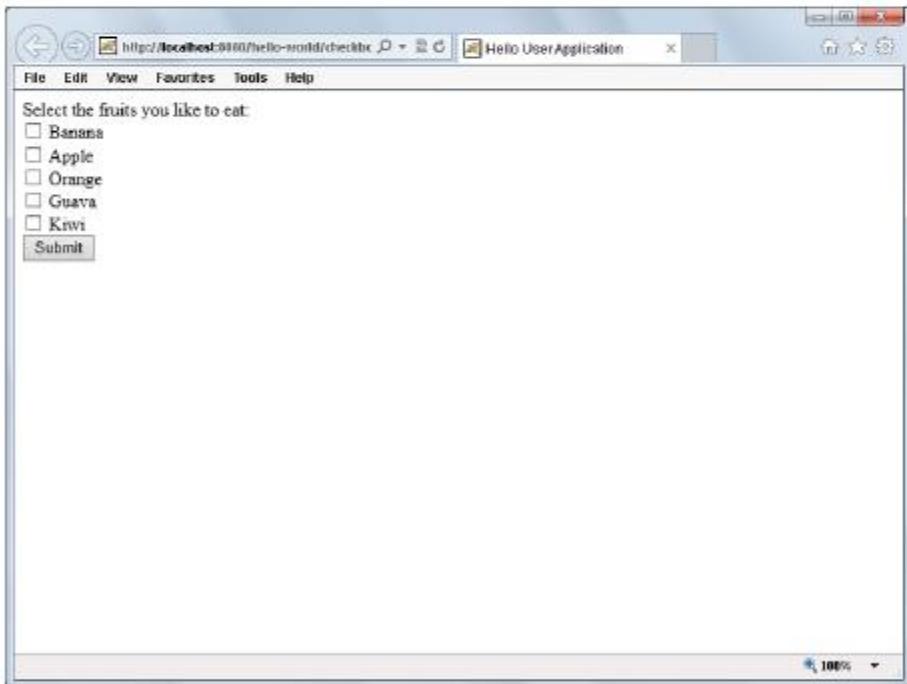


FIGURE 4-1

Now compile, debug, and navigate to `http://localhost:8080/hello-world/contextParameters.jsp`. As with the Servlet-based Hello-User project, you should see the values of the context init parameters.

Why You Shouldn't Use Java in a JSP

There are plenty of advantages to using Java within a JSP, and in addition to the uses previously pointed out so far in this chapter, you are likely thinking of other uses as you read this paragraph. The coolest thing about using Java in a JSP is that almost anything you can do in a normal Java class you can do in a JSP. However, one of the biggest dangers of using Java in a JSP is that almost anything you can do in a normal Java class you can do in a JSP. These sentences might sound crazy, but it's true. Think about all the things you can do in Java code. Here are a few to help you out.

You could connect to, query, and manipulate a relational database (or NoSQL database, as the case may be). You could also access and write to files on the server file system. You could connect to remote servers, perform REST web service transactions, and interact with system peripherals. You

could even do some number crunching, sort a binary tree with one billion nodes, traverse a large data set looking for suspicious data, or search a Document Object Model for a particular set of nodes. Now raise your hand if you think any of these things are good ideas in a JSP.

Java is a powerful language, and the problem with having all that power at your fingertips is that it's so hard not to use it. Depending on the application, any one of those tasks might be tasks you need to perform within a web application. But consider this: In a cleanly structured application, would it be appropriate to put all the database access, file manipulation, and number crunching code in a single class? Probably not. Most likely, you would have several classes that performed specialized functions and then use those classes wherever needed. JavaServer Pages is a technology that was designed for the *presentation layer*, also known as the *view*. Although it's possible to mix database access with the presentation layer, or to mix number crunching with the presentation layer, it is not a good idea. Functional languages, scripting languages, and other languages that execute from the top of a file to the bottom of a file, such as PHP, certainly have their uses. But it's likely you didn't pick Java as your platform of choice so that you could make pages written in this manner. Chances are you picked Java for its elegance, strong typing, and strict object-oriented structure, among other reasons.

Additionally, in most organizations, user interface developers are responsible for creating the presentation layer. These developers rarely have experience writing Java code, and providing them with that ability can be dangerous. Instead, it often makes sense to provide them with a less-powerful set of tools to work with.

In a well-structured, cleanly coded application, the presentation layer is separated from the business logic, which is likewise separated from the data persistence layer. It's actually possible to create JSPs that display dynamic content without a single line of Java inside the JSP. This enables application developers to concentrate on the business and data logic while user interface developers work on the JSPs. You may wonder how this is possible, but you will not be disappointed. You learn the first step in the next section, and explore even more powerful JSP technologies in Chapters 6, 7, and 8.

COMBINING SERVLETS AND JSPS

For the rest of this chapter you improve the customer support application you began working on in Chapter 3. You can follow along with the examples and find the entire source code in the Customer-Support-v2 project on the wrox.com download site. When dealing with complex logic, data validation, data persistence, and a detailed presentation layer, it makes the most sense to use a combination of Servlets and JSPs instead of using exclusively one or the other. In this section, you separate the business logic of customer support from the presentation layer.

Configuring JSP Properties in the Deployment Descriptor

Earlier in the chapter you learned about the `page` directive and the many attributes it provides to enable you to customize how your JSP is translated, compiled, and processed. If you have many JSPs with similar properties, however, it can be cumbersome to place this `page` directive at the top of every JSP file. Fortunately, there is a way to configure common JSP properties within the deployment descriptor. In the `web.xml` file, which should be empty except for the `<display-name>`, add the following contents:

```

<jsp-config>
  <jsp-property-group>
    <url-pattern>*.jsp</url-pattern>
    <url-pattern>*.jspx</url-pattern>
    <page-encoding>UTF-8</page-encoding>
    <scripting-invalid>false</scripting-invalid>
    <include-prelude>/WEB-INF/jsp/base.jspf</include-prelude>
    <trim-directive-whitespaces>true</trim-directive-whitespaces>
    <default-content-type>text/html</default-content-type>
  </jsp-property-group>
</jsp-config>

```

Understanding JSP Property Groups

The `<jsp-config>` tag contains any number of `<jsp-property-group>` tags. These property groups are used to differentiate properties for different groups of JSPs. For example, you may want to define one set of common properties for all JSPs in the `/WEB-INF/jsp/admin` folder and a different set of common properties for all the JSPs in the `/WEB-INF/jsp/help` folder.

You differentiate these property groups by defining distinct `<url-pattern>` tags for each `<jsp-property-group>`. In the previous code example, the `<url-pattern>` tags indicate that this property group applies to all files ending in `.jsp` and `.jspx`, anywhere in the web application. If you want to treat JSPs in one folder differently from JSPs in another in the fashion mentioned just earlier, you could have two (or more) `<jsp-property-group>` tags, with one having `<url-pattern>/WEB-INF/jsp/admin/*.jsp</url-pattern>` and the other having `<url-pattern>/WEB-INF/jsp/help/*.jsp</url-pattern>`.

Consider some important rules when dealing with the `<url-pattern>` tag:

- If some file in your applications matches a `<url-pattern>` in both a `<servlet-mapping>` and a JSP property group, whichever match is more specific wins. For example, if one matching `<url-pattern>` were `*.jsp` and the other were `/WEB-INF/jsp/admin/*.jsp`, the one with `/WEB-INF/jsp/admin/*.jsp` would win. If the `<url-pattern>` tags are identical, the JSP property group wins over the Servlet mapping.
- If some file matches a `<url-pattern>` in more than one JSP property group, the more specific match wins. If two or more most-specific matches are identical, the first matching JSP property group in the order it appears in the deployment descriptor wins.
- If some file matches a `<url-pattern>` in more than one JSP property group and more than one of those property groups contains `<include-prelude>` or `<include-coda>` rules, the include rules from *all* the JSP property groups are applied for that file, even though only one of the property groups is used for the other properties.

To understand that last bullet point, consider the following hypothetical property groups:

```

<jsp-property-group>
  <url-pattern>*.jsp</url-pattern>
  <url-pattern>*.jspx</url-pattern>
  <page-encoding>UTF-8</page-encoding>
  <include-prelude>/WEB-INF/jsp/base.jspf</include-prelude>
</jsp-property-group>
<jsp-property-group>
  <url-pattern>/WEB-INF/jsp/admin/*.jsp</url-pattern>
  <url-pattern>/WEB-INF/jsp/admin/*jspx</url-pattern>

```

```
<page-encoding>ISO-8859-1</page-encoding>
<include-prelude>/WEB-INF/jsp/admin/include.jspf</include-prelude>
</jsp-property-group>
```

A file named `/WEB-INF/jsp/user.jsp` would match only the first property group. It would have a character encoding of UTF-8 and the `/WEB-INF/jsp/base.jspf` file would be included at the beginning. On the other hand, `/WEB-INF/jsp/admin/user.jsp` would match both property groups. Because the second property group is a more specific match, this file would have a character encoding of ISO-8859-1. However, both `/WEB-INF/jsp/base.jspf` and `/WEB-INF/jsp/admin/include.jspf` would be included at the beginning of this file. This can get very confusing, so you are urged to keep your JSP property groups as simple as possible.

Using JSP Properties

The `<include-prelude>` tag in the Customer Support project's deployment descriptor tells the container to include the `/WEB-INF/jsp/base.jspf` file *at the beginning* of every JSP that belongs in this property group. This is useful for defining common variables, tag library declarations, or other resources that should be made available to all JSPs in the group. Similarly, an `<include-coda>` tag defines a file to be included *at the end* of every JSP in the group. You can use both of these tags more than once in a single JSP group. You might, for example, create `header.jspf` and `footer.jspf` files to include at the beginning and end, respectively, of every JSP. These files could contain header and footer HTML content to work as a sort of template for your application. Of course, you must take care when doing this, because you could easily include these files in places you don't intend.

The `<page-encoding>` tag is identical to the `pageEncoding` attribute of the `page` directive. Because JSPs already have a content type of `text/html` by default, you could simply specify a `<page-encoding>` of UTF-8 to change the content type character encoding of your JSPs from `text/html; ISO-8859-1` to `text/html; UTF-8`. You could also use the `<default-content-type>` tag to override `text/html` with some other default content type.

A particularly useful property is `<trim-directive-whitespaces>`. This property instructs the JSP translator to remove from the response output any white space only text created by directives, declarations, scriptlets, and other JSP tags. Earlier in this chapter you learned how to chain the end of one directive to the beginning of the next to prevent extra new lines from appearing in the response. This tag takes care of that for you so that you can write cleaner code.

Also mentioned earlier was the possibility to use the deployment descriptor to completely disable Java within JSPs. The `<scripting-invalid>` tag serves that purpose. The default value and value in your code, `false`, permits Java in all JSPs in the group. Later in the book you change this value to `true`. Once `true`, using Java within a matching JSP results in a translation error. The `<el-ignored>` tag is similar and corresponds to the `isELIgnored` attribute of the `page` directive. If `true`, expression language is prohibited in the group's JSPs (resulting in a translation error if EL is used). This defaults to `false` (allow expression language), and you can leave it that.

There are a handful of other JSP property group tags that you will probably never use. `<is-xmle>` indicates that matching JSPs are JSP documents (which you learn about in the next section). The `<deferred-syntax-allowed-as-literal>` tag is an expression language feature you learn about in Chapter 6. `<buffer>` corresponds to the `buffer` attribute of the `page` directive that you learned about earlier in the chapter. Finally, `<error-on-undeclared-namespace>` indicates whether an error is raised if a tag with unknown namespace is used within a matching JSP, and defaults to `false`.

Except for `<url-pattern>`, all of the tags within `<jsp-property-group>` are optional, but they must appear in the following order, with unused tags omitted: `<url-pattern>`, `<el-ignored>`, `<page-encoding>`, `<scripting-invalid>`, `<is-xml>`, `<include-prelude>`, `<include-coda>`, `<deferred-syntax-allowed-as-literal>`, `<trim-directive-whitespace>`, `<default-content-type>`, `<buffer>`, `<error-on-undeclared-namespace>`.

In the Customer Support project you have included `/WEB-INF/jsp/base.jspf` in all JSPs in the application. (The web container is smart enough not to apply this include rule to `base.jspf` itself.) Its contents are simple:

```
<%@ page import="com.wrox.TicketServlet, com.wrox.Attachment" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
```

This accomplishes two things: It imports these classes for all JSPs and declares the JSTL core tag library with an XMLNS prefix of `c`. You learn more about the JSTL in Chapter 7. You may wonder why this file is placed in the `/WEB-INF/jsp` directory instead of in the web root. Remember that files within the `WEB-INF` directory are protected from web access. Placing the JSP file in this directory prevents users from accessing the JSP from their browser. You would want to do this for any JSP that you do not want browsers to access directly, such as JSPs that rely on session and request attributes provided by a forwarding Servlet and JSPs that are only included.

The last thing you should look at before moving on is the `index.jsp` file in the web root of the Customer Support project. This is a web application *directory index file*, and its existence in the web root means it can respond to requests for the deployed application root (`/`) without being directly identified in the URL. It has two simple lines of code in it:

```
<%@ page session="false" %>
<c:redirect url="/tickets" />
```

The second line of code redirects the user to the `/tickets` Servlet URL relative to the deployed application. The first line of code disables sessions in the `index.jsp` file to prevent the unnecessary `JSESSIONID` parameter from being automatically appended to the redirect URL (which happens when a session is created and the client is redirected in the same request).

Forwarding a Request from a Servlet to a JSP

A typical pattern when combining Servlets and JSPs is to have the Servlet accept the request, do any business logic processing and data storage or retrieval necessary, prepare a model that can easily be used in a JSP, and then forward the request to the JSP. The methods in the Customer Support application's `TicketServlet` need a few changes to make this happen. You can apply these changes yourself or just view them in the project you downloaded.

Using the Request Dispatcher

You should first address the `showTicketForm` method because it is the simplest to change. You need to change its signature to also accept an `HttpServletRequest` and then replace the entire contents with a simple forward to the JSP:

```
private void showTicketForm(HttpServletRequest request,
                           HttpServletResponse response)
                           throws ServletException, IOException
{
```

```
        request.getRequestDispatcher("/WEB-INF/jsp/view/ticketForm.jsp")
            .forward(request, response);
    }
```

The new code for this method introduces you to a new feature of the `HttpServletRequest`. The `getRequestDispatcher` method obtains a `javax.servlet.RequestDispatcher`, which handles internal forwards and includes for a specific path (in this case `/WEB-INF/jsp/view/ticketForm.jsp`). With this object, you can forward the current request to that JSP by calling the `forward` method. Note that this is not a redirect. The user's browser does not receive a redirect status code, and the browser URL bar does not change. Instead, the internal request handling is forwarded to a different part of the application. After you call `forward`, your Servlet code should never manipulate the response again. Doing so could result in errors or erratic behavior. Now create the JSP file that this method forwards to (or view it in the project you downloaded):

```
<%@ page session="false" %>
<!DOCTYPE html>
<html>
    <head>
        <title>Customer Support</title>
    </head>
    <body>
        <h2>Create a Ticket</h2>
        <form method="POST" action="tickets" enctype="multipart/form-data">
            <input type="hidden" name="action" value="create"/>
            Your Name<br/>
            <input type="text" name="customerName"><br/><br/>
            Subject<br/>
            <input type="text" name="subject"><br/><br/>
            Body<br/>
            <textarea name="body" rows="5" cols="30"></textarea><br/><br/>
            <b>Attachments</b><br/>
            <input type="file" name="file1"/><br/><br/>
            <input type="submit" value="Submit"/>
        </form>
    </body>
</html>
```

Designing for the Presentation Layer

This isn't an impressive example because all you've done is copy some code from Java to JSP — not a new thing at this point. You are not using sessions yet, so that has been disabled in the JSP. You should next change the `TicketServlet`'s `viewTicket` method, which is more complicated. A good approach to take is to think of your presentation, first — what data elements does it need to work? — and then code your Servlet method to provide that information. With this in mind, start with the `/WEB-INF/jsp/view/viewTicket.jsp` file:

```
<%@ page session="false" %>
<%
    String ticketId = (String)request.getAttribute("ticketId");
    Ticket ticket = (Ticket)request.getAttribute("ticket");
%>
<!DOCTYPE html>
<html>
    <head>
```

```

<title>Customer Support</title>
</head>
<body>
<h2>Ticket #<%= ticketId %>: <%= ticket.getSubject() %></h2>
<i>Customer Name - <%= ticket.getCustomerName() %></i><br />
<%= ticket.getBody() %><br /><br />
<%
    if(ticket.getNumberOfAttachments() > 0)
    {
        %>Attachments: <%
        int i = 0;
        for(Attachment a : ticket.getAttachments())
        {
            if(i++ > 0)
                out.print(", ");
            %><a href=<c:url value="/tickets">
                <c:param name="action" value="download" />
                <c:param name="ticketId" value=<%= ticketId %>" />
                <c:param name="attachment" value=<%= a.getName() %>" />
            </c:url>><%= a.getName() %></a><%
        }
    }
    %>
    <a href=<c:url value="/tickets" />>Return to list tickets</a>
</body>
</html>

```

Creating this JSP should show you that the presentation layer needs a *ticketId* and a *ticket* to display correctly (the code in bold). The `viewTicket` method can be changed to provide these variables and forward the request to the JSP:

```

private void viewTicket(HttpServletRequest request,
                       HttpServletResponse response)
throws ServletException, IOException
{
    String idString = request.getParameter("ticketId");
    Ticket ticket = this.getTicket(idString, response);
    if(ticket == null)
        return;

    request.setAttribute("ticketId", idString);
    request.setAttribute("ticket", ticket);

    request.getRequestDispatcher("/WEB-INF/jsp/view/viewTicket.jsp")
        .forward(request, response);
}

```

The first few lines of the method perform the business logic of parsing the request parameter and getting the ticket from the database. Then the code in bold adds two attributes to the request. This is the primary purpose of *request attributes*. They can be used to pass data between different elements of the application that are handling the same request, such as between a Servlet and a JSP. Request attributes are different from request parameters: Request attributes are *Objects* while request parameters are *Strings*, and clients cannot pass in attributes like they can parameters. Request attributes exist solely for internal use within your application. If the Servlet

places a `Ticket` into a request attribute, the JSP retrieves it as a `Ticket`. During the life of the request, any component of the application that has access to the `HttpServletRequest` instance has access to the request attributes. When the request has completed, the request attributes are discarded.

The last method you need to change is the `listTickets` method. Again, begin by creating the `/WEB-INF/jsp/view/listTickets.jsp` presentation file in the Customer Support application. Because request attributes are `Objects`, you must cast them when you retrieve them. In this case, the cast to a `Map<Integer, Ticket>` is an unchecked operation, so you need to suppress the warning.

```
<%@ page session="false" import="java.util.Map" %>
<%
    @SuppressWarnings("unchecked")
    Map<Integer, Ticket> ticketDatabase =
        (Map<Integer, Ticket>) request.getAttribute("ticketDatabase");
%>
<!DOCTYPE html>
<html>
    <head>
        <title>Customer Support</title>
    </head>
    <body>
        <h2>Tickets</h2>
        <a href="
```

As you can see, this JSP needs the `ticketDatabase`, so you should change the `listTickets` method to provide this variable and forward the request:

```
private void listTickets(HttpServletRequest request,
    HttpServletResponse response)
    throws ServletException, IOException
{
```

```
request.setAttribute("ticketDatabase", this.ticketDatabase);  
request.getRequestDispatcher("/WEB-INF/jsp/view/listTickets.jsp")  
    .forward(request, response);  
}
```

Testing the Updated Customer Support Application

At this point your Servlet code should look much less cluttered. You have moved the presentation code into JSPs and focused on business logic in the Servlet. There are two methods from the previous version of the `TicketServlet`, `writeHeader` and `writeFooter`, that are now unused and can be removed. These made writing presentation code in the Servlet slightly easier, but now you don't need that. Finally, `doGet` and `doPost` had to be updated to reflect the changed signature of the methods they call.

Compile the customer support application and run Tomcat in your IDE debugger. Navigate in your favorite browser to `http://localhost:8080/support/`. You should be redirected to `http://localhost:8080/support/tickets` because of the redirect code in the `index.jsp` file. You should see the page in Figure 4-2. Create a few tickets, uploading attachments with some and not with others; view tickets; and download attachments. Overall, the application should function identically to version 1 created in Chapter 3. However, now that you are no longer writing presentation layer code in Java, it is much easier to improve and expand the application.

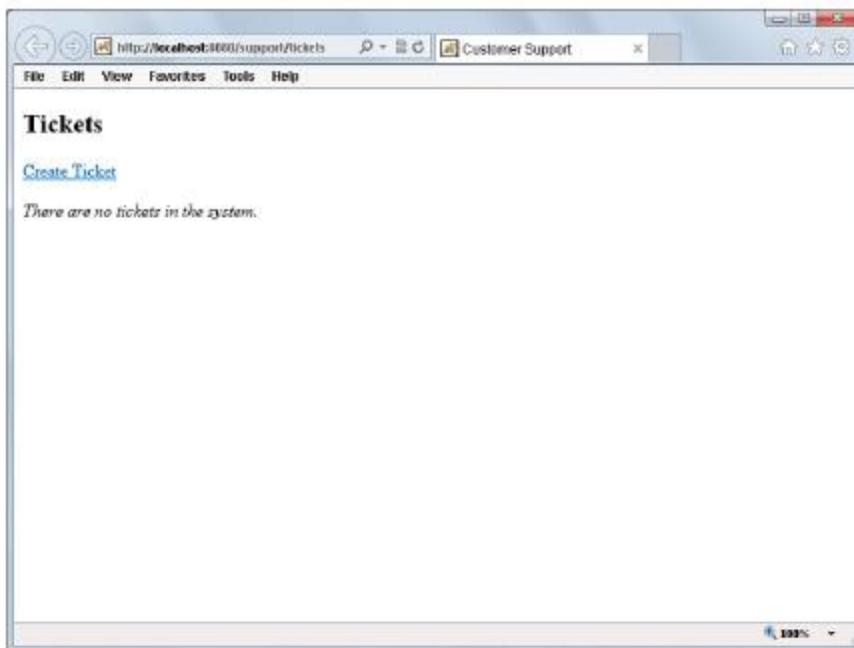


FIGURE 4-2

In the next chapter you continue improving the customer support application by introducing session support and the ability to add comments to support tickets.

A NOTE ABOUT JSP DOCUMENTS (JSPX)

Earlier in the chapter you saw a passing reference to a technology known as JSP Documents, which end in the `.jspx` extension. They are not as widely used as standard JSPs, and although they support the same features, they do so in different ways. Overall, the increased difficulty and code that comes with using JSP Documents instead of JSPs can be nontrivial, as demonstrated in this section. Also, due to the lesser popularity of JSP Documents, you can find fewer examples and code samples online that use JSP Documents, and it may be harder to find forum and mailing list users with experience in JSP Documents to help you with any questions you might have. For this reason, JSP Documents are not used in this book. Only in this chapter do you see an example of JSP Documents for the purpose of understanding the difference between the two related technologies. Nevertheless, the technology exists in case you prefer working with pure XML.

JSP Documents are XML Documents (hence their name), and therefore many of the features you have seen, such as directives, cannot work the same way. XML Documents must adhere to a strict schema or they will fail to parse correctly. The main advantage to using JSP Documents over standard JSPs is that it's slightly easier to detect problems with the JSPs at compile time instead of run time. However, in many cases this benefit is not worth the added cost of dealing with JSP Documents. Table 4-1 lists several JSP features and compares their JSP syntax to their JSP Document syntax.

TABLE 4-1: Comparison of JSP Features and JSP Document Features

FEATURE	JSP SYNTAX	JSP DOCUMENT SYNTAX
Page Directive	<code><%@ page %></code>	<code><jsp:directive.page /></code>
Include Directive	<code><%@ include %></code>	<code><jsp:directive.include /></code>
Tag Library Directive	<code><%@ taglib %></code>	<code>xmlns:prefix="Library URI"</code>
Declaration	<code><%! ... %></code>	<code><jsp:declaration> ... </jsp:declaration></code>
Scriptlet	<code><% ... %></code>	<code><jsp:scriptlet> ... </jsp:scriptlet></code>
Expression	<code><%= ... %></code>	<code><jsp:expression> ... </jsp:expression></code>
Comment	<code><%-- ... --%></code>	<code><!-- ... --></code>

You should notice two patterns in this table:

- Everything is a `jsp` tag. Directives, declarations, scriptlets, and expressions are all XML tags now, with the `jsp` namespace prefix. The only exception is the tag library directive, which becomes an attribute of the root document tag, instead.
- You no longer differentiate between JSP comments and XML comments. All comments are XML comments. (Of course, inside declarations and scriptlets, you can still use Java comments.)

To demonstrate how this can change a document, consider Listing 4-1. This is a simple JSP file with all the features covered in this chapter. Then, compare that code to Listing 4-2, the JSP

Document-equivalent of Listing 4-1. Notice how the directives, declarations, scriptlets, expressions, and comments all change. Pay particular attention to the XML doctype, the `<jsp:root>` element, and the XMLNS attributes. As you can see, JSPs are noticeably easier to work with than JSP Documents.

LISTING 4-1: A standard JSP file

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<%@ include file="/WEB-INF/jsp/base.jspf" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%
    private static final String DEFAULT_USER = "Guest";
%>
<%
    String user = request.getParameter("user");
    if(user == null)
        user = DEFAULT_USER;
%>
<%--<%= "This code is commented" %>--%>
<!DOCTYPE html>
<html>
    <head>
        <title>Hello User Application</title>
    </head>
    <body>
        Hello, <%= user %>!<br /><br />
        <form action="greeting.jsp" method="POST">
            Enter your name:<br />
            <input type="text" name="user" /><br />
            <input type="submit" value="Submit" />
        </form>
    </body>
</html>
```

LISTING 4-2: The JSP Document-equivalent of Listing 4-1

```
<?xml version="1.0" encoding="UTF-8"?>
<jsp:root xmlns="http://www.w3.org/1999/xhtml" version="2.0"
    xmlns:jsp="http://java.sun.com/JSP/Page"
    xmlns:c="http://java.sun.com/jsp/jstl/core">
    <jsp:directive.page contentType="text/html;charset=UTF-8" language="java" />
    <jsp:directive.include file="/WEB-INF/jsp/base.jspx" />
    <jsp:declaration>
        private static final String DEFAULT_USER = "Guest";
    </jsp:declaration>
    <jsp:scriptlet>
        String user = request.getParameter("user");
        if(user == null)
            user = DEFAULT_USER;
    </jsp:scriptlet>
```

continues

LISTING 4-2 (continued)

```
</jsp:scriptlet>
<!--<jsp:expression>"This code is commented"</jsp:expression> -->
<!DOCTYPE html>
<html>
    <head>
        <title>Hello User Application</title>
    </head>
    <body>
        Hello, <jsp:expression>user</jsp:expression>!<br /><br />
        <form action="greeting.jsp" method="post">
            Enter your name:<br />
            <input type="text" name="user" /><br />
            <input type="submit" value="Submit" />
        </form>
    </body>
</html>
</jsp:root>
```

SUMMARY

In this chapter you explored the world of JSPs and learned how they can make your life easier by simplifying the task of writing HTML markup to the response output. You were introduced to directives, declarations, scriptlets, and expressions. You learned about the various ways you can comment out code in JSPs and about the many ways you can include Java code in a JSP file. You also discovered the nine implicit Java variables available in your JSP and read about why using Java scriptlets and declarations is discouraged. Finally, you applied these principles and improved the Customer Support application by adding JSP properties to the deployment descriptor and separating the business logic in the Servlet from the presentation code in the JSP.

In the next chapter you learn about HTTP sessions, their purpose, and how to use them in Java EE web applications.

5

Maintaining State Using Sessions

IN THIS CHAPTER

- Why sessions are necessary
- Working with cookies and URL parameters
- How to store data in a session
- Making sessions useful
- How to cluster an application that uses sessions

WROX.COM CODE DOWNLOADS FOR THIS CHAPTER

You can find the [wrox.com](http://www.wrox.com/go/projavaforwebapps) code downloads for this chapter at www.wrox.com/go/projavaforwebapps on the Download Code tab. The code for this chapter is divided into the following major examples:

- Shopping-Cart Project
- Session-Activity Project
- Customer-Support-v3 Project

NEW MAVEN DEPENDENCIES FOR THIS CHAPTER

There are no new Maven dependencies for this chapter. Continue to use the Maven dependencies introduced in all previous chapters.

UNDERSTANDING WHY SESSIONS ARE NECESSARY

So far you've learned about web applications, web containers, Servlets, JSPs, and how Servlets and JSPs work together. You have also learned about the life cycle of a request, and it should be clear at this point that the tools you have been introduced to so far do not enable you to associate multiple requests coming from the same client and share data between those requests. You might think that you can use the IP address as a unique identifier and all requests from an IP address within some timeframe must belong to the same client. Unfortunately, due to Network Address Translation (NAT) this is not reliable. Thousands of students at a college campus can literally all use the same IP address, hidden behind a NAT router. For this reason the concept of *HTTP sessions* has achieved nearly universal adoption by all HTTP server-side technologies, and Java EE has session support written into its specification.

Not every application needs sessions. The Hello World examples you've seen in this book certainly don't need sessions. So far the Customer Support application hasn't needed sessions. It has been more like an anonymous message board. But if you think about the requirements Multinational Widget Corporation has for its customer support site, you may quickly realize that at some point you must create user accounts, and those users need to log in to the application. Customer support requests may contain private information, such as server configuration files that other customers shouldn't see. Certainly you need a way to restrict access to certain support tickets so that only the posting customer and members of MWC's support team can access any given ticket. You could have users provide a username and password on every page they access, but it's a fair bet customers aren't going to be happy with that solution.

Maintaining State

Sessions are used to maintain state between one request and the next. HTTP requests are completely stateless on their own. From the server's perspective, the request begins when the user's web browser opens a socket to the server, and it ends when the server sends the last packet back to the client and closes the connection. At that point there is no longer a link between the user's browser and the server, and when the next connection comes in, there is no way to tie the new request to the previous request.

Applications often cannot function correctly in such a stateless manner. A classic example is the online shopping website. Nearly every online shopping site these days requires you to create a username and password before purchasing, but consider even the few that don't. When browsing the store, you find a product you like, so you add that product to your shopping cart. You continue browsing the store and find another product you like. You add it to your shopping cart as well. When you view your shopping cart, you see that both products you added remain in your shopping cart. Somehow, between every request you made, the website knew those requests were coming from the same browser on the same computer and associated that with your shopping cart. Nobody else can see your shopping cart or the items in it — it is exclusively tied to your computer and browser. This scenario is an analogy to a real-life shopping experience. You enter your favorite grocery store, and as you walk in the door, you grab a shopping cart or basket. (You get a session from the server.) You walk through the store and pick up items as you go, placing them in your cart (adding them to the session). When you get to the cash register, you remove the items from the cart and give them to the cashier, who scans them and takes

your money. (You check out using your session.) As you walk out the door, you return your shopping cart or basket. (You close your browser or log out, ending your session.)

In this example, the cart or basket maintains your state as you walk through the store. Without the cart, neither you nor the store could keep up with everything you needed to purchase. If no state were maintained between requests, you would have to “walk in,” grab one item, pay for it, “walk out” (end the request), and repeat the entire process again for each item you wanted to purchase. Sessions are the engine behind maintaining state between requests, and without them the web would be a very different place.

Remembering Users

Another scenario to consider is the user forum website. Almost universally in online forums, users are known by their usernames or “handles.” As a user enters the forums, he logs in, providing a username and password to prove his identity. (The merit of username/password authentication as proof of identity is an argument reserved for Chapter 25.) From that point he can add forum threads, respond to threads, participate in private messages with other users, report threads or responses to moderators, and possibly mark threads as favorites. Notice that the user logged in only a single time during that entire timeline. The system needed a way to remember who he was between each request, and sessions provided that.

Enabling Application Workflow

Often users need some form of workflow to complete a task using an advanced web application. In the case of creating a news article for publication on a news site, for example, the journalist might first go to a screen where she can enter a title, tagline, and body and format the elements appropriately. On the next page she might then select one or more photos associated with the article and indicate how they should be displayed. She might also upload or record some video to be placed in the article. Finally, she would probably be presented with a list of similar articles or a search field to find similar articles so that she could indicate which ones should be placed in a Related Articles box.

After all these steps had been completed, the article would be published. This entire scenario represents the idea of a workflow. The workflow contains many steps in it, each step part of the completion of a single task. To tie all these steps together to complete the workflow, the requests must have state maintained between them. The shopping cart example is actually a subset of the broader idea of workflows.

USING SESSION COOKIES AND URL REWRITING

Now that you understand the importance of sessions, you are probably wondering how they work. There are two different components to this: first, the generic theory behind web sessions and how they are implemented; and second, the specifics behind the session implementation in Java EE web applications. Both are covered in this section.

In the general theory of web sessions, a session is some file, memory segment, object, or container managed by the server or web application that contains various data elements assigned to it.

These data elements could be a username, a shopping cart, workflow details, and more. The user's browser does not hold or maintain any of this data. It is managed solely by the server or web application code. The only missing piece is a link between this container and the user's browser. For this purpose, sessions are assigned a randomly generated string called a session ID. The first time a session is created (as a result of a request being received), the session ID for that session is conveyed back to the user's browser as part of the response. Every subsequent request from that user's browser includes the session ID in some fashion. When the application receives the request with the session ID, it can then link the existing session to that request. This is demonstrated in Figure 5-1.

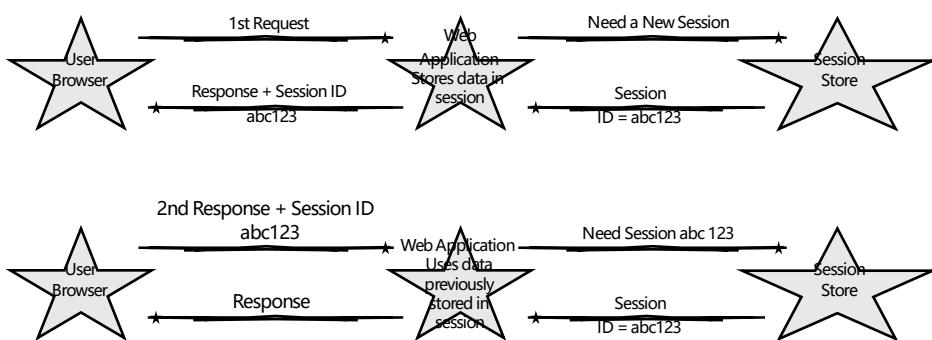


FIGURE 5-1

NOTE You may wonder why the session ID is random instead of a simple sequential ID. The reason for this is compelling: A sequential ID would be predictable, and a predictable ID would make hijacking other users' sessions trivial.

The remaining problem to be solved is how the session ID is passed from server to browser and back. There are two techniques used to accomplish this: *session cookies* and *URL rewriting*.

Understanding the Session Cookie

Fortunately, a solution already exists in HTTP 1.1 that enables servers to send session IDs back to browsers so that the browsers include the session IDs in future requests. This is the technology called *HTTP cookies*. If you are unfamiliar with cookies, they are essentially a mechanism whereby arbitrary data can be passed from the server to the browser via the `Set-Cookie` response header, stored locally on the user's computer, and then transmitted back from the browser to the server via the `Cookie` request header. Cookies can have various attributes, such as a domain name, a path, an expiration date or maximum age, a secure flag, and an HTTP-only flag.

The `Domain` attribute instructs the browser for which domain names it should send the cookie back, whereas the `Path` attribute enables the cookie to further be restricted to a certain URL relative to the domain. Every time a browser makes a request of any type, it finds all cookies that match the domain and path for the site and sends those cookies along with the request. `Expires` defines an absolute expiration date for the cookie, whereas the mutually exclusive `Max-Age` attribute defines the number of seconds before the cookie expires. If a cookie's expiration date is in the past, the browser

deletes it immediately. (This is how you delete a cookie — set its expiration date to the past.) If a cookie does not have an `Expires` or `Max-Age` attribute, it is deleted when the browser is closed. If the `Secure` attribute is present (it does not need to have a value) the browser will send the cookie back only over HTTPS. This protects the cookie from being transmitted unencrypted. Finally, the `HttpOnly` attribute restricts the cookie to direct browser requests. Other technologies, such as JavaScript and Flash, will not have access to the cookie.

Web servers and application servers use cookies to store session IDs on the client side so that they can be transmitted back to the server with each request. With Java EE application servers, the name of this session cookie is `JSESSIONID` by default. Examine the following headers from a series of requests and responses between a client browser and a Java EE web application deployed at `http://www.example.com/support`. This is what you would expect to see if tracing the HTTP requests and responses with a network-sniffing tool like Fiddler or Wireshark.

REQUEST 1

```
GET /support HTTP/1.1
Host: www.example.com
```

RESPONSE 1

```
HTTP/1.1 302 Moved Temporarily
Location: https://www.example.com/support/login
Set-Cookie: JSESSIONID=NRxclGg2vG7kI4Md1Ln; Domain=.example.com; Path=/; HttpOnly
```

REQUEST 2

```
GET /support/login HTTP/1.1
Host: www.example.com
Cookie: JSESSIONID=NRxclGg2vG7kI4Md1Ln
```

RESPONSE 2

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=UTF-8
Content-Length: 21765
```

REQUEST 3

```
POST /support/login HTTP/1.1
Host: www.example.com
Cookie: JSESSIONID=NRxclGg2vG7kI4Md1Ln
```

RESPONSE 3

```
HTTP/1.1 302 Moved Temporarily
Location: http://www.example.com/support/home
Set-Cookie: remusername=Nick; Expires=Wed, 02-Jun-2021 12:15:47 GMT;
Domain=.example.com; Path=/; HttpOnly
```

REQUEST 4

```
GET /support/home HTTP/1.1
Host: www.example.com
Cookie: JSESSIONID=NRxclGg2vG7kI4Md1Ln; remusername=Nick
```

RESPONSE 4

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=UTF-8
Content-Length: 56823
```

The `Set-Cookie` headers in the responses are used to send cookies to the user's browser for storage. Likewise, the `Cookie` headers in the requests are used to send cookies back to the web server. In this imaginary scenario, the user navigates to some support site and gets redirected to the login page. While redirected, the user's browser also gets a session ID cookie from the server. When the user's browser goes to the login page, it includes the session ID cookie in its request. From then on, each time the browser sends a new request, it includes the `JSESSIONID` cookie. The server does not send it again because it knows the browser already has it.

After a successful login, the server also sends back a `remusername` cookie. This is unrelated to the session and in this case represents a technique the site uses to auto-populate the user's username whenever he goes to the login page. Future requests will always contain this cookie; although, future responses do not reset it. Notice that the `JSESSIONID` cookie has no expiration date, whereas the `remusername` cookie does. The `remusername` cookie will expire in the year 2021 (a long time from now, after which the user will probably have a different computer), whereas the `JSESSIONID` cookie will expire as soon as the user closes his browser.

NOTE *The `remusername` cookie is used here simply to demonstrate another use for cookies and how multiple cookies are transmitted in the `Cookie` request header. The actual feature — remembering usernames — is not related to this discussion.*

One of the obstacles to using cookies to transmit session IDs is that users can disable cookie support in their browsers, thereby completely eliminating this method of transmitting session IDs. However, over the past decade this has become less and less of a concern, with one major search and e-mail provider and one major social network requiring cookies to be enabled for users of their websites.

Session IDs in the URL

Another popular method for transmitting session IDs is through URLs. The web or application server knows to look for a particular pattern containing the session ID in the URL and, if found, retrieves the session from the URL. Different technologies use different strategies for embedding and locating session IDs in the URL. For example, PHP uses a query parameter named `PHPSESSID`:

```
http://www.example.com/support?PHPSESSID=NRxc1Gg2vG7kI4Md1Ln&foo=bar&high=five
```

Java EE applications use a different approach. The session ID is placed in a matrix parameter in the last path segment (or directory) in the URL. This frees up the query string so that the session ID does not conflict with other parameters in the query string.

```
http://www.example.com/support;JSESSIONID=NRxc1Gg2vG7kI4Md1Ln?foo=bar&high=five
```

The specific technique that a given technology uses is immaterial to the end result: Embed the session ID in the URL and you avoid needing to use cookies. You might wonder, however, how the session ID in a request URL gets to the browser in the first place. A request URL is only effective for conveying the session ID from the browser to the server. So where does the session ID come from? The answer is that the session ID must be embedded in every URL that the application sends back in every response, including links on the page, form actions, and 302 redirects. Consider the previous example

of the login scenario using cookies. The following headers demonstrate the same set of transactions using URL embedding instead of cookies:

REQUEST 1

```
GET /support HTTP/1.1
Host: www.example.com
```

RESPONSE 1

```
HTTP/1.1 302 Moved Temporarily
Location: https://www.example.com/support/login;JSESSIONID=NRxclGg2vG7kI4Md1Ln
```

REQUEST 2

```
GET /support/login;JSESSIONID=NRxclGg2vG7kI4Md1Ln HTTP/1.1
Host: www.example.com
```

RESPONSE 2

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=UTF-8
Content-Length: 21796
...
<form action="http://www.example.com/support/login;JSESSIONID=NRxclGg2vG7kI4Md1Ln"
      method="post">
...

```

REQUEST 3

```
POST /support/login;JSESSIONID=NRxclGg2vG7kI4Md1Ln HTTP/1.1
Host: www.example.com
```

RESPONSE 3

```
HTTP/1.1 302 Moved Temporarily
Location: http://www.example.com/support/home;JSESSIONID=NRxclGg2vG7kI4Md1Ln
```

REQUEST 4

```
GET /support/home;JSESSIONID=NRxclGg2vG7kI4Md1Ln HTTP/1.1
Host: www.example.com
```

RESPONSE 4

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=UTF-8
Content-Length: 56854
...
<a href="http://www.example.com/support/somewhere;JSESSIONID=NRxclGg2vG7kI4Md1Ln">
...

```

In this case, notice that the session ID is being returned to the browser via the `Location` header, form action, and link tag. As you can see, the browser is never actually “aware” of the session ID like it is with a session cookie. Instead, the server rewrites the `Location` header URL and the URLs in any response content (links, form actions, and other URLs) so that any URLs the browser uses to access the server already have the session ID embedded in them. The important point about this is that the session ID *must be embedded in the `Location` header URL and in every single URL in the markup*. This is no trivial task and can often be downright inconvenient. For this purpose, the Java EE Servlet API comes with a few conveniences that make this simple.

For starters, the `HttpServletResponse` interface defines two methods that rewrite URLs to include embedded session IDs if necessary: `encodeURL` and `encodeRedirectURL`. Any URL that will be placed in a link, form action, or other markup can first be passed to the `encodeURL` method, which returns an appropriately “encoded” version of the URL. Any URL passed to the `sendRedirect` response method can be passed to the `encodeRedirectURL` method, which returns an appropriately “encoded” version of that URL. The word “encoded” here means that the `JSESSIONID` matrix parameter will be embedded in the last path segment of the URL only if all four of the following conditions are met:

- A session is active for the current request. (Either it requested a session by passing in a session ID, or the application code created a new session.)
- The `JSESSIONID` cookie was not present in the request.
- The URL is not an absolute URL and is a URL within the same web application.
- Session URL rewriting is enabled in the deployment descriptor (more on this in the section “Storing Data in a Session”).

The second condition is the troublesome condition. The only way to detect if a user’s browser allows cookies is to set a cookie and then look for that cookie to be returned on the next request. However, you need a session to associate one request with another; otherwise, how would you know whether the request was simply the first request from a different user or a second request from the same user without a cookie? Therefore, the second condition always assumes that the lack of a `JSESSIONID` cookie means the user’s browser doesn’t support cookies, with the understanding that this means URLs will *always* get encoded on the *first* request to a session-enabled application even if the user’s browser supports cookies. The unfortunate side effect is that sometimes URLs contain the `JSESSIONID` matrix parameter even if the user’s browser accepted the `JSESSIONID` cookie.

Of course, the `HttpServletRequest` methods are just part of the toolset available to help you embed session IDs in URLs. The `<c:url>` tag, which is discussed more in Chapter 7, also embeds session IDs in URLs.

Session Vulnerabilities

As you can imagine, sessions are not without their vulnerabilities, and I would be remiss if I did not warn you about them. The bad news is that these vulnerabilities can cause serious problems for your users, and if you transact sensitive or personal information (such as credit card numbers or healthcare data) it can mean huge penalties for your business. The good news is that there are easy ways to address these vulnerabilities, which you will learn about as well. Of course, I cannot possibly cover all potential vulnerabilities in your applications as there are thousands of ways to compromise web applications. The developer should always be diligent and well informed on matters of security. In mission-critical, sensitive applications, it would be wise to use a commercial scanner of some type that scans your application for weaknesses.

For more information about web application and session vulnerabilities and how to detect and address them, visit the Open Web Application Security Project (OWASP) <https://www.owasp.org/> website.

The Copy and Paste Mistake

Perhaps one of the easiest ways a session can be compromised is for an unsuspecting user to copy and paste the URL from his browser into an e-mail, forum posting, chat room, or other public area. Embedding session IDs in URLs, which you read about earlier in this section, is the source of this problem. Remember the URLs passed back and forth between the client and server? Those URLs, session ID and all, appear in the address bar in the client's browser. If the user decides to share a page in your application with his friends and copies and pastes the URL from the address bar, the session ID is included in the URL his friends see. If they go to that URL before the session expires, *they then assume the identity of the user who shared the URL*. The obvious problem with this is that the user's friends might see personal information accidentally.

The more dangerous scenario is that a nefarious character finds the link and uses it to hijack the user's session. He can then change the account e-mail address, obtain a password reset link, and finally change the password — giving the attacker complete control over the user's account and everything in it.

As innocent as the origin of this problem is — a user copying and pasting a URL from his address bar — the only infallible method of addressing this vulnerability is to completely disable embedding session IDs in URLs. Although this may sound like a drastic measure with potentially catastrophic consequences for the usability of your application, remember what was said earlier about how commonplace it has become for major Internet companies to require cookies when using their sites. Cookies have become a fact of life for web users today, and the vulnerabilities inherent in cookies are far less common and dangerous than this one.

Session Fixation

The *session fixation attack* is similar to the copy-and-paste mistake, except that the "unsuspecting user" in this case is the attacker, and the victims are the users who use a link containing a session ID. An attacker might go to some website known to accept session IDs embedded in the URL. The attacker will obtain a session ID in this manner (either through a URL or by examining the browser's cookies) and then send a URL containing that session ID to a victim, through a forum or (most often) an e-mail. At this point, when the user clicks the link to go to the website, his session ID is fixed to what was in the URL — a session ID the attacker knows about. If the user then logs in to the website during this session, the attacker will also be logged in because he shares the session ID, giving him access to the user's account.

There are two ways to address the issue:

- As with the copy-and-paste mistake, you can simply disable the embedding of session IDs in URLs *and also* disallow your application from accepting session IDs via URLs (something you explore in the section "Storing Data in a Session").
- Employ *session migration* after login. When the user logs in, change the session ID, or copy the session details to a new session and invalidate the original session. (Either method achieves the same thing: assigning a different session ID to the newly "logged in" session.) The attacker still has the original session ID, which is no longer valid and not connected to the user's session.

WARNING *There is another type of session fixation attack in which a malicious website writes a session ID cookie using another website's domain name, effectively setting the session ID for the other website in the victim's browser. This attack has the same effect as the URL session fixation attack. However, there is no way for web applications to protect against this vulnerability without disabling sessions altogether. This vulnerability is actually a browser vulnerability, not a vulnerability of web applications.*

All modern browsers have fixed this vulnerability for cross-domain attacks (site example.net sets a cookie for site example.com). However, site malicious .example.net could still set a session cookie for domain .example.net, which would then be picked up by site vulnerable.example.net. This problem can be avoided altogether by following a simple rule: Don't share a domain name with untrusted applications.

Cross-Site Scripting and Session Hijacking

You have already read about the copy-and-paste mistake which, when exploited by a malicious party, becomes a session fixation attack. There is another form of *session hijacking* that utilizes JavaScript to read the contents of a session cookie. An attacker, who exploits a site's vulnerability to *cross-site scripting attacks*, injects JavaScript into a page to read the contents of a session ID cookie using the JavaScript DOM property `document.cookie`. After the attacker retrieves a session ID from an unsuspecting user, he can then assume that session ID by creating a cookie on his own machine or using URL embedding, thereby assuming the identity of the victim.

The most obvious defense against this attack is to secure your site against cross-site scripting, which is a topic outside the scope of this book (see the previously mentioned OWASP website). However, doing this can be tricky and difficult, and attackers are constantly finding new ways to effect cross-site scripting attacks. An alternative defense, which you should *always* use in conjunction with this, is flagging all your cookies with the `HttpOnly` attribute. This attribute allows the cookie to be used only when the browser makes an HTTP (or HTTPS) request, whether that request happens via link, manual entry of a URL in the address bar, form submission, or AJAX request. More important, `HttpOnly` completely disables the ability of JavaScript, Flash, or some other browser scripting or plugin to obtain the contents of the cookie (or even know of its existence). This stops the cross-site scripting session hijacking attack in its tracks. Session ID cookies should always include the `HttpOnly` attribute.

NOTE *Although the `HttpOnly` attribute prevents JavaScript from accessing the cookie using the `document.cookie` DOM property, AJAX requests originating from JavaScript code will still include the session ID cookie because the browser, not the JavaScript code, is responsible for forming the AJAX request headers. This means the server will still be able to associate the AJAX requests with the user's session.*

Insecure Cookies

The final vulnerability you should consider is the *man-in-the-middle attack (MitM attack)*, the classic data interception attack whereby an attacker observes a request or response as it travels between the client and server and obtains information from the request or response. This attack gave rise to Secure Sockets Layer and Transport Layer Security (SSL/TLS), the foundation of the HTTPS protocol. Securing your web traffic using HTTPS effectively foils the MitM attack and prevents session ID cookies from being stolen. The problem, however, is that a user might first try to go to your site using HTTP. Even if you redirect them to HTTPS, the damage is already done: Their browser has transmitted the session ID cookie to your server unencrypted, and an observing attacker can steal the session ID.

The `Secure` cookie flag was created to address this very issue. When your server sends the session ID cookie to the client in the response, it sets the `Secure` flag. This tells the browser that the cookie should be transmitted only over HTTPS. From then on, the cookie will only be transmitted encrypted, and attackers cannot intercept it. The drawback is that your site must *always* be behind HTTPS for this to work. Otherwise, as soon as you redirect the user to HTTP, the browser can no longer transmit the cookie and the session will be lost. For this reason, you must weigh the security needs of your application and determine if the data you are protecting is sensitive enough to warrant the performance overhead and hassle of securing every request with HTTPS.

The Strongest Possible Defense

One final option you should understand when dealing with the security of your sessions is the SSL/TLS Session ID. To improve the efficiency of the SSL protocol by eliminating the need to perform an SSL handshake on every request, the SSL protocol defines its own type of session ID. The *SSL Session ID* is established during the SSL handshake and then used in subsequent requests to tie requests together for determining which keys should be used for encryption and decryption. This very concept duplicates the notion of the HTTP session ID. However, the SSL Session ID is not transmitted or stored using cookies or URLs and is extremely secure. (You can learn more about how the SSL Session ID works by reviewing RFC 2246 “The TLS Protocol.”) It is inordinately difficult to obtain an SSL Session ID for which you are not authorized. Some extremely high-security websites, such as those of financial institutions, reuse the SSL Session ID as the HTTP session ID, thereby eliminating cookies *and* URL encoding and still maintaining state between requests.

This is an extremely secure method of establishing a session ID across requests and is nearly invulnerable. Plus, when SSL vulnerabilities are found, they are usually dealt with in a matter of weeks and eliminated by browser updates. However, there are understandably some drawbacks to using this technique; otherwise, everyone would use it. In older versions of the Java EE specification, there was no standard way to specify this, so developers had to use container-specific classes to achieve using SSL Session IDs, and this configuration was sometimes hit-or-miss. In the Java EE 6.0 specification, an option was added (which you learn about in the next section) to easily instruct the web container to use SSL session IDs, so configuration is no longer a major concern (though not many sites are using this yet). In addition, as with the `Secure` cookie flag, it requires that your site *always* be behind HTTPS. If you are concerned enough about security to enable this feature, however, you probably intend for your entire site to always be behind HTTPS, so this will likely not be an issue for you.

Another problem with reusing the SSL Session ID is that the web container must be responsible for the SSL communications. If you use a web server or load balancer to manage your SSL communications — something common in clustered server environments — the web container will not know what the SSL Session ID is. In such a clustered environment, the user's request must also always be routed to the same server. Finally, depending on server and browser, the life of the SSL Session ID can be very long or very short, so it's hard to rely on this as an HTTP session ID replacement.

Now that you have been introduced to sessions, learned about the `JSESSIONID` cookie and URL rewriting, and explored some of the vulnerabilities inherent in sessions and how to address them, it's time to start using sessions in your Java EE applications.

STORING DATA IN A SESSION

As you learn about using sessions in Java EE, you will be using the Shopping-Cart example project found on the [wrox.com](http://www.wrox.com) code download site. You will not create an entire shopping site with payment systems and related features. You will simply explore the concept of using sessions to aggregate data collected across multiple pages (in this case, products added to a shopping cart). You can create the project yourself or follow along in the Shopping-Cart project. Your project should start with the deployment descriptor `<jsp-config>` from Chapter 4 and the following `/WEB-INF/jsp/base.jspf` file:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
```

Also, you should have a simple `index.jsp` file in your web root for redirecting to your store Servlet:

```
<c:redirect url="/shop" />
```

Configuring Sessions in the Deployment Descriptor

In many cases, HTTP sessions are ready to go in Java EE and require no explicit configuration. However, configure them you can, and for security purposes you should. You configure sessions in the deployment descriptor using the `<session-config>` tag. Within this tag, you can configure the method by which sessions are tracked, the age after which sessions timeout, and the details of the session ID cookie, if you use that. Many of these have default values that you never need to change. The following code demonstrates all the possible deployment descriptor settings for sessions.

```
<session-config>
  <session-timeout>30</session-timeout>
  <cookie-config>
    <name>JSESSIONID</name>
    <domain>example.org</domain>
    <path>/shop</path>
    <comment><![CDATA[Keeps you logged in. See our privacy policy for
more information.]]></comment>
    <http-only>true</http-only>
    <secure>false</secure>
    <max-age>1800</max-age>
  </cookie-config>
  <tracking-mode>COOKIE</tracking-mode>
```

```
<tracking-mode>URL</tracking-mode>
<tracking-mode>SSL</tracking-mode>
</session-config>
```

All of the tags within `<session-config>` and `<cookie-config>` are optional, but they must appear in the order shown in this example (excluding omitted tags). The `<session-timeout>` tag specifies how long sessions should remain inactive, in minutes, before being invalidated. If the value is 0 or less, the session never expires. If this tag is omitted, the container default applies. Tomcat's default container is 30, which can be changed in the Tomcat configuration. If you want consistency, you should explicitly set the timeout using this tag. In this example the timeout is 30 minutes. Each time a user with a certain session ID makes a request to your application, the timer resets on his session's inactivity. If he goes more than 30 minutes without making a request, his session is considered invalid and he is given a new session. The `<tracking-mode>` tag, which was added in Servlet 3.0/Java EE 6, indicates which technique the container should use for tracking session IDs. The legal values are:

- `URL` — The container only embeds session IDs in URLs. It does not use cookies or SSL session IDs. This approach is not very secure.
- `COOKIE` — The container uses session cookies for tracking session IDs. This technique is very secure.
- `SSL` — The container uses SSL Session IDs as HTTP session IDs. This method is the most secure approach available but requires all requests to be HTTPS for it to work properly.

You may use `<tracking-mode>` more than once to tell the container it can use multiple strategies. For example, if you specify both `COOKIE` and `URL`, the container prefers cookies but uses URLs when cookies are not available (as described in the previous section). Specifying `COOKIE` as the only tracking mode tells the container to *never* embed sessions in URLs and always assume the user has cookies enabled. Likewise, specifying `URL` as the only tracking mode tells the container to *never* use cookies. If you enable the `SSL` tracking mode, you cannot also enable the `COOKIE` or `URL` modes. SSL Session IDs must be used on their own; the container cannot fall back to cookies or URLs in the absence of HTTPS.

The `<cookie-config>` tag applies only when `COOKIE` is specified as one of the (or the only) tracking modes. Tags within it customize the session cookies that the container returns to the browser:

- The `<name>` tag enables you to customize the name of the session cookie. The default is `JSESSIONID`, and you will probably never need to change that.
- The `<domain>` and `<path>` tags correspond to the `Domain` and `Path` attributes of the cookie. The web container appropriately defaults these for you so that you should usually not need to customize them. The `Domain` defaults to the domain name used to make the request during which the session was created. The `Path` defaults to the deployed application context name.
- The `<comment>` tag adds a `Comment` attribute to the session ID cookie, providing the opportunity to add arbitrary text. This is often used to explain the purpose of the cookie and point users to the site's privacy policy. Whether you use this is entirely up to you. If you omit this tag, the `Comment` attribute is not added to the cookie.

- The `<http-only>` and `<secure>` tags correspond to the `HttpOnly` and `Secure` cookie attributes, and both default to `false`. For increased security you should always customize `<http-only>` to `true`. `<secure>` should be changed to `true` only if you have HTTPS enabled.
- The final tag, `<max-age>`, specifies the `Max-Age` cookie attribute that controls when the cookie expires. By default, the cookie has no expiration date, which means it expires when the browser closes. Setting this to `-1` has the same effect. Expiring the cookie when the browser closes is almost always what you want. You customize this value in seconds (unlike `<session-timeout>`, which is in minutes), but doing so could cause the cookie to expire and session tracking to fail while the user is in the middle of actively using your application. It's best to leave this one alone and not use this tag.

NOTE *As of Servlet 3.0/Java EE 6, you can skip the deployment descriptor and configure most of these options programmatically using the `ServletContext`. Use the `setSessionTrackingModes` method to specify a `Set` of one or more `javax.servlet.SessionTrackingMode` enum constants. `getSessionCookieConfig` returns a `javax.servlet.SessionCookieConfig` — use this object to configure any of the `<cookie-config>` settings. You can configure the tracking modes or cookie configuration only within a `ServletContextListener`'s `contextInitialized` method or a `ServletContainerInitializer`'s `onStartup` method. You learn about listeners in the "Applying Sessions Usefully" section, and `ServletContainerInitializers` in Chapter 12. Currently you cannot configure the session timeout programmatically — this oversight should be corrected in Java EE 8.*

Now that you understand the available options, the session configuration for the Shopping-Cart project is as follows:

```
<session-config>
  <session-timeout>30</session-timeout>
  <cookie-config>
    <http-only>true</http-only>
  </cookie-config>
  <tracking-mode>COOKIE</tracking-mode>
</session-config>
```

This causes sessions to last 30 minutes, instructs the container to only use cookies for session tracking and makes session cookies contain the `HttpOnly` attribute for security. It accepts all the other default values and does not specify a comment for the cookie. URL session tracking is disabled because it is not secure. For the rest of the book, you will always use this session configuration.

NOTE *As noted earlier, the most secure approach would be to use SSL Session IDs. A secure compromise uses cookies but sets the cookie `Secure` attribute to require HTTPS. This book does not demonstrate either of these techniques because doing so would require generating a self-signed SSL certificate and learning the complexities of configuring SSL in Tomcat. Both of these topics are beyond the scope of this book and can be explored more in the Tomcat documentation.*

Storing and Retrieving Data

In your project create a Servlet called `com.wrox.StoreServlet` and annotate it as a Servlet with the URL pattern `/shop`. In addition, create a simple map in your Servlet representing a product database. (Or, just use the Shopping-Cart project.)

```

@WebServlet(
    name = "storeServlet",
    urlPatterns = "/shop"
)
public class StoreServlet extends HttpServlet
{
    private final Map<Integer, String> products = new Hashtable<>();

    public StoreServlet()
    {
        this.products.put(1, "Sandpaper");
        this.products.put(2, "Nails");
        this.products.put(3, "Glue");
        this.products.put(4, "Paint");
        this.products.put(5, "Tape");
    }
}

```

You can use this product database to "browse" products and link cart items back to product names.

Using Sessions in Your Servlets

Create a simple implementation of the `doGet` method supporting three actions: `browse`, `addToCart`, and `viewCart`:

```

@Override
protected void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException
{
    String action = request.getParameter("action");
    if(action == null)
        action = "browse";

    switch(action)
    {
        case "addToCart":
            this.addToCart(request, response);
            break;

        case "viewCart":
            this.viewCart(request, response);
            break;

        case "browse":
        default:
            this.browse(request, response);
            break;
    }
}

```

The `browse` and `viewCart` methods of your Servlet should be quite simple, adding a request attribute and forwarding on to a JSP:

```
private void viewCart(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException
{
    request.setAttribute("products", this.products);
    request.getRequestDispatcher("/WEB-INF/jsp/view/viewCart.jsp")
        .forward(request, response);
}

private void browse(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException
{
    request.setAttribute("products", this.products);
    request.getRequestDispatcher("/WEB-INF/jsp/view/browse.jsp")
        .forward(request, response);
}
```

These methods are similar in that they both add the products database to a request attribute, but they forward to different JSPs. Now take a look at the `addToCart` method:

```
private void addToCart(HttpServletRequest request,
    HttpServletResponse response)
    throws ServletException, IOException
{
    int productId;
    try
    {
        productId = Integer.parseInt(request.getParameter("productId"));
    }
    catch (Exception e)
    {
        response.sendRedirect("shop");
        return;
    }

    HttpSession session = request.getSession();
    if(session.getAttribute("cart") == null)
        session.setAttribute("cart", new Hashtable<Integer, Integer>());

    @SuppressWarnings("unchecked")
    Map<Integer, Integer> cart =
        (Map<Integer, Integer>) session.getAttribute("cart");
    if(!cart.containsKey(productId))
        cart.put(productId, 0);
    cart.put(productId, cart.get(productId) + 1);

    response.sendRedirect("shop?action=viewCart");
}
```

This method is definitely more complicated. First, it gets and parses the product ID for the product being added to the cart. After that the code in bold calls some new session-related methods that you haven't looked at yet. The `getSession` method on `HttpServletRequest` comes in two forms: `getSession()` and `getSession(boolean)`.

A call to `getSession()` calls `getSession(true)`, which returns the existing session if one exists and creates a new session if a session does not already exist. (It never returns `null`.) A call to `getSession(false)`, on the other hand, returns the existing session if one exists and `null` if no session exists. There are reasons for calling `getSession` with an argument of `false` — for example, you may want to test whether a session has already been created — but in most cases you simply call `getSession()`. The `getAttribute` method returns an object stored in the session. It has a counterpart, `getAttributeNames`, which returns an enumeration of the names of all the attributes in the session. The `setAttribute` method binds an object to the session. In this example, the code looks for the `cart` attribute, adds it if it does not exist, and then retrieves the simple `cart` map from the session. It then looks for the product ID in the cart and adds it with a quantity of zero if it does not exist. Finally, it increments the quantity of that product in the cart.

Using Sessions in Your JSPs

The Servlet code can handle the logic in your application, but you need some JSPs to display the product list and shopping cart. Start by creating `/WEB-INF/jsp/view/browse.jsp`:

```
<%@ page import="java.util.Map" %>
<!DOCTYPE html>
<html>
    <head>
        <title>Product List</title>
    </head>
    <body>
        <h2>Product List</h2>
        <a href="

```

This JSP has little new in it, and simply lists out all the products. You will explore the `<c:url>` and `<c:param>` tags further in Chapter 7. Clicking a product name adds it to the cart. Next create `/WEB-INF/jsp/view/viewCart.jsp`:

```
<%@ page import="java.util.Map" %>
<!DOCTYPE html>
<html>
    <head>
        <title>View Cart</title>
    </head>
    <body>
```

```

<h2>View Cart</h2>
<a href=<c:url value="/shop" />>Product List</a><br /><br />
<%
    @SuppressWarnings("unchecked")
    Map<Integer, String> products =
        (Map<Integer, String>) request.getAttribute("products");
    @SuppressWarnings("unchecked")
    Map<Integer, Integer> cart =
        (Map<Integer, Integer>) session.getAttribute("cart");

    if(cart == null || cart.size() == 0)
        out.println("Your cart is empty.");
    else
    {
        for(int id : cart.keySet())
        {
            out.println(products.get(id) + " (qty: " + cart.get(id) +
                ")<br />");
        }
    }
%>
</body>
</html>

```

This JSP uses the implicit `session` variable you learned about in Chapter 4 to access the shopping cart `Map` stored in the session. It then lists out all the items in the cart and their quantities. Notice that the `session` attribute of the `page` directive is no longer set to `false` (it defaults to `true`), which enables you to use the `session` variable in the JSP.

Compiling and Testing

Now that everything is in place, compile your project, and run Tomcat in your IDE debugger.

1. Navigate in your browser to `http://localhost:8080/shopping-cart/` and you see the list of products.
2. Click View Cart to view your cart, which will be empty because you haven't added anything yet.
3. Click Product List to return to the product list and then click a product name to add it to your cart. You should now see the cart, which has the item in it.
4. Return to the product list and add a different product to the cart. Now you should see both items in your cart. The session is successfully storing data between requests.
5. Add another product and also add some of the same products. More products should appear in your cart, and the quantities should increase for products you've added again.

After a while, your cart should look like Figure 5-2.

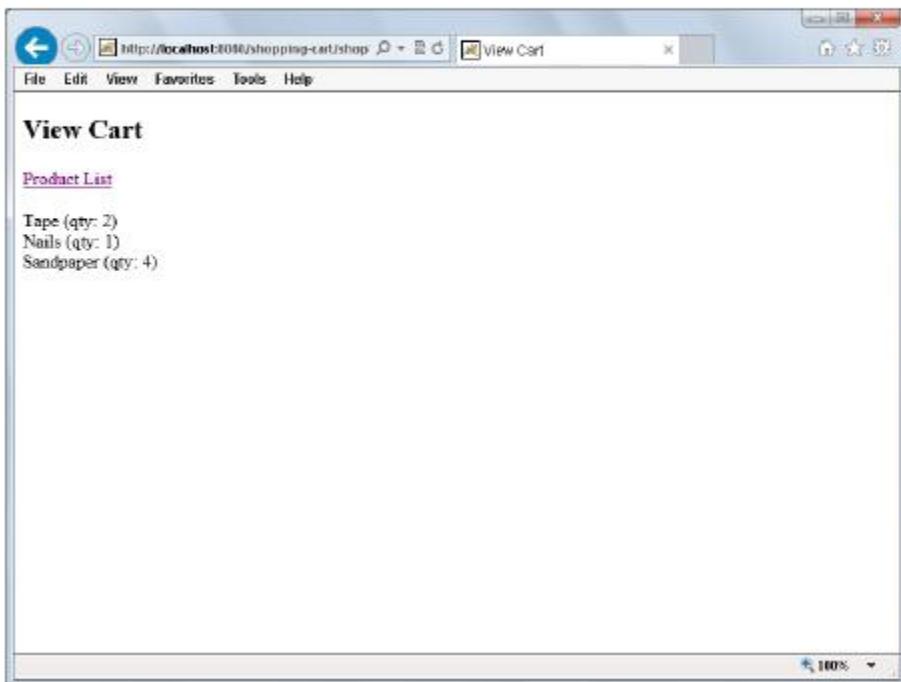


FIGURE 5-2

To further test that the session is working properly, open your application in a different browser, and click View Cart. The cart in the new browser should be empty, whereas the cart in your original browser should still have items in it. This demonstrates that not only is your cart persisting between requests, but also that it belongs *only* to your individual session in that browser. No other users can see it.

The final test is to close and re-open the original browser window that had cart items in it. Now the cart should be empty. This is because your session cookie expired when you closed the browser, and when you went back to your application, you got a new session. The old session, however, sticks around for a while until you undeploy the application or shut down Tomcat, or the session times out due to inactivity. There is no (easy) way to get that session back in your browser.

Removing Data

So far the session is useful, but you shouldn't have to close and re-open your browser to empty your cart. That's where the `removeAttribute` method of the session comes in.

1. Add a new case to your `doGet` method:

```
case "emptyCart":  
    this.emptyCart(request, response);  
    break;
```

2. Add the `emptyCart` method implementation:

```
private void emptyCart(HttpServletRequest request,
```

```

        HttpServletResponse response)
throws ServletException, IOException
{
    request.getSession().removeAttribute("cart");
    response.sendRedirect("shop?action=viewCart");
}

```

As you can see, this is the simplest method in your Servlet. The code removes the `cart` attribute from your session and then redirects you to view your empty cart.

NOTE *It should be pointed out that you could have instead called `getAttribute` to retrieve the `Map` and then called the `clear` method on the `Map`. This would also empty the cart and would be slightly more efficient because over time it would lead to fewer garbage collections. However, this example demonstrates the use of the `removeAttribute` method.*

3. You now need a way to navigate to the link to empty the cart. Modify `/WEB-INF/jsp/view/viewCart.jsp` and add the following link to it:


```
<a href=">">Empty Cart</a><br /><br />
```
4. Compile and debug your application and add some products to your cart.
5. After your cart starts to fill up, click Empty Cart. All the products in your cart should go away, leaving you with an empty cart.

You can do some other things with sessions that you won't experiment with here but that you need to know about. The most obvious thing you might want to do is retrieve the session ID to use for some purpose. Calling the `getId` method on the `HttpSession` object easily accomplishes this. Also there are the `getCreationTime` and `getLastAccessedTime` methods. Although `getCreationTime` obviously returns the time (Unix timestamp in milliseconds) that the session object was created, the `getLastAccessedTime` method can be a bit counterintuitive.

This is not the last time that your code used the session object in some way. Instead, it is the timestamp of the last request that included the session ID for that session in it (URL, cookie, or SSL session) — in other words, the last time the *user* accessed the session. The `isNew` method can be handy: It returns `true` if the session was created during the current request, which means the user's browser has not yet received the session ID.

`getMaxInactiveInterval` returns the maximum time (in seconds) that this session can be inactive (no requests containing the session ID) before it expires. Its counterpart is `setMaxInactiveInterval`, which enables you to change the inactivity window. By default, `getMaxInactiveInterval` returns the value you set in `<session-timeout>`. The `setMaxInactiveInterval` method overrides this configured setting to make it shorter or longer for this specific session.

To understand why you might need to do this, consider an application where certain users (administrators) have a lot of power and can see sensitive information. You might want their

inactivity interval to be shorter than other users'. So, when the user first signs in, you call `setMaxInactiveInterval` to change this value depending on the user's permissions.

Perhaps one of the most important `HttpSession` methods to know about is the `invalidate` method. This is a method that you would call when a user logs out (although that is just one example). `invalidate` destroys the session and unbinds all the data bound to it. Even if the client's browser makes another request with the same session ID, the invalidated session is not used. Instead, a new session is created and the response contains the new session ID.

Storing More Complex Data in Sessions

So far you've learned how to use the `HttpSession` object and how to add data to and remove it from the session. However, you worked only with a simple `Map` with integer keys and values. Is this all that a session can do? The answer is no. Theoretically speaking, a session can store just about anything you want to put in it.

Of course, you have size considerations to think about. If you put too much data in your sessions, you could begin to exhaust the virtual machine's memory pool. Then there's clustering to keep in mind. Clustering is discussed in the section "Clustering an Application That Uses Sessions," but you want to make sure that you can serialize and transmit your session data throughout the cluster (so the session attributes would need to implement `Serializable`). Other than those two restrictions, there's really not a lot you can't put in a session.

To demonstrate this, consider the Session-Activity example project available on the wrox.com download site. It has the same deployment descriptor and `/WEB-INF/jsp/base.jspf` file and a slightly different `index.jsp`:

```
<c:redirect url="/do/home" />
```

In the `com.wrox` package there is a POJO called `PageVisit`. The class and its fields are shown in the following code. The simple accessor (getter) and mutator (setter) methods for this class are left up to the reader to complete.

```
import java.io.Serializable;
import java.net.InetAddress;

public class PageVisit implements Serializable
{
    private long enteredTimestamp;
    private Long leftTimestamp;
    private String request;
    private InetAddress ipAddress;
    // accessor and mutator methods
}
```

Notice that although `enteredTimestamp` is a primitive `long`, `leftTimestamp` is a wrapper `Long`. This is so that `leftTimestamp` can be `null`. The `ActivityServlet` in Listing 5-1 isn't very complex. The standard `doGet` method calls `recordSessionActivity` and then

`viewSessionActivity`. The `viewSessionActivity` method simply forwards to a JSP. `recordSessionActivity` is doing all the fun work: It gets the session; ensures the `activity` Vector exists in the session; updates the `leftTimestamp` for the last `PageVisit` in the Vector, if there is one; and then adds information about the current request to the Vector. `Vector` is used here because, unlike `ArrayList`, it is a thread-safe `List`. The URL pattern for the Servlet has a wildcard in it. This URL pattern means that this Servlet answers any request starting with `/do/`, which can come in handy when you test this out.

LISTING 5-1: ActivityServlet.java

```

@.WebServlet(
    name = "storeServlet",
    urlPatterns = "/do/*"
)
public class ActivityServlet extends HttpServlet
{
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException
    {
        this.recordSessionActivity(request);

        this.viewSessionActivity(request, response);
    }

    private void recordSessionActivity(HttpServletRequest request)
    {
        HttpSession session = request.getSession();

        if(session.getAttribute("activity") == null)
            session.setAttribute("activity", new Vector<PageVisit>());
        @SuppressWarnings("unchecked")
        Vector<PageVisit> visits =
            (Vector<PageVisit>) session.getAttribute("activity");

        if(!visits.isEmpty())
        {
            PageVisit last = visits.lastElement();
            last.setLeftTimestamp(System.currentTimeMillis());
        }

        PageVisit now = new PageVisit();
        now.setEnteredTimestamp(System.currentTimeMillis());
        if(request.getQueryString() == null)
            now.setRequest(request.getRequestURL().toString());
        else
            now.setRequest(request.getRequestURL()+"?"+request.getQueryString());
        try
        {
            now.setIpAddress(InetAddress.getByName(request.getRemoteAddr()));
        }
    }
}

```

```

        catch (UnknownHostException e)
        {
            e.printStackTrace();
        }
        visits.add(now);
    }

    private void viewSessionActivity(HttpServletRequest request,
                                    HttpServletResponse response)
        throws ServletException, IOException
    {
        request.getRequestDispatcher("/WEB-INF/jsp/view/viewSessionActivity.jsp")
            .forward(request, response);
    }
}

```

The final thing to look at in this project is the `/WEB-INF/jsp/view/viewSessionActivity.jsp` file in Listing 5-2. It's less complicated than it looks. All it's doing is displaying all the page visit data accrued in the session in a readable manner. Now to test this, follow these steps:

1. Compile and debug your application and navigate to `http://localhost:8080/session-activity/do/home/` in your browser. You should see some information about your session, an indication that the session is new, and information about the request you just made.
2. Start adding paths and query parameters to the end of the URL. Try different URLs and wait different amounts of time between each request. You can even replace `home/` with something else — just make sure you leave `/do/` in the URL.

After a while, you should start to see something like Figure 5-3 emerge. Your application is tracking request activity and persisting it between requests to display to the user.

LISTING 5-2: viewSessionActivity.jsp

```

<%@ page import="java.util.Vector, com.wrox.PageVisit, java.util.Date" %>
<%@ page import="java.text.SimpleDateFormat" %>
<%
    private static String toString(long timeInterval)
    {
        if(timeInterval < 1_000)
            return "less than one second";
        if(timeInterval < 60_000)
            return (timeInterval / 1_000) + " seconds";
        return "about " + (timeInterval / 60_000) + " minutes";
    }
%>
<%
    SimpleDateFormat f = new SimpleDateFormat("EEE, d MMM yyyy HH:mm:ss Z");
%>
<!DOCTYPE html>
<html>
    <head>
        <title>Session Activity Tracker</title>
    </head>
    <body>

```

continues

LISTING 5-2 (continued)

```

<h2>Session Properties</h2>
Session ID: <%= session.getId() %><br />
Session is new: <%= session.isNew() %><br />
Session created: <%= f.format(new Date(session.getCreationTime())) %><br />

<h2>Page Activity This Session</h2>
<%
    @SuppressWarnings("unchecked")
    Vector<PageVisit> visits =
        (Vector<PageVisit>) session.getAttribute("activity");

    for (PageVisit visit : visits)
    {
        out.print(visit.getRequest());
        if (visit.getIpAddress() != null)
            out.print(" from IP " + visit.getIpAddress().getHostAddress());
        out.print(" (" + f.format(new Date(visit.getEnteredTimestamp())));
        if (visit.getLeftTimestamp() != null)
        {
            out.print(", stayed for " + toString(
                visit.getLeftTimestamp() - visit.getEnteredTimestamp()
            ));
        }
        out.println("<br />");
    }
%>
</body>
</html>

```

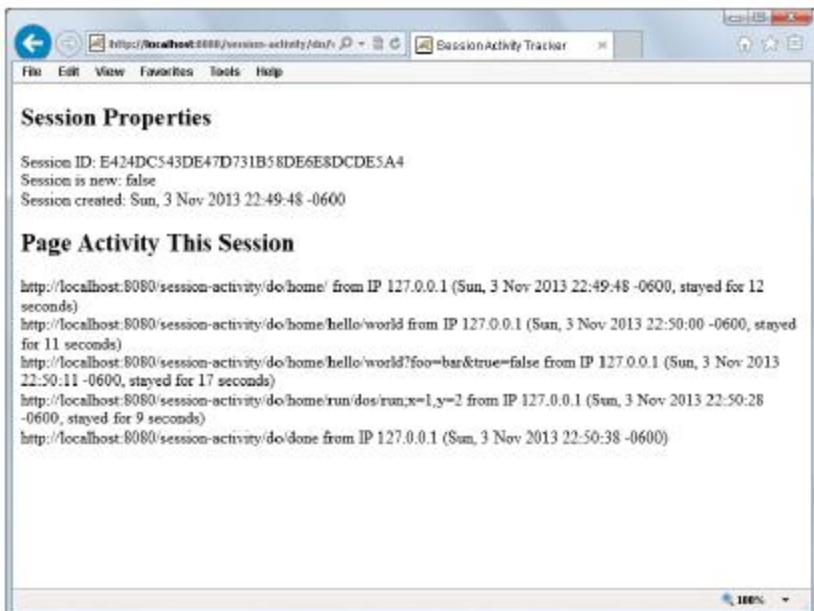


FIGURE 5-3

APPLYING SESSIONS USEFULLY

At this point you should be well acquainted with how sessions work and how to use sessions in Java EE web applications. There are many things you can do with sessions. In addition, some extra tools are available to help you track when sessions are created, destroyed, and updated. You explore those further in this section. For the rest of the chapter, you'll work with the Customer-Support-v3 project found on the wrox.com code download site and integrate sessions into the Customer Support application.

Adding Login to the Customer Support Application

In the last chapter you disabled sessions in the customer support application by adding `session="false"` to the `page` attributes in all the JSPs. You want to use sessions now, and this can prevent you from doing that, so remove the `session="false"` attribute from all the JSPs in version 3 of the Customer Support application. Remember that this attribute value defaults to `true`, so removing the attribute altogether enables sessions.

You should also add the `<session-config>` XML from the Shopping-Cart application to the deployment descriptor so that sessions are configured for better security and session IDs don't end up in URLs. It should be obvious at this point that the Customer Support application needs some form of user database with logins. In this section, you'll add a very rudimentary, unsecure login capability to your application. In the last part of the book several chapters cover securing your application with a more comprehensive authentication and authorization system, so you can keep it simple for now.

Setting Up the User Database

Add a `LoginServlet` class to your application and create a static, in-memory user database in it:

```

@WebServlet(
    name = "loginServlet",
    urlPatterns = "/login"
)
public class LoginServlet extends HttpServlet
{
    private static final Map<String, String> userDatabase = new Hashtable<>();

    static {
        userDatabase.put("Nicholas", "password");
        userDatabase.put("Sarah", "drowssap");
        userDatabase.put("Mike", "wordpass");
        userDatabase.put("John", "green");
    }
}

```

As you can see, the user database is a simple map of usernames to passwords without respect to any sort of varying permissions level. Users can either access the system or they can't, and passwords are not stored in a secure manner. The `doGet` method is responsible for displaying the login screen, so create that now.

```

@Override
protected void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException
{
    HttpSession session = request.getSession();
    if(session.getAttribute("username") != null)
    {
        response.sendRedirect("tickets");
        return;
    }

    request.setAttribute("loginFailed", false);
    request.getRequestDispatcher("/WEB-INF/jsp/view/login.jsp")
        .forward(request, response);
}

```

The first thing the method in the previous example does is check to see if a user is already logged in (a `username` attribute exists) and redirect them to the ticket screen if they are. If the user is not logged in, it sets a `loginFailed` request attribute to `false` and forwards the request to the login JSP. When the login form on the JSP is submitted, it posts to the `doPost` method:

```

@Override
protected void doPost(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException
{
    HttpSession session = request.getSession();
    if(session.getAttribute("username") != null)
    {
        response.sendRedirect("tickets");
        return;
    }

    String username = request.getParameter("username");
    String password = request.getParameter("password");
    if(username == null || password == null ||
        !LoginServlet.userDatabase.containsKey(username) ||
        !password.equals(LoginServlet.userDatabase.get(username)))
    {
        request.setAttribute("loginFailed", true);
        request.getRequestDispatcher("/WEB-INF/jsp/view/login.jsp")
            .forward(request, response);
    }
    else
    {
        session.setAttribute("username", username);
        request.changeSessionId();
        response.sendRedirect("tickets");
    }
}

```

There's not a lot new in the `doPost` method. It again makes sure that the user isn't already logged in, and then checks the `username` and `password` against the "database." If the login failed it sets the `loginFailed` request attribute to `true` and sends the user back to the login JSP. If the credentials match, it sets the `username` attribute on the session, changes the session ID, and then redirects the

user to the ticket screen. The `changeSessionId` method (code in bold) is a new feature in Servlet 3.1 from Java EE 7 that protects against the session fixation attacks you read about earlier in the chapter by migrating the session (changing the session ID).

Creating the Login Form

Next create `/WEB-INF/jsp/view/login.jsp` and put a login form in it:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Customer Support</title>
  </head>
  <body>
    <h2>Login</h2>
    You must log in to access the customer support site.<br /><br />
    <%
      if(((Boolean)request.getAttribute("loginFailed")))
      {
        %>
        <b>The username or password you entered are not correct. Please try
        again.</b><br /><br />
        <%
      }
    %>
    <form method="POST" action=<c:url value="/login" />>
      Username<br />
      <input type="text" name="username" /><br /><br />
      Password<br />
      <input type="password" name="password" /><br /><br />
      <input type="submit" value="Log In" />
    </form>
  </body>
</html>
```

This simple page writes out a login form to the screen and, using the `loginFailed` attribute, notifies users when their login credentials were rejected. Together with the `LoginServlet`, it completes the simple login feature. However, this doesn't stop users from accessing the ticket screens. You need to add a check in the `TicketServlet` to make sure users are logged in before displaying ticket information or letting them post tickets. This is easily accomplished by adding the following code to the top of the `doGet` and `doPost` methods in the `TicketServlet`:

```
if(request.getSession().getAttribute("username") == null)
{
  response.sendRedirect("login");
  return;
}
```

Now that users log in before creating tickets, your code already has access to their names when they create new tickets. This means you don't need the name field on the ticket form anymore. In the `TicketServlet`'s `createTicket` method, change the current code, which sets the ticket's customer name using the `request` parameter, so that it now uses the username from the session as shown in the following code. You can also remove the "Your Name" (`customerName`) input field from `/WEB-INF/jsp/view/ticketForm.jsp`.

```
        ticket.setCustomerName(
            (String) request.getSession().getAttribute("username")
        );
    
```

Testing the Log In

Now that your application requires logins, follow these steps to test it:

1. Compile the project and debug it using your IDE.
2. Navigate to the application in your browser (<http://localhost:8080/support/>) and you should immediately be taken to the login page.
3. Try logging in with incorrect usernames and passwords (both of which are case-sensitive) and you should be denied entry.
4. Try a valid username and password, and you should land on the list of tickets.
5. Create a few tickets like you did in previous chapters, and your username should be attached to them.
6. Close your browser, re-open it, and log back in using a different username and password.
7. Create another ticket and you can see that the new ticket has the name of the user you're currently logged in as, while the old tickets have the other user's name.

Adding a Logout Link

When testing, you had to close your browser to log out of the Customer Support application. This may not be desirable and is not the hallmark of an enterprise application. Adding a logout link is trivial enough. First, tweak the code at the top of the `LoginServlet`'s `doGet` method to add support for logging the user out:

```
HttpSession session = request.getSession();
if(request.getParameter("logout") != null)
{
    session.invalidate();
    response.sendRedirect("login");
    return;
}
else if(session.getAttribute("username") != null)
{
    response.sendRedirect("tickets");
    return;
}
```

The only other thing you need to do is add a logout link to the top of the `listTickets.jsp`, `ticketForm.jsp`, and `viewTicket.jsp` files in `/WEB-INF/jsp/view`, just above the `<h2>` headers:

```
<a href=<c:url value="/login?logout" />>Logout</a>
```

Now rebuild and run again, and log in to your application. You should see a logout link on top of every page. Click the logout link and you will return to the login page, indicating that you have successfully been logged out.

Detecting Changes to Sessions Using Listeners

One of the more useful features of sessions in Java EE is the idea of session events. When changes are made to sessions (for example, session attributes are added or removed), the web container can notify your application of these changes. This is achieved through a form of the publish-and-subscribe model, enabling you to decouple the code in your application that needs to be aware of session changes from the code that makes changes to sessions. This is especially useful if some third-party code — such as Spring Framework or Spring Security — makes changes to sessions in your application because it enables you to detect these changes without changing the third-party code. The tools that you use to detect these changes are called listeners.

Several listeners are defined in the Servlet API and most, though not all of them, listen for some form of session activity. You subscribe to an event by implementing the listener interface corresponding to that event and then (in most cases) either adding a `<listener>` configuration to your deployment descriptor or (as of Servlet 3.0/Java EE 6) annotating the class with `@javax.servlet.annotation.WebListener` (but not both).

You may implement as few or as many listener interfaces as you need in a single class; although of course, you wouldn't want to put code that didn't logically belong together in the same class. When something happens that triggers the publication of an event to which your code is subscribed, the container invokes the method on your class corresponding to that event.

NOTE *Starting in Servlet 3.0/Java EE 6, instead of annotating a listener class with `@WebListener` or declaring it in your deployment descriptor you can programmatically register it using `ServletContext`'s `addListener` method. You can only call this method within a `ServletContextListener`'s `contextInitialized` method or a `ServletContainerInitializer`'s `onStartup` method. Of course, any `ServletContextListener` you use to do this has to be registered as well (using one of these three approaches). You learn more about `ServletContainerInitializers` in Chapter 12.*

One of the listener interfaces you can implement is the `javax.servlet.http.HttpSessionAttributeListener` interface. It has three methods that are notified when session attributes are added, updated (replaced) or removed.

A particularly interesting listener is `javax.servlet.http.HttpSessionBindingListener`. Unlike most other listeners, you do not add deployment descriptor configurations for or annotate `HttpSessionBindingListener`s. If a class implements this interface, it becomes aware of its status as a session attribute. For example, if class `Foo` implements `HttpSessionBindingListener` and you add an instance of `Foo` to an `HttpSession` using `setAttribute`, the container calls that instance's `valueBound` method. Likewise, the container calls the instance's `valueUnbound` method when you remove it from the session using `removeAttribute`.

The two listeners you look at more closely in this section are `HttpSessionListener` and `HttpSessionIdListener` in the `javax.servlet.http` package. Create a `SessionListener` class in your project that implements both of these interfaces and annotate it with `@WebListener` (or follow along in the Customer-Support-v3 project):

```

@WebListener
public class SessionListener implements HttpSessionListener, HttpSessionIdListener
{
...
}

```

`@WebServlet` is not the only way to notify the container that your code is subscribing to these events. You could instead register it programmatically or declare the listener in the deployment descriptor as follows (though the example will stick to the annotation because it is the easiest technique).

```

<listener>
    <listener-class>com.wrox.SessionListener</listener-class>
</listener>

```

The `HttpSessionListener` interface defines the `sessionCreated` and `sessionDestroyed` methods. `sessionCreated`, intuitively, is called whenever a new session is created. `sessionDestroyed` is called whenever something causes the session to no longer be valid. This could be an explicit call to the session's `invalidate` method in code, or it could be an implicit invalidation due to an inactivity timeout. The following code implements these methods:

```

@Override
public void sessionCreated(HttpSessionEvent e)
{
    System.out.println(this.date() + ": Session " + e.getSession().getId() +
        " created.");
}

@Override
public void sessionDestroyed(HttpSessionEvent e)
{
    System.out.println(this.date() + ": Session " + e.getSession().getId() +
        " destroyed.");
}

```

As you can see, you use these events to log when a session is created or destroyed. This is a common use case for this particular listener because often administrators want to log this information in some way for record-keeping purposes. `HttpSessionIdListener` defines only one method, `sessionIdChanged`. This method, called whenever the session ID is changed using the request's `changeSessionId` method, is implemented in the following code:

```

@Override
public void sessionIdChanged(HttpSessionEvent e, String oldSessionId)
{
    System.out.println(this.date() + ": Session ID " + oldSessionId +
        " changed to " + e.getSession().getId());
}

```

All three of these methods use a simple helper method to add a timestamp to the session activity log entries.

```

private SimpleDateFormat formatter =
    new SimpleDateFormat("EEE, d MMM yyyy HH:mm:ss");
...

```

```
private String date()
{
    return this.formatter.format(new Date());
}
```

Now compile, debug, and navigate to your application. Immediately a logging message should appear in your debug window indicating that a session was created. Log in to the application, and you should observe another logging message that the session ID changed. This is the session fixation protection that you added to the project a few pages ago. Finally, when you log out of the application, two more log entries appear — one indicating that your session was destroyed and another indicating that a new session was created (because you returned to the login page). You now have a mechanism to log session activity in your application.

NOTE *When you first start the debugger but before you open your browser, you may already see a logging message indicating that one or more sessions were destroyed. This is completely normal. Tomcat persists sessions to the filesystem when it is shut down so that the data in them is not lost and then attempts to restore the serialized sessions to memory when Tomcat starts back up. If the persisted sessions expired before Tomcat restored them, Tomcat notifies HttpSessionListeners that the sessions expired just as if Tomcat was never stopped. This is fairly standard behavior among web containers and can be disabled in most cases, but that is outside the scope of this book. Consult your container's documentation.*

Maintaining a List of Active Sessions

In addition to logging session activity, you can use the `HttpSessionListener` and `HttpSessionIdListener` to maintain a list of active sessions in the application, something the Servlet API specification does not provide for directly.

To accomplish this, start by creating the `SessionRegistry` class in Listing 5-3. This class is fairly simple. It maintains a static `Map` with session IDs as keys and corresponding session objects as values. This may seem inefficient at first, but remember that these session objects already exist in memory for another purpose. The session objects are not being duplicated; this class simply stores another set of references to them, which is a relatively lightweight thing to do compared to the potential memory footprint of the session objects themselves. Because the class contains only static methods, its constructor is private to prevent instantiation.

LISTING 5-3: `SessionRegistry.java`

```
public final class SessionRegistry
{
    private static final Map<String, HttpSession> SESSIONS = new Hashtable<>();
```

continues

LISTING 5-3 (continued)

```

public static void addSession(HttpSession session)
{
    SESSIONS.put(session.getId(), session);
}

public static void updateSessionId(HttpSession session, String oldSessionId)
{
    synchronized(SESSIONS)
    {
        SESSIONS.remove(oldSessionId);
        addSession(session);
    }
}

public static void removeSession(HttpSession session)
{
    SESSIONS.remove(session.getId());
}

public static List<HttpSession> getAllSessions()
{
    return new ArrayList<>(SESSIONS.values());
}

public static int getNumberOfSessions()
{
    return SESSIONS.size();
}

private SessionRegistry() { }

}

```

This registry stores references to all the active sessions, but you must add and remove sessions somehow. For that, follow these steps:

1. Expand the `SessionListener` you created earlier. Add the following code to the `sessionCreated` method:

```
SessionRegistry.addSession(e.getSession());
```

2. Add the following code to the `sessionDestroyed` method:

```
SessionRegistry.removeSession(e.getSession());
```

3. Add the following code to the `sessionIdChanged` method:

```
SessionRegistry.updateSessionId(e.getSession(), oldSessionId);
```

Now sessions will be added to and removed from your registry at the appropriate times, but you still need a way to display these sessions. A simple `SessionListServlet` handles the request:

```

@WebServlet(
    name = "sessionListServlet",
    urlPatterns = "/sessions"
)

```

```
public class SessionListServlet extends HttpServlet
{
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException
    {
        if(request.getSession().getAttribute("username") == null)
        {
            response.sendRedirect("login");
            return;
        }

        request.setAttribute("numberOfSessions",
            SessionRegistry.getNumberOfSessions());
        request.setAttribute("sessionList", SessionRegistry.getAllSessions());
        request.getRequestDispatcher("/WEB-INF/jsp/view/sessions.jsp")
            .forward(request, response);
    }
}
```

The code for `/WEB-INF/jsp/view/sessions.jsp`, which takes care of displaying the sessions, is contained in Listing 5-4.

To test this you need two different Internet browsers (not just two windows of the same browser):

1. Rebuild and debug your application, and open the first browser to the support application URL.
2. After logging in, navigate to `http://localhost:8080/support/sessions`. You should see your current session listed in the list of sessions.
3. Open the second browser, log in to the support application, and navigate to `http://localhost:8080/support/sessions` in that browser as well. You should see a screen similar to the one in Figure 5-4.
4. Reload the first browser you opened, and the new session should appear there too. This means you are successfully maintaining a list of sessions.

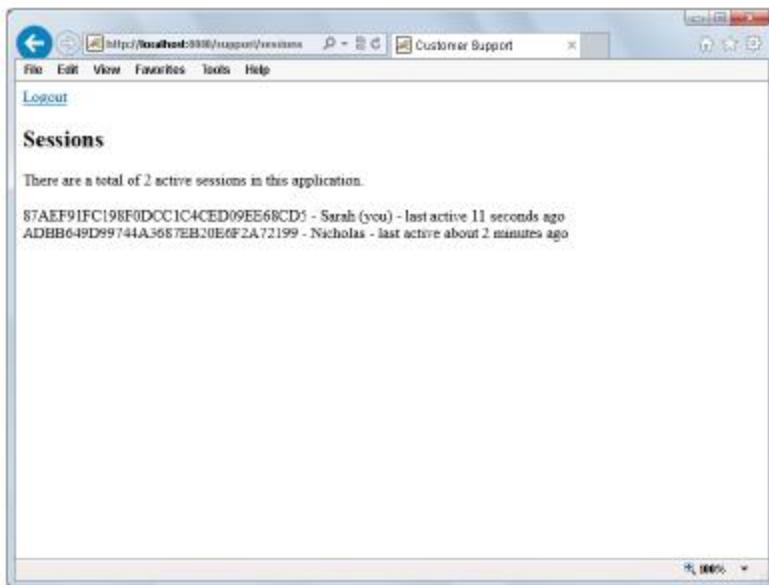


FIGURE 5-4

LISTING 5-4: sessions.jsp

```

<%@ page import="java.util.List" %>
<%
    private static String toString(long timeInterval)
    {
        if(timeInterval < 1_000)
            return "less than one second";
        if(timeInterval < 60_000)
            return (timeInterval / 1_000) + " seconds";
        return "about " + (timeInterval / 60_000) + " minutes";
    }
%>
<%
    int numberOfSessions = (Integer)request.getAttribute("numberOfSessions");
    @SuppressWarnings("unchecked")
    List< HttpSession > sessions =
        (List< HttpSession >)request.getAttribute("sessionList");
%>
<!DOCTYPE html>
<html>
    <head>
        <title>Customer Support</title>
    </head>
    <body>
        <a href=<c:url value="/login?logout" />>Logout</a>
        <h2>Sessions</h2>
        There are a total of <%= numberOfSessions %> active sessions in this
        application.<br /><br />
        <%
            long timestamp = System.currentTimeMillis();
            for(HttpSession aSession : sessions)

```

```

    {
        out.print(aSession.getId() + " - " +
                  aSession.getAttribute("username"));
        if(aSession.getId().equals(session.getId()))
            out.print(" (you)");
        out.print(" - last active " +
                  toString(timestamp - aSession.getLastAccessedTime()));
        out.println(" ago<br />");
    }
%>
</body>
</html>

```

NOTE *The sessions listed in this example are only the ones in the currently running instance of Tomcat. If your application was deployed to multiple Tomcat instances, you would see different sessions listed, depending on which Tomcat instance your request to the application landed on, because the page would still list only sessions on that particular Tomcat instance. The solution to this problem involves properly configuring your application for clustering and setting up session replication in your container. These topics are explored in the next section.*

CLUSTERING AN APPLICATION THAT USES SESSIONS

In the time you spend working with enterprise applications, you will undoubtedly come across the need to cluster an application. Clustering provides several advantages, notably adding redundancy and scalability to your application. Properly clustered applications can suffer failures or even endure routine maintenance without end users ever experiencing downtime. In a very well-managed environment, administrators can even roll out upgrades to applications without causing downtime. As you can tell, clustering is an invaluable member of the web application toolset.

Clustering does not come without its downsides, however, and there are challenges that must be overcome. One of the biggest of these challenges is the passing of messages between instances of an application running on separate machines, sometimes even on disparate or disconnected networks or in different regions of the world. For decades engineers have been re-imagining and redesigning cluster messaging systems, constantly searching for that “perfect” messaging framework that is stable, reliable, and fast. Advanced Message Queuing Protocol (AMQP), Java Message Service (JMS), and Microsoft Message Queuing (MSMQ) are three competing technologies that have emerged as a result. Of course, there are other challenges with application clustering than just messaging, and the one you look at in this section is managing sessions in a cluster.

Understanding this section requires you to have some basic knowledge of what load balancing is, how it works, and what some of the common load balancing strategies are. These are topics that would require considerable time to discuss and are outside the scope of this book.

Using Session IDs in a Cluster

The immediate problem you might see with session clustering is that sessions exist as objects in memory and as such only reside on a single instance of a web container. In a purely round-robin or load-smart load balancing scenario, two consecutive requests from the same client may go to

different web containers. The first web container instance would assign a session ID to the first request it received, and then when the next request came in to a different instance of the web container, the second instance would not recognize the session ID and would create and assign a new session ID. At this point, sessions would be useless.

One solution to this problem is to employ sticky sessions. The idea of sticky sessions is that the load balancing mechanism is session-aware and always sends a request from the same session to the same server. This can be accomplished in a number of ways and depends largely on the load balancing technology. For example, the load balancer may be made aware of the web container's session cookie and know that it is a session cookie, therefore using it as a mechanism for determining when requests should go to the same server. Or some load balancers can add their own session cookies to responses and recognize those cookies in subsequent requests. (Yes, a single request can belong to many different sessions, as long as the session cookie names or session ID transmission techniques are all different.)

A potential downside to both of these techniques is that the web container cannot use SSL/HTTPS because that would prevent the load balancer from inspecting or modifying requests or responses. However, many load balancers support handling the encryption and decryption of HTTPS traffic, so you haven't really made your application less secure; you've just moved the encryption mechanism from the server to the load balancer. (Some organizations even prefer this setup, but remember that it prevents you from using SSL Session IDs as your HTTP session IDs.) Finally, some load balancers use a combination of source and destination IP addresses to determine when to send multiple requests to the same server, but this can be troublesome for the same reason that using IP addresses to establish HTTP sessions is a bad idea.

The most common load balancing approach administrators of a Tomcat environment take is to use an Apache HTTPD or Microsoft IIS web server to load balance requests between Apache Tomcat instances. The Apache Tomcat Connector <http://tomcat.apache.org/connector-doc/> provides a mechanism for interfacing these web servers with Tomcat. The connector's mod_jk component is an Apache HTTPD module that forwards requests to Tomcat and provides sticky sessions capability using Tomcat's session IDs. Likewise, isapi_redirect is the IIS connector that provides the same capability when using IIS. As load increases even more, you can set up a dumb round-robin load balancer to balance requests between multiple HTTPD or IIS web servers.

This multi-layer approach, demonstrated in Figure 5-5, can achieve extremely high performance and availability while maintaining session affinity. The connector (mod_jk or isapi_redirect) uses a Tomcat concept known as the session ID *jvmroute* to determine which Tomcat instance to send each request to. Consider the following session ID:

AA64E92624FFEA976C4148DF5BC6BA03

In a load-balanced environment with multiple Tomcat instances, each Tomcat instance would have a *jvmroute* configured in the `<Connector>` element in Tomcat's `conf/server.xml` configuration file. That *jvmroute* is appended to the end of all session IDs. In a cluster with three Tomcat instances having *jvmroutes* `tcin01`, `tcin02` and `tcin03`, that same session ID would instead look like this if the session originated on instance `tcin02`:

AA64E92624FFEA976C4148DF5BC6BA03.tcin02

From then on the web server connector (mod_jk or isapi_redirect) would recognize that this session belonged to Tomcat instance `tcin02` and would always send requests in that session to that instance. If your application were secured with HTTPS, the web server would have to be in charge of certificates and encryption/decryption for this to work. The advantage of using mod_jk or isapi_redirect for this is that they have access to the SSL Session ID and re-transmit that ID to Tomcat,

allowing SSL session tracking to work properly. This exact sticky-session load balancing approach also works with GlassFish behind Apache HTTPD/mod_jk and IIS/isapi_redirect.

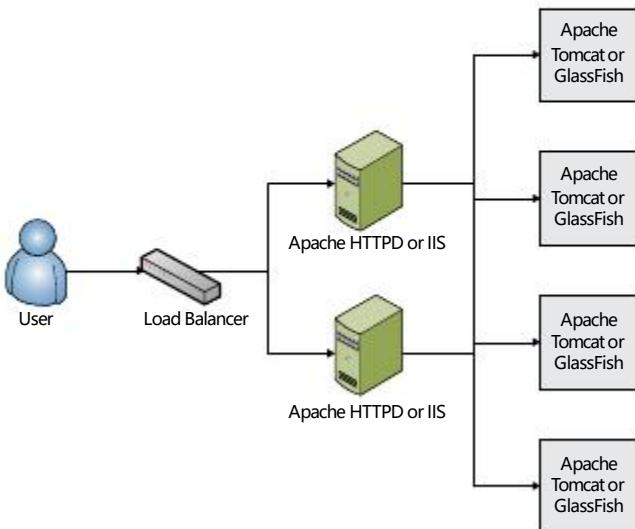


FIGURE 5-5

The exact details of configuring mod_jk, isapi_redirect, and Tomcat's and GlassFish's jvmroute are outside the scope of this book and vary from one version to the next. Consult the Tomcat and GlassFish documentation for instructions. WebLogic, WebSphere, and other containers offer similar but ultimately different approaches that are covered in detail in their documentation as well.

Understand Session Replication and Failover

The major problem with using sticky sessions is that it may support scalability, but it does not support high availability. If the Tomcat instance that created a particular session goes down, the session is lost and the user must log in again. Even worse, the user could potentially lose unsaved work. For this purpose sessions can be replicated throughout the cluster so that all sessions are available to all web container instances regardless of the instances from which they originated. Enabling session replication in your application is easy to accomplish. You just need to add the `<distributable>` tag to the deployment descriptor:

```
<distributable />
```

That's all there is to it. There are no attributes, nested tags, or content for this tag. The presence of this in the deployment descriptor tells the web container to replicate sessions across the cluster, if one exists. When a session is created in one instance, it is replicated to the other instances. If a session attribute is changed, that session is re-replicated to the other instances so that they have the latest version of the session.

Of course, it isn't *actually* this simple. For instance, this only *marks* your application as supporting distributable sessions. It does not configure your web container's session replication mechanism (which is a complex topic not discussed in this book). It also does not automatically mean your application follows best practices. You must be careful which session attributes you set (if they are not `Serializable`, an `IllegalArgumentException` is thrown when you call `setAttribute`) and how you update those session attributes. Consider this code snippet from the Shopping-Cart project:

```
@SuppressWarnings("unchecked")
Map<Integer, Integer> cart =
    (Map<Integer, Integer>) session.getAttribute("cart");
if(!cart.containsKey(productId))
    cart.put(productId, 0);
cart.put(productId, cart.get(productId) + 1);
```

The web container does not (and cannot) know that the Map containing the cart items has changed in this way. Because of this, the change to the session will not be replicated, which means that other container instances cannot know about the new item in the cart. This can be addressed simply:

```
@SuppressWarnings("unchecked")
Map<Integer, Integer> cart =
    (Map<Integer, Integer>) session.getAttribute("cart");
if(!cart.containsKey(productId))
    cart.put(productId, 0);
cart.put(productId, cart.get(productId) + 1);
session.setAttribute("cart", cart);
```

Notice the code in bold that has been added. This may seem silly because you replaced the `cart` session attribute with the same object that was already assigned to it. However, calling this method tells the container that the session has changed and causes the session to be replicated again. Any time you change an object assigned to a session attribute, you must call `setAttribute` again to ensure the change is replicated.

There is also a listener associated with the concept of session replication. Any objects added to sessions as attributes can implement the `javax.servlet.http.HttpSessionActivationListener` interface. When a session is about to be serialized to replicate to other servers, the `sessionWillPassivate` method is called, giving the object bound to the session an opportunity to perform some action first. When the session is deserialized in another container, the `sessionDidActivate` method is called to notify the attribute that it has been serialized.

One final note: Sticky sessions and session replication are not mutually exclusive concepts. Often the two are combined to achieve session failover — sessions are still replicated, but requests in the same session are sent to the same instance until that instance fails, at which point the requests are sent to a different instance that already knows about the session. You can use several techniques to increase the efficiency of your application using sticky session failover, but they are outside the scope of this book. The documentation for your web container should describe the replication features it supports and how to use them.

SUMMARY

In this chapter you have been introduced to the concept of sessions and how sessions are established between the client and server. You learned about some of the many potential security vulnerabilities associated with sessions and how each of them can be addressed, and you also learned about the most secure session ID transmission method of all: using the SSL session ID. You explored employing sessions in Java EE using a shopping cart application and added login support to the Customer Support application. You also discovered how to detect changes to sessions and used that to establish a registry of sessions within your application. Finally, you were introduced to the concepts behind clustering sessions and learned about some of the challenges and approaches to session clustering. In the next three chapters, you explore some technologies that make working with JSPs easier than ever before and help you get rid of Java within JSPs for good.

6

Using the Expression Language in JSPs

IN THIS CHAPTER

- All about Expression Language
- How to write with the EL syntax
- How to use scoped variables in EL expressions
- How to access collections with Java 8 streams in EL expressions
- Switching out Java code with Expression Language

WROX.COM CODE DOWNLOADS FOR THIS CHAPTER

You can find the [wrox.com](http://www.wrox.com) code downloads for this chapter at www.wrox.com/go/projavaforwebapps on the Download Code tab. The code for this chapter is divided into the following major examples:

- User-Profile Project
- Customer-Support-v4 Project

NEW MAVEN DEPENDENCIES FOR THIS CHAPTER

In addition to Maven dependencies introduced in previous chapters, you also need the following Maven dependency:

```
<dependency>
    <groupId>javax.el</groupId>
    <artifactId>javax.el-api</artifactId>
    <version>3.0.0</version>
    <scope>provided</scope>
</dependency>
```

UNDERSTANDING EXPRESSION LANGUAGE

Up to this point, you have used Java to output dynamic content from your JSPs. However, recall that Chapter 4 covers how the use of declarations, scriptlets, and expressions is discouraged. Not only does this provide a great deal of power (sometimes too much) to your JSPs, but it also makes writing JSPs difficult for UI developers with little or no Java background. There must be an easier way to display data and perform simple operations than using Java code. You might think that the `<jsp>` tags could provide a solution, and indeed these tags can be used to replace certain Java operations. However, these tags are clunky and awkward to use. What's needed is something easily read, familiar to both Java developers and UI developers, and with a simple set of rules and operators to make data access and manipulation easier.

What It's For

Expression Language (EL) was originally developed as part of the JSP Java Standard Tag Library (JSTL), something you learn about in the next chapter, to support the rendering of data on JSP pages without the use of scriptlets, declarations, or expressions. It was inspired by and largely based on the ECMAScript (the foundation of JavaScript) and XPath languages. At the time it was referred to as the Simplest Possible Expression Language (SPEL) but later shortened to Expression Language. EL was part of the JSTL 1.0 specification that came out with JSP 1.2 and could be used only in attributes of JSTL tags. In JSP 2.0 and JSTL 1.1 the EL specification (due to its popularity) was moved from the JSTL specification to the JSP specification and became available for use anywhere in a JSP, not just within JSTL tag attributes.

While this was happening, work had commenced on JavaServer Faces, built on JSP 1.2 as an alternative to plain JSP. JSF also needed its own expression language. However, there were several drawbacks to reusing the EL as it existed for JSPs. For one, JSF needed to control the evaluation of expressions to certain points of the JSF life cycle. An expression might need to be evaluated during page rendering but also during a postback to the JSF page. In addition, JSF needed better support for method expressions than the EL offered. As a result, two separate but extremely similar expression languages formed — one for JSP 2.0 and one for JSF 1.0.

Obviously having two separate Java expression languages was not ideal, so when work began on the JSP 2.1 specification, an effort was underway to merge the JSP 2.0 Expression Language with the JSF 1.1 Expression Language. The result was the Java Unified Expression Language (JUEL) for JSP 2.1 and JSF 1.2.

Despite being shared by JSP and JSF, EL did not get its own JSR but continued to be a part of the JSP specification, although it did have its own specification document and JAR artifact. This remained the case for EL in JSP 2.2. EL continues to expand and improve, and as of Java EE 7 it was moved into its own JSR (JSR 341) and updated to support lambda expressions and an equivalent of the Java 8 Collections Stream API, marking Java Unified Expression Language 3.0 (or EL 3.0 for short). EL 3.0 was released with Java EE 7, Servlet 3.1, JSP 2.3, and JSF 2.2 in 2013. In this chapter you explore EL 3.0 as it pertains to JSPs, learning about JSF-related features only where pertinent comparisons can be made. Most of the chapter centers on syntax, and where features that are new to EL 3.0 are demonstrated, this is indicated.

Understanding the Base Syntax

The base syntax for EL delineates expressions that require evaluation from the rest of the JSP page syntax. The JSP interpreter must detect when an EL expression begins and ends so that it can parse and evaluate the expression separately from the rest of the page. There are two different types of the base EL syntax: *immediate evaluation* and *deferred evaluation*.

Immediate Evaluation

Immediate evaluation EL expressions are those that the JSP engine should parse and evaluate at the time of page rendering. This means that as the JSP code is being executed from top to bottom, the EL expression is evaluated as soon as the JSP engine comes across it and before the execution of the rest of the page continues. EL expressions that should be immediately evaluated look like the following example, where `expr` is a valid EL expression.

```
 ${expr}
```

The dollar sign and opening and closing brackets define the boundaries of the EL expression.

Everything inside the brackets gets evaluated as an EL expression. More important, this means that you can't use this syntax for any other purpose in your JSPs; otherwise, it will get evaluated as an EL expression and could result in an EL syntax error. If you ever needed to write something with this syntax out to the response, you would need to escape the dollar sign:

```
\${not an EL expression}
```

The backslash before the dollar sign indicates to the JSP engine that this is not, in fact, an EL expression and should not be evaluated. The previous example would have literally been written to the response as `\${not an EL expression}`. You could also have used the dollar sign XML entity `$` instead of `\$` and it would have resulted in the same outcome.

```
&#36;{not an EL expression}
```

Although the JSP engine would also ignore this, many find using the backslash easier. It's simply a matter of personal preference. Of course, you might legitimately need to put a backslash before an expression that you do, actually, want evaluated. This *requires* the use of the backslash XML entity:

```
&#92;${EL expression to evaluate}
```

In this case, the EL expression will be evaluated and rendered after the backslash.

Deferred Evaluation

Deferred evaluation EL expressions are a part of the Unified Expression Language that primarily supports the needs of JavaServer Faces. Although the deferred syntax is legal in JSPs, it is not normally seen in JSPs. Deferred syntax looks nearly identical to immediate syntax, again where `expr` is a valid EL expression:

```
#${expr}
```

In JSF, deferred expressions can be evaluated either when the page is rendered or during a postback to the page, or possibly even both. The specifics of this are not pertinent to this book, but you must understand that this is different from JSP, which does not have a sense of life cycles that JSF has.

In JSP, the `#{}` deferred syntax, which is only valid in JSP tag attributes, can be used to defer the evaluation of the EL expression until later in the rendering process *of the tag*. Instead of the EL expression being evaluated before the attribute value is bound to the tag (like it would be with `${}`), the tag attribute gets a reference to the unevaluated EL expression. The tag can then later invoke a method to evaluate the EL expression when it's appropriate. This can be useful and is explored more in Chapter 8, but it is rarely used.

One potential problem with deferred syntax is that some templating languages and JavaScript frameworks use the `#{}` syntax for substitutions. Because of this, if you use these substitutions, you would normally have to escape them so that they aren't confused with deferred evaluation EL expressions:

```
\#{not an EL expression}  
&#35;{also not an EL expression}
```

However, this may not work for some frameworks that utilize this syntax, and it can be a real pain if you need to use this often or if you have a lot of existing JSPs that need to work with EL 2.1 or higher. (Also, the XML entity isn't compatible with JavaScript.) Because of this, there is another option for preventing a `#{}` literal from being evaluated as a deferred expression. Within the `<jsp-config>` section of the deployment descriptor, you can add the following tag to any `<jsp-property-group>`:

```
<deferred-syntax-allowed-as-literal>true</deferred-syntax-allowed-as-literal>
```

This permits the `#{}` syntax to be used in a literal manner and prevents you from having to escape the hash tag in this case. If you need to control this for individual JSPs, you can use the `deferredSyntaxAllowedAsLiteral="true"` attribute of the `<page>` directive in any JSP, instead.

For the remainder of this book, you will only see immediate evaluation EL syntax in example code and you will not use deferred evaluation EL syntax, with one exception. In the Chapter 8 discussion on custom tag and function libraries, you'll explore the `<deferred-value>` and `<deferred-method>` options when defining custom tags. This also necessitates demonstrating the deferred syntax.

Placing EL Expressions

Simply put, EL expressions can be used just about anywhere in a JSP, with a few minor exceptions. To start, EL expressions cannot be used within any directives, so don't even try it. Directives (`<%@ page %>`, `<%@ include %>`, and `<%@ taglib %>`) are evaluated when the JSP is compiled, but EL expressions are evaluated later when the JSP is rendered, so it cannot work. Also, EL expressions are not valid within JSP declarations (`<%! %>`), scriptlets (`<% %>`), or expressions (`<%= %>`). If used within any of these, an EL expression will simply be ignored or, worse, could result in a syntax error.

Other than that, EL expressions can be placed just about anywhere. One place you might see EL expressions is within simple literal text written to the screen:

The user will see `${expr}` text and will know that `${expr}` is good.

This example includes two EL expressions that, when evaluated, are placed inline with the text that displays. If the first expression evaluated to “red” and the second expression evaluated to “it,” the user would see the following:

The user will see red text and will know that it is good.

In addition, expressions can be used within standard HTML tag attributes as in the following example.

```
<input type="text" name="something" value="${expr}" />
```

HTML tag attributes are not the only place that EL expressions are allowed. You can also use them in JSP tag attributes, as demonstrated with the following code.

```
<c:url value="/something/${expr}/${expr}" />
<c:redirect url="${expr}" />
```

As you can see, EL expressions do not have to make up the entire attribute value. Instead, any one or more parts of the attribute value can include EL expressions. You might wonder about other HTML features, such as JavaScript or Cascading Style Sheets. The JSP engine does not parse things of this nature and writes them out to the response as if they were literal text, so these, also, may contain EL expressions in either quoted or literal form:

```
<script type="text/javascript" lang="javascript">
    var employeeName = '${expr}';
    var booleanValue = ${expr};
    var numericValue = ${expr};
</script>
<style type="text/css">
    span.error {
        color: ${expr};
        background-image: url('/some/place/${expr}.png');
    }
</style>
```

So far you have learned about the different types of EL expressions and where EL expressions can be placed, but you may wonder what exactly *expr* looks like. In the next section you learn about what you can put within an EL expression.

WRITING WITH THE EL SYNTAX

EL expressions, like any other language, have a specific syntax. Like Java, JavaScript, and most other languages, that syntax is strict, and violating it will result in syntax errors when your JSP is rendered. Unlike Java, however, EL syntax is loosely typed and has many implicit type conversions built in, similar to languages like PHP or JavaScript. The primary rule for an expression is that it should evaluate to some value. You cannot declare variables within an expression or perform some kind of assignment or operation that does not result in a value. (For example, \${object.method()} is only valid if `method` has a non-`void` return type.) EL is not designed to replace Java; instead, it is designed to provide you with the tools you need to create JSPs without Java.

NOTE *Although you cannot declare variables within an EL expression, you can assign variables as of the EL 3.0 specification. Using the standard assignment operator `=`, you can assign `A = B` within an expression as long as `B` is some value that can be written out to the page. So, the expression `${x = 5}` will result in assigning `5` to `x` and also in rendering `5` in place of the EL expression.*

Reserved Keywords

As with any other language, one of the first things you should know about EL is its list of reserved keywords. These are words that should be used only for their prescribed purpose. Variables, properties, and methods should have names equal to these reserved words.

- | | |
|--------------|-------|
| ➤ true | ➤ or |
| ➤ false | ➤ not |
| ➤ null | ➤ eq |
| ➤ instanceof | ➤ ne |
| ➤ empty | ➤ lt |
| ➤ div | ➤ gt |
| ➤ mod | ➤ le |
| ➤ and | ➤ ge |

You'll recognize the first four words as also being Java reserved keywords. You can use these in the same manner you would use their counterparts in Java. The `empty` keyword is used to validate whether some `Collection`, `Map`, or `array` contains any values, or whether some `String` has a length of one or more characters. If any of these are `null` or "empty," the expression evaluates to `true`; otherwise, it evaluates to `false`.

`${empty x}`

The `div` and `mod` keywords map to the Java mathematical operations `divide (/)` and `modulus (%)`, respectively, and are merely alternatives to the mathematical symbols. You can still use `/` and `%` if you prefer. The `and`, `or`, and `not` keywords map to the Java logical operators `&&`, `||`, and `!`, respectively. As with the mathematical operators, you can still use traditional logical operators if you prefer. Finally, the `eq`, `ne`, `lt`, `gt`, `le`, and `ge` operators are alternatives to the Java relational operators `==`, `!=`, `<`, `>`, `<=`, and `>=`, respectively, which can also still be used if you prefer.

Operator Precedence

Just like with other languages, all the previous operators, together with other operators in the EL, have an order of precedence that is important to understand. This order is mostly intuitive and

not dissimilar from operator precedence in Java. More important, as with Java and arithmetic equations, operators of equal precedence are considered in the order they appear in an expression, from left to right.

The first operators evaluated in an EL expression are the bracket [] and dot (.) resolution operators. Consider the following expression:

```
 ${myCollection["key"].memberName["anotherKey"]}
```

The engine first resolves the value mapped to `key` in the `myCollection` object. It then resolves the `memberName` method, field, or property within the `key` value found in `myCollection`. Finally, it locates the `anotherKey` value within the value that `memberName` evaluates to. After these operators are considered, the grouping parentheses operators () are considered. These operators are used to change the precedence of other operators, just as they are in Java or arithmetic equations.

The third set of operators considered includes the unary negative sign (-), `not`, `!`, and `empty`. Next, the EL engine evaluates the arithmetic operators `multiply` (*), `divide` (/), and `div`, and `modulus` (%) and `mod`, which are followed by the addition (+) and binary subtraction (-) operators, just like they are ordered in mathematical equations. After this the EL string concatenation operator `+=` (new to EL 3.0) is evaluated. Next, it evaluates the comparison relational operators `<` (or `lt`), `>` (or `gt`), `<=` (or `le`), and `>=` (or `ge`), followed by the equality relational operators `==` (or `eq`) and `!=` (or `ne`). After this, it evaluates all the `&&` and `and` operators from left to right, then all the `||` and `or` operators from left to right, and then all the `?` and `:` conditional operators from left to right.

The next thing that the EL engine evaluates is the lambda expression operator (->), new in the EL 3.0 specification. This has the same syntactic and semantic purpose as the Java 8 lambda expression operator. However, you do not need to be running on Java 8 for EL lambda expressions to be valid. After this, the EL engine evaluates the assignment = operator, which was also added in the EL 3.0 specification. This operator assigns the value of some expression on the right side of the operator to the variable on the left side of the operator. The resulting value of the expression is then the value of the variable on the left side of the operator. Consider the following expression.

```
 ${x = y + 3}
```

Now assume that, at execution time, the value of `y` is 4. The result of the expression `y + 3` is 7, thus 7 is assigned to `x`. Because the resulting value of the expression is `x`, the value of `${x = y + 3}` is 7.

The final operator that the EL engine evaluates is the semicolon (;) operator, also a new feature in EL 3.0. This operator mimics the comma (,) operator in C, allowing the specification of several expressions with the values of all but the last expression being discarded. To understand this, refer to the following expression.

```
 ${x = y + 3; object.callMethod(x); 'Hello, World!'}
```

This combination EL expression has four expressions within it.

- The expression `y + 3` is evaluated, resulting in 7 assuming `y` is 4.
- That value is assigned to `x`.

- The `callMethod` method is invoked on the `object` variable and passed `x` (7) as its argument.
- The string literal "Hello, World!" is evaluated. The result of this expression is the result of the expression after the last semicolon only: "Hello, World!"

The results of the `x = y + 3` expression and the `object.callMethod(x)` expression are discarded. This is especially useful to assign some value to an EL variable and then include that value in some other part of the expression instead of just outputting the value.

To help you keep all this straight, the following list summarizes the order of precedence from top (highest) to bottom (lowest) with only the symbols and none of the clutter. Remember that operators with the same precedence are evaluated in the order they appear in an expression, from left to right.

```
[], .
()
unary -, !, not, empty
*, /, div, %, mod
+ math, binary -
+= string
<, lt, >, gt, <=, le, >=, ge
==, eq, !=, ne
&&, and
||, or
?, :
->
=
;
```

NOTE *In Java, you test for equality between objects using the `equals` method.*

For example, to test if two Strings were equal, you would use `"Hello".equals("Hello")`, not `"Hello" == "Hello"`. The latter tests that the two references are the same instance, not that the two objects are equal. However, in EL expressions you use the `==` or `eq` operators to test for object equality instead of calling the `equals` method. (There is no equivalent for testing if two references are the same in EL.) Likewise, you use `!=` or `ne` instead of `!"Hello".equals("Hello")`.

The use of the relational comparison operators `<`, `lt`, `>`, `gt`, `<=`, `le`, `>=`, and `ge` is similar to the equality operators. Any two objects that implement the `java.lang.Comparable` interface can be compared with comparison operators as long as the types are the same or one can be coerced to the other. So,

`${o1 >= o2} and ${o1 ge o2} in EL are equivalent to o1.compareTo(o2) >= 0 in Java, and ${o1 < o2} and ${o1 lt o2} are equivalent to o1.compareTo(o2) < 0.`

Literal Values

The Unified Expression Language has a support for specifying literal values with a specific syntax. You have already seen the `true`, `false`, and `null` keywords, which are all literal values.

In addition, EL can have string literal values. Unlike Java, where string literals are always surrounded by double quotes, string literals in EL can be surrounded by *either* double *or* single quotes, similar to PHP and JavaScript. So, both of the expressions in the following example are valid.

```
 ${"This string will be rendered on the user's screen."}
 ${'This string will also be "rendered" on the screen.'}
```

As you can see, there are advantages and disadvantages to using either type of string literal, and in many cases you will simply use the one that's easiest for the particular case. If some string has a single quote within it, it's probably easiest to use double quotes for the literal. Similarly, if the string has a double quote in it, it's probably easiest to use a single quote literal.

One thing you must be careful about, however, is using EL expression string literals within JSP tag attributes. Because these are both evaluated by the JSP engine, the quotes surrounding an attribute value and the quotes surrounding a string literal conflict. Thus, both of the EL expression attribute values in the following example are invalid and result in syntax errors:

```
<c:url value="${value}" />
<c:url value='${value}' />
```

There are two valid ways to address this conflict. You can either use opposite quote types for the attribute and literal, or you can escape the literal quotes. All four lines of code in the following example are valid.

```
<c:url value="${'value'}" />
<c:url value='${"value"}' />
<c:url value="${\"value\"}" />
<c:url value='${\\'value\\'}' />
```

Generally, you will find it is easier to simply use opposite quotes instead of escaping. But what if your string literal itself contains a single or double quote and you need to put the expression in an attribute value? There's no way around it at this point. You must escape *something*. The six lines of code in the following example are all valid ways of dealing with this.

```
<c:url value='${some \"value\"}' />
<c:url value='${some \'value\"}' />
<c:url value='${some \'value\'}' />
<c:url value='${some \"value\'}' />
<c:url value='${\"some 'value'\"}' />
<c:url value='${\\'some \"value\"\\'}' />
```

Need to mix and match single and double quotes within a string literal that exists within an attribute value? This is where things start to get hairy:

```
<c:url value='${some attribute\'s \"value\"}' />
<c:url value='${some \"attribute\" \'value\"}' />
```

As you can tell, this can quickly spiral out of control. Where possible, it's best to keep your string literals simple. The last thing to note about string literals is that, as of Expression Language 3.0 in Java EE 7, you can concatenate string literals within EL expressions somewhat like you do within Java. All three lines in the following example are equivalent and result in the same output.

```
The user will see ${expr} text and will ${expr}.\n${'The user will see ' += expr += " text and will " += expr += '.'}\n${"The user will see " += expr += ' text and will ' += expr += ".")
```

If `expr` results in some object that is not a `String`, it will be coerced to a `String` by calling the `toString` method on that object.

Numeric literals in EL are simplified over those in Java, and you can even perform arithmetic between certain objects that you could not in Java. Consider the following three integer-type numeric literals:

```
 ${105}  
 ${-132147483648}  
 ${139223372036854775807}
```

- The first literal is an implicit `int` and is treated like one when the expression is evaluated.
 - The second literal is too large to be an `int`. In Java, this would be a syntax error unless you appended an `L` to the end of the number to indicate it was a `long`, but in EL it simply becomes a `long` implicitly.
 - The third literal is too large to even be a `long`, so it is treated as a `BigInteger` implicitly.

All these conversions happen under the hood without your involvement. Then there are the decimal types:

Similar to the integer types, these literals are an implicit `float`, `double`, and `BigDecimal`, respectively. It should be noted that although the default literal decimal type in Java is `double`, the default decimal type in EL is `float` unless a larger precision is required. Keep this in mind when you work with EL expressions. You cannot explicitly specify the literal type — it is always handled implicitly.

EL expressions make mathematical operations much easier because all type conversions and precision upgrades are implicit and because the arithmetic operators can be used on `BigInteger` and `BigDecimal` types. Consider the following expression, which adds two numbers and returns the resulting value:

§{12 + 1.79769313486231570e+309}

The number on the left side of the addition operator is an `int`, whereas the number on the right side is an implicit `BigDecimal`. To do this in Java would normally require the following code:

```
new BigDecimal(12).add(new BigDecimal("1.79769313486231570e+309"));
```

However, the EL engine takes care of everything for you. First, it coerces `12` from an `int` to a `BigDecimal`; then it turns the addition operator into a call to the `add` method.

NOTE *In Java, numbers can be expressed as standard (base-10, 83) literals, octal (base-8, 0123) literals, hexadecimal (base-16, 0x53) literals, or binary (base-2, 0b01010011) literals. In EL expressions, only base-10 literals are permitted.*

There is no equivalent for literals in the other bases. Also, while underscores are permitted within numeric literals (1_491_188, 0b0101_0011) in Java to make it easier to distinguish groups of numbers in a literal (as a replacement for commas, for example), this is not permitted in EL expressions. Number literals must be contiguous.

Three other primitive literals to consider are `chars`, `bytes`, and `shorts`. You do not normally need to use these data types in EL expressions, but it is possible that some method you might call in an EL expression could expect a `char`, `byte`, or `short` as an argument. EL does not contain specific literals for these types but will coerce other literals into `chars`, `bytes`, and `shorts` when necessary.

For `chars`, a `null`, '' string literal, or "" string literal will be coerced into the null byte character (`0x00`). A single-character string literal (single or double quote) will be coerced into its equivalent `char`. An integer-type number will also be coerced into a `char` as long as its value is between 0 and 65,535. Any other type, any multicharacter string, or any number outside the range of 0 and 65,535 will result in an error.

Any integer-type number will also be coerced into a `byte` or `short` when necessary, as long as the number does not extend beyond the range of the `byte` or `short` it is being coerced to. Otherwise, the attempted coercion will result in an error.

The final literal type is not a primitive but rather a literal for creating various collections. Collection literals construction is a feature proposed as an improvement to the Java Collections API in Java 8 that did not make the final feature cut and instead was deferred to Java 9 (for now). It did make it into Expression Language 3.0, however. You can create a collection within an EL expression whenever needed. The syntax is rather intuitive, is quite similar to syntaxes in JavaScript and other languages, and is in line with the proposed syntax for Java 9. You can construct `Sets`, `Lists`, and `Maps` with EL collection literals, and they will all be constructed as instances of the default implementations. A literal `Set` will become a `HashSet<Object>`, a literal `List` will become an `ArrayList<Object>`, and a literal `Map` will become a `HashMap<Object, Object>`. Consider first the `Set` literal:

```
{1, 2, 'three', 4.00, x}
```

This constructs a `HashSet<Object>` with five elements of varying types. The fifth object, `x`, could be anything. Commas separate elements in the literal `Set`. You might need to create a `Set`, for example, to pass in as an argument to a method call:

```
 ${someObject.someMethod({1, 2, 'three', 4.00, x})}
```

Constructing a `List` is nearly identical to constructing a `Set` except that it uses brackets instead of braces, and it works exactly the same as arrays in JavaScript/JSON:

```
[1, 2, 'three', [x, y], {'foo', 'bar'}]
```

Notice that the fourth element of this `ArrayList<Object>` is another `List`, and the fifth element is a `Set`. You can nest collection literals in this manner to insert collection objects into other collection objects. As with `Sets`, elements in `Lists` are separated with commas.

The final collection literal, which creates a `HashMap<Object, Object>`, is identical to the object literal syntax in JavaScript and JSON:

```
{'one': 1, 2: 'two', 'key': x, 'list': [1, 2, 3]}
```

Elements here, too, are separated with commas. However, `Maps` are more complicated because they require keys mapped to values instead of just values. So each element in this literal is a pair of objects separated by a colon, with the object on the left of the colon being the key and the object on the right of the colon being the value. The `list` key in this literal is mapped to a `List` object with values 1, 2, and 3.

Object Properties and Methods

EL provides a simplified syntax for accessing properties in JavaBeans in addition to the standard syntax you are used to for accessing public accessor methods. You cannot access public fields from EL expressions. Consider a class named `Shirt` with a public field named `size`. Assuming a variable name of `shirt`, you might think that you could access `size` with the following EL expression:

```
 ${shirt.size}
```

However, this is not allowed. When the EL engine sees this syntax, it is looking for a property on `shirt`, not a field. But what is a property? Consider an altered `Shirt` where `size` is a properly encapsulated private field with standard JavaBean accessor and mutator methods `getSize` and `setSize`. Now the expression `shirt.size` becomes a shortcut for calling `shirt.getSize()`. This can work for any field of any type. As long as it has a standard JavaBean accessor method, it can be accessed in this way. If `Shirt` had a field named `styleCategory` with an accessor `getStyleCategory`, it could be accessed with `shirt.styleCategory`. For boolean fields (and only boolean fields) the accessor can start with either `get` or `is`. So for a field named `expired` with either a `getExpired` or `isExpired` accessor, you could access the field with `shirt.expired`.

This is not the only technique that you can use to access properties within a JavaBean. In the spirit of the ECMAScript and XPath languages, you can also access properties using the `[]` operator. The following expressions also access the `size`, `styleCategory`, and `expired` properties using the `getSize`, `getStyleCategory`, and `getExpired` or `isExpired` methods, respectively.

```
 ${shirt["size"]}  
 ${shirt["styleCategory"]}  
 ${shirt["expired"]}
```

In earlier versions of EL, you could access only JavaBeans properties. You could not call methods on objects. However, EL 2.1 added the ability to call object methods in JSPs. So, you could get the size of a `Shirt` with `${shirt.getSize()}` instead of ${shirt.size}`, but why would you? The latter`

is certainly easier. Method invocation mostly comes in handy when a value-returning method also requires some input.

Suppose you had an immutable class `ComplexNumber` that represents mathematical complex numbers (combination of a real number and an imaginary number in the form $a + bi$). That class would undoubtedly have a `plus` method that enables you to add some other number to it. (Possibly that method is overloaded so that you could add an `integer`, a `double`, or another `ComplexNumber`.) You can call the `plus` method and pass in an argument, and the resulting `ComplexNumber` would be the value of the expression:

```
 ${complex.plus(12)}
```

In this example, the `toString` method is implicitly called on the resulting `ComplexNumber` so that the string representation of the `ComplexNumber` is rendered. However, suppose you wanted the `i` in the string representation to be properly italicized so that it looks like a proper mathematical representation of a complex number. You might have a `toHtmlString` method on the `ComplexNumber` class to achieve this. You can thus render it like so:

```
 ${complex.plus(12).toHtmlString()}
```

These are chained method calls identical to the way you would perform this operation in standard Java code.

EL Functions

In EL, a function is a special tool mapped to a static method on a class. Like schema-compliant XML tags, functions are mapped to a namespace. The overall syntax of a function call is as follows, where `[ns]` is the namespace, `[fn]` is the function name, and `[a1]` through `[an]` are arguments:

```
 ${[ns]:[fn]([a1[, a2[, a3[, ...]]]])}
```

Functions are defined within Tag Library Descriptors (TLDs), which may sound strange because functions are not tags. This is a carryover from the earliest days of EL when it was part of the Java Standard Tag Library (JSTL) specification and EL could be used only within JSP tag attributes. Because the TLD concept already supported the idea of namespaces, it made sense for EL function definitions to remain within TLDs.

You learn more about TLDs and defining tags and functions in Chapter 8. However, there is already a set of functions defined in the JSTL that meet many of the needs developers have within JSPs today. All the functions deal with strings in some way—trimming, searching, joining, splitting, escaping, and more. By convention, the JSTL function library has a namespace of `fn`; however, you may make it whatever you like in the `taglib` directive. You experiment with using EL functions in the next section, but here are some of the more common JSTL EL functions and what they do.

- `${fn:contains(String, String)}` — This function tests whether the first string contains one or more instances of the second string and returns `true` if it does.
- `${fn:escapeXml(String)}` — If a string you are outputting could contain special characters, you can use this function to escape those special characters. `<` becomes `<`, `>` becomes `>`, `&` becomes `&`, and `"` becomes `"`. This is an especially important tool in the prevention of cross-site scripting (XSS) attacks.

- `${fn:join(String[], String)}` — This function joins an array of strings together using the specified string as a delimiter. For example, this could be useful for comma-separating an array of e-mail addresses together into one string for display on the page.
- `${fn:length(Object)}` — If the argument is a string, this function invokes and returns the result of calling the `length` method on the specified string. If it is a `Collection`, `Map`, or array, it returns the size of that `Collection`, `Map`, or array. No other types are supported. This is perhaps the most useful function in the JSTL.
- `${fn:toLowerCase(String)}` and `${fn:toUpperCase(String)}` — You can use these functions to change the case of a string to all lowercase or all uppercase.
- `${fn:trim(String)}` — This function trims all white space from both ends of the specified string.

There are still more functions available in the JSTL, and you can read about the rest of them by clicking. This is the documentation for JSTL 1.1 in Java EE 5. Unfortunately, there is no readily available HTML documentation for JSTL 1.2 in Java EE 6 and 7.

Static Field and Method Access

New in Expression Language 3.0, you can now access the public static fields and public static methods within any class on your JSP's class path. You could argue (and some have) that this puts too much power in the hands of JSP authors and enables them to do practically anything they could normally do with a scriptlet. It's up to you to decide whether that is a good thing or a bad thing, but the feature exists and cannot be disabled in EL 3.0.

You access static fields and methods the same way you would in Java — using the fully-qualified class name and field or method name separated with the dot operator. For example, you can access the `MAX_VALUE` constant on the `Integer` class with the following expression:

```
 ${java.lang.Integer.MAX_VALUE}
```

The class name must be fully qualified unless the class is imported using the JSP `page` directive. Remember that in JSPs, like Java, all classes in `java.lang` are implicitly imported for you. Because of this, the previous expression could be written like this instead:

```
 ${Integer.MAX_VALUE}
```

With this you can access static fields or methods on any class your JSP has access to. It's important to note that you can only *read* the value of these fields. You cannot write to them. (Of course, if a field is also final, you couldn't normally write to it anyway.) Calling a static method on a class is just as easy. Suppose you wanted to reverse the order of the bits in a number and see how the value of the number changed:

```
 ${java.lang.Integer.reverse(42)}
 ${Integer.reverse(24)}
```

This expression calls the static `reverse` method on the `Integer` class and passes the number 42 as its argument. In addition to calling named static methods, you can also invoke a constructor on a class, which returns an instance of that class that you can further access properties of, invoke methods on, or simply coerce to a string for output.

```

${com.wrox.User()}
${com.wrox.User('First', 'Last').firstName}

```

Although the static method access can entirely replace the behavior of EL functions and function libraries, that doesn't mean that function libraries are unnecessary. The previous static method call to `Integer.reverse` might be convenient, but with a theoretical `int` function library mapped to the static methods of `Integer`, the following expression is still more convenient:

```

${int:reverse(42)}

```

That may not seem much shorter, but imagine a much longer class name, and you should quickly see why function libraries are still of great use. One of the areas in which static field access could be most handy is with enums, which you learn about next.

Enums

Chances are you've been exposed to Java enums at some point, and if you've been using Java for a while, you are probably familiar with how useful and powerful they can be. Traditionally, enums in EL have been coerced to and from strings when necessary. For example, say your JSP had an in-scope variable named `dayOfWeek` and it represented one of the values from the `java.time.DayOfWeek` enum in the new Java 8 Date and Time API. You could test whether `dayOfWeek` is Saturday with the following boolean expression:

```

${dayOfWeek == 'SATURDAY'}

```

The `dayOfWeek` variable here is converted to a `String` and compared to "SATURDAY." This is unlike Java, where this conversion would never happen automatically. Although this is handy, it is certainly not type-safe. If you misspell Saturday (or if Saturday ever ceases being a day of the week) your IDE would probably not catch it, and if you compile JSPs during a continuous integration build to check for JSP compile-time errors, that would not catch it either. However, as of EL 3.0 you can use the static field access syntax to achieve type-safe enum constant reference. After all, enum constants are just public static final fields of their enum types:

```

${dayOfWeek == java.time.DayOfWeek.SATURDAY}

```

And, if you import `DayOfWeek` into your JSP, the expression is nearly as simple as the string-as-enum expression (and more like what you'd see in Java code):

```

${dayOfWeek == DayOfWeek.SATURDAY}

```

These last two techniques are type-safe and will be validated by your IDE and at compile time. Whichever you use is up to you, but we recommend a type-safe way.

Lambda Expressions

Expression Language 3.0 counts lambda expressions among its many new features. A *lambda expression* is an anonymous function that, typically, is passed as an argument to a higher-order function (such as a Java method). In the most general sense, lambda expressions are a list of parameter names (or some placeholder if the function has no parameters), followed by some type of operator, and finally the function body. In some languages supporting lambda expressions, this

order is reversed or otherwise different. Lambda expression syntax in EL is nearly identical to that of Java 8 lambda expressions. The primary difference between the two is that in Java the body of a lambda expression can contain anything that's legal in a Java method, whereas in EL the body of a lambda expression is another EL expression.

Just like with Java lambda expressions, EL lambda expressions use the arrow operator `->` to separate the expression parameters on the left side from the expression in the right side. Also, again as with Java lambda expressions, the parentheses around the expression parameter are optional if there is exactly one parameter. The following expressions are valid EL lambda expressions:

```
a -> a + 5
(a, b) -> a + b
```

Of course, by themselves these lambda expressions are not complete EL expressions. Something must be done with the lambda expressions. They could be evaluated immediately:

```
$( (a -> a + 5) (4) )
${((a, b) -> a + b) (4, 7) }
```

In the preceding EL expressions, the lambda expressions are declared and evaluated immediately. The resulting outputs of the two EL expressions are 9 and 11, respectively. Note that the lambda expression itself is surrounded by parentheses. This disambiguates the lambda expression from everything around it and enables you to execute it immediately. You could also define an EL lambda expression for use at a later time:

```
$(v = (a, b) -> a + b; v(3, 15) )
```

The output of the second expression in this case is 18 because it executes the lambda expression defined before the semicolon. The lambda expression `v` can now be used in any other EL expression that follows this expression on the page. This is especially useful if the lambda is very complex.

Finally, you could also pass an EL lambda expression as an argument to a method called within an EL expression.

```
$(users.stream().filter(u -> u.lastName == 'Williams' ||
u.lastName == 'Sanders ').toArray() )
```

Collections

Collections can be easily accessed in EL using the dot `.` and bracket `[]` operators. How you use the operators depends on what type of collection it is. Remember that in the Java Collections API, all collections are either `Collections` or `Maps`. Within the hierarchy of `Maps`, you simply have many different types of maps, all of which share a common foundation: Some key is associated with some value. The `Collection` hierarchy is a little more complicated. Within it you have `Sets`, `Lists`, and `Queues`. Because each type of collection has a different way in which you access its values, EL supports each one slightly differently.

Accessing values in a `Map` is quite simple and mimics the accessing of properties on JavaBeans. Suppose you have a `Map` named `map` with a key `username` mapped to the value "Jonathon" and a key `userId` mapped to the value "27." You could access these two properties of the map using the bracket operators, as in the following example:

```
 ${map["username"] }
 ${map["userId"] }
```

However, this is not the only technique you can use to access the `Map` values. You could also treat the keys like bean properties and access their values using the dot operator:

```
 ${map.username}  
 ${map.userId}
```

Although the second technique certainly involves fewer characters (always three fewer, to be exact), some people find the first technique more natural and more like how you would access `Map` values in languages that support operator overloading. You should use whichever you are more comfortable with. However, you should also note some restrictions on using the dot operator for accessing `Map` values. Simply put, if a key couldn't be an identifier in Java, you must use brackets instead of the dot operator to access the value mapped to that key. This means that your key can't contain spaces, periods, or hyphens, can't start with a number, and can't contain most special characters. (Although, there are a few surprising special characters that Java supports in identifiers, such as the dollar sign (\$) and accented characters such as å, é, ï, ö, ü, ñ, and so on.) If it contains any characters that aren't valid in Java identifiers, you must use brackets. If you're not sure, err on the side of caution and use brackets.

Accessing `Lists` is equally simple; however, it may surprise you just how forgiving it is. Consider a `List` (cleverly named `list`) with values "blue," "red," and "green," in order from 0 to 2. You would access the values using the bracket operator just as if the `List` were actually an array. The following code demonstrates this.

```
 ${list[0]}  
 ${list[1]}  
 ${list[2]}
```

You cannot treat the `List` indexes as properties and access them with the dot operator. This results in syntax errors:

```
 ${list.0} <%-- The EL interpreter will complain about a syntax error --%>
```

However, EL does permit you to use string literals instead of numbers to index the `List`, just as if it were a `Map` with the `List` indexes serving as the keys:

```
 ${list["0"]}  
 ${list['1']}  
 ${list[2]}
```

The only rule when using string literals is that the strings must be convertible to integers; otherwise, your code results in runtime errors. Although this is certainly flexible, there is no reason to use string literals as `List` indexes, and doing so can result in other developers' confusing your `List` (probably not named `list`) for a `Map`. We recommend using numeric literals instead.

The values of the other two types of collections, `Sets` and `Queues`, cannot be accessed using EL. These collections do not provide a means of directly accessing a value, such as an index with a `List` or a key with a `Map`. There are no "get" methods on `Sets` and `Queues`. You can access the values in these types of collections only by using iteration — something you explore in the next chapter. However, as with all types of collections, you can test whether `Sets` and `Queues` are empty using the `empty` operator:

```
 ${empty set}  
 ${empty queue}
```

You can do more things with collections using EL collection streams, and you explore that more in the collection streams section later in this chapter.

USING SCOPED VARIABLES IN EL EXPRESSIONS

Expression Language's sense of scoped variables and how variables are resolved makes it especially useful and powerful. Recall from Chapter 4 that JSPs have a set of implicit variables (`request`, `response`, `session`, `out`, `application`, `config`, `pageContext`, `page`, and `exception`) that you can use to obtain information from the request, session, and execution environment and affect the response. EL has a similar set of implicit variables; however, it also has an idea of implicit scope in which unknown variables are resolved. This enables you to obtain information from a variety of sources with minimal code. You explore these topics in this section.

For this section you use the User-Profile project available for download on the wrox.com code site. If you create it from scratch, be sure to create your `web.xml` file using the `<jsp-config>` from Chapter 4 and the `<session-config>` from Chapter 5, and create an `index.jsp` with the lone tag `<c:redirect url="/profile" />`.

The `/WEB-INF/jsp/base.jspf` file, which you have used in previous chapters, has changed slightly. Instead of just declaring the `c` tag library, it now also declares the `fn` function library, which you use in this section:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="fn" uri="http://java.sun.com/jsp/jstl/functions" %>
```

A NOTE ABOUT SCOPES

There are four different attribute scopes mentioned in this section (page, request, session, and application), but you may not understand the difference between them or what exactly they are. Each of these scopes has a progressively larger and longer scope than the previous one. You should already be familiar with the request scope: It begins when the server receives the request and ends when the server completes sending the response back to the client. The request scope exists anywhere that has access to the `request` object, and the attributes bound to the request are no longer bound after the request completes.

In Chapter 5 you learned about sessions and session attributes, so by now you may have figured out that the session scope persists between requests, and that any code with access to the `HttpSession` object can access the session scope. When the session has been invalidated, its attributes are unbound and the scope ends.

The page and application scopes are somewhat different. The page scope encapsulates attributes for a particular page (JSP) and request. When a variable is bound to the page scope, it is available only to that JSP page and only during the life of the

request. Other JSPs and Servlets cannot access the page scope-bound variable, and when the request completes, the variable is unbound. With access to the `JspContext` or `PageContext` object, you can store and retrieve attributes that exist within the page scope using the `setAttribute` and `getAttribute` methods. The application scope is the broadest scope, existing across all requests, sessions, JSP pages, and Servlets. The `ServletContext` object you learned about in Chapter 3 represents the application scope, and attributes that are stored in it live in the application scope.

Using the Implicit EL Scope

The EL defines 11 implicit variables in the scope of EL expressions, and you will learn about them all later in this section. However, the implicit scope is more useful and more commonly used because of its capability to resolve an attribute in the request, session, page, or application scope. When an EL expression references a variable, the EL evaluator resolves the variable using the following procedure:

1. It checks if the variable is one of the 11 implicit variables.
2. If the variable is not one of the 11 implicit variables, the EL evaluator next looks for an attribute in the page scope (`PageContext.getAttribute("variable")`) that has the same name (case-sensitive) as the variable. If it finds a matching page scope attribute, it uses the attribute value as the variable's value.
3. Finding no matching page attribute, the evaluator next looks for a request attribute (`HttpServletRequest.getAttribute("variable")`) with the same name as the variable and uses the attribute if it is found.
4. The evaluator looks for a session attribute (`HttpSession.getAttribute("variable")`) and uses it if found.
5. The evaluator looks for an application attribute (`ServletContext.getAttribute("variable")`) and uses it if found.
6. After the evaluator looks in all these places, if it finds no implicit variable or attribute matching the variable name, it raises an error.

The beauty of this feature is you do not need to retrieve an instance of the `HttpServletRequest` or `HttpSession` to use attributes on either of those objects. This is demonstrated in the `ProfileServlet` and `profile.jsp` file in the User-Profile project. Start by looking at the `com.wrox.User` class, which has several private fields with matching accessor and mutator methods:

```
public class User
{
    private long userId;
    private String username;
    private String firstName;
    private String lastName;
    private Map<String, Boolean> permissions = new Hashtable<>();
```

```
... // mutators and accessors
...
}
```

This is a simple POJO that you can use to hold information about your "user." You need to view this information somehow, so next create a very simple `ProfileServlet`:

```
@WebServlet(
    name = "profileServlet",
    urlPatterns = "/profile"
)
public class ProfileServlet extends HttpServlet
{
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException
    {
        User user = new User();
        user.setUserId(19384L);
        user.setUsername("Coder314");
        user.setFirstName("John");
        user.setLastName("Smith");

        Hashtable<String, Boolean> permissions = new Hashtable<>();
        permissions.put("user", true);
        permissions.put("moderator", true);
        permissions.put("admin", false);
        user.setPermissions(permissions);

        request.setAttribute("user", user);
        request.getRequestDispatcher("/WEB-INF/jsp/view/profile.jsp")
            .forward(request, response);
    }
}
```

So far, you haven't seen anything new. The Servlet creates a new `User` instance, sets some values on it, adds some permissions to it, creates a request attribute to hold the `user` object, and then forwards the request on to the view. The important code is contained in the `/WEB-INF/jsp/view/profile.jsp` file, which displays the user profile information in the browser:

```
<%--@elvariable id="user" type="com.wrox.User"--%>
<!DOCTYPE html>
<html>
    <head>
        <title>User Profile</title>
    </head>
    <body>
        User ID: ${user.userId}<br />
        Username: ${user.username} (${user.username.length()} characters)<br />
        Full Name: ${fn:escapeXml(user.lastName)} += ', '
                    += fn:escapeXml(user.firstName)}
        <br /><br />
        <b>Permissions (${fn:length(user.permissions)})</b><br />
```

```
User: ${user.permissions["user"]}<br />
Moderator: ${user.permissions["moderator"]}<br />
Administrator: ${user.permissions["admin"]}<br />
</body>
</html>
```

There is a lot of interesting stuff in this JSP, and you will dissect it in a minute. For now, compile and fire up your debugger; then navigate to <http://localhost:8080/user-profile/profile> in your browser. You should see the page from the screen shot in Figure 6-1.

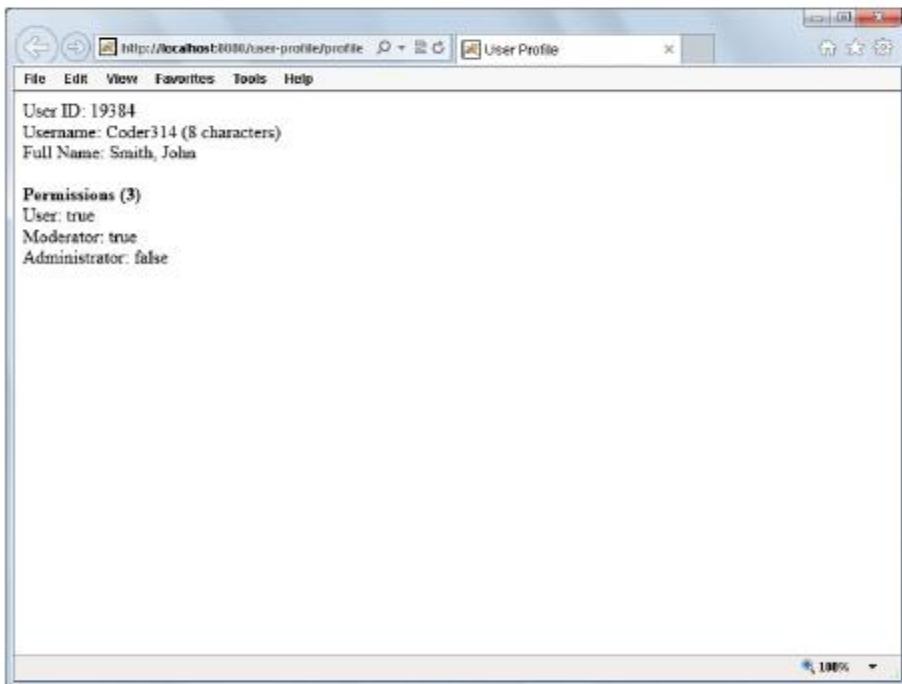


FIGURE 6-1

Now take a look at this JSP line by line to get a better understanding of how it works. First there's the new, weird JSP comment at the top of the file:

```
<%--@elvariable id="user" type="com.wrox.User"--%>
```

This comment tag is not really needed, and in fact if you remove it, recompile, and rerun your application, it still works. (Go ahead; try it!) So what does it do? The special `@elvariable` comment is a convention that developers use to type-hint for their IDE. This comment tells the IDE "Yes, a `user` variable exists in the implicit scope on this page, and its type is `com.wrox.User`." The advantage this gains you is that because the IDE knows the variable exists and what its type is, it can now provide auto-completion and intelligent suggestions that it could not otherwise provide. It can also validate that your EL expression is correct.

Even if you do not use an IDE or use one that does not support this convention, other developers maintaining your JSPs at a later date can quickly know what the EL variable type is. Your JSP-writing time will be much easier spent if you get in the habit of using `@elvariable` comments.

Next, you should notice the User ID line:

```
User ID: ${user.userId}<br />
```

Here, the `user` attribute that you added to the request in the Servlet code has been accessed as an EL variable using the implicit scope in the JSP page, and you have used the bean property `userId` instead of calling the accessor method directly. The line directly below it does the same with the `username` but also calls the `length` method on the `username` String.

```
Username: ${user.username} (${user.username.length()} characters)<br />
```

Note that instead of calling the `length` method directly you could have used the `fn:length` function, but that is used later in the code for a collection, and this serves as a good example of the alternative. Next, your JSP escapes the last and first name and concatenates them with a comma:

```
Full Name: ${fn:escapeXml(user.lastName) += ', ' += fn:escapeXml(user.firstName)}
```

Note the use of the `fn:escapeXml` function to escape HTML characters that might be in the name and the `+=` string concatenation operator to combine all the strings. The final part of your JSP prints out the user's permissions:

```
<b>Permissions (${fn:length(user.permissions)})</b><br />
User: ${user.permissions["user"]}<br />
Moderator: ${user.permissions["moderator"]}<br />
Administrator: ${user.permissions["admin"]}<br />
```

The `fn:length` function outputs the number of elements in the user's `permissions` collection, and the other three lines are all using the bracket operators to access values in the `permissions` Map.

As an exercise, edit your `ProfileServlet` and change the `request.setAttribute("user", user)` line to put the user on the session instead of the request:

```
request.getSession().setAttribute("user", user);
```

Now compile and rerun your application. You don't need to make any changes to the JSP. The `user` attribute may be in a different scope (session instead of request) but is still in the implicit scope so that you can access it from EL expressions as an EL variable. When it was bound to the request, the `user` attribute existed until the request was complete, and then it was made eligible for garbage collection. Now that it is bound to the session, it is available to other requests from the same client, even if they go to different pages. However, this is not the only scope you could bind the `user` attribute to. Replace `request.getSession()` with `this.getServletContext()` and bind it to the application context:

```
this.getServletContext().setAttribute("user", user);
```

Now compile and rerun again without making any changes to the JSP. Again the `user` attribute was still in the implicit scope and accessible from your EL expression. You can access anything in the four supported scopes in this manner, which greatly simplifies your task of writing JSPs.

Using the Implicit EL Variables

As mentioned earlier in this section, there are 11 implicit EL variables available for use within EL expressions. With one exception, they are all `Map` objects. Most are used to access attributes from some scope, request parameters, or headers.

- `pageContext` is an instance of the `PageContext` class and is the only implicit EL variable that is not a `Map`. You should be familiar with `PageContext` from Chapter 4 and earlier in this section. Using this variable you can access the page error data and exception object (if applicable), the expression evaluator, the output writer, the JSP Servlet instance, the request and response, the `ServletContext`, the `ServletConfig`, and the session.
- `pageScope` is a `Map<String, Object>` containing all the attributes bound to the `PageContext` (page scope).
- `requestScope` is a `Map<String, Object>` of all the attributes bound to the `ServletRequest`. Using this, you can access these attributes without calling a method on the `request` object.
- `sessionScope` is also a `Map<String, Object>`, and it contains all the session attributes from the current session.
- `applicationScope` is the last of the scopes, a `Map<String, Object>` containing all the attributes bound to the `ServletContext` instance.
- `param` and `paramValues` are similar in that they both provided access to the request parameters. The `param` variable is a `Map<String, String>` and contains only the first value from any parameter with multiple values (similar to `getParameter` from `ServletRequest`), whereas the `Map<String, String[]>` `paramValues` contains all the values of every parameter (`getParameterValues` from `ServletRequest`). `param` is easier to use if you know a request parameter has only one value.
- `header` and `headerValues` provide access to the request headers, with `Map<String, String>` `header` containing only the first value of any multivalue headers and `Map<String, String[]>` `headerValues` containing all values for every header. Like `param`, `header` is easier to use if you know a header has only one value.
- `initParam` is a `Map<String, String>` containing all the context init parameters from the `ServletContext` instance for this application.
- `cookie` is a `Map<String, javax.servlet.http.Cookie>` containing all the cookies that the user's browser sent along with the request. The keys in this map are the cookie names. It should be noted that it is possible to have two cookies with the same name (but different paths), and in that case this `Map` will contain only the first cookie with a given name in the order it existed in the request. This order might vary from one request to the next. There is no way in EL to access any of the other duplicate cookies with the same name without iterating over all the cookies. (Iteration with EL is something you learn how to do in the next chapter.)

To demonstrate the various EL implicit variables and how they can be used, create a file named `info.jsp` in the web root of your project and put the following code in it:

```
<%  
    application.setAttribute("appAttribute", "foo");  
    pageContext.setAttribute("pageAttribute", "bar");  
    session.setAttribute("sessionAttribute", "sand");  
    request.setAttribute("requestAttribute", "castle");  
%>  
<!DOCTYPE html>  
<html>  
    <head>  
        <title>Information</title>  
    </head>  
    <body>  
        Remote Address: ${pageContext.request.remoteAddr}<br />  
        Request URL: ${pageContext.request.requestURL}<br />  
        Session ID: ${pageContext.request.session.id}<br />  
        Application Scope: ${applicationScope["appAttribute"]}<br />  
        Page Scope: ${pageScope["pageAttribute"]}<br />  
        Session Scope: ${sessionScope["sessionAttribute"]}<br />  
        Request Scope: ${requestScope["requestAttribute"]}<br />  
        User Parameter: ${param["user"]}<br />  
        Color Multi-Param: ${fn:join(paramValues["colors"], ', ') }<br />  
        Accept Header: ${header["Accept"]}<br />  
        Session ID Cookie Value: ${cookie["JSESSIONID"].value}<br />  
    </body>  
</html>
```

The first four lines of the JSP set attributes within the various scopes, for the purposes of demonstration. The lines within the HTML body print out varying information about the request, attributes on the different scopes, parameters in the URL, headers, and cookies.

Compile and debug your application; then go to `http://localhost:8080/user-profile/info.jsp?user=jack&colors=green&colors=red` in your browser. You should see a good deal of information printed out to the screen. If the Session ID Cookie Value line is empty, this means that your session was just created and the browser did not send a cookie yet; refresh the page and a value should appear here.

One last JSP demonstrates the priority of the various scopes when resolving variables in the implicit EL scope. Create a file named `scope.jsp` in the web root, and put the following code in it.

```
<%  
    pageContext.setAttribute("a", "page");  
    request.setAttribute("a", "request");  
    session.setAttribute("a", "session");  
    application.setAttribute("a", "application");  
  
    request.setAttribute("b", "request");  
    session.setAttribute("b", "session");  
    application.setAttribute("b", "application");  
  
    session.setAttribute("c", "session");
```

```

        application.setAttribute("c", "application");

        application.setAttribute("d", "application");
    %>
<!DOCTYPE html>
<html>
    <head>
        <title>Scope Demonstration</title>
    </head>
    <body>
        a = ${a}<br />
        b = ${b}<br />
        c = ${c}<br />
        d = ${d}<br />
    </body>
</html>

```

The majority of this JSP is setup code, with only four EL expressions making up the demonstration. The `a` attribute has conflicting values in all four scopes, `b` in three, and `c` in two. The `d` attribute is present only in the application scope. The value displayed next to each name on the page will be the name of the scope with the highest precedence among the scopes with conflicting values. Compile and run your application and navigate to `http://localhost:8080/user-profile/scope.jsp`. The output should look identical to what follows, indicating that the EL engine looks for implicitly scoped variables first in the page scope and then in the request, session, and application scopes, in that order.

```

a = page
b = request
c = session
d = application

```

ACCESSING COLLECTIONS WITH THE STREAM API

One of the biggest additions to Expression Language 3.0 in Java EE 7 is support for the Collections Stream API introduced in Java SE 8. Because the API is supported natively in EL 3.0, you do not need to run your application in Java 8 to take advantage of this new EL feature. In this section, you learn the basics of the Stream API and how to use it in your JSPs.

NOTE *In an early, prerelease version of Expression Language 3.0, the specification included an implementation of Microsoft LINQ (Language Integrated Query). This added collection-querying capabilities using the LINQ standard query operators. The final specification was rewritten to remove the LINQ features and replace them with an equivalent to the Stream API. This provides consistency across the Java language and Expression Language specifications.*

The basis of the Stream API is the no-argument `stream` method present on every `Collection`. This method returns a `java.util.stream.Stream` that can filter and otherwise manipulate a copy of the collection. The `java.util.Arrays` class also provides many static methods for retrieving `Streams` from various arrays. Using this `Stream`, you can perform many different operations. Some of these operations return other `Streams`, allowing you to create a *chained pipeline* of operations. This

pipeline consists of a *pipeline source* (the `Stream`), the *intermediate operations* (such as filtering and sorting), and finally a *terminal operation* (such as converting the results to a `List` that can be iterated and displayed).

In EL 3.0, you can call the `stream` method on any EL variable that is a Java array or a `Collection`. The returned `Stream` isn't actually a `java.util.stream.Stream` because EL 3.0 must work in Java 7, where `Streams` do not exist yet. Instead, the returned `Stream` is an EL-specific implementation of the Stream API. For example, the following EL expression filters a `Collection` of books by title, reduces the properties available for each book to just the title and author, and returns a `List` of the results:

```
books.stream().filter(b->b.title == 'Professional Java for Web Applications')
  .map(b->{ 'title':b.title, 'author':b.author })
  .toList()
```

Understanding Intermediate Operations

As mentioned earlier, intermediate operations filter, sort, reduce, transform, or otherwise alter a collection of values so that the collection ends up in the desired state. It's important to understand that when performing intermediate operations on a `Stream`, the original `Collection` or array is never altered. The operations affect only the contents of the stream. You'll find many different intermediate operations, and you learn about the most common and useful ones in this section. You can learn about the rest of them by downloading and reading the JSR 341 specification PDF from the specification download page.

Filtering the Stream

The `filter` operation is probably the operation you will use most often. It filters the contents of the `Stream`, typically reducing the number of objects contained therein. The `filter` operation accepts a *predicate argument*—a lambda expression that returns a `boolean` and accepts a single argument whose type is the element type of the `Stream`. Given a `List<E>` where `E` is the element type, `stream` returns a `Stream<E>`. Calling `filter` on this `Stream<E>`, you supply a `Predicate<E>` with the signature `E -> boolean`. You then use properties of `E` to determine whether to include that particular `E` in the resulting `Stream<E>`. To better understand this, consider the following expression:

```
$(books.stream().filter(b -> b.author == "John F. Smith"))
```

The predicate in this case is the lambda expression that accepts a book as an argument and tests whether the book's author is John F. Smith. When passed to the `filter` operation, the predicate applies to every book in the `Stream`, and the resulting `Stream` contains only those books for which the predicate returns `true`.

You can also use the special `distinct` operation to filter out duplicate values. The following expression removes the duplicate 3s and 5s from the `List`:

```
$([1, 2, 3, 3, 4, 5, 5, 5, 6].stream().distinct())
```

Manipulating Values

You can manipulate the values in a `Stream` using the `forEach` operation. Like `filter`, `forEach` accepts a lambda expression that is evaluated for every element in the `Stream`. However, this lambda expression is a *consumer*, meaning it has no return value. You can use this to manipulate the values in the `Stream`, likely to transform them in some way. Here is one potential use case:

```
 ${books.stream().forEach(b -> b.setLastViewed(Instant.now()))}
```

Sorting the Stream

You sort the stream using the `sorted` operation. For a `Stream<E>`, the `sorted` operation accepts a `java.util.Comparator<E>`. As a Java developer, you are probably familiar with this interface, which can be represented with the lambda expression `(E, E) -> int`. This lambda expression or `Comparator` compares two elements in the `Stream` using an efficient sorting algorithm that is unspecified and implementation-specific. The following expression sorts books by their title:

```
 ${books.stream().sorted((b1, b2) -> b1.title.compareTo(b2.title))}
```

A variation of the `sorted` operation exists that does not accept any arguments. Instead, it assumes that the elements in the `Stream` implement the `java.lang.Comparable` interface, meaning you can naturally sort them. The following naturally orders the list of numbers from least to greatest. The resulting list is -2, 0, 3, 5, 7, 8, 19.

```
 ${[8, 3, 19, 5, 7, -2, 0].stream().sorted()}.
```

Limiting the Stream Size

You can limit the number of elements in the `Stream` using the `limit` and `substream` operations. Use `limit` to simply truncate the `Stream` after the specified number of elements. `substream` is more useful for pagination because you can specify a start index (inclusive) and end index (exclusive).

```
 ${books.stream().limit(10)}
 ${books.stream().substream(10, 20)}
```

Transforming the Stream

Using the `map` operation, you can transform the elements in the `Stream` to some other type of element. The `map` operation accepts a *mapper* that expects one type of element and returns a number. Given a `Stream<S>`, `map` expects a lambda expression whose sole argument is of type `S`. If the lambda expression then returns a different type `R`, the resulting `Stream` is a `Stream<R>`. The following takes a `List<Book>`, retrieves a `Stream<Book>`, and transforms it into a `Stream<String>` containing only the book titles:

```
 ${books.stream().map(b -> b.title)}
```

Of course, you can return more complex types. You might have a different type, `DisplayableBook`, with a limited set of properties. Or you could create an implicit `List` or `Map`, returning a `Stream<List<Object>>` or `Stream<Map<Object, Object>>`:

```
 ${books.stream().map(b -> [b.title, b.author])}
 ${books.stream().map(b -> {"title":b.title, "author":b.author})}
```

Using Terminal Operations

After you filter, sort, or otherwise transform your `Stream`, you need to perform some final operation that converts the `Stream` back into a useful value, `Collection`, or array. This type of operation is a terminal operation. It is terminal because unlike intermediate operations, which all return `Streams` that can be further acted on, this operation does not return a `Stream`. It evaluates any intermediate operations deferred for performance reasons and then converts the final result as desired. Ultimately, you must always perform a terminal operation. A `Stream` is not very useful by itself; you need a final value to act on.

Returning a Collection

You can use the `toArray` and `toList` operations to return a Java array or `List` of the final result element type. For example, the following expressions return a `String[]` and `List<String>` of book titles, respectively:

```
 ${books.stream() .map(b -> b.title) .toArray() }  
 ${books.stream() .map(b -> b.title) .toList() }
```

If you performed any `sorted` intermediate operations on the `Stream`, the resulting array or `List` will be in the order indicated with those operations. You can also use the `iterator` operation to return a suitable `java.util.Iterator`.

Using Aggregate Functions

You can aggregate the values in the `Stream` using the `min`, `max`, `average`, `sum`, and `count` operations. The `count` operation can operate on any type of `Stream`, whereas the `average` and `sum` operations require the final `Stream` element types to be coercible to `Number`s. `count` returns the number of elements in the `Stream` as a `long`; `average` returns the average of all the `Stream` elements as an `Optional<? extends Number>`; and `sum` returns the sum of all the `Stream` elements as a `Number`. An `Optional` is a placeholder that can report whether the returned value was `null` and provide the returned value when requested.

The `min` and `max` operations are both interesting. They both return `Optional<E>` where `E` is the element type of the resulting `Stream`. Without any arguments, these operations require that the `Stream` elements implement `Comparable`. However, you can provide a `Comparator` argument to these operations when wanted.

The following expressions represent some common use cases for these aggregating terminal operations:

```
 ${books.stream() .map(b -> b.price()) .min() }  
 ${books.stream() .map(b -> b.price()) .max() }  
 ${books.stream() .filter(b -> b.author == "John F. Smith")  
     .map(b -> b.price()) .average() }  
 ${books.stream() .filter(b -> b.author == "John F. Smith") .count() }  
 ${cartItems.stream() .map(i -> i.price() * i.quantity()) .sum() }
```

Returning the First Value

You can use the `findFirst` operation to return the first element in the resulting `Stream`. For a `Stream<E>` it returns an `Optional<E>` because the `Stream` might be empty, meaning there is no first element to return.

```
 ${books.stream().filter(b -> b.author == "John F. Smith").findFirst()}
```

Putting the Stream API to Use

For a simple exercise in the use of the Stream API, you add a JSP to the User-Profile project and filter, map, and sort a `List` of `User`s. Start by adding a constructor to the `User` object (and also adding a default constructor so that previous code won't break).

```
public User() { }

public User(long userId, String username, String firstName, String lastName)
{
    this.userId = userId;
    this.username = username;
    this.firstName = firstName;
    this.lastName = lastName;
}
```

Now create a `collections.jsp` file in the web root of the project and put the following code in it:

```
<%@ page import="com.wrox.User" %>
<%@ page import="java.util.ArrayList" %>
<%
    ArrayList<User> users = new ArrayList<>();
    users.add(new User(19384L, "Coder314", "John", "Smith"));
    users.add(new User(19383L, "geek12", "Joe", "Smith"));
    users.add(new User(19382L, "jack123", "Jack", "Johnson"));
    users.add(new User(19385L, "farmer-dude", "Adam", "Fisher"));
    request.setAttribute("users", users);
%>
<!DOCTYPE html>
<html>
    <head>
        <title>Collections and Streams</title>
    </head>
    <body>
        ${users.stream()
            .filter(u -> fn:contains(u.username, '1'))
            .sorted((u1, u2) -> (x = u1.lastName.compareTo(u2.lastName));
                    x == 0 ? u1.firstName.compareTo(u2.firstName) : x))
            .map(u -> {'username':u.username, 'first':u.firstName,
                        'last':u.lastName})
            .toList()}
    </body>
</html>
```

The setup code at the top of the file creates some users and adds them to the list. Then the EL expression filters the list to users whose usernames contain the number 1; orders by the last name and then first name; selects the username, first name, and last name from each matching user; and then evaluates immediately to a `List`. Finally, the `List` is automatically coerced to a `String` for display on the screen (using the `List`'s `toString` method). Notice the use of the semicolon and assignment (`=`) operators in the `sorted` lambda expression — this allows you to compare the last names only once, assign the comparison to a variable (`x`), and then test the value of `x`, returning it if the last names are different and comparing the first names if the last names are the same. The body of the `sorted` lambda expression is surrounded by parentheses (in bold) because the lambda operator (`->`) has a higher precedence than the assignment and semicolon operators.

You can test this out by compiling and running your application and going to `http://localhost:8080/user-profile/collections.jsp` in your browser.

NOTE In Chapter 4 you explored using Java code in JSPs and learned about some of the many reasons using Java within JSPs is discouraged. The introduction of the Stream API to the Expression Language provides a lot of additional power for the JSP author to manipulate collections significantly. If you get in the habit of using the Stream API in JSPs routinely, you may find that you have started putting business logic in the presentation layer instead of the Java code. Only you can decide whether this is appropriate for your needs, but it is something to keep in mind. You will not see use of the Stream API in JSPs anywhere else in the book — these types of operations are performed only in the Java code from now on.

REPLACING JAVA CODE WITH EXPRESSION LANGUAGE

In this section you begin replacing some of the Java code in your JSPs in the ongoing customer support application with EL expressions. The Customer-Support-v4 project on the wrox.com code download site contains these changes. You cannot replace all the Java code yet. For that, we will need the next chapter. Start by updating your `/WEB-INF/jsp/base.jspf` file to contain a tag library declaration for the JSTL function library:

```
<%@ page import="com.wrox.Ticket, com.wrox.Attachment" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="fn" uri="http://java.sun.com/jsp/jstl/functions" %>
```

You do not need to make any changes to Java code in this section. Everything you change will be within JSPs. The `ticketForm.jsp` view in `/WEB-INF/jsp/view` is already devoid of any Java code, so there is nothing you can do to improve that. `viewTicket.jsp`, on the other hand, has several things that can be replaced. The new code for this file is in Listing 6-1.

Notice that the new code has `@elvariable` type hints at the top for `ticketId` and `ticket`, and that the `ticketId` Java variable has been removed. The `ticket` Java variable has not been removed, however, because EL expressions cannot replace everything the `ticket` variable is being used for — such as iterating over the attachments. The new EL expressions have been highlighted in bold.

LISTING 6-1: `viewTicket.jsp`

```

<%--@elvariable id="ticketId" type="java.lang.String"--%>
<%--@elvariable id="ticket" type="com.wrox.Ticket"--%>
<%
    Ticket ticket = (Ticket)request.getAttribute("ticket");
%>
<!DOCTYPE html>
<html>
    <head>
        <title>Customer Support</title>
    </head>
    <body>
        <a href=<c:url value="/login?logout" />>Logout</a>
        <h2>Ticket #${ticketId}: ${ticket.subject}</h2>
        <i>Customer Name - ${ticket.customerName}</i><br /><br />
        ${ticket.body}<br /><br />
        <%
            if(ticket.getNumberOfAttachments() > 0)
            {
                %>Attachments: <%
                int i = 0;
                for(Attachment a : ticket.getAttachments())
                {
                    if(i++ > 0)
                        out.print(", ");
                    %><a href=<c:url value="/tickets">
                        <c:param name="action" value="download" />
                        <c:param name="ticketId" value="${ticketId}" />
                        <c:param name="attachment" value="<%= a.getName() %>" />
                    </c:url>><%= a.getName() %></a><%
                }
                %><br /><br /><%
            }
        %>
        <a href=<c:url value="/tickets" />>Return to list tickets</a>
    </body>
</html>

```

The `/WEB-INF/jsp/view/sessions.jsp` file is another JSP that could use EL expressions. You can find the new code for this file in Listing 6-2. The only changes to this JSP are the `@elvariable` type hint and the lone EL expression in bold. None of the rest of the Java code can be replaced at this time because of the need for recursion and formatting the time interval.

LISTING 6-2: sessions.jsp

```

<%--@elvariable id="numberOfSessions" type="java.lang.Integer"--%>
<%@ page import="java.util.List" %>
<%!
    private static String toString(long timeInterval)
    {
        if(timeInterval < 1_000)
            return "less than one second";
        if(timeInterval < 60_000)
            return (timeInterval / 1_000) + " seconds";
        return "about " + (timeInterval / 60_000) + " minutes";
    }
%>
<%
    @SuppressWarnings("unchecked")
    List<HttpSession> sessions =
        (List<HttpSession>)request.getAttribute("sessionList");
%>
<!DOCTYPE html>
<html>
    <head>
        <title>Customer Support</title>
    </head>
    <body>
        <a href=">Logout</a>
        <h2>Sessions</h2>
        There are a total of ${numberOfSessions} active sessions in this
        application.<br /><br />
        <%
            long timestamp = System.currentTimeMillis();
            for(HttpSession aSession : sessions)
            {
                out.print(aSession.getId() + " - " +
                    aSession.getAttribute("username"));
                if(aSession.getId().equals(session.getId()))
                    out.print(" (you)");
                out.print(" - last active " +
                    toString(timestamp - aSession.getLastAccessedTime()));
                out.println(" ago<br />");
            }
        %>
    </body>
</html>

```

Now compile and run the customer support application and go to <http://localhost:8080/support/> in your browser. Log in, create a few tickets, and view the tickets. Go to <http://localhost:8080/support/sessions> and view the list of sessions. Everything should work the same way it did in Chapter 5, but now EL takes care of some of your output.

SUMMARY

In this chapter you learned about the history of the Java Unified Expression Language, the basic EL syntax, and what EL expressions are used for. You explored reserved words, operators, literal values, accessing object properties and methods, EL functions and the JSTL function library, static field and method access, enums, lambda expressions, and collections operators. You were introduced to the four different scopes and the implicit EL scope, and learned about the eleven implicit EL variables. You also learned about the Stream API and its addition to EL 3.0. Finally, you replaced some Java code with EL expressions in the customer support application that you started in Chapter 3.

It should be clear by now that although EL expressions can replace a lot of Java code, they do not do everything you need to replace *all* the Java code in your JSPs. For example, you cannot loop within EL expressions or have blocks of code evaluated based on whether some expression is true. For this you need the Java Standard Tag Library, which you explore in the next chapter.

7

Using the Java Standard Tag Library

IN THIS CHAPTER

- Understanding JSP tags and the JSTL
- How to implement the Core tag library (C namespace)
- How to use the Formatting tag library (FMT namespace)
- How to use the Database Access tag library (SQL namespace)
- How to use the XML Processing tag library (XML namespace)
- Swapping Java code with JSP tags

WROX.COM CODE DOWNLOADS FOR THIS CHAPTER

You can find the `wrox.com` code downloads for this chapter at <http://www.wrox.com/go/projavaforwebapps> on the Download Code tab. The code for this chapter is divided into the following major examples:

- Address-Book Project
- Address-Book-i18n Project
- Customer-Support-v5 Project

NEW MAVEN DEPENDENCIES FOR THIS CHAPTER

There are no new Maven dependencies for this chapter. Continue to use the Maven dependencies introduced in all previous chapters.

INTRODUCING JSP TAGS AND THE JSTL

Up to this point it has been absolutely necessary to use Java to do certain things in JSPs. The ability to add Java to your JSPs is convenient to be sure, but remember that your UI developers, accustomed to HTML, JavaScript, and CSS, will likely not write Java code. Your goal is to have Java-free JSPs, but so far you do not have the tools to do that yet. The Expression Language was helpful in replacing some Java code, but you experienced in the last chapter how much you still have to rely on Java code even using EL. In fact, you really haven't even tapped in to the power of EL yet because you are still too limited by having to use Java code.

You have already seen a small sampling of JSP tags and the JSTL in previous chapters with the `<c:url>` and `<c:redirect>` tags. This was simply unavoidable because the alternatives were too unfriendly to even show you. However, these tags were mentioned only in passing, and the details of them and the JSTL were left to this chapter. You also saw part of the JSTL when you explored the JSTL function library (with the `fn` namespace) in the previous chapter, but this was necessary because it is really a function library, not a tag library, and it is meant exclusively for use in EL expressions. You can't use the JSTL function library outside of EL expressions. In this chapter you learn about the concept of JSP tags and the JSTL in detail, and you finally replace the unsightly Java code in your JSPs with a combination of JSP tags and EL expressions.

Working with Tags

JSP tags are a special syntax of the JavaServer Pages technology that looks like any normal HTML or XML tag. JSP tags are also called *actions* because that's what they do. A JSP tag performs some action, such as creating or restricting output. The JSP and JSTL specifications refer almost exclusively to actions, but this book calls them *tags*. Because they are outside of the scope of any standard HTML-specified tag, JSP tags require an XML namespace to be referenced correctly. However, writing XML can be a very tedious and unforgiving task, as you saw with the brief introduction to JSP Documents (`.jspx`) in Chapter 4. In particular, the need to adhere to a strict XML document syntax is sometimes difficult even for seasoned programmers. Thus the idea of the JSP tag syntax includes some shortcuts to make writing JSPs easier. The first of these shortcuts is the `taglib` directive, which you explored in Chapter 4 and have used since then.

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="fn" uri="http://java.sun.com/jsp/jstl/functions" %>
```

This directive is an alternative to the XMLNS technique for referencing XML namespaces in XML documents:

```
<jsp:root xmlns="http://www.w3.org/1999/xhtml" version="2.0"
           xmlns:jsp="http://java.sun.com/JSP/Page"
           xmlns:c="http://java.sun.com/jsp/jstl/core"
           xmlns:fn="http://java.sun.com/jsp/jstl/functions">
```

The use of this directive prevents XML document parsers from parsing your JSP, but it also prevents you from having to worry about XML standards compliance (the other important shortcut). Instead, the JSP engine in your web container understands the special JSP syntax and knows how to parse it, and (these days) all major Java IDEs also understand the JSP syntax and can alert you to syntax errors and other issues in your JSP.

The `prefix` attribute in a `taglib` directive (or the XML namespace) represents the namespace with which the tag library is referenced throughout the JSP page. The tag prefix is suggested in the Tag Library Descriptor file (TLD) for that tag library but is declared in the `taglib` directive using the `prefix` attribute. Thus the prefix can be whatever you set it to in the `prefix` attribute, but generally developers stick to using the prefix suggested in the TLD to prevent confusion among other developers.

The `uri` attribute indicates the URI defined in the TLD for that tag library. This is how the JSP parser locates the appropriate TLD for the referenced tag library: It finds the TLD containing the same URI.

NOTE *The URI is a naming convention, not actually the location of the TLD (and not a real URL). In fact, in most cases navigating to the URI in your browser can result in a 404 Not Found or similar error. The TLDs you use are included with your application in some fashion, whether in the container, in your application's JAR files, or in your application's `WEB-INF` directory. The URI is merely a technique for uniquely identifying a TLD so that the tags you use can be correctly associated with the appropriate TLD.*

When the JSP parser encounters a `taglib` directive, it locates the TLD file for that tag library, using the URI, by looking for it in a variety of locations. These locations are indicated in the JSP specification as follows, in order from highest to lowest precedence:

1. If the container is a Java EE-compliant container, the parser looks for any matching TLD files that are part of the Java EE specification, including the JSP tag library, the Java Standard Tag Library, and any JavaServer Faces libraries.
2. It then checks explicit `<taglib>` declarations within the `<jsp-config>` section of the deployment descriptor.
3. If the parser still hasn't located a matching TLD file, it checks any TLD files contained within the `META-INF` directory of any JAR files placed in your application's `/WEB-INF/lib` directory, or any TLD files placed in your application's `/WEB-INF` directory or in any subdirectories of `/WEB-INF`, recursively.
4. Finally, the parser checks any other TLD files that ship as part of the web container or application server. (These are usually custom to the web container, and as such using them ties your application specifically to that web container and makes it non-portable.)

An explicit `<taglib>` declaration is normally not needed unless the TLD you are referencing does not contain a URI (legal, but unusual), it is not located within one of the other locations previously listed (something you can avoid by putting it in the right place), or you need to override a TLD with a conflicting URI supplied in some third-party JAR file that you don't

have control over (a more likely but still unusual scenario). Explicit `<taglib>` declarations look like this:

```
<jsp-config>
  ...
  <taglib>
    <taglib-uri>http://www.example.org/xmlns/jsp/custom</taglib-uri>
    <taglib-location>/tld/custom.tld</taglib-location>
  </taglib>
  ...
</jsp-config>
```

In this example the `<taglib-uri>` value `http://www.example.org/xmlns/jsp/custom` would be compared against the `taglib` directive `uri` attribute. If they matched, it would use the TLD specified (`/tld/custom.tld`), relative to the root of the web application. Notice this configuration does not specify a prefix. This is because it is not a tag library declaration, like the `taglib` directive. It's simply a map telling the container where the TLD file for the specified tag library URI lives. The use of explicit `<taglib>` declarations is almost universally avoidable, so you will not use them in any examples in this book.

After a `taglib` directive is correctly configured to resolve to the appropriate TLD, you can use the tags within that library in your JSP. All JSP tags follow the same basic syntax:

```
<prefix:tagname[ attribute=value[ attribute=value[ ...]]] />
<prefix:tagname[ attribute=value[ attribute=value[ ...]]]>
  content
</prefix:tagname>
```

In this syntactic notation, `prefix` denotes the JSP tag library prefix, also known as the *namespace* (which is the standard XML nomenclature). `tagname` is the name of the tag as defined in the TLD. Attribute values are quoted with either single quotes ('') or double quotes ("") but are never unquoted. Two attributes in the same tag can use different quoting styles, but if an attribute value starts with a single quote, it must end with a single quote, and if it starts with a double quote, it must end with a double quote. There must be white space between attributes, but in a self-closing tag, the white space before the `/>` is optional. All JSP tags must either be valid XML self-closing tags (`<prefix:tagname />`) or they must have matching closing tags (`<prefix:tagname></prefix:tagname>`). Non-XML self-closing tags without matching closing tags (`<prefix:tagname>`) are syntax errors.

When you write a JSP, note that one tag library is already implicitly included for use in all your JSPs. This is the JSP tag library (`prefix_jsp`), and you do not need to place a `taglib` directive in a JSP to use it. (In a JSP document, however, you do need to add an XMLNS declaration for the `jsp` tag library.) You have already seen uses of tags in the JSP tag library in previous chapters, such as `<jsp:include>`, `<jsp:forward>`, `<jsp:plugin>`, `<jsp:useBean>`, and so on. You have also seen how the JSP tag library can be used in JSP Documents with `<jsp:root>`, `<jsp:directive>`, `<jsp:declaration>`, `<jsp:scriptlet>`, and `<jsp:expression>`. All these tags are already available to you in any JSP you write.

Remember from Chapter 2 that there are full, Java EE-compliant application servers, and then there are more limited Java EE web containers. Application servers implement the entire Java EE

specification, whereas web containers implement the Servlet and JSP specifications—and maybe a handful of other specifications that the creators of the web container thought important. Most web containers also implement the EL specification because it used to be part of the JSP specification and today remains inextricably linked to the JSP specification. All web containers support using tag libraries with JSPs because this support is part of the JSP specification. However, some web containers do not implement the Java Standard Tag Library (JSTL) specification, because the specific tag libraries in the JSTL are easily decoupled from the generic concept of tag libraries. Tomcat has historically been one of these web containers, and to this day, it does not implement the JSTL. However, this does not mean that you cannot use JSTL in applications you plan on deploying in Tomcat!

NOTE *Tomcat implements the Servlet API, JSP, Expression Language, and WebSocket API implementations. Other web containers may implement more or fewer specifications, and this may vary from one version to the next. Be sure to consult the documentation for your particular web container to determine which specifications it supports.*

Recall from Chapter 4 that you added three new Maven dependencies to your example code. One of these was the JSP API, which simply enables you to compile against the JSP features in your IDE. Another dependency is for the Servlet API. These Maven dependencies have “provided” scope because Tomcat already includes the JSP API library, and as such you do not need to include it in your deployed application. The other two dependencies you added were the JSTL API (the interfaces, abstract classes, and tag descriptions for the JSTL) and the JSTL implementation provided by GlassFish (the JSTL TLD, concrete classes, and implementations of the interfaces). If Tomcat provided a JSTL implementation, you still would need JSTL Maven dependencies, but they would have “provided” scope. Because Tomcat does not provide a JSTL implementation, these libraries are in “compile” scope so that they deploy with your application. This enables you to use JSTL in your application despite Tomcat’s lack of a JSTL implementation.

There are five tag libraries in the Java Standard Tag Library specification:

- Core (core)
- Formatting (fmt)
- Functions (fn)
- SQL (sql)
- XML (x)

You already learned about the Functions library in Chapter 6 while you were exploring the Expression Language. The rest of this chapter is devoted to using the other four libraries and also touches on why using the XML and SQL libraries is generally discouraged. For reference, you can view the TLD documentation for Java EE 5’s JSTL 1.1 at <http://docs.oracle.com/javaee/5/jstl/1.1/docs/tlddocs/>. Unfortunately, there is no public documentation for Java EE 6’s JSTL 1.2, but the changes between these versions were very minor. No new tags were added—just clarifications in the specification. There was no new JSTL version in Java EE 7.

USING THE CORE TAG LIBRARY (C NAMESPACE)

The Core tag library, as the name implies, contains nearly all the core functionality you need to replace the Java code in your JSPs. This includes tools for conditional programming, looping and iterating, and outputting content. When you work on the Customer Support application at the end of this chapter, you will find that almost every line of Java code is replaced with some tag from the Core library. You have already seen a couple of tags from the Core library, so you should be familiar with its `taglib` directive:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
```

There are many tags in the Core library, none of them unimportant. However, some of them are more commonly used than others, and you learn about them first.

<c:out>

The `<c:out>` tag is probably the most commonly used (and sometimes the most misunderstood) tag in the Core tag library. Its purpose is to assist in the outputting of content to your JSPs. You might immediately wonder how this is different from simply using an EL expression to output content. Perhaps more confusing is that `<c:out>` is almost always used with one or more EL expressions!

```
<c:out value="${someVariable}" />
```

Although in this case `<c:out>` may very well be equivalent to simply writing `${someVariable}`, there are some differences. First, in this case, the use of `<c:out>` is actually equivalent to `${fn:escapeXml(someVariable)}`. This is because, by default, `<c:out>` escapes reserved XML characters (`<`, `>`, `',`, `",`, and `&`) just like `fn:escapeXml` does. You can disable this behavior by setting the `escapeXml` attribute to `false`:

```
<c:out value="${someVariable}" escapeXml="false" />
```

However, in most cases you would never want to do this. The default escaping of reserved XML characters helps protect your site from cross-site scripting and various injection attacks, and also helps prevent unexpected special characters from breaking the functionality of your site. There is also the `default` attribute, which specifies a default value if the one provided in the `value` attribute is `null`.

```
<c:out value="${someVariable}" default="Value not specified." />
```

The `default` attribute can also contain an EL expression. (Any attribute in almost any tag can, for that matter.)

```
<c:out value="${someVariable}" default="${someOtherValue}" />
```

Instead of the `default` attribute, you could use nested contents and achieve the same thing. This enables you to use HTML tags, JavaScript, and other JSP tags to generate the default value.

```
<c:out value="${someVariable}">default value</c:out>
```

Finally, note how the `value` attribute works. Normally, the value specified by the EL expression in the `value` attribute is coerced to a `String` and that `String` is written to the output. However, if the EL expression returns a `java.io.Reader`, the contents of that reader are read and then written to the output.

<c:url>

The `<c:url>` tag properly encodes URLs, and rewrites them if necessary to add the session ID, and can also output URLs in your JSP. (In the examples in this book, session IDs in URLs are disabled to prevent session fixation attacks, so you will not see URL rewriting with this tag in action.) The tag accomplishes this behavior together with the `<c:param>` tag, which specifies query parameters to include in the URL. If the URL is a relative URL, the tag prepends the URL with the context path for your application so that the browser receives the correct absolute URL. Consider the following use of the `<c:url>` tag:

```
<c:url value="http://www.example.net/content/news/today.html" />
```

Because this URL is an absolute URL and contains no spaces or other special characters to encode, it is not changed in any way. Using the `<c:url>` tag for such a purpose is really pointless. However, if you had query parameters that you need to include in the URL, that would be a different story.

```
<c:url value="http://www.example.net/content/news/today.jsp">
  <c:param name="story" value="${storyId}" />
  <c:param name="seo" value="${seoString}" />
</c:url>
```

In this case the `<c:url>` tag will properly form and encode the query string. The `story` parameter might not be a problem, but the `seo` parameter (likely a search engine optimization string) could contain spaces, question marks, ampersands, and other special characters, all of which are encoded to ensure that they do not corrupt the URL. Where the `<c:url>` tag is probably most helpful, however, is in the encoding of relative URLs.

Consider that your application is deployed to `http://www.example.org/forums/` and you place the following link tag in your HTML:

```
<a href="/view.jsp?forumId=12">Product Forum</a>
```

When a user clicked that link, they would be taken to `http://www.example.org/view.jsp?forumId=12`. You probably meant for that URL to be relative to the forums application, not the entire website. You could easily change your link to point to `/forums/view.jsp?forumId=12`, but what if you write a forums application that anyone can download and use on their website. You don't know whether they're going to deploy the application to `/forums`, `/discussion`, `/boards`, or even just `/`. This is where the `<c:url>` tag shines.

```
<a href="
  <c:param name="forumId" value="12" />
</c:url">">Product Forum</a>
```

Notice that the `<c:url>` tag here is actually embedded within the attribute of an HTML tag. This is completely legal and quite common. The `<c:url>` tag gets parsed and replaced when the JSP engine renders your JSP, and it treats everything that isn't special JSP syntax as plain text. If your application is deployed to `/forums`, the resulting link points to `/forums/view.jsp?forumId=12`. If it's deployed to `/boards`, it'll be `/boards/view.jsp?forumId=12`. This saves you the trouble of worrying about what context path your application is deployed to. Of course, it's possible that you really wanted the URL to go back to the root of the site. Or maybe you needed it to go to some other deployed application. This is easily accomplished, too, by adding the `context` attribute to the tag.

```
<c:url value="/index.html" context="/" />
<c:url value="/item.jsp?itemId=15" context="/store" />
```

The first tag produces a URL going to the root context, `/index.html`. The second tag produces a URL going to the `/store` context, `/store/item.jsp?itemId=15`.

By default, the `<c:url>` tag outputs the resulting URL to the response. If you have a URL you are going to use multiple times on the page, you can save the resulting URL to a scoped variable instead:

```
<c:url value="/index.jsp" var="homepageUrl" />
<c:url value="/index.jsp" var="homepageUrl" scope="request" />
```

The `var` attribute specifies the name of the EL variable to create and save the resulting URL to. By default it is saved to the page scope (remember the four EL variable scopes: page, request, session, and application), which is normally sufficient. If for some reason you need to save it to a different scope, you can use the `scope` attribute to explicitly specify the scope. Notice that the value of `var` is a plain string, not an EL expression. Although an EL expression can work here, it is useless that way. You are telling the JSP the name of the attribute you want created in the scope, so it should always be a plain string value.

The first tag in the previous example creates a `homepageUrl` attribute in the page scope. The second tag creates the same attribute, but in the request scope instead. Regardless of which scope you save the URL to, you can then reference the URL later in the page with (in this example)

```
 ${homepageUrl}.
```

```
<a href="${homepageUrl}">Home</a>
```

For maximum safety, flexibility, and portability, it is recommended that *all* URLs in JSPs get encoded with `<c:url>` unless the URL is an external URL with no query parameters. Even in that case, using `<c:url>` is still legal and even encouraged in case the URL contains special characters that need encoding.

<c:if>

It is likely obvious to you that the `<c:if>` tag is a conditional tag for controlling when certain content is rendered. Using the `<c:if>` tag is quite straightforward:

```
<c:if test="${something == somethingElse}">
  execute only if test is true
</c:if>
```

The `test` attribute specifies a condition that must evaluate to `true` for the nested content within the `<c:if>` tag to be evaluated. If `test` evaluates to `false`, everything within the tag is ignored. If you have some complex condition that you want to test only once but use multiple times on the page, you can save it to a variable using the `var` attribute (and optionally specify a different `scope`):

```
<c:if test="${someComplexExpressionIsTrue}" var="itWasTrue" />
...
<c:if test="${itWasTrue}">
  do something
</c:if>
...
<c:if test="${itWasTrue}">
  do something else
</c:if>
```

You might immediately wonder if there is a `<c:else>` to accompany `<c:if>`. There is not. `<c:if>` is meant for simple, all-or-nothing conditional blocks. For more complex if/else-if/else logic, you need something more powerful than `<c:if>`.

`<c:choose>`, `<c:when>`, and `<c:otherwise>`

The `<c:choose>`, `<c:when>`, and `<c:otherwise>` tags are the more powerful counterpart to the `<c:if>` tag and provide more complex if/else-if/else logic. The `<c:choose>` tag acts as a frame to indicate the beginnings and end of the complex conditional block. It has no attributes and may contain only white space, `<c:when>`, and `<c:otherwise>` nested within. There must be at least one and may be unlimited `<c:when>` tags within a `<c:choose>`, and all the `<c:when>` tags must come before the `<c:otherwise>` tag. `<c:when>` has one attribute, `test`, which indicates the condition that must evaluate to `true` for the content within that `<c:when>` to execute. Any content or other JSP tags may be nested within `<c:when>`. Only one `<c:when>` tag will have its contents evaluated: The first one whose `test` is `true`. The `<c:choose>` short-circuits to the end after a `<c:when>` tag has evaluated to `true`.

There may be at most one (optional) `<c:otherwise>` tag, and it must be the last tag within `<c:choose>`. It has no attributes, may contain any nested content, and always has its contents executed if and only if none of the `<c:when>` tags within the same `<c:choose>` evaluate to `true`.

```
<c:choose>
  <c:when test="#">{something}">
    "if"
  </c:when>
  <c:when test="#">{somethingElse}">
    "else if"
  </c:when>
  ...
  <c:otherwise>
    "else"
  </c:otherwise>
</c:choose>
```

From the previous code, you can see that the first `<c:when>` is like the initial `if` in Java. It is evaluated first, and if it's `true` everything within it is executed and everything else is ignored. The second `<c:when>` and all other `<c:when>`s are analogous to `else if`. They are tested only if every `<c:when>` before them evaluate to `false`. The `<c:otherwise>` tag is like the final `else`: the backup case that is always evaluated when everything else fails. Of course, your `<c:choose>` could have just one `<c:when>` and nothing else:

```
<c:choose>
  <c:when test="#">{something}">
    "if"
  </c:when>
</c:choose>
```

However, in this case it's much easier to just write `<c:if test="#">{something}"...</c:if>`. When complexity is required, `<c:choose>` can manage it. When your `if` needs no `else`s, `<c:if>` is probably the way to go.

<c:forEach>

The `<c:forEach>` tag is used for iteration and repeats its nested body content some fixed number of times or while iterating over some collection or array of objects. It can act like a standard Java `for` loop or a Java `for-each` loop depending on which attributes you use. For example, say you want to replace the following Java loop with a `<c:forEach>`:

```
for(int i = 0; i < 100; i++)
{
    out.println("Line " + i + "<br />");
}
```

The equivalent `<c:forEach>` tag is as follows:

```
<c:forEach var="i" begin="0" end="100">
    Line ${i}<br />
</c:forEach>
```

In this case, every number between 0 and 100 prints to the screen. The `begin` attribute must be at least 0. If `end` is less than `begin`, the loop never executes. You can also increment `i` by more than 1, if you want to, using the `step` attribute (which must be greater than or equal to 1):

```
<c:forEach var="i" begin="0" end="100" step="3">
    Line ${i}<br />
</c:forEach>
```

In this case, every third number between 0 and 100 prints to the screen.

Using `<c:forEach>` to iterate over some collection of objects is a matter of utilizing different attributes.

```
<c:forEach items="${users}" var="user">
    ${user.lastName}, ${user.firstName}<br />
</c:forEach>
```

The expression within `items` must evaluate to some `Collection`, `Map`, `Iterator`, `Enumeration`, object array, or primitive array. If `items` is a `Map`, the `Map.Entry`s are iterated by calling the `entrySet` method. If `items` is an `Iterator` or `Enumeration`, remember that you cannot rewind these types to the beginning after iteration begins, so you can only iterate over them once. If `items` is `null`, no iteration is performed, just as if the collection were empty. This does not cause a `NullPointerException`. If you have some other class that implements `Iterable` but is not one of these types, you can use it by calling the `iterator` method on the object (`items="${object.iterator()}"`). The `var` attribute specifies the name of the variable to which each element should be assigned for each loop iteration. The previous example is equivalent to the following Java `for-each` loop:

```
for(User user : users)
{
    out.println(user.getLastName() + ", " + user.getFirstName() + "<br />");
```

You can skip collections elements in `<c:forEach>` using the `step` attribute, just like you would for iterating over numbers. You can also use the `begin` attribute to begin iteration at the specified index (inclusive) and the `end` attribute to end iteration at the specified index (inclusive). This is useful to implement paging of a collection of objects, for example.

Finally, whether you use `<c:forEach>` as a `for` loop of numbers or a `for-each` loop over a collection of objects, you can use the `varStatus` attribute to make a variable available within the loop containing the current status of the iteration.

```
<c:forEach items="${users}" var="user" varStatus="status">
    ${status.begin}
    ${status.end}
    ${status.step}
    ${status.count}
    ${status.current}
    ${status.index}
    ${status.first}
    ${status.last}
</c:forEach>
```

In this example the `status` variable encapsulates the status of the current iteration. The properties of this status object (an instance of `javax.servlet.jsp.jstl.core.LoopTagStatus`) follow:

- `begin`—Contains the value of the `begin` attribute from the loop tag.
- `end`—Contains the value of the `end` attribute from the loop tag.
- `step`—Contains the value of the `step` attribute from the loop tag.
- `index`—Returns the current index from the iteration. This value increases by `step` for each iteration.
- `count`—Returns the count of the number of iterations performed so far (including the current iteration). This value increases by 1 for each iteration, even if `step` is greater than 1. The value starts at 1 with the first iteration and is never equal to `status.index`.
- `current`—This contains the current item from the iteration. If you also use the `var` attribute to export the item as a variable, this is the same as that. In the previous example, `status.current equals user`.
- `first`—This is `true` if the current iteration is the very first iteration (if `status.count` is equal to 1). Otherwise it is `false`.
- `last`—This is `true` if the current iteration is the very last iteration. Otherwise it is `false`.

One final thing to consider when using `<c:forEach>` is the impact of EL deferred syntax (`#{}). If you intend to use some tag within the loop that requires deferred syntax in an attribute, and you want to use the variable created as specified in var within that deferred syntax, you must also use deferred syntax for the EL expression in <c:forEach>'s items attribute. Otherwise, the deferred syntax referencing the element variable will not work.`

<c:forTokens>

The `<c:forTokens>` tag is nearly identical to the `<c:forEach>` tag. It contains many of the same attributes (`var`, `varStatus`, `begin`, `end`, and `step`) that behave in the same way as their `<c:forEach>` counterparts when operating in a `for-each` loop over a collection of objects. The major difference with `<c:forTokens>` is that the `items` attribute accepts a `String`, not a collection, and the additional `delims` attribute specifies one or more characters with which to split the `String` into tokens. The tag, then, loops over those tokens.

```
<c:forTokens items="This,is,a,cool,tag." delims="," var="word">
    ${word}<br />
</c:forTokens>
```

<c:redirect>

You have already seen the `<c:redirect>` tag in use in the `index.jsp` files of many of the sample projects. This tag redirects the user to another URL, just as the name implies. After adding the `Location` header to the response and changing the HTTP response status code, it aborts execution of the JSP. Because it changes response headers, `<c:redirect>` must be called before the response has started streaming back to the client. Otherwise, it is not successful in redirecting the client, and the client instead receives a truncated response (with everything after the `<c:redirect>` tag in the JSP missing). `<c:redirect>` follows the same rules as `<c:url>` regarding URL encoding, rewriting for session IDs, and adding query parameters using nested `<c:param>` tags. The following examples are all possible uses of the `<c:redirect>` tag; however, this is certainly not an exhaustive example.

```
<c:redirect url="http://www.example.com/" />

<c:redirect url="/tickets">
    <c:param name="action" value="view" />
    <c:param name="ticketId" value="${ticketId}" />
</c:redirect>

<c:redirect url="/browse" context="/store" />
```

<c:import>

The `<c:import>` tag is a particularly interesting action that enables the retrieval of the contents of the resource at a particular URL. Those contents can then be inlined to the response, saved to a `String` variable, or saved to a `Reader` variable. As with `<c:url>` and `<c:redirect>`, the URL can be for the local context, for another context, or for an external site and is properly encoded and rewritten when necessary. Nested `<c:param>` tags can also specify query parameters to encode into the URL. The `var` attribute specifies the name of the `String` variable to which the content should be saved, and the `scope` attribute can specify the scope of the `String` variable. The `varReader` attribute specifies the name of the `Reader` variable that should be made available for reading the content.

If you use `varReader` to export a `Reader` variable, you cannot use `<c:param>`, and you must use the `Reader` within the nested content of the `<c:import>` tag. The `Reader` variable will not be available after the closing `</c:import>` tag. (This ensures that the JSP engine has the opportunity to close the `Reader`.)

You should never use `var` and `varReader` together; doing so will result in an exception. With neither the `var` nor the `varReader` attributes specified, the content of the resource at the URL is inlined in the JSP. The following examples demonstrate some of the ways that you can use `<c:import>`.

```
<c:import url="/copyright.jsp" />

<c:import url="/ad.jsp" context="/store" var="advertisement" scope="request">
    <c:param name="category" value="${forumCategory}" />
</c:import>

<c:import url="http://www.example.com/embeddedPlayer.do?video=f8ETe9238MNTte" />
```

```

        varReader="player" charEncoding="UTF-8">
<wrox:writeVideoPlugin reader="${player}" />
</c:import>

```

The first example inlines the contents from the application's local `copyright.jsp` in the page. The second saves the contents of `ad.jsp?category=${forumCategory}` to a String named `advertisement` in the request scope, properly encoding the `category` query parameter. The third fetches some external resource and exports it as a `Reader` object named `player`. The imaginary `<wrox:writeVideoPlugin>` tag then uses `player` in some way that is unimportant here.

Notice also the use of the `charEncoding` attribute. Chances are you will never use this attribute. However, if the target resource does not return a `Content-Type` header (very unusual) and the content type is something other than ISO-8859-1, you need to specify what the character encoding is by specifying the `charEncoding` attribute.

<c:set> and <c:remove>

You can use the `<c:set>` tag to set the value of a new or existing scope variable and use its counterpart, `<c:remove>`, to remove a variable from scope.

```

<c:set var="myVariable" value="Hello, World!" />
...
${myVariable}
...
<c:remove var="myVariable" scope="page" />
<c:set var="complexVariable" scope="request">
    nested content including other JSP tags
</c:set>
...
${complexVariable}
...
<c:remove var="complexVariable" scope="request" />

```

As with most other tags that expose scope variables, you can use the `scope` attribute to specify what scope the variable is defined in. (The default is page scope.) Be careful with `<c:remove>` because its `scope` attribute does not work the same: If you do not specify `scope`, all attributes with matching names in all scopes are removed. This is probably not what you want to happen, so you should always use the `scope` attribute.

In addition to the uses previously shown, you can also use `<c:set>` to change the value of a property on a bean.

```

<c:set target="${someObject}" property="propertyName" value="Hello, World!" />
<c:set target="${someObject}" property="propertyName">
    nested content including other JSP tags
</c:set>

```

When used in this manner, the `target` attribute should always be an EL expression that evaluates to either a `Map` or some other bean with mutator methods ("setters") that are used to set property values. The previous examples are both equivalent to calling `someObject.setProperty("propertyName", ...)` for a bean or `someObject.put("propertyName", ...)` for a `Map` in Java code.

Putting Core Library Tags to Use

To get a feel for how the Core tag library works, this section creates an application that lists out contacts in an address book. You can create the address book from scratch or follow along with the Address-Book project available on the wrox.com code download site. Create the standard application with a deployment descriptor containing the standard JSP and session configurations you used in Chapter 6. The `index.jsp` welcome file should redirect to the `/list` servlet using the `<c:redirect>` tag:

```
<c:redirect url="/list" />
```

The base JSP page `/WEB-INF/jsp/base.jsp` should contain `taglib` directives for the Core and Function libraries from the JSTL:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="fn" uri="http://java.sun.com/jsp/jstl/functions" %>
```

You need something to list, so start by creating a simple `Contact` POJO with basic information you would find in an address book. Notice in the following code sample that `Contact` uses the new Java 8 Date and Time API and implements `Comparable` so that it can be sorted appropriately. The mutator and accessor methods and contents of the constructor have been omitted for brevity.

```
public class Contact implements Comparable<Contact>
{
    private String firstName;
    private String lastName;
    private String phoneNumber;
    private String address;
    private MonthDay birthday;
    private Instant dateCreated;

    public Contact() { }

    public Contact(String firstName, String lastName, String phoneNumber,
                  String address, MonthDay birthday,
                  Instant dateCreated) { ... }

    ...

    @Override
    public int compareTo(Contact other)
    {
        int last = lastName.compareTo(other.lastName);
        if(last == 0)
            return firstName.compareTo(other.firstName);
        return last;
    }
}
```

The `ListServlet` in Listing 7-1 responds to requests; it contains a static `Set` of contacts that serves as a database of sorts and is prepopulated with a handful of contacts. The `doGet` method adds an empty `contacts` `Set` to a request attribute if the `empty` parameter is present, or the static `set` if the parameter is not present, and then redirects to the `/WEB-INF/jsp/view/list.jsp` file found in Listing 7-2.

Notice in `list.jsp` the use of `<c:choose>`, `<c:when>`, and `<c:otherwise>` to display a message if the address book is empty and otherwise execute the code to loop over the list. `<c:forEach>` performs this task, and `<c:out>` ensures that the `String` values are properly escaped so that they do not contain XML characters. Finally, the `<c:if>` tag makes sure that the birthday displays only if it is not null.

LISTING 7-1: ListServlet.java

```

@WebServlet(
    name = "listServlet",
    urlPatterns = "/list"
)
public class ListServlet extends HttpServlet
{
    private static final SortedSet<Contact> contacts = new TreeSet<>();

    static {
        contacts.add(new Contact("Jane", "Sanders", "555-1593", "394 E 22nd Ave",
            MonthDay.of(Month.JANUARY, 5),
            Instant.parse("2013-02-01T15:22:23-06:00")
        ));
        contacts.add(new Contact("John", "Smith", "555-0712", "315 Maple St",
            null, Instant.parse("2012-10-15T09:31:17-06:00")
        ));
        contacts.add(new Contact("Scott", "Johnson", "555-9834", "424 Oak Dr",
            MonthDay.of(Month.NOVEMBER, 17),
            Instant.parse("2013-04-04T19:45:01-06:00")
        ));
    }

    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException
    {
        if(request.getParameter("empty") != null)
            request.setAttribute("contacts", Collections.<Contact>emptySet());
        else
            request.setAttribute("contacts", contacts);
        request.getRequestDispatcher("/WEB-INF/jsp/view/list.jsp")
            .forward(request, response);
    }
}

```

LISTING 7-2: list.jsp

```

<%--@elvariable id="contacts" type="java.util.Set<com.wrox.Contact>"--%>
<!DOCTYPE html>
<html>
    <head>
        <title>Address Book</title>
    </head>
    <body>
        <h2>Address Book Contacts</h2>

```

continues

LISTING 7-2 (continued)

```

<c:choose>
    <c:when test="${fn:length(contacts) == 0}">
        <i>There are no contacts in the address book.</i>
    </c:when>
    <c:otherwise>
        <c:forEach items="${contacts}" var="contact">
            <b>
                <c:out value="${contact.lastName}, ${contact.firstName}" />
            </b><br />
            <c:out value="${contact.address}" /><br />
            <c:out value="${contact.phoneNumber}" /><br />
            <c:if test="${contact.birthday != null}">
                Birthday: ${contact.birthday}<br />
            </c:if>
            Created: ${contact.dateCreated}<br /><br />
        </c:forEach>
    </c:otherwise>
</c:choose>
</body>
</html>

```

Compile and debug your project and navigate to `http://localhost:8080/address-book/list?empty` in your browser. You should see the message that there are no contacts in the address book, meaning that the `<c:when>` test evaluated to `true`. Now take the `empty` parameter off the URL, and you should see a screen like that in Figure 7-1. For the most part this looks okay, but the dates are not formatted in a friendly manner, and there is no support for alternative languages. In the next section you learn more about implementing these.

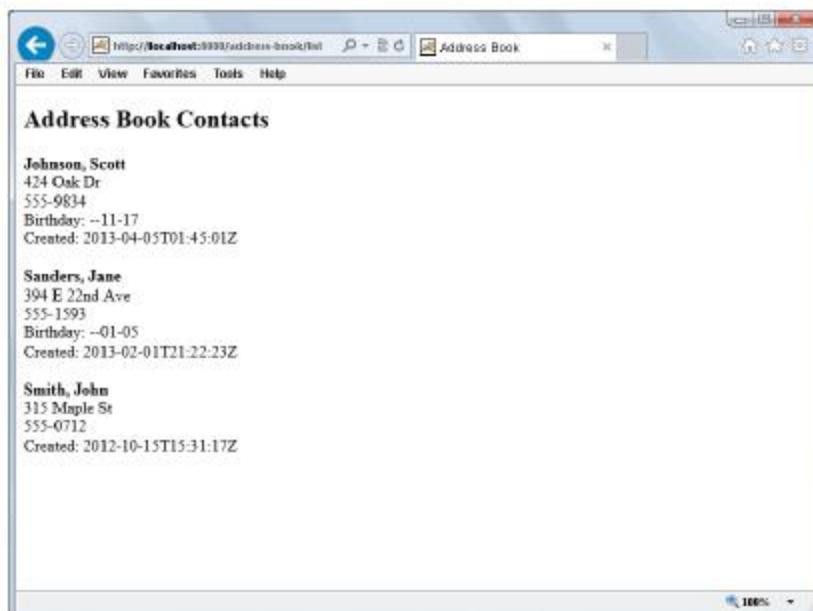


FIGURE 7-1

USING THE INTERNATIONALIZATION AND FORMATTING TAG LIBRARY (FMT NAMESPACE)

If you intend to create enterprise Java applications for deployment on the web and target large, international audiences, you will ultimately need to *localize* your application to particular regions of the world. This is achieved through *internationalization* (often abbreviated i18n), which is the process of designing an application so that it can adapt to different regions, languages, and cultures without redesigning or rewriting the application for new regions.

After an application is internationalized, you can utilize the internationalization framework to *localize* the application by adding support for the target regions, languages, and cultures. Often the terms localization (often abbreviated L10n) and internationalization are confused, which is understandable considering that they are so interrelated. One handy way to remember the difference is, "First you internationalize through architecture, and then you localize through translation."

Internationalization and Localization Components

There are three components to internationalization and localization:

- Text must be translatable and translated so that users who speak and understand other languages can use your application.
- Dates, times, and numbers (including currencies and percentages) must be formatted correctly for different locales. For example, 12,934.52 in the United States is actually 12 934,52 in France and 12.934,52 in Germany.
- Prices must be convertible so that they can display in a local currency to match the client's region of the world.

Often, the conversion of currencies is omitted — and for good reason. Displaying prices in different currencies to users can be extremely challenging, inaccurate, and out of date, and today most business financial institutions provide mechanisms for converting your currency to the user's currency during checkout. In many cases it is perfectly sufficient to simply display prices in U.S. dollars. Due to the complexities of this topic, this book does not cover currency conversion.

The JSTL provides a tag library that supports both internationalization and localization efforts in your applications: the Internationalization and Formatting library, whose prefix is `fmt`. In this section you learn about how to use and configure the Formatting library to internationalize your application so that you can later localize it. You often see the term *locale*, which is an identifier representing a specific region, culture, and/or political area. Locales always contain a *two-letter lowercase language code* as specified by ISO 639-1. You can view a list of all these codes by clicking http://www.loc.gov/standards/iso639-2/php/code_list.php. This page displays 2-letter ISO 639-1 and 3-letter ISO 639-2 codes. You will only use the 2-letter codes for this book.

For languages where the language code is too ambiguous (Mexican Spanish is slightly different from Spanish in Spain, for example) the locale can optionally contain a *two-letter uppercase country code* as specified by ISO 3166-1, which you can view by clicking http://www.iso.org/iso/home/standards/country_codes/iso-3166-1_decoding_table.htm.

In a locale code, the language code always comes first. If a country code is specified, it comes next, with an underscore separating the language and country codes. Occasionally, a locale is also associated with a variant, which is not included as part of the locale code. You do not usually need to worry about variants, but for more information on them, you should see the API documentation for the `java.util.Locale` class. `Locale` is used to represent locales in Java and the JSTL.

The Internationalization and Formatting tag library is divided into two main categories:

- Tags that support internationalization (i18n tags)
- Tags that support date, time, and number formatting (formatting tags)

I18n tags have a sense of resource bundles, which define locale-specific objects. Resource bundles consist of keys that correspond to entries in the bundle and are defined using an arbitrary basename of the developer's choosing with the locale code appended to the basename to form the full resource bundle name. A given key typically has entries in every resource bundle, one for each language and country supported. In this section you learn first about all the i18n tags and then the formatting tags.

The `taglib` directive for the Internationalization and Formatting library follows:

```
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
```

NOTE *One of the traditional complaints about the JSTL Internationalization and Formatting tag library is that it is more difficult to use than it should be. However, this is sometimes also seen as a matter of personal preference, with other developers thinking the JSTL support for i18n and localization is quite adequate. A lot of this depends on your particular needs, and in Chapter 15 you will use Spring Framework i18n tools to explore internationalization more thoroughly and understand your other options.*

<fmt:message>

Perhaps the i18n tag that you will use the most (maybe the only i18n tag you will ever use), `<fmt:message>` resolves a localized message in a resource bundle and then either inlines that message on the page or saves it to an EL variable. The required `key` attribute specifies the key of the localization message to resolve in the resource bundle. You use the optional `bundle` attribute to indicate which localization context, created with `<fmt:setBundle>`, should be used for locating the key. This overrides the default bundle. The optional `var` attribute specifies an EL variable to save the localized message to, and the corresponding `scope` attribute can control which scope the variable goes in. Localization messages can also be parameterized using the `<fmt:param>` tag nested within the `<fmt:message>` tag. Say your application had one resource bundle translated for two languages—U.S. English (`en_US`) and Mexican Spanish (`es_MX`)—and each translation contained an entry for the localization message `store.greeting`. The English value for `store.greeting` might look like this:

```
store.greeting=There are {0} products in the store.
```

The Spanish value is similar, except that it is translated:

```
store.greeting=Hay {0} productos en la tienda.
```

The `{0}` token in each message indicates a placeholder that some parameter should replace. Placeholders are zero-based but they do not have to appear in numerical order within the message. Some languages use very different word orders, meaning a parameter may need to be replaced at different places in different languages. Placeholders can also be duplicated when desirable—you may want to insert the same parameter into a message multiple times. Within your JSP, you reference the localization message using the following code.

```
<fmt:message key="store.greeting">
    <fmt:param value="${numberOfProducts}" />
</fmt:message>
```

If `numberOfProducts` were 63, users would see the following output depending on their selected locale:

English: There are 63 products in the store.

Spanish: Hay 63 productos en la tienda.

The nested `<fmt:param>` tags correspond to the placeholders in order starting from 0. So the first `<fmt:param>` tag specifies a value for `{0}`, the second for `{1}`, and so on, regardless of what order the placeholders actually appear in the localized message. (Messages do not need to have placeholders in them, and you can use `<fmt:message>` without `<fmt:param>`. If a message does contain a placeholder and no `<fmt:param>` is specified for it, the placeholder is left as-is in the message.) Instead of the `value` attribute, you can specify the parameter value by placing it within the `<fmt:param>` opening and closing tags.

```
<fmt:message key="store.greeting">
    <fmt:param>${numberOfProducts}</fmt:param>
</fmt:message>
```

You should also note that the `key` attribute isn't strictly required; rather, the message key is. There is an alternative way of specifying the message key without using the `key` attribute:

```
<fmt:message>some.message.key</fmt:message>

<fmt:message>
    store.greeting
    <fmt:param value="${numberOfProducts}" />
</fmt:message>
```

Although this method is supported, it is rarely used, and many IDEs do not validate message keys unless they are placed within the `key` attribute. In this book, message keys are always specified in the `key` attribute.

NOTE *If you ever see a `<fmt:message>` replaced with `??????`, this means you failed to validly specify a message key (either with the `key` attribute or in the tag body). If some message (such as `store.greeting`) is ever replaced with `???<key>???` (such as `???store.greeting???`), this means that the message key could not be found in the configured resource bundle.*

<fmt:setLocale>

The `<fmt:setLocale>` tag sets the locale used to resolve resource bundle messages for i18n and formatting. The `value` attribute specifies the locale and can be either a `String` locale code (such as `en_US`) or an EL expression evaluating to an instance of `Locale`. If `value` is a locale code, the `variant` attribute can also be specified to indicate a variant of the locale. The locale is saved to an EL variable with the name `javax.servlet.jsp.jstl.fmt.locale` and becomes the default locale in the specified `scope` (which defaults to page scope). If you use this tag, it should come before any other i18n or formatting tags to ensure that the correct locale is used. However, you should not normally need the `<fmt:setLocale>` tag. Internationalized applications typically have a mechanism (such as loading saved locale settings from a user account) by which the locale is automatically set before the request ever gets forwarded to a JSP. You explore this further in Chapter 15.

```
<fmt:setLocale value="en_US" />  
<fmt:setLocale value="${locale}" />
```

<fmt:bundle> and <fmt:setBundle>

I18n tags in JSTL rely on a localization context to inform them of the current resource bundle and locale. You can use `<fmt:setLocale>` and other techniques to specify the locale in the current localization context. `<fmt:bundle>` and `<fmt:setBundle>` are two ways to indicate the resource bundle that should be used. When an i18n tag needs to know its localization context, it looks in several places, using the first specified localization context it finds in this order of precedence:

- If the `bundle` attribute of the `<fmt:message>` tag is specified, it uses that value with preference over all other bundles that might apply to the tag.
- If the i18n tag is nested with a `<fmt:bundle>` tag, it uses this bundle (unless overridden by the `bundle` attribute on `<fmt:message>`).
- If the default localization context is specified using the context `init` parameter or EL variable named `javax.servlet.jsp.jstl.fmt.localizationContext`, it uses that bundle. You learn how to specify this later in this section.

Although the `<fmt:bundle>` tag creates an ad-hoc localization context that affects only the nested tags within, `<fmt:setBundle>` exports a localization context to an EL variable that i18n tags can later use by referencing the bundle variable in the `bundle` attribute. The name of the exported variable is specified in the `var` attribute and has the scope specified in `scope` (which defaults to page scope). If you do not specify `var`, the localization context is saved to the EL variable named `javax.servlet.jsp.jstl.fmt.localizationContext` and becomes the default localization context for that scope. The following example demonstrates the precedence of bundle definitions.

```
<fmt:setBundle basename="Errors" var="errorsBundle" />  
  
<fmt:bundle basename="Titles">  
  <fmt:message key="titles.homepage" />  
  
  <fmt:message key="errors.notFound" bundle="${errorsBundle}" />  
</fmt:bundle>  
  
<fmt:message key="others.greeting" />
```

The `basename` attribute indicates the base name of the resource bundle—that is, the beginning of the resource bundle file, before the locale code is appended to it. The `<fmt:message>` tag outputting the `titles.homepage` message will use the `Titles` bundle defined by the `<fmt:bundle>` tag within which it is nested, whereas the `errors.notFound` message will resolve using the `Errors` bundle defined in the `<fmt:setBundle>` tag. Finally, the `others.greeting` message will resolve using the default localization context.

As with `<fmt:setLocale>`, you will rarely (if ever) use the `<fmt:bundle>` and `<fmt:setBundle>` tags. There are tools that make managing this easier, and they are discussed later in this section and more in depth in Chapter 15.

`<fmt:requestEncoding>`

The `<fmt:requestEncoding>` tag sets the character encoding for the current request using the `var` attribute so that request parameters are correctly decoded with the character encoding appropriate to the given locale. You do not need to use this tag and do not see it used in this book for two reasons:

- Your Java servlets process request parameters before any `<fmt:requestEncoding>` tag has a chance to change the request character encoding.
- All modern browsers include a `Content-Type` request header with a character encoding for any requests whose encoding differs from ISO-8859-1. They all also use the character encoding returned by the last response from the site the request is sent to. This eliminates the need to manually set the request character encoding.

This tag is a legacy tag that originates from an era where the encoding of request attributes was rarely known and had to be guessed. The developer could make an educated guess about the encoding based on the selected language. Today, because of the behavior of modern browsers, the need to do this has gone away.

`<fmt:timeZone>` and `<fmt:setTimeZone>`

The formatting tags that handle dates and times need a locale and a time zone to function properly. The locale comes from the localization context, which you read about earlier in the discussion about `<fmt:bundle>` and `<fmt:setBundle>`, and is determined using the same rules defined in that section. However, the idea of a time zone, while sometimes correlated to the region or language specified in the locale, is not strictly tied to the locale. Someone from the United States may visit Tokyo, for example, and want to use your application in English with standard U.S. date and time formatting while seeing the dates and times in Tokyo time. For this reason and many others, time zones and locales are separate in Java and in the JSTL. Formatting tags that require time zones resolve the time zone to use with the given strategy, in order of precedence:

- If a date and time formatting tag has a value specified for its `timeZone` attribute, use that value with preference over all other time zones.
- If the tag is nested within a `<fmt:timeZone>` tag, use the time zone specified by `<fmt:timeZone>`.

- If the context init parameter or EL variable named `javax.servlet.jsp.jstl.fmt.timeZone` is specified, use that time zone.
- Otherwise, use the time zone provided by the container (typically the JVM time zone, the time zone of the underlying operating system).

The `<fmt:timeZone>` tag is the time zone analog to the `<fmt:bundle>` action. It creates an ad-hoc time zone scope within which any nested tags use the given time zone. Its only attribute is `value`, which can be an EL expression evaluating to a `java.util.TimeZone`, or to a `String` matching any legal time zone ID as specified in the IANA Time Zone Database. You can learn about these IDs on the IANA website or in the API documentation for `TimeZone`. If `value` is `null` or empty, the GMT time zone is assumed.

The `<fmt:setTimeZone>` tag, on the other hand, acts like the `<fmt:setBundle>` tag and exports the value specified time zone to a scope variable. The `var` attribute specifies the name of the EL variable to export the time zone to in the scope specified by `scope` (which defaults to page scope, as usual). If `var` is omitted, the time zone is saved to the EL variable named `javax.servlet.jsp.jstl.fmt.timeZone` and becomes the default time zone for that scope.

```
<fmt:setTimeZone value="America/Chicago" var="timeZoneCst" />

<fmt:timeZone value="${someTimeZone}">
    tags nested here use someTimeZone
</fmt:timeZone>

<fmt:timeZone value="${timeZoneCst}">
    tags nested here use America/Chicago
</fmt:timeZone>
```

<fmt:formatDate> and <fmt:parseDate>

The `<fmt:formatDate>` tag, as the name implies, formats the specified date (and/or time) using the default or specified locale and the default or specified time zone. The formatted date is then either inlined or saved to the variable specified in the attribute `var` with the indicated `scope`. The `timeZone` attribute specifies a different `TimeZone` or `String` time zone ID to format the date.

The date value is specified using the `value` attribute. Currently, `value` must be an EL expression evaluating to an instance of `java.util.Date`; neither `java.util.Calendar` nor the Java 8 Date & Time API classes are supported. In the next chapter, you create a custom tag library with an improved date formatting tag that supports these newer classes. A future version of the JSTL will likely support these types.

How a date is formatted is determined using the locale combined with the `type`, `dateStyle`, `timeStyle`, and `pattern` attributes. `type` should be one of "date," "time," or "both" to indicate whether to output just the date, just the time, or the date followed by the time, respectively. The `dateStyle` and `timeStyle` attributes both follow the semantics defined in the API documentation for `java.text.DateFormat` and must be one of "default" (conveniently, the default), "short," "medium," "long," or "full." These attributes specify how the date and time, respectively, are formatted in relation to their locales. If you need to you can also specify a custom formatting

pattern according to the `java.text.SimpleDateFormat` rules using the `pattern` attribute. In this case the `type`, `dateStyle`, and `timeStyle` attributes are ignored. Doing this also ignores the styles that come with the locale (though the months are still localized to the language), so it's best to avoid using `pattern` if at all possible.

```
<fmt:formatDate value="${someDate}" type="both" dateStyle="long"
                 timeStyle="long" />

<fmt:formatDate value="${someDate}" type="date" dateStyle="short"
                 var="formattedDate" timeZone="${differentTimeZone}" />
```

Given a date of the 3rd day of October in the year 2013 at the time 15:22:37 in the default time zone of America/Chicago, the first example outputs "October 3, 2013 3:22:37 PM CDT" for the U.S. English locale and "3 October 2013 15:22:37 CDT" for the France French locale. The second example formats the date only and in a shorter format in the time zone specified by `${differentTimeZone}` and saves it to the `formattedDate` variable. The value of `formattedDate` is "10/3/13" for the U.S. English locale and "03/10/13" for the France French locale.

If `<fmt:parseDate>` sounds to you like the opposite of `<fmt:formatDate>`, you're exactly right. `<fmt:parseDate>` has all the same attributes and rules as `<fmt:formatDate>`, but it reverses the process. It takes formatted Strings like what `<fmt:formatDate>` would output and parses them into `Date` objects. Typically, you would always assign this to a variable using the `var` attribute; otherwise, it's not very useful. Also, instead of specifying the date to parse using the `value` attribute, you can specify it as the body content within the tag.

<fmt:formatNumber> and <fmt:parseNumber>

The `<fmt:formatNumber>` tag is an extremely powerful action that enables the formatting of numbers (integer-style and decimals), currencies, and percentages. It has many attributes and not all of them apply to all situations. First, know that this tag, like so many others you have seen, has `var` and `scope` attributes that behave as you are accustomed. If `var` is omitted, the formatted number is inlined in the JSP. Now consider the need to format a currency and assume `number` is a scope variable with the value 12349.15823.

```
<fmt:formatNumber type="currency" value="${number}" />
```

This outputs "\$12,349.16" for the U.S. English locale and "12.349,16 €" for the Spain Spanish locale. You should immediately see the problem: The number was represented with two different currency symbols without a currency conversion taking place. This would be inaccurate and confusing for any users of your application. Because of this, you should always specify the `currencyCode` attribute, which can be any valid ISO 4217 currency code. (You can view this list by clicking http://en.wikipedia.org/wiki/ISO_4217).

```
<fmt:formatNumber type="currency" value="${number}" currencyCode="USD" />
```

The output of this is still "\$12,349.16" for the U.S. English locale but is now "12.349,16 USD" for the Spain Spanish locale, which is correct. The `currencySymbol` attribute can also be used to override the currency symbol used, but it is best to leave this alone. The tag can correctly determine the

currency symbol based on the currency code you specify. Both of these attributes are ignored if `type` is not "currency."

Another valid `type` value (and the default) is "number." This formats a number as a generic number. By default it rounds numbers to three digits and groups digits of numbers according to the locale.

```
<fmt:formatNumber type="number" value="${number}" />
```

This outputs "12,349.158" for the U.S. English locale and "12.349,158" for the Spain Spanish locale. You use the `maxFractionDigits` attribute to increase and decrease the rounding accuracy by specifying the number of digits after the decimal separator and the `maxIntegerDigits` to specify the maximum number of digits before the decimal separator. (Don't ever use `maxIntegerDigits` because it can truncate your numbers. For example, if set to 3, the number 12345 becomes 345.) The `minFractionDigits` is used to pad zeroes onto the end of the decimal portion of the number out to the specified number of digits, and likewise `minIntegerDigits` is used to pad zeroes onto the beginning of the integer portion of the number. The `groupingUsed` attribute (default `true`) specifies whether digits should be grouped in the formatted number. If `false`, the output from the previous example would have been "12349.158" and "12349,158." These five attributes are all accepted for all three number `types`.

The third and final valid `type` value is "percent" and is used to format the number as a percentage.

```
<fmt:formatNumber type="percent" value="0.8572" />
```

The output of this is "86%" for both the U.S. English locale and the Spain Spanish locale because the default strategy is to round percentages to whole numbers. If the `maxFractionDigits` attribute were set to 2, the values would be "85.72%" and "85,72%". Notice that the number is automatically multiplied by 100 for you so that it is converted to a percentage. The default value for `maxFractionDigits` is locale-specific for currencies, 3 for numbers and 0 for percentages. There is also a `pattern` attribute with which you can specify a custom pattern to format the number according to the `java.text.DecimalFormat` rules. You generally want to avoid using this attribute.

Like `<fmt:parseDate>`, `<fmt:parseNumber>` reverses the process of `<fmt:formatNumber>`. It does not have the `maxFractionDigits`, `maxIntegerDigits`, `minFractionDigits`, `minIntegerDigits`, or `groupingUsed` attributes because that information is not needed to parse numbers. It does contain the additional attributes `integerOnly` (which specifies whether to ignore the fraction part of the number and defaults to `false`) and `parseLocale` (which specifies the locale to use when parsing the number, if other than the default locale). Also like `<fmt:parseDate>`, you can specify the number to parse in the `value` attribute or in the tag body content.

Putting i18n and Formatting Library Tags to Use

To explore the i18n and Formatting library further, you can expand upon the Address-Book project you created earlier in the book and internationalize it. You can continue to add to the previous project, or you can follow along in the Address-Book-i18n project from the wrox.com code download site. The first thing you should do is add a new context init parameter in the deployment descriptor.

```
<context-param>
    <param-name>javax.servlet.jsp.jstl.fmt.localizationContext</param-name>
    <param-value>AddressBook-messages</param-value>
</context-param>
```

This establishes a resource bundle from which localized messages can be loaded. But where does the container locate this bundle? It looks for a file anywhere on your class path named `AddressBook-messages_[language]_[region].properties`. If it does not find that file, it looks for `AddressBook-messages_[language].properties`. If it doesn't find that either, it switches to the fallback locale (English) and looks for its bundle. To satisfy this need, create a file named `AddressBook-messages_en_US.properties` in the source/production/resources directory of your IDE project. All the files in this resources directory will get copied to the `/WEB-INF/classes` directory at build time.

```
title.browser=Address Book
title.page=Address Book Contacts
message.noContacts=There are no contacts in the address book.
label.birthday=Birthday
label.creationDate=Created
```

You also want a translated version of this file so that you can test switching languages, so create a file named `AddressBook-messages_fr_FR.properties` in the same directory.

```
title.browser=Carnet d'Adresses
title.page=Contacts du Carnet d'Adresses
message.noContacts=Il n'y a pas des contacts dans le carnet d'adresses.
label.birthday=Anniversaire
label.creationDate=Établi
```

You need a way to easily change the language the page displays in, so add the following code to the top of the `doGet` method in the `ListServlet`. The `Config` class here is imported from `javax.servlet.jsp.jstl.core.Config`.

```
String language = request.getParameter("language");
if("french".equalsIgnoreCase(language))
    Config.set(request, Config.FMT_LOCALE, Locale.FRANCE);
```

Because the `<fmt:formatDate>` tag doesn't support the Java 8 Date and Time API, you need a way to access the old-style `Date` object, so add the following method to the `Contact` POJO. (If this seems like a hack to you, that's because it is. It's the easiest way right now to get the `Date` object for formatting. In the next chapter you create a way to format the new API without this hack.)

```
public Date getOldDateCreated()
{
    return new Date(this.dateCreated.toEpochMilli());
}
```

Finally, add the `taglib` directive you learned about earlier in the chapter to the `/WEB-INF/jsp/base.jspf` file. Now that all the groundwork has been laid, take a look at the new (substantially changed) `list.jsp` in Listing 7-3. All literal text is replaced with `<fmt:message>` tags referencing the keys from your properties files. The process of placing the `<fmt:message>` tags in the JSP is the internationalization of your application. The process of creating the properties files that contain the translations is the localization of your application.

Also note the use of `<fmt:formatDate>` to format the creation date for display on the page and how the code that displays the birthday has changed. Now test it all out by compiling and debugging your project and going to `http://localhost:8080/address-book/list` in your browser. You can see that it looks the same for the most part, except that the birthday and creation date are both displaying in a friendly manner. Add `?language=french` to the URL, and the page should now display in French instead of English, as shown in Figure 7-2. Add `&empty` to the URL and you see in French the message that there are no contacts. You have successfully internationalized and localized your application.

LISTING 7-3: list.jsp

```
<%--@elvariable id="contacts" type="java.util.Set<com.wrox.Contact>"--%>
<!DOCTYPE html>
<html>
    <head>
        <title><fmt:message key="title.browser" /></title>
    </head>
    <body>
        <h2><fmt:message key="title.page" /></h2>
        <c:choose>
            <c:when test="${fn:length(contacts) == 0}">
                <i><fmt:message key="message.noContacts" /></i>
            </c:when>
            <c:otherwise>
                <c:forEach items="${contacts}" var="contact">
                    <b>
                        <c:out value="${contact.lastName}, ${contact.firstName}" />
                    </b><br />
                    <c:out value="${contact.address}" /><br />
                    <c:out value="${contact.phoneNumber}" /><br />
                    <c:if test="${contact.birthday != null}">
                        <fmt:message key="label.birthday" />:
                        ${contact.birthday.month.getDisplayName(
                            'FULL', pageContext.response.locale
                        )}&nbsp;${contact.birthday.dayOfMonth}<br />
                    </c:if>
                    <fmt:message key="label.creationDate" />:
                    <fmt:formatDate value="${contact.oldDateCreated}" type="both"
                        dateStyle="long" timeStyle="long" />
                    <br /><br />
                </c:forEach>
            </c:otherwise>
        </c:choose>
    </body>
</html>
```



FIGURE 7-2

USING THE DATABASE ACCESS TAG LIBRARY (SQL NAMESPACE)

The JSTL contains a library of tags that provides transactional access to relational databases. The standard prefix for this library is `sql`, and its `taglib` directive is similar to previous directives you have seen.

```
<%@ taglib prefix="sql" uri="http://java.sun.com/jsp/jstl/sql" %>
```

Generally speaking, performing database actions within the presentation layer (JSPs) is frowned upon and should be avoided if at all possible. Instead, such code should go in the business logic of the application, typically a Servlet or, more appropriately, a repository that the Servlet uses. For this reason, this book does not dive deeply into the details of the SQL tag library and strongly discourages you from using this library. However, this tag library is sometimes useful, especially for prototyping new applications or quickly testing theories or concepts, so an overview of its use is warranted.

The actions in the SQL library provide you with the capability of querying data with `SELECT` statements; accessing and iterating the results of those queries; updating data with `INSERT`, `UPDATE`, and `DELETE` statements; and performing any number of these actions within a transaction. Typically, the tags in this library operate using a `javax.sql.DataSource`. The `<sql:query>`, `<sql:update>`, `<sql:transaction>`, and `<sql:setDataSource>` tags all have `dataSource` attributes for specifying the data source that should be used to perform that action.

The `dataSource` attribute must either be a `DataSource` or a `String`. If it's a `DataSource`, it is used as-is. If it's a `String`, the container attempts to resolve the `String` as a JNDI name for a `DataSource`. If a matching `DataSource` is not found, the container makes a last-ditch effort by treating the `String` as a JDBC connection URL and attempts to connect using the `java.sql.DriverManager`. If none of these works, an exception is thrown. For all tags the `dataSource` attribute is optional, in which case the container looks for the EL variable named `javax.servlet.jsp.jstl.sql.dataSource` in the default scope (set using a context init parameter or `<sql:setDataSource>`). If it is a `String` or a `DataSource`, the same logic previously described is applied. Otherwise, an exception is thrown.

Queries are performed using the `<sql:query>` tag and update actions using the `<sql:update>` tag. For either tag, the SQL statement can be specified in the `sql` attribute or in the nested body content. Nested tags can be used to specify prepared statement parameters. You can create a transaction with `<sql:transaction>`, and all nested query and update tags will use that transaction, but you must keep in mind two rules:

- Only the `<sql:transaction>` tag may specify a `dataSource` attribute (nested tags within it may not).
- If you query data, you must iterate over that data within the transaction as well.

The following code demonstrates some of the things possible with the SQL tag library:

```
<sql:transaction dataSource="${someDataSource}" isolation="read_committed">
    <sql:update sql="UPDATE dbo.Account
        SET Balance = Balance - ?, LastTransaction = ?
        WHERE AccountId = ?">
        <sql:param value="${transferAmount}" />
        <fmt:parseDate var="transactionDate" value="${effectiveDate}" />
        <sql:dateParam value="${transactionDate}" />
        <sql:param value="${sourceAccount}" />
    </sql:update>
    <sql:update>
        UPDATE dbo.Account SET Balance = Balance + ?, LastTransaction = ?
        WHERE AccountId = ?
        <sql:param value="${transferAmount}" />
        <sql:dateParam value="${someLaterDate}" />
        <sql:param value="${destinationAccount}" />
    </sql:update>
</sql:transaction>

<sql:query var="results" sql="SELECT * FROM dbo.User WHERE Status = ?">
    <sql:param value="${statusParameter}" />
</sql:query>

<c:forEach items="${results.rows}" var="user">
    ...
</c:forEach>
```

USING THE XML PROCESSING TAG LIBRARY (X NAMESPACE)

Like the SQL tag library, the XML Processing tag library is not recommended for use and is not covered in-depth in this book. When it was invented, XML was the only widespread standard with which applications shared data, and having the ability to parse and traverse XML was crucial. Today, more and more applications support the JSON standard as an alternative to XML, and several highly efficient libraries can map objects to JSON or XML and back to objects. These tools are easier to use than the XML tag library and can take care of data transformation where it belongs—in the business logic.

The XML tag library, whose prefix is `x`, is based on the XPath standard and consists of nodes or node sets, variable bindings, functions, and namespace prefixes. It contains many actions similar to the tags in the Core tag library but designed specifically to work with XPath expressions against an XML document. The `taglib` directive and a small sampling of what can be done with the XML library are demonstrated in the following code.

```
<%@ taglib prefix="x" uri="http://java.sun.com/jsp/jstl/xml" %>

<c:import url="http://www.example.news/feed.xml" var="feed" />
<x:parse doc="${feed}" var="parsedDoc" />
<x:out select="$parsedDoc/feed/title" />
<x:forEach select="$parsedDoc/feed/stories//story">
    <x:out select="@title" /><br />
    <x:out select="@url" /><br /><br />
</x:forEach>
```

REPLACING JAVA CODE WITH JSP TAGS

In Chapter 6, you replaced a few lines of Java code with Expression Language (EL) in the JSPs for the Customer Support application. However, you simply could not change much because you did not have the tools yet. In this section you see nearly all of the JSP Java code replaced with JSTL tags. You can follow along in the Customer-Support-v5 project from the wrox.com code download site. Start by looking at the `/WEB-INF/jsp/view/login.jsp` page. As shown in the following code sample, not much has changed (because there wasn't much Java code to begin with). The `@elvariable` type hint has been added to the first line of the page above the doctype, and the only scriptlet on the page was replaced with the `<c:if>` tag.

```
<%--@elvariable id="loginFailed" type="java.lang.Boolean"--%>
...
    You must log in to access the customer support site.<br /><br />
    <c:if test="${loginFailed}">
        <b>The username and password you entered are not correct. Please try
        again.</b><br /><br />
    </c:if>
    <form method="POST" action="

```

/WEB-INF/jsp/view/viewTicket.jsp has been changed more, and again all the Java code in the file is now gone. You can see this change in Listing 7-4. The series of scriptlets and expressions that listed links to the attachments has been replaced with a `<c:if>` test of the number of attachments, a `<c:forEach>` loop over the list of attachments, and another `<c:if>` tag using the loop tag status variable to determine whether to print a comma before the current attachment. Also, most EL expressions have been placed within the `value` attribute of `<c:out>` tags. This protects the application from injection of HTML and JavaScript. The `<c:out>` tags escape any XML reserved characters as long as the `escapeXml` attribute is left with its default `true` value. It's good to get into the habit of always using `<c:out>` to output `String` variables. However, variables that you know with certainty are non-`char` primitives (integers, decimals, and so on) do not need escaping.

LISTING 7-4: viewTicket.jsp

```

<%--@elvariable id="ticketId" type="java.lang.String"--%>
<%--@elvariable id="ticket" type="com.wrox.Ticket"--%>
<!DOCTYPE html>
<html>
    <head>
        <title>Customer Support</title>
    </head>
    <body>
        <a href=<c:url value="/login?logout" />>Logout</a>
        <h2>Ticket #${ticketId}: <c:out value="${ticket.subject}" /></h2>
        <i>Customer Name - <c:out value="${ticket.customerName}" /></i><br /><br />
        <c:out value="${ticket.body}" /><br /><br />
        <c:if test="${ticket.numberOfAttachments > 0}">
            Attachments:
            <c:forEach items="${ticket.attachments}" var="attachment"
                varStatus="status">
                <c:if test="${!status.first}">, </c:if>
                <a href=<c:url value="/tickets">
                    <c:param name="action" value="download" />
                    <c:param name="ticketId" value="${ticketId}" />
                    <c:param name="attachment" value="${attachment.name}" />
                </c:url>><c:out value="${attachment.name}" /></a>
            </c:forEach><br /><br />
        </c:if>
        <a href=<c:url value="/tickets" />>Return to list tickets</a>
    </body>
</html>

```

The /WEB-INF/jsp/view/listTickets.jsp file in Listing 7-5 uses many of the same features to replace Java code as Listing 7-4 but also utilizes the more complex `<c:choose>`, `<c:when>`, and `<c:otherwise>` to replace the old `if-else` code. The `<c:when>` tests whether the ticket database is empty and prints, "There Are No Tickets in the System" if it is. Otherwise (hence the name of the tag) the `<c:forEach>` loop iterates over the database. Notice here that the `items` attribute of `<c:forEach>` resolves to a `Map`, so each iteration exposes a `Map.Entry<Integer, Ticket>` variable. It's easy to access the `int` key with `entry.key` and the corresponding `Ticket` value with `entry.value`. The `<c:out>` tag outputs user input in a safe way.

One final note about Listing 7-5: The `@elvariable` type hint is wrapped for the purposes of printing in this book; however, it must be all on one line for your IDE to recognize it.

LISTING 7-5: listTickets.jsp

```
<%--@elvariable id="ticketDatabase"
    type="java.util.Map<Integer, com.wrox.Ticket"--%>
<!DOCTYPE html>
<html>
    <head>
        <title>Customer Support</title>
    </head>
    <body>
        <a href=">Logout</a>
        <h2>Tickets</h2>
        <a href="

```

The only JSP remaining that still contains Java code at this point is `/WEB-INF/jsp/view/sessions.jsp`. This JSP contains a definition for a special method that converts a time interval into friendly text, such as “less than a second” or “about x minutes.” Unfortunately, nothing you have learned in this chapter can easily replace that method or how it’s used. If you really need to, you could calculate this value in some Servlet code and forward to the JSP with a list of POJOs instead of a list of `HttpSessions`. However, that would be more work than necessary, so you can leave replacing this Java code as an exercise to complete in the next chapter.

Now compile and fire up your application and go to `http://localhost:8080/support/` to log in. Create some tickets, view the list of tickets, and view the tickets you created. Include some HTML tags, quotes, and apostrophes in your ticket titles and bodies. In the previous version of the project these would have been printed literally and interpreted as HTML in your browser. You can view the page source to see how they are now escaped properly and don’t pose a danger to your application anymore.

SUMMARY

In this chapter you have learned about all the features that the Java Standard Tag Library (JSTL) brings to the table, and also a little bit more about JSP tags in general and how they are created. You've explored the various facets of the Core tag library and the Internationalization and Formatting tag library, and also took a brief look at accessing databases and parsing XML in JSPs. You saw how to replace Java code with JSP tags and how the JSTL covers almost everything you could need to do in a JSP, and you replaced nearly all the presentation layer Java code in the Customer Support application that you are working on for Multinational Widget Corporation.

You may have noticed that some things could be done better or more easily (such as formatting dates using the Java 8 Date and Time API) if you could create your own tags, and that the clever time interval formatter in the session list still uses Java code. In the next chapter you learn about creating custom tags, functions, and tag and function libraries, and you apply that to get rid of any remaining Java in your JSPs. Finally, recall that you did not internationalize and localize the Customer Support application. In Chapter 15 you study internationalization more in-depth using tools provided by Spring Framework that make the task substantially easier.

8

Writing Custom Tag and Function Libraries

IN THIS CHAPTER

- All about TLDs, tag files, and tag handlers
- Creating an HTML template using a tag file
- How to create a more useful date formatting tag handler
- Abbreviating strings using an EL function
- How to replace Java code with custom JSP tags

WROX.COM CODE DOWNLOADS FOR THIS CHAPTER

You can find the `wrox.com` code downloads for this chapter at www.wrox.com/go/projavaforwebapps on the Download Code tab. The code for this chapter is divided into the following major examples:

- `c.tld`
- `fn.tld`
- Template-Tags Project
- Customer-Support-v6 Project

NEW MAVEN DEPENDENCIES FOR THIS CHAPTER

In addition to the Maven dependencies introduced in previous chapters, you also need the following Maven dependency:

```
<dependency>
  <groupId>org.apache.commons</groupId>
  <artifactId>commons-lang3</artifactId>
```

```

<version>3.1</version>
<scope>compile</scope>
</dependency>

```

UNDERSTANDING TLDS, TAG FILES, AND TAG HANDLERS

In Chapter 7 you explored the Java Standard Tag Library (JSTL) and also got a brief introduction to the Tag Library Descriptor (TLD), a special file that describes the tags and/or functions in a library. In this chapter you learn more about the TLD and how tags and functions are declared within them. You also learn how to create tag handlers and tag files.

All JSP tags result in execution of some tag handler. The tag handler is an implementation of `javax.servlet.jsp.tagext.Tag` or `javax.servlet.jsp.tagext.SimpleTag` and contains the Java code necessary to achieve the tag's wanted behavior. A tag handler is specified within a tag definition in a TLD, and the container uses this information to map a tag in a JSP to the Java code that should execute in place of that tag.

However, tags do not always have to be written as Java classes explicitly. Just like the container can translate and compile JSPs into `HttpServlets`, it can also translate and compile tag files into `SimpleTags`. Tag files are not as powerful as straight Java code, and you cannot do things like parse nested tags within a tag file like you can with an explicit tag handler, but tag files do have the advantages of using simple markup like JSPs and allowing the use of other JSP tags within them. A tag definition in a TLD can point to either a tag handler class or to a tag file. However, you do not have to create a TLD to use a tag defined in a tag file. The `taglib` directive enables you to do this using the `tagdir` attribute:

```
<%@ taglib prefix="myTags" tagdir="/WEB-INF/tags" %>
```

Notice how this `taglib` directive is different from others you have seen. Instead of specifying a URI for a TLD file containing the tag library's definitions, it specifies a directory in which tag files can be found. Any `.tag` or `.tagx` files within the `tagdir` directory are bound to the `myTags` namespace in this case. Tag files in an application *must* be within the `/WEB-INF/tags` directory, but they may also be within a subdirectory of this directory. You could use this to have multiple name spaces of tag files in your application, as in the following example.

```
<%@ taglib prefix="t" tagdir="/WEB-INF/tags/template" %>
<%@ taglib prefix="f" tagdir="/WEB-INF/tags/formats" %>
```

NOTE *The difference between `.tag` and `.tagx` is the same as the difference between `.jsp` and `.jspx`. The `.tag` files contain JSP syntax while `.tagx` files contain JSP Document (XML) syntax. For more information on the difference between JSP and JSP Document syntax, refer to "A Note About JSP Documents" in Chapter 4.*

JSP tag files can also be defined in JAR files within your application's `/WEB-INF/lib` directory, but the rules are slightly different. Whereas tag files in your application must be in `/WEB-INF/tags` and can be declared either in a TLD or with a `taglib` directive pointing to the directory, tag files in a JAR file are placed in the `/META-INF/tags` directory and *must* be declared within a TLD in the `/META-INF` directory (or subdirectory) of the same JAR file.

Reading the Java Standard Tag Library TLD

To write custom tag and function libraries, you must understand the JSP tag library XSD and how to write tags with it. The best way to demonstrate this is by example, and using something you are already familiar with should be helpful, so take a look at the TLD for the Core tag library from the JSTL. You can find this within the `org.glassfish.web:javax.servlet.jsp.jstl` Maven artifact JAR file (look for the `/META-INF/c.tld` file), or you can download it from the `wrox.com` code download site. Start by looking at the initial declaration within the file:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!--...Copyright (c) 2010 Oracle and/or its affiliates. All rights reserved...-->
<taglib xmlns="http://java.sun.com/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
                           http://java.sun.com/xml/ns/javaee/web-jsptaglibrary_2_1.xsd"
         version="2.1">
...
</taglib>
```

If you are at all familiar with XML, you know that this just sets up the document root element and declares that the document is using the XML schema definition `web-jsptaglibrary_2_1.xsd`. This XML schema defines how a TLD is structured, just like `web-app_3_1.xsd` in `web.xml` defines how the deployment descriptor is structured. The only thing of importance about the JSP tag library XSD that is not obvious from reading through a TLD file is that the schema uses a strict element sequence, meaning that any elements you use must come in a certain order, or your TLD file will not validate. The first five elements within the document root declare general information about the tag library.

```
<description>JSTL 1.2 core library</description>
<display-name>JSTL core</display-name>
<tlib-version>1.2</tlib-version>
<short-name>c</short-name>
<uri>http://java.sun.com/jsp/jstl/core</uri>
```

For this code:

- The `<description>` and `<display-name>` elements provide useful names that XML tools (such as your IDE) can display, but are irrelevant to the actual content of the TLD and are completely optional. You can actually have as many `<description>`s and `<display-name>`s as you want (as long as order is maintained), allowing you to specify different display names and descriptions for different languages. Another optional element not shown here is `<icon>`, which must always come after `<display-name>` and before `<tlib-version>`. Don't worry about `<icon>`; you will never need to use it.
- `<tlib-version>` is a required element. Any TLDs you create must have exactly one of these. It defines the version of your tag library and must contain only numbers and periods (groups of numbers that can be separated only with a single period).
- `<short-name>` indicates the preferred and default prefix (or namespace) for this tag library and is also required. It cannot contain white space or start with a number or underscore.
- The fifth element shown here, `<uri>`, defines the URI for this tag library. This element is not required, and (as explained in Chapter 7) the lack of a URI means that the `<jsp-config>`

in the deployment descriptor must contain a `<taglib>` declaration for this tag library to be useable. It's best to always use `<uri>` despite the fact that it's optional, and remember that it doesn't have to be (and really shouldn't be) a URL to an actual resource. Your TLD can have only one `<uri>`.

Defining Validators and Listeners

The next element in the Core TLD defines a validator.

```
<validator>
  <description>
    Provides core validation features for JSTL tags.
  </description>
  <validator-class>
    org.apache.taglibs.standard.tlv.JstlCoreTLV
  </validator-class>
</validator>
```

Validators extend the `javax.servlet.jsp.tagext.TagLibraryValidator` class to validate a JSP at compile time to ensure it uses the library correctly. The validator in this case performs checks that, for example, make sure the `<c:param>` tag is nested only within tags that support it (such as `<c:url>` and `<c:import>`). Validator elements have nested, in order, zero or more `<description>` elements, exactly one required `<validator-class>` element, and zero or more `<init-param>` elements (which work like you're accustomed to in the deployment descriptor). A tag library can have zero or more validators. Validators are difficult to implement because doing so requires you to actually parse the JSP syntax. Because validators are almost never needed and extremely complicated, they are not covered in this book.

Validators are mentioned here only because they must be declared after the `<uri>` element and before any `<listener>` elements. Listener declarations are identical to their counterparts in the deployment descriptor, and any valid Java EE listener class (`ServletContextListener`, `HttpSessionListener`, and others) can be declared here. However, it is extremely unusual for listeners to be declared within a TLD, and you do not see examples of that in this book. You may declare zero or more listeners in a TLD, and immediately following that you may declare zero or more tags, which are the next things you see in the Core TLD.

Defining Tags

The `<tag>` element is the workhorse of the TLD and is responsible for defining tags in your tag library.

```
<tag>
  <description>
    Catches any Throwable that occurs in its body and optionally
    exposes it.
  </description>
  <name>catch</name>
  <tag-class>org.apache.taglibs.standard.tag.common.core.CatchTag</tag-class>
  <body-content>JSP</body-content>
  <attribute>
    <description>
      Name of the exported scoped variable for the
    </description>
  </attribute>
```

exception thrown from a nested action. The type of the scoped variable is the type of the exception thrown.

```

    </description>
    <name>var</name>
    <required>false</required>
    <rtpvalue>false</rtpvalue>
  </attribute>
</tag>
```

It may have zero or more nested `<description>`, `<display-name>`, and `<icon>` elements, just like `<taglib>`. Generally, only `<description>` is used here. After these comes the required `<name>` element, which specifies the name of the JSP tag. In this case, the full tag name is `<c:catch>`, where `c` is the `<short-name>` (prefix) for the tag library and `catch` is the `<name>` of the tag. A tag can obviously have only one name. The `<tag-class>` element comes next and indicates the tag handler class (`Tag` or `SimpleTag`) responsible for executing the tag. Not shown in this example is the optional `<tei-class>` element, which comes next and specifies an extension of `javax.servlet.jsp.tagext.TagExtraInfo` for this tag. `TagExtraInfo` classes can validate the attributes used in a tag at translation time to ensure that their uses are proper. You may see these occasionally, but not often. In the Core tag library only the `<c:import>` and `<c:forEach>` tags provide extra info classes.

Next, `<body-content>` specifies what type of content is allowed nested within the tag. Its valid values are:

- `empty` — This indicates that the tag may not contain any nested content and must be an empty tag.
- `scriptless` — Tags with this body content type may have template text, EL expressions, and JSP tags within them, but no scriptlets or expressions. (Declarations are never allowed within the nested body content of a tag.)
- `JSP` — This indicates that the tag's nested content may be any content that is otherwise valid in the JSP (including scripting, if that is enabled in the JSP, but no declarations).
- `tagdependent` — This tells the container not to evaluate the nested content of the tag, but instead to let the tag evaluate the content itself. Usually this means the content is a different language, such as SQL, XML, or encoded data.

After `<body-content>` come zero or more `<variable>` elements, which is not shown in the previous example and, indeed, cannot be found in any of the JSTL TLDs. These elements provide information about variables defined as a result of using this tag. The `<variable>` element has the following subelements that provide additional information about defined variables:

- `<description>` — This is an optional description for the variable.
- `<name-given>` — The (required) name of the variable created by this tag. This should be a valid Java identifier.
- `<name-from-attribute>` — The (required) name of an attribute whose value determines what the name of the variable will be. Note that this element conflicts with `<name-given>`. You must specify both of these elements, but only one of them should ever have a value. The other should always be empty. The `<name-from-attribute>` element must always refer to the name of a tag attribute whose type is `String`, which does not allow runtime expressions in its value, and which is used to specify the name of the variable.

- `<variable-class>` — This optional element indicates the fully qualified Java class name of the defined variable's type. If not provided, it is assumed to be `String`.
- `<declare>` — This Boolean element, which defaults to `true`, indicates whether the variable defined is a new variable that will require a declaration. If `false`, it means the variable is already defined elsewhere and is merely changing.
- `<scope>` — Indicates the scope in which this variable will be defined. The default value is `NESTED`, which means that the variable is only available to code and actions *nested within the tag*. The other valid values are `AT_BEGIN`, which indicates that the variable is in scope for code nested within the tag *and* code coming after the tag, and `AT_END`, which means that the variable is only in scope for code that comes after the tag and *not* for code nested within the tag.

NOTE *What exactly is a variable in this case? Why doesn't the JSTL use this element in its TLDs? Unfortunately, there's a lot of incomplete or inaccurate information online about `<variable>`s in the TLD (even the XML schema document doesn't make it clear), and as always the whole story can be found in the official JSP specification document. A variable, in this case, kind of refers to both an EL variable and a Java (scripting) variable. In your tag handler or tag file, you create or assign a value to an EL variable by calling `pageContext.setAttribute`, `jspContext.setAttribute`, or `getJspContext().setAttribute` (depending on what type of handler or file it is). The `<variable>` attribute in your TLD tells the container to expect this EL variable and to copy its value to a scripting variable. (Some IDEs also use variable definitions to assist the developer in using the tag.) Without a `<variable>`, users of your tag can still access the variables you define, using EL. With a `<variable>`, on the other hand, users of your tag can access the variables you define using EL or Java code. Because the use of scripting in JSPs is discouraged (see Chapter 4), developers rarely use `<variable>`, and you will not see examples of it in this book.*

WARNING *The `TagExtraInfo` class also provides a way to specify variables defined by a tag. However, that is an old method of achieving this and should not be used anymore. You should only use `<variable>` elements in your TLD for this purpose.*

After the `<variable>`s for your tag, you can define zero or more `<attribute>`s, something demonstrated in the `<c:catch>` tag defined previously. Attributes, as the name implies, define the attributes that can be specified for this tag. There are several subelements of an `<attribute>` that specify the details of the attribute it defines.

- `<description>` — This optionally specifies a description for this attribute.
- `<name>` — This required element indicates the name of this attribute. The value must be a valid Java identifier.

- `<required>` — This is an optional Boolean value that indicates whether this attribute is required when using this tag. The default value is `false`.
- `<rtpvalue>` — This optional Boolean element, which defaults to `false`, indicates whether the attribute allows runtime expressions (EL or scripting expressions) to specify the attribute value. If `true`, runtime expressions are permitted. If `false`, attribute values are considered static and runtime expressions result in a translation-time error.
- `<type>` — This optional element specifies the fully qualified Java class name of the attribute type. If not specified, the type is assumed to be an `Object`.
- `<deferred-value>` — The presence of this optional element indicates that the attribute value is a deferred EL value expression, and as a result the tag handler will be passed the attribute value as a `javax.el.ValueExpression`. Normally, the container evaluates an expression before binding its return value to the attribute value. With deferred value, the unevaluated `ValueExpression` is bound to the attribute, instead. The tag handler can then later evaluate that expression zero or more times as needed. By default, it is assumed that the value of this expression will be coerced into an `Object`, but a nested `<type>` element can specify a more precise type, which is especially helpful when using this attribute in an IDE.
- `<deferred-method>` — The presence of this optional element indicates that the attribute value is a deferred EL method expression, and as a result the tag handler will be passed the attribute value as a `javax.el.MethodExpression`. This is similar to `<deferred-value>` except for the expression type. By default it is assumed that the signature of the method in the expression is `void method()`, but you can use the nested `<method-signature>` element to specify a more precise signature (such as `void execute(java.lang.String)` or `boolean test(java.lang.Object)`) so that the expected return type and the number and type of parameters are correctly documented. The method name here isn't actually important and the container ignores it.
- `<fragment>` — If this optional Boolean element is set to `true`, it makes the attribute type `javax.servlet.jsp.tagext.JspFragment` and tells the container not to evaluate the JSP content contained in the attribute value. The tag handler can then manually evaluate the fragment zero or more times. If omitted, the default is `false`. Using `<fragment>`, code that uses the tag can specify the value of the attribute using a nested `<jsp:attribute>` tag instead of an actual XML attribute, as in the following example. This is the case even if the body content is set to `empty`.

```
<myTags:doSomething>
  <jsp:attribute name="someAttribute">
    Any <b>content</b> <fmt:message key="including.jsp.tags" />.
  </jsp:attribute>
</myTags:doSomething>
```

A related element not often seen, `<dynamic-attributes>`, follows any `<attribute>` tags. You can specify this Boolean element exactly once or you can omit it. The default value is `false`, and it indicates whether attributes not otherwise specified by `<attribute>` elements are still permitted. Where this is most often seen is in a JSP tag that ultimately outputs an HTML tag. The code for that tag could obtain the dynamic attributes and then copy them from the JSP tag to the HTML tag at run time. Spring Framework's form tag library, which you explore in Part II of this book,

uses dynamic attributes for this exact reason. Dynamic attributes always permit EL and scriptlet expressions as values. To use dynamic attributes your tag handler class must implement `javax.servlet.jsp.tagext.DynamicAttributes`.

Following `<dynamic-attributes>` is the optional `<example>` element (of which there can only be one). This is related to `<description>` and contains simple text providing examples of using the tag. Finally, your tag can have zero or more `<tag-extension>`s. This tag provides additional information about a tag useful for consumption by some tool, such as an IDE or validator. Tag extensions never affect the behavior of the tag or the container. They are abstract and do not contain any subelements; instead, the developer is responsible for defining a schema for the tag extensions he wants to implement. Tag extensions are unusual and complicated and are not covered in this book.

Defining Tag Files

You have already seen how you can use the `taglib` directive to collect a directory of tag files into a namespace of custom tags, and you now know that tag files are essentially JSPs with slightly different semantics. You learn the details of these semantics later in this section. However, you should also know how to define tag files within a Tag Library Descriptor. Remember that tag files shipped inside of JAR libraries must be defined inside a TLD. Also, if you have one or more tag files that you would like to group with one or more tag handlers or JSP functions into the same namespace, you need to define those tag files within the TLD even if they are not shipped inside of JAR libraries.

After all the `<tag>` elements in your TLD, you can place zero or more `<tag-file>` elements to define tag files that belong to your library. Within the `<tag-file>` element are the optional `<description>`, `<display-name>`, and `<icon>` elements that you should be accustomed to by now. Typically, only the `<description>` is specified. The `<name>` element is analogous to `<tag>`'s `<name>` element and specifies what the name of the tag following the prefix is. The next element is `<path>`, which specifies the path to the `.tag` file implementing this custom tag. The value of `<path>` must start with `/WEB-INF/tags` in web applications and `/META-INF/tags` in JAR files. The final two elements are `<example>` and `<tag-extension>`, which serve the same purpose as their counterparts within the `<tag>` element. You will not find an example of `<tag-file>` within the JSTL TLDs, but the following XML demonstrates its basic use.

```
<tag-file>
  <description>This tag outputs bar.</description>
  <name>foo</name>
  <path>/WEB-INF/tags/foo.tag</path>
</tag-file>
```

Defining Functions

After defining tag files in your TLD, you may define zero or more JSP functions using the `<function>` element. This is not demonstrated in `c.tld`, but you can view the JSTL functions defined within `fn.tld`, which you can also find in `/META-INF/` in the Maven artifact or download from the wrox.com download site. If you open this file, you can see the familiar header with the tag library description, display name, version, short name, and URI. The next thing you see is the first function:

```

<function>
  <description>
    Tests if an input string contains the specified substring.
  </description>
  <name>contains</name>
  <function-class>
    org.apache.taglibs.standard.functions.Functions
  </function-class>
  <function-signature>
    boolean contains(java.lang.String, java.lang.String)
  </function-signature>
  <example>
    &lt;c:if test="${fn:contains(name, searchString)}">
  </example>
</function>

```

As you can see, defining a function in a TLD is extraordinarily simple. Bypassing the `<description>`, `<display-name>`, `<icon>`, and `<name>` elements you are already familiar with for `<tag>` and `<tag-file>`, the important elements in this example are `<function-class>` and `<function-signature>`. The function class is just the fully qualified name of a standard Java class, and the function signature is literally the signature of a static method on that class. Any public static method on any public class can become a JSP function in this manner. The last two elements that you may use within `<function>` are `<example>` and `<function-extension>`, which are analogous to the `<example>` and `<tag-extension>` elements within `tag` and `tag file` definitions.

WARNING *The `<example>` for the `fn:contains` function contains characters reserved in the XML language. This is bad practice, and very strict XML validators may flag this as an issue. You should always place content like this within a CDATA block (`<![CDATA[special content goes here]]>`).*

Defining Tag Library Extensions

After all `<tag>`s, `<tag-file>`s, and `<function>`s in your TLD, you may define zero or more tag library extensions using the `<taglib-extension>` element. Tag library extensions, like tag extensions and function extensions, do not affect tag or container behavior and simply exist to support tooling. Their concept is abstract and there are no predefined subelements within `<taglib-extension>`; instead, the developer is expected to know what he wants to use the extension for and define the schema himself. This author has never seen a tag extension, function extension, or tag library extension in practice, and so they are not covered in this book.

Comparing JSP Directives and Tag File Directives

As discussed earlier in this chapter, tag files work essentially like JSP files do. They contain the same syntax and must follow the same basic rules, and at run time they get translated and compiled into Java just like JSPs do. Tag files can use any normal template text (including HTML), any other JSP tag, declarations, scriptlets, expressions, and expression language. It should not be surprising, however, that there are some minor differences between the two file formats, mainly concerning

the directives available for tag files. In Chapter 4 you learned about the `page`, `include`, and `taglib` directives and how to use them in JSPs. Tag files can also use the `include` and `taglib` directives to include files and other tag libraries in the JSP, but there is no `page` directive in tag files. The `include` directive can be used to include `.jsp`, `.jspx`, and other `.tag` files in a `.tag` file, or `.jspx` and other `.tagx` files in a `.tagx` file. Using a `taglib` directive in a tag file is identical to using one in a JSP file.

Instead of the `page` directive, tag files have a `tag` directive. This directive replaces the necessary functionality from JSP's `page` directive and also replaces many of the configuration elements from a `<tag>` element in a TLD file. The `tag` directive has the following attributes, none of which are required:

- `pageEncoding` — This is equivalent to the `page` directive `pageEncoding` attribute and sets the character encoding of the tag's output.
- `isELIgnored` — Equivalent to its counterpart in the `page` directive, this instructs the container not to evaluate EL expressions in the tag file and defaults to `false`.
- `language` — This specifies the scripting language used in the tag file (currently only Java is supported), just like the `language` attribute in the `page` directive.
- `deferredSyntaxAllowedAsLiteral` — Just like the `page` directive attribute, this tells the container to ignore and not parse deferred EL syntax within the tag file.
- `trimDirectiveWhitespaces` — This tells the container to trim white space around directives, equivalent to the same attribute on the `page` directive.
- `import` — This attribute works just like the `page` directive's `import` attribute. You can specify one or more comma-separated Java classes to import in this attribute, and you can use the attribute multiple times in the same `tag` directive or across multiple `tag` directives.
- `description` — This is the equivalent of the `<description>` element in a TLD file, and specifying it can be helpful for developers to understand your tag better.
- `display-name` — Equivalent to the `<display-name>` element in a TLD, there is usually no need to specify this.
- `small-icon` and `large-icon` — These attributes essentially replace the `<icon>` element in a TLD, and you should never need to specify them.
- `body-content` — This is the replacement for `<body-content>` in a TLD, with one minor change: Its valid values are `empty`, `scriptless`, and `tagdependent`. The `JSP` value available in a TLD is not valid for the body content of a tag specified in a tag file. Due to the limitations of how tag files work, you cannot use scriptlets or expressions within the nested body content when using a tag that was defined in a tag file. `scriptless` is the default value for this attribute.
- `dynamic-attributes` — This string attribute is the counterpart of the `<dynamic-attributes>` element in a TLD and indicates whether dynamic attributes are enabled. By default the value is `blank`, which means that dynamic attributes are not supported. To enable dynamic attributes, set its value to the name of the EL variable you want created to hold all of the dynamic attributes. The EL variable will have a type of `Map<String, String>`. The map keys will be dynamic attribute names, and the values attribute values.

- `example` — You can use this attribute to indicate example tag usage just like with the `<example>` element in a TLD, but it is very difficult to effectively do this in a directive attribute.

Notice that equivalent directive attributes are missing for the `<name>`, `<tag-class>`, `<tei-class>`, `<variable>`, `<attribute>`, and `<tag-extension>` elements. The tag name is inferred from and always equal to the tag filename (minus the `.tag` extension), and `<tag-class>` is not needed because the tag file is the tag handler (or will be, after the container compiles it). There is no way to specify a `TagExtraInfo` class or a tag extension for tag files because there is simply no equivalent. This leaves `<variable>` and `<attribute>`, which are replaced with the `variable` and `attribute` directives, respectively.

The `variable` directive provides `description`, `name-given`, `name-from-attribute`, `variable-class`, `declare`, and `scope` attributes that are equivalent to their identically named elements in a TLD. It also provides an additional attribute, `alias`, which enables you to specify a local variable name that you can use to reference the variable within your tag file. The `attribute` directive has `description`, `name`, `required`, `rtexprvalue`, `type`, `fragment`, `deferredValue`, `deferredValueType`, `deferredMethod`, and `deferredMethodSignature` attributes that correspond to elements in a TLD.

CREATING YOUR FIRST TAG FILE TO SERVE AS AN HTML TEMPLATE

Now that you are familiar with the details of Tag Library Descriptors and tag files, it's time to create your first custom JSP tag. The simplest way to create a custom tag is by writing a tag file and using a `taglib` directive with the `tagdir` attribute. You need to follow along in the Template-Tags project on the wrox.com code download site because some of the code in the next several sections is too long to print in this book. Note the standard deployment descriptor has changed slightly since Chapter 7: `<scripting-invalid>false</scripting-invalid>` has changed to `<scripting-invalid>true</scripting-invalid>` to disable Java in JSPs. The project also has an `index.jsp` file in the web root that redirects to `/index` with `<c:url>`, and a `/WEB-INF/jsp/base.jspf` file with the following tag library declarations:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
<%@ taglib prefix="fn" uri="http://java.sun.com/jsp/jstl/functions" %>
<%@ taglib prefix="template" tagdir="/WEB-INF/tags/template" %>
```

Next, create a simple Servlet that can respond to requests to `/index`. Right now it may look like overkill for such a simple action, but in later sections you will add some more logic to this.

```
@WebServlet(
    name = "indexServlet",
    urlPatterns = "/index"
)
public class IndexServlet extends HttpServlet
{
    @Override
```

```
protected void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException
{
    String view = "hello";

    request.getRequestDispatcher("/WEB-INF/jsp/view/" + view + ".jsp")
        .forward(request, response);
}
}
```

One of the most powerful things you can do with tag files is establish a system of HTML templates for your application to use. This templating system can take care of many of the repetitive tasks needed on pages across your application, cutting down on duplicated code and making it easier to change the design of your site. To demonstrate this, create a /WEB-INF/tags/template/main.tag file containing the basic JSP layout that most pages in your application use.

```
<%@ tag body-content="scriptless" dynamic-attributes="dynamicAttributes"
       trimDirectiveWhitespaces="true" %>
<%@ attribute name="htmlTitle" type="java.lang.String" rtexprvalue="true"
       required="true" %>
<%@ include file="/WEB-INF/jsp/base.jspf" %>
<!DOCTYPE html>
<html><c:forEach items="${dynamicAttributes}" var="a">
    <c:out value=' ${a.key}="${fn:escapeXml(a.value)}"' escapeXml="false" />
</c:forEach>>
<head>
    <title><c:out value="${fn:trim(htmlTitle)}" /></title>
</head>
<body>
    <jsp:doBody />
</body>
</html>
```

Now examine the previous code example:

- The first directive in this file establishes that uses of the tag can contain body content, that it supports dynamic attributes, that dynamic attributes are accessible with the `dynamicAttributes` EL variable, and that directive white space should be trimmed. Note the use of the `trimDirectiveWhitespaces` attribute to accomplish this; the `<jsp-config>` in the deployment descriptor *does not affect tag files*, so you need to do this manually in each tag file.
- The second directive establishes the explicit `htmlTitle` attribute.
- The third attribute includes the `base.jspf` file, again because `<jsp-config>` does not affect tag files.
- The `<c:forEach>` loop copies all the dynamic attributes into the `<html>` tag. This isn't something you would normally need to do, but it does demonstrate how useful dynamic attributes can be.
- The `<c:out>` tag in the loop may seem unnecessary, but it is done this way for a reason: This ensures that the space between each attribute is not ignored, that the attribute values are escaped properly, and that the quotes around the attribute values are not escaped.

- The `htmlTitle` attribute is output as the document `<title>` using `<c:out>` and the EL expression trimming its value.
- The special `<jsp:doBody>` tag is used within the HTML `<body>`. This tag, which can be used only within tag files, tells the container to evaluate the content of the JSP tag call and place it inline. You could also specify the `var` or `varReader` attributes and the `scope` attribute to output the evaluated body content to a variable instead of inlining it.

Now to put all this to use, create the `/WEB-INF/jsp/view/hello.jsp` file that calls the `<template:main>` tag you have created.

```
<template:main htmlTitle="Template Homepage">
    Hello, Template!
</template:main>
```

That's all there is to it! Compile the application and start your debugger; then navigate to `http://localhost:8080/template-tags/index`. If you view the response source for the resulting page, you see that the document title is "Template Homepage" and that the text "Hello, Template!" was placed within the document body. You have successfully created your first custom JSP tag! In the next section you demonstrate the power of this HTML template more fully when you create another page to show off your next custom tag.

CREATING A MORE USEFUL DATE FORMATTING TAG HANDLER

In the previous chapter, you explored the JSTL and learned about internationalization, localization, and the standard `fmt` tag library. You may recall that you were promised you would create a replacement for the `<fmt:formatDate>` tag, which was so limited in its capabilities. Some of the major drawbacks you may have noticed with this tag were its inability to format anything other than a `java.util.Date`, place the time before the date when needed, or place a token string between the date and time. The first drawback required you to always convert all date and time instances to a `Date`, and the second and third required you to either use two `<fmt:formatDate>` tags or use the `pattern` attribute, neither of which is ideal.

To create a better formatting tag, start by thinking about the date types you would like to support. Obviously, you would want to support `Date` and probably also `Calendar`. Then there's the Java 8 Date and Time API, which has many different types representing dates and times. Thankfully, the API has a common interface, `java.time.temporal.TemporalAccessor`, that all the major types representing dates and times implement. Add some code to the `doGet` method of `IndexServlet` to put dates of each type on the page model.

```
if(request.getParameter("dates") != null)
{
    request.setAttribute("date", new Date());
    request.setAttribute("calendar", Calendar.getInstance());
    request.setAttribute("instant", Instant.now());
    view = "dates";
}
```

Now that you know what you need your tag to do, think about how you want to use it. Create a TLD file that specifies the behavior and attributes for the new date formatting tag. Doing this before actually writing the tag handler is an way of writing an interface based on use cases and then coding to the interface. The tag definition in your TLD is the interface in this case. Listing 8-1 declares a `wrox` tag library with the URI `http://www.wrox.com/jsp/tld/wrox` and specifies a single tag conveniently named `formatDate`.

The entire tag declaration is more than 100 lines long—too long to print in this book. You need to download the Template-Tags project from the `wrox.com` code download site to see the whole thing. The tag has many of the same attributes as `<fmt:formatDate>`, and as you read the TLD, you can understand the purpose for each attribute through its description.

LISTING 8-1: `wrox.tld`

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<taglib xmlns="http://java.sun.com/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
         http://java.sun.com/xml/ns/javaee/web-jsptaglibrary_2_1.xsd"
         version="2.1">

    <tlib-version>1.0</tlib-version>
    <short-name>wrox</short-name>
    <uri>http://www.wrox.com/jsp/tld/wrox</uri>

    <tag>
        <description><![CDATA[...]]></description>
        <name>formatDate</name>
        <tag-class>com.wrox.tag.FormatDateTag</tag-class>
        <body-content>empty</body-content>
        <attribute>
            <description>...</description>
            <name>value</name>
            <required>true</required>
            <rteprvalue>true</rteprvalue>
        </attribute>
        <attribute>
            ...
        </attribute>
        ...
        <dynamic-attributes>false</dynamic-attributes>
    </tag>
</taglib>
```

Implementing this tag is not a simple task. The `FormatDateTag` class, which extends `TagSupport` and utilizes Java 8 lambda expressions and method references to take care of some of the harder work, contains several hundred lines of code, and is too long to print in this book. You can find the implementation of this class in the Template-Tags project on the `wrox.com` code download site. There are many important things to note about this class.

First, the static fields are used to make the reflective access to the Apache/GlassFish JSTL implementation more efficient. The Apache classes have already implemented the hard work of

resolving the locale and time zone information according to the JSTL specification, which consists of several hundred lines of code that it would not be ideal to duplicate. Of course, using these classes requires you to stick to that specific JSTL implementation, which isn't always ideal either. In a true enterprise application, you would probably want to copy the relevant Open Source code into your application so that it was more portable. For demonstration purposes, using the Apache classes is adequate. The private `getLocale` and `getTimeZone` methods use these static fields to invoke the protected methods in the Apache classes.

The `init` method, which is called from the constructor and the `release` method, resets all the tag attributes to their default values. This is important because containers pool and reuse tag handlers to improve efficiency, and between each use of the tag they call the `release` method to reset it before setting all the attributes again. A series of mutator methods ("setters") set all of the values for the tag attributes. The `org.apache.commons.lang3.StringUtils` class from Apache Commons Lang makes testing the string attributes for `null` or blank easier in these methods. The `setTimeZone` method enables a `String`, `java.util.TimeZone`, `java.time.ZoneId`, or `null` value and throws an exception if some other type is passed in.

The `doEndTag` method is the method that the container invokes when it is ready to close the tag. For tags that permit body content, `doStartTag` and `doAfterBody` could also be overridden to execute code at different times, but because this tag doesn't allow body content, only `doEndTag` needs be overridden. The `doEndTag` outputs nothing and removes the scope variable (if applicable) if the date value is `null`. It then invokes one of two `formatDate` methods based on the value type and either outputs the formatted date or sets it to the specified scope variable. The two `formatDate` methods set up the appropriate formatters for the differing date types and then pass those formatters to the third `formatDate` method, which executes them in the proper order depending on the `timeFirst` and `separateDateTimeWith` attribute values. Before Java 8, this would require `DateFormat` and `DateTimeFormatter` to inherit from a common interface, but they do not. Using Java 8 lambdas, references to the `format` methods of each formatter are passed instead and invoked when necessary.

To use the newly created tag, add a `taglib` directive to `/WEB-INF/jsp/base.jspf`:

```
<%@ taglib prefix="wrox" uri="http://www.wrox.com/jsp/tld/wrox" %>
```

Now create the `/WEB-INF/jsp/view/dates.jsp` view to format the dates previously created in the Servlet.

```

<b>Instant, time first with separator:</b>
<wrox:formatDate value="${instant}" type="both" dateStyle="full"
    timeStyle="full" timeFirst="true"
    separateDateTimeWith=" " /><br />
</template:main>

```

Compile and debug your application and go to `http://localhost:8080/template-tags/index?dates` in your browser. You should see all three dates formatted, two of them twice each with different formats, just as specified. This new tag you have created is much more flexible and powerful than the stock `<fmt:formatDate>` tag that comes with the JSTL.

UNDERSTANDING THE DIFFERENT TYPES OF TAG HANDLERS

You can write tag handlers in different ways to achieve many different tasks, and this book simply cannot cover all the possibilities. But it's important to understand the general differences between these different tag types. All tags must implement either the `Tag` interface or `SimpleTag` interface, both of which extend the `JspTag` marker interface. `Tag` is the classic tag handler that has been around since the earliest days of JSP tags, whereas `SimpleTag` was added in JSP 2.0 to make tags—well—simpler to write.

`SimpleTag` may be simpler to use for many tasks, but among these advantages come a key disadvantage: `SimpleTag` classes are instantiated, used exactly once, and then thrown away. `Tag` instances can be pooled and reused, which can net significant performance gains on heavily used tags or tags that carry around many resources. However, you can't ignore how much easier it is to implement a loop, for example, in a `SimpleTag`. (The `invoke` method in this case is writing the invoked body to the JSP output, but passing in a `Writer` instead of `null` causes the body to be written to that `Writer`.)

```

public void doTag() throws JspException, IOException
{
    while(condition-is-true)
    {
        this.getJspContext().setAttribute("someElVariable", value);
        this.getJspBody().invoke(null);
    }
}

```

Compare this to how complex it is to implement the same loop in a `Tag` (implementing `IterationTag`, which extends `Tag`):

```

public int doStartTag() throws JspException
{ // this is invoked exactly once
    if(condition-is-true)
    {
        this.pageContext.setAttribute("someElVariable", value);
        return Tag.EVAL_BODY_INCLUDE;
    }
}

```

```
        return Tag.SKIP_BODY;
    }

    public int doAfterBody() throws JspException
    { // this is invoked as many times as needed
        if(condition-is-true)
        {
            this.pageContext.setAttribute("someElVariable", value);
            return Tag.EVAL_BODY_AGAIN;
        }
        return Tag.SKIP_BODY;
    }

    public int doEndTag() throws JspException
    { // this is invoked exactly once
        return Tag.EVAL_PAGE;
    }
```

You must evaluate each tag you need to create to determine whether it necessitates prioritizing performance over convenience. Keep in mind that `doStartTag` may return only `Tag.SKIP_BODY` or `Tag.EVAL_BODY_INCLUDE`; `doAfterBody` may return only `Tag.SKIP_BODY` or `IterationTag.EVAL_BODY_AGAIN`; and `doEndTag` may return only `Tag.SKIP_PAGE` or `Tag.EVAL_PAGE`.

The following list indicates the various tag interfaces you can implement (and the helper classes you should actually extend) and when you would implement each.

- `SimpleTag` extends `JspTag` — Use this for all tags that do not require the performance gains achieved through pooling. You should normally extend `SimpleTagSupport`.
- `Tag` extends `JspTag` — Use this poolable tag handler for most simple tags that do not do things like loop or access their body content. (The body content can still be evaluated with this tag, just not accessed.) You should normally extend `TagSupport`.
- `LoopTag` extends `Tag` — This tag is not often used, as it is only useful for looping over collections (or arrays) of objects. You should normally extend `LoopTagSupport`.
- `IterationTag` extends `Tag` — This is a more useful iteration tag that can iterate based on any condition, including looping over collections of objects. You should normally extend `IterationTagSupport`.
- `BodyTag` extends `IterationTag` — This iteration tag enables the output of the evaluated body to be buffered into a `BodyContent` object that can later be used for some other purpose (like saving the output to a variable). This is achieved by returning `BodyTag.EVAL_BODY_BUFFERED` from `doStartTag` instead of `Tag.EVAL_BODY_INCLUDE` and then accessing the body content in `doEndTag`. You should normally extend `BodyTagSupport`.

CREATING AN EL FUNCTION TO ABBREVIATE STRINGS

Recall from earlier in the chapter that you can use a TLD to define EL functions as well as tags. Remember that EL function definitions are simple; all you need is a public static method of some class and you can define an EL function that maps to that. You don't even have to write the static method yourself. It can be something from the Java SE library, the Java EE library, or a completely different third-party library. Consider `StringUtils` from the Apache Commons Lang library. It has a very handy method, `abbreviate`, that ensures a string does not exceed a certain length. If the string is too long, it gets shortened and an ellipsis (...) is added to the end of the string. This can be a very useful action in the world of web applications, in which user input can often be long and sometimes needs to be shortened for summary display or to keep the page layout from being thrown off.

To turn this useful method into an EL function that can easily be called from your JSPs, follow these steps:

1. Open the `wrox.tld` file and add the following `<function>` below the date formatting `<tag>`:

```
<function>
    <description>...</description>
    <name>abbreviateString</name>
    <function-class>org.apache.commons.lang3.StringUtils</function-class>
    <function-signature>
        java.lang.String abbreviate(java.lang.String,int)
    </function-signature>
</function>
```

It should be obvious how much easier it is to define an EL function than a JSP tag. The `<function-class>` element specifies the fully qualified class name that the method belongs to and the `<function-signature>` element specifies which method on the class makes up the function.

2. Now add a little bit more logic to the `doGet` method of your `IndexServlet`:

```
else if(request.getParameter("text") != null)
{
    request.setAttribute("shortText", "This is short text.");
    request.setAttribute(
        "longText",
        "This is really long text that should get cut
        off at 32 chars.");
    view = "text";
}
```

3. Create a `/WEB-INF/jsp/view/text.jsp` view to demonstrate using the EL function.

```
<%--@elvariable id="shortText" type="java.lang.String"--%>
<%--@elvariable id="longText" type="java.lang.String"--%>
<template:main htmlTitle="Abbreviating Text">
    <b>Short text:</b> ${wrox:abbreviateString(shortText, 32)}<br />
    <b>Long text:</b> ${wrox:abbreviateString(longText, 32)}<br />
</template:main>
```

4. Compile and start up the Template-Tags project one more time and navigate to `http://localhost:8080/template-tags/index?text` in your browser.

You should see that your new EL function has abbreviated only the second string and has added an ellipsis to the end of that string.

REPLACING JAVA CODE WITH CUSTOM JSP TAGS

Previously you were promised that you would finally get to replace all Java code in the Customer Support application JSPs with JSP tags you created in this chapter. Indeed, you can do this with everything you have learned in this chapter. You can follow along in the Customer-Support-v6 project available on the `wrox.com` code download site, or you can make the changes noted here. You should start by changing `<scripting-invalid>false</scripting-invalid>` in your deployment descriptor to `<scripting-invalid>true</scripting-invalid>`. This change proves that you have replaced all Java code in your JSPs because any JSPs with Java code cannot compile with this setting enabled.

Now copy the `com.wrox.tag.FormatDateTag` class and `/WEB-INF/tld/wrox.tld` file from the Template-Tags project to the support project; you can use the date formatting tag and string abbreviating function in the support application. You can also add another function to the TLD. Create a `TimeUtils` class like the following:

```
public final class TimeUtils
{
    public static String intervalToString(long timeInterval)
    {
        if(timeInterval < 1_000)
            return "less than one second";
        if(timeInterval < 60_000)
            return (timeInterval / 1_000) + " seconds";
        return "about " + (timeInterval / 60_000) + " minutes";
    }
}
```

You can then add a function to the bottom of the TLD that calls the method in the `TimeUtils` class.

```
<function>
    <description>
        Formats a time interval in an attractive way, such as "less than one
        second" or "ten seconds" or "about 12 minutes".
    </description>
    <name>timeIntervalToString</name>
    <function-class>com.wrox.TimeUtils</function-class>
    <function-signature>
        java.lang.String intervalToString(long)
    </function-signature>
</function>
```

Make sure the `/WEB-INF/jsp/base.jspf` file has the appropriate `taglib` declarations in it.

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
<%@ taglib prefix="fn" uri="http://java.sun.com/jsp/jstl/functions" %>
<%@ taglib prefix="wrox" uri="http://www.wrox.com/jsp/tld/wrox" %>
<%@ taglib prefix="template" tagdir="/WEB-INF/tags/template" %>
```

You're almost ready to begin changing your JSPs. You'll also create a few tag files to help you template the support application and avoid duplicating presentation code. Start with `/WEB-INF/tags/template/main.tag` in Listing 8-2, which is much more complex than the template tag from the *Template-Tags* project.

Notice that two of the attributes for this tag are simple strings, but the `headContent` and `navigationContent` attributes are JSP fragments. This allows those attributes to contain JSP content that you can later evaluate. In the proper place, the tag evaluates these fragments using the `<jsp:invoke>` tag. This is similar to `<jsp:body>` except that it acts on fragment attributes instead of the tag body content.

LISTING 8-2: main.tag

```
<%@ tag body-content="scriptless" trimDirectiveWhitespaces="true" %>
<%@ attribute name="htmlTitle" type="java.lang.String" rtexprvalue="true"
   required="true" %>
<%@ attribute name="bodyTitle" type="java.lang.String" rtexprvalue="true"
   required="true" %>
<%@ attribute name="headContent" fragment="true" required="false" %>
<%@ attribute name="navigationContent" fragment="true" required="true" %>
<%@ include file="/WEB-INF/jsp/base.jspf" %>
<!DOCTYPE html>
<html>
  <head>
    <title>Customer Support :: <c:out value="${fn:trim(htmlTitle)}" /></title>
    <link rel="stylesheet"
      href="

```

You might wonder how you're supposed to specify an attribute that contains JSP content; after all, such a value might be many lines long and contain other JSP tags. `/WEB-INF/tags/template/loggedOut.tag` in Listing 8-3 demonstrates this. When you use a JSP tag, you typically specify all the attributes as normal XML attributes and then the contents within the tag make up the entire tag body. However, when an attribute value is too long or contains JSP content, you can use the `<jsp:attribute>` tag within the tag body to specify the attribute value.

At this point the attribute is specified within the tag body content, so to specify the *real* tag body content you need to use the `<jsp:body>` tag. Everything in this tag becomes the tag body for the enclosing tag (`<template:main>` in this case). Listing 8-4, `/WEB-INF/tags/template/basic.tag`, demonstrates this again but with a lot more content in the fragment attributes. This tag specifies several links that go within the sidebar of the page on every page that uses this template tag. It also provides `extraHeadContent` and `extraNavigationContent` attributes to allow consuming pages to add extra content to the head tag or sidebar.

Although tag files cannot technically extend one another, this is essentially what you are doing. The `main` tag sets up a foundation for the template and the `loggedOut` and `basic` tags build upon that foundation. Now you can see the full power of tag files.

LISTING 8-3: `loggedOut.tag`

```
<%@ tag body-content="scriptless" trimDirectiveWhitespaces="true" %>
<%@ attribute name="htmlTitle" type="java.lang.String" rtexprvalue="true"
   required="true" %>
<%@ attribute name="bodyTitle" type="java.lang.String" rtexprvalue="true"
   required="true" %>
<%@ include file="/WEB-INF/jsp/base.jspf" %>
<template:main htmlTitle="${htmlTitle}" bodyTitle="${bodyTitle}">
  <jsp:attribute name="headContent">
    <link rel="stylesheet"
      href=<c:url value="/resource/stylesheet/login.css" />> />
  </jsp:attribute>
  <jsp:attribute name="navigationContent" />
  <jsp:body>
    <jsp:doBody />
  </jsp:body>
</template:main>
```

LISTING 8-4: `basic.tag`

```
<%@ tag body-content="scriptless" trimDirectiveWhitespaces="true" %>
<%@ attribute name="htmlTitle" type="java.lang.String" rtexprvalue="true"
   required="true" %>
<%@ attribute name="bodyTitle" type="java.lang.String" rtexprvalue="true"
   required="true" %>
<%@ attribute name="extraHeadContent" fragment="true" required="false" %>
<%@ attribute name="extraNavigationContent" fragment="true" required="false" %>
<%@ include file="/WEB-INF/jsp/base.jspf" %>
<template:main htmlTitle="${htmlTitle}" bodyTitle="${bodyTitle}">
  <jsp:attribute name="headContent">
    <jsp:invoke fragment="extraHeadContent" />
```

continues

LISTING 8-4 (continued)

```

</jsp:attribute>
<jsp:attribute name="navigationContent">
    <a href=<c:url value="/tickets" />>List Tickets</a><br />
    <a href=<c:url value="/tickets">
        <c:param name="action" value="create" />
    </c:url>>Create a Ticket</a><br />
    <a href=<c:url value="/sessions" />>List Sessions</a><br />
    <a href=<c:url value="/login?logout" />>Log Out</a><br />
    <jsp:invoke fragment="extraNavigationContent" />
</jsp:attribute>
<jsp:body>
    <jsp:doBody />
</jsp:body>
</template:main>

```

NOTE You may have noticed two CSS files used in these tags, /resource/stylesheets/main.css and /resource/stylesheets/login.css. These style sheets make the application more attractive but are not required for the application to function properly, so they are not printed in this book. You can find them in the Customer-Support-v6 project on the wrox.com code download site.

Now you need just a few simple changes to the Java classes in the application. First, add a creation date field and appropriate mutator and accessor to the Ticket class:

```
private OffsetDateTime dateCreated;
```

In the `TicketServlet` class, add the following line to the `createTicket` method to assign a value to the creation date field:

```
ticket.setDateCreated(OffsetDateTime.now());
```

In the `doGet` method of `SessionListServlet`, add the following request attribute before the request dispatcher forwards the request:

```
request.setAttribute("timestamp", System.currentTimeMillis());
```

You are now ready to change the JSPs in /WEB-INF/jsp/view to use the new template and wrox tag libraries. `login.jsp` is simple and uses the `<template:loggedOut>` tag:

```

<%--@elvariable id="loginFailed" type="java.lang.Boolean"--%>
<template:loggedOut htmlTitle="Log In" bodyTitle="Log In">
    You must log in to access the customer support site.<br /><br />
    <c:if test="${loginFailed}">
        <b>The username and password you entered are not correct. Please try
        again.</b><br /><br />
    </c:if>
    <form method="POST" action=<c:url value="/login" />>
        Username<br />
        <input type="text" name="username" /><br /><br />
        Password<br />
        <input type="password" name="password" /><br /><br />
        <input type="submit" value="Log In" />
    </form>
</template:loggedOut>

```

All the other tickets use the `<template:basic>` tag. `ticketForm.jsp` hasn't changed at all except for using the template:

```
<template:basic htmlTitle="Create a Ticket" bodyTitle="Create a Ticket">
    <form method="POST" action="tickets" enctype="multipart/form-data">
        <input type="hidden" name="action" value="create"/>
        Subject<br />
        <input type="text" name="subject"><br /><br />
        Body<br />
        <textarea name="body" rows="5" cols="30"></textarea><br /><br />
        <b>Attachments</b><br />
        <input type="file" name="file1"/><br /><br />
        <input type="submit" value="Submit"/>
    </form>
</template:basic>
```

In addition to using the template, `viewTicket.jsp` now uses the `<wrox:formatDate>` tag to display the date the ticket was created:

```
<%--@elvariable id="ticketId" type="java.lang.String"--%>
<%--@elvariable id="ticket" type="com.wrox.Ticket"--%>
<template:basic htmlTitle="${ticket.subject}"
    bodyTitle="Ticket ${ticketId}: ${ticket.subject}">
    <i>Customer Name - <c:out value="${ticket.customerName}" /><br />
    Created <wrox:formatDate value="${ticket.dateCreated}" type="both"
        timeStyle="long" dateStyle="full" /></i><br /><br />
    <c:out value="${ticket.body}" /><br /><br />
    <c:if test="${ticket.numberAttachments > 0}">
        Attachments:
        <c:forEach items="${ticket.attachments}" var="attachment"
            varStatus="status">
            <c:if test="${!status.first}">, </c:if>
            <a href="

```

`listTickets.jsp` uses not only the date formatter, but also the `wrox:abbreviateString` EL function to truncate ticket subjects to 60 characters:

```
<%--@elvariable id="ticketDatabase"
    type="java.util.Map<Integer, com.wrox.Ticket>" --%>
<template:basic htmlTitle="Tickets" bodyTitle="Tickets">
    <c:choose>
        <c:when test="${fn:length(ticketDatabase) == 0}">
            <i>There are no tickets in the system.</i>
        </c:when>
        <c:otherwise>
            <c:forEach items="${ticketDatabase}" var="entry">
                Ticket ${entry.key}: <a href="

```

```
<c:out value="${wrox:abbreviateString(entry.value.subject, 60)}" />
</a><br />
<c:out value="${entry.value.customerName}" /> created ticket
<wrox:formatDate value="${entry.value.dateCreated}" type="both"
    timeStyle="short" dateStyle="medium" /><br />
<br />
</c:forEach>
</c:otherwise>
</c:choose>
</template:basic>
```

Notice that `sessions.jsp` experienced the largest change. All the Java code for declaring the method, looping over the sessions, and outputting data is gone, replaced by 100 percent JSP code using `<c:forEach>`, `<c:out>`, `<c:if>`, and `wrox:timeIntervalToString`.

```
<%--@elvariable id="timestamp" type="long"--%>
<%--@elvariable id="numberOfSessions" type="int"--%>
<%--@elvariable id="sessionList"
    type="java.util.List<javax.servlet.http.HttpSession>" --%>
<template:basic htmlTitle="Active Sessions" bodyTitle="Active Sessions">
    There are a total of ${numberOfSessions} active sessions in this
    application.<br /><br />
    <c:forEach items="${sessionList}" var="s">
        <c:out value="${s.id} - ${s.getAttribute('username')}" />
        <c:if test="${s.id == pageContext.session.id}">&nbsp; (you)</c:if>
        &nbsp;- last active
        ${wrox:timeIntervalToString(timestamp - s.lastAccessedTime)} ago<br />
    </c:forEach>
</template:basic>
```

You can now compile and run your application and go to `http://localhost:8080/support/login` in your browser. Log in to the support application to create, view, and list tickets. Notice the formatted dates on the view and list pages, and the convenient sidebar on the left side of the page. Create a ticket with a long subject and see how it gets chopped off before wrapping on the list page. You now have all the tools you need to create useful, dynamic JSPs without any Java code embedded in them.

SUMMARY

In this chapter you learned about creating custom JSP tags and EL functions. You explored the Tag Library Descriptor (TLD) by taking a look at TLDs from the Java Standard Tag Library (JSTL) and by creating your own TLD. You examined the concept of tag files and used this technology to create powerful HTML templates to serve as a base for your application's pages. You created a better date and time formatting tag that provides more flexible formatting options and supports `Date`, `Calendar`, and the Java 8 Date and Time API, while maintaining the locale and time zone contract established in the JSTL. At this point you know everything you could need to know to create Java-free JSPs.

From here on you switch gears and look at more advanced technologies. In the next chapter you explore filters and how you can usefully apply them to your applications.

9

Improving Your Application Using Filters

IN THIS CHAPTER

- The purpose of filters
- How to create, declare, and map filters
- How to properly order your filters
- Using filters with asynchronous request handling
- Exploring practical uses for filters
- Using filters to simplify authentication

WROX.COM CODE DOWNLOADS FOR THIS CHAPTER

You can find the [wrox.com](http://www.wrox.com/go/projavaforwebapps) code downloads for this chapter at <http://www.wrox.com/go/projavaforwebapps> on the Download Code tab. The code for this chapter is divided into the following major examples:

- Filter-Order Project
- Filter-Async Project
- Compression-Filter Project
- Customer-Support-v7 Project

NEW MAVEN DEPENDENCIES FOR THIS CHAPTER

There are no new Maven dependencies for this chapter. Continue to use the Maven dependencies introduced in all previous chapters.

UNDERSTANDING THE PURPOSE OF FILTERS

Filters are application components that can intercept requests to resources, responses from resources, or both, and act on those requests or responses in some manner. Filters can inspect and modify requests and responses, and they can even reject, redirect, or forward requests.

A relatively new addition to the Servlet specification, filters were added in Servlet 2.3, improved in Servlet 2.4, and haven't really changed much since then. The `javax.servlet.Filter` interface is very simple and involves the `HttpServletRequest` and `HttpServletResponse` that you are already familiar with. Like Servlets, filters can be declared in the deployment descriptor, programmatically, or with annotations, and they can have init parameters and access the `ServletContext`. The number of uses for filters is really limited only by your imagination; in this section you explore some of the most common applications.

Logging Filters

Filters can be especially useful for logging activity in your application. You'll learn about the concepts of and tools involved with logging in Chapter 11, but one scenario applications developers sometimes face is the need to log every request to the application and what the result of each request is (status code, length, and possibly other information). Usually the web container provides facilities for request logging, but if you need proprietary information displayed in your request log, you can use a filter to log requests instead. You can also use filters to add tracking information that will be used for all logging operations for the request. This is called fish tagging, and you'll learn more about it in Chapter 11.

Authentication Filters

When you need to ensure that only authorized people access your application, this typically involves performing a check on each request to ensure the user is "logged in." The meaning of that term may vary from application to application, but the tedium of performing this check in every Servlet is universal. Filters can make the job easier by centralizing authentication and authorization checks into one location that intercepts all secured requests.

At the end of this chapter, you add a filter to the ongoing Customer Support project that does just this and removes duplicate code throughout the application. In Part IV, you explore Spring Security and use it to add authentication and authorization to your applications. Filters are the foundation for Spring Security and make up a large part of its functionality.

Compression and Encryption Filters

Though it is not always the case (and is becoming less and less common every day), there are still times in which Internet bandwidth is extremely limited and CPU resources are more abundant. In these cases, it's often desirable to spend the necessary CPU cycles compressing data before transmitting it over the wire. You can use a filter to accomplish this: When the request comes in, it remains unaltered, but as the response goes back out to the user, the filter compresses it. This, of course, requires that the user can decompress the response. In the section "Investigating Practical Uses for Filters," you see what's involved in writing a response compression filter.

Filters can also be useful for handling decryption of requests and encryption of responses. Typically, you rely on HTTPS for this; something the web container or web server can natively handle. However, if the consumers of your resources are not a browser or other web client but instead some other application, they could employ a proprietary encryption system understood only by those two applications. In this case, a filter is a prime candidate for decrypting requests as they come in and encrypting responses as they go out. Understand that this is an unusual scenario, and you really should rely on industry-standard tools such as HTTPS for securing your application's communications.

Error Handling Filters

Let's face it: As hard as software developers try to handle errors that arise during the execution of their software, sometimes errors slip through the cracks. When the software is an operating system, the nasty symptom is often a system halt, affectionately referred to in the technology world as "the blue screen of death." With desktop applications, the user might receive a notice that "the application has quit unexpectedly."

With web applications, the result is typically an HTTP response code of 500, often accompanied by a plain HTML page with the words "Internal Server Error" and some diagnostic information. For applications run locally (such as on an intranet), this diagnostic information is usually not harmful and it might actually be useful for the developer to figure out what went wrong. But for web applications run remotely, this diagnostic information can potentially reveal sensitive system information that hackers can use to compromise a system.

For these reasons, you should display a more friendly, generic error page to users (often styled like the rest of your web application) and log the error or notify a system administrator as necessary. A filter is the perfect tool for this task. You can wrap request handling in a `try-catch` block, thus catching and logging any errors. This results in the request being forwarded to a generic error page that does not contain any diagnostic or sensitive information.

CREATING, DECLARING, AND MAPPING FILTERS

Creating a filter is as simple as implementing the `Filter` interface. Its `init` method is called when the filter is initialized and provides access to the filter's configuration, its `init` parameters, and the `ServletContext`, just like the `init` method in a `Servlet`. Similarly, the `destroy` method is called when the web application is shut down. The `doFilter` method is called when a request comes in that is mapped to the filter. It provides access to the `ServletRequest`, `ServletResponse`, and `FilterChain` objects. Although you can use filters to filter more than just HTTP requests and responses, in reality, for your uses, the request is always an `HttpServletRequest` and the response is always an `HttpServletResponse`. In fact, at the moment the Servlet API specification does not support any protocols besides HTTP. Within `doFilter` you can either reject the request or continue it by calling the `doFilter` method of the `FilterChain` object; you can alter the request and the response; and you can wrap the request and response objects.

Understanding the Filter Chain

Although only one `Servlet` can handle a request, any number of filters may intercept a request. Figure 9-1 demonstrates how the filter chain accepts an incoming request and passes it from filter

to filter until all matching filters have been processed, finally passing it on to the Servlet. Calling `FilterChain.doFilter()` triggers the continuation of the filter chain. If the current filter is the last filter in the chain, calling `FilterChain.doFilter()` returns control to the Servlet container, which passes the request to the Servlet. If the current filter does not call `FilterChain.doFilter()`, the chain is interrupted, and the Servlet and any remaining filters never handle the request.



FIGURE 9-1

In this way, the filter chain is very much like a stack (and, indeed, the series of method executions do go on the Java stack). When a request comes in, it goes to the first filter, which is added to the stack. When that filter continues the chain, the next filter is added to the stack. This continues until the request goes to the Servlet, which is the last item added to the stack. As the request completes and the Servlet's `service` method returns, the Servlet is removed from the stack and control passes back to the last filter. When its `doFilter` method returns, that filter is removed from the stack and control returns to the previous filter. This continues until control has returned to the first filter. When its `doFilter` method returns, the stack is empty and request processing completes. Because of this, a filter can take action on a request both before and after the destination Servlet services it.

Mapping to URL Patterns and Servlet Names

Just like Servlets, filters can be mapped to URL patterns. This determines which filter or filters will intercept a request. Any request matching a URL pattern to which a filter is mapped first goes to that filter before going to any matching Servlets. Using URL patterns, you can intercept not only requests to your Servlets, but also to other resources, such as images, CSS files, JavaScript files, and more.

Sometimes, mapping to a particular URL is inconvenient. Possibly you have several URLs — even dozens — that are already mapped to a Servlet, and you want to also map a filter to those URLs. Instead of mapping your filter to a URL or URLs, you can map it to one or more Servlet names. If a request is matched to a Servlet, the container looks for any filters mapped to that Servlet's name and applies them to the request. Later in this section, you learn about how to map filters to URLs and Servlet names. Whether you map your filters using URL patterns, Servlets names, or both, a filter can intercept multiple URL patterns and Servlet names, and multiple filters can intercept the same URL pattern or Servlet name.

Mapping to Different Request Dispatcher Types

In a Servlet container, you can dispatch requests in any number of ways. There are:

- **Normal requests** — These originate from the client and include a URL targeted for a particular web application in the container.

- **Forwarded requests**—These trigger when your code calls the `forward` method on a `RequestDispatcher` or uses the `<jsp:forward>` tag. Though they are related to the original request, they are handled internally as a separate request.
- **Included requests**—Similarly, using `<jsp:include>` and calling `include` on a `RequestDispatcher` result in separate, internal include requests related to the original requests. (Remember that this is contrary to `<%@ include %>`.)
- **Error resource requests**—These are requests to error pages for handling HTTP errors such as `404 Not Found`, `500 Internal Server Error`, and so on.
- **Asynchronous requests**—These are requests that are dispatched from the `AsyncContext` during the handling of any other request.

Prior to Servlet 2.4, filters applied only to resources for typical requests. Servlet 2.4 added the ability to map filters to forwarded requests, include requests, and error resources, greatly expanding their capabilities. In Servlet 3.0 (Java EE 6), the new asynchronous request handling presented a challenge for filter writers: Because the Servlet's `service` method returns before the request's response has been sent, the capability of the filter chain is compromised. To compensate for this, Servlet 3.0 added the new asynchronous dispatcher type for filters intercepting requests dispatched from the `AsyncContext`. Asynchronous filters should be implemented with caution because they can be invoked multiple times (potentially in different threads) for a single asynchronous request. This is covered in more detail in the next section.

You indicate which dispatcher type or types a filter should apply to when you declare and map a filter, which you learn about in the remainder of this section.

Using the Deployment Descriptor

Before any filters you write can intercept requests, you must declare and map them just like you do your Servlets. As with Servlets, you can accomplish this in multiple ways. The traditional way is in the deployment descriptor using the `<filter>` and `<filter-mapping>` elements (analogous to the `<servlet>` and `<servlet-mapping>` elements). `<filter>` elements must contain at least a name and a class but may also include a description, display name, icon, and one or more init parameters.

```
<filter>
  <filter-name>myFilter</filter-name>
  <filter-class>com.wrox.MyFilter</filter-class>
</filter>
```

The previous code snippet demonstrates a simple filter declaration within the deployment descriptor. Unlike Servlets, filters cannot be loaded on the first request. A filter's `init` method is always called on application startup: After `ServletContextListeners` initialize, before Servlets initialize, and in the order the filter appears in the deployment descriptor.

After a filter has been declared, you can map it to any number of URLs or Servlet names. Filter URL mappings can also include wildcards, just like Servlet URL mappings.

```
<filter-mapping>
  <filter-name>myFilter</filter-name>
  <url-pattern>/foo</url-pattern>
  <url-pattern>/bar/*</url-pattern>
```

```
<servlet-name>myServlet</servlet-name>
<dispatcher>REQUEST</dispatcher>
<dispatcher>ASYNC</dispatcher>
</filter-mapping>
```

In this case, the filter responds to any request to the application-relative URLs `/foo` and `/bar/*`, as well as any requests that end up being serviced by the Servlet named `myServlet`. The two `<dispatcher>` elements mean that it can respond to normal requests and requests dispatched from the `AsyncContext`. The valid `<dispatcher>` types are `REQUEST`, `FORWARD`, `INCLUDE`, `ERROR`, and `ASYNC`. A filter mapping may have zero or more `<dispatcher>` elements. If none are specified, a single, default `REQUEST` dispatcher is assumed.

Using Annotations

As with Servlets, you can declare and map filters using annotations. The `@javax.servlet.annotation.WebFilter` annotation contains attributes that substitute for all the options in the deployment descriptor. The following code has the equivalent effect of the previous filter declaration and mapping in the deployment descriptor:

```
@WebFilter(
    filterName = "myFilter",
    urlPatterns = { "/foo", "/bar/*" },
    servletNames = { "myServlet" },
    dispatcherTypes = { DispatcherType.REQUEST, DispatcherType.ASYNC }
)
public class MyFilter implements Filter
```

The primary disadvantage of using annotations to declare and map filters is the inability to order those filters on the filter chain. Filters have a particular order (which you learn about in the next section) that is very important to their proper interaction. If you want to control the order in which your filters execute without using the deployment descriptor, you need to use programmatic configuration. Hopefully, the future Java EE 8 will include the capability of ordering annotated filters.

Using Programmatic Configuration

You can configure filters programmatically with the `ServletContext`, just like Servlets, listeners, and other components. Instead of using the deployment descriptor or annotations, you can call methods on the `ServletContext` to register and map filters. Because this must be done before the `ServletContext` finishes starting, it is typically accomplished within a `ServletContextListener`'s `contextInitialized` method. (You can also add filters within a `ServletContainerInitializer`'s `onStartup` method, which you learn more about in Part II.)

```
@WebListener
public class Configurator implements ServletContextListener
{
    @Override
    public void contextInitialized(ServletContextEvent event)
    {
```

```
ServletContext context = event.getServletContext();

FilterRegistration.Dynamic registration =
        context.addFilter("myFilter", new MyFilter());
registration.addMappingForUrlPatterns(
        EnumSet.of(DispatcherType.REQUEST, DispatcherType.ASYNC),
        false, "/foo", "/bar/*"
);
registration.addMappingForServletNames(
        EnumSet.of(DispatcherType.REQUEST, DispatcherType.ASYNC),
        false, "myServlet"
);
}
```

In this example, the filter is added to the `ServletContext` using the `addFilter` method.

This returns a `javax.servlet.FilterRegistration.Dynamic`, which you can use to add filter mappings for URL patterns and Servlet names. The `addMappingForUrlPatterns` and `addMappingForServletNames` methods both accept a `Set` of `javax.servlet.DispatcherTypes` as the first argument. As with the deployment descriptor, if the `dispatcher types` argument is `null`, the default `REQUEST` dispatcher is assumed:

```
registration.addMappingForUrlPatterns(null, false, "/foo", "bar/*");
```

The second method parameter indicates the filter's order relative to filters in the deployment descriptor. If `false` (as in this case), the programmatic filter mapping is ordered before any filter mappings in the deployment descriptor. If `true`, mappings in the deployment descriptor come first. You'll learn more about filter order in the next section. The final parameter is a vararg parameter for specifying the URL patterns (for `addMappingForUrlPatterns`) or Servlet names (for `addMappingForServletNames`) to map the filter to.

ORDERING YOUR FILTERS PROPERLY

So far throughout this chapter, you have seen several references to the filter order, but you are undoubtedly wondering what exactly that is. Filter order determines where in the filter chain a filter appears, which in turn determines when one filter processes a request in relation to other filters. In some cases it does not matter what order your filters process requests; in other cases, however, it can be critical — it completely depends on how you are using filters. For example, a filter that sets up logging information for a request (or enters the request in a log) should probably come before any other filters because other filters might alter the fate of the request. As discussed earlier, you cannot order filters declared using annotations, making them virtually useless for most enterprise applications. You will use the deployment descriptor or programmatic configuration extensively, but you will probably never use annotations for configuring filters.

URL Pattern Mapping versus Servlet Name Mapping

Defining the filter order is simple: Filters that match a request are added to the filter chain in the order their mappings appear in the deployment descriptor or programmatic configuration. (And remember, if you configure some filters in the deployment descriptor and others programmatically,

you can determine whether a programmatic mapping comes before the XML mappings using the second argument to the `addMapping*` methods.) Filter order is demonstrated in Figure 9-2, where different requests match different filters but always in the same order. However, the order is not quite as simple as that: URL mappings are given preference over Servlet name mappings. If two filters match a request, one by a URL pattern and the other by a Servlet name, the filter that matches by URL pattern always is present on the chain *before* the filter that matches by Servlet name, as shown in Figure 9-3, even if its mapping appears afterward. To demonstrate this, consider the following mappings:

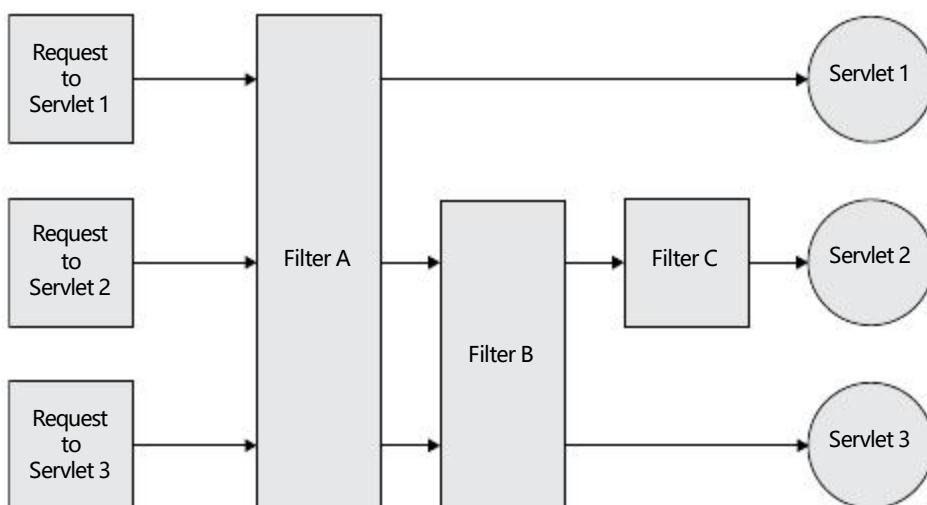


FIGURE 9-2

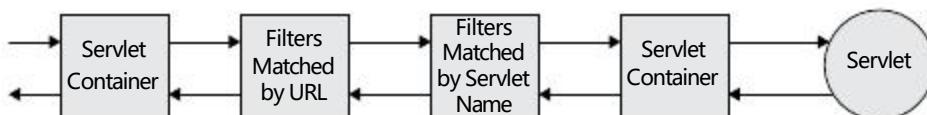


FIGURE 9-3

```

<servlet-mapping>
  <servlet-name>myServlet</servlet-name>
  <url-pattern>/foo*</url-pattern>
</servlet-mapping>

<filter-mapping>
  <filter-name>servletFilter</filter-name>
  <servlet-name>myServlet</servlet-name>
</filter-mapping>

<filter-mapping>
  <filter-name>myFilter</filter-name>
  <url-pattern>/foo*</url-pattern>

```

```

</filter-mapping>

<filter-mapping>
    <filter-name>anotherFilter</filter-name>
    <url-pattern>/foo/bar</url-pattern>
</filter-mapping>

```

If a normal request comes in to the URL `/foo/bar`, it will match all three filters. The filter chain will consist, in order, of `myFilter`, `anotherFilter`, and then `servletFilter`. `myFilter` executes before `anotherFilter` because that's the order they appear in the deployment descriptor. They both execute before `servletFilter` because URL mappings always come before Servlet-name mappings. If the request is a forward, include, error dispatch, or asynchronous dispatch, it will not match any of these filters, because the mappings do not specify any `<dispatcher>`s explicitly.

Exploring Filter Order with a Simple Example

To better understand how filter order works, take a look at the Filter-Order project on the [wrox.com](http://www.wrox.com) code download site. It contains three Servlets and three filters. The following code snippet, `ServletOne`, is identical to its counterparts `ServletTwo` and `ServletThree`, except that all occurrences of "One" have been replaced with "Two" and "Three," respectively:

```

@WebServlet(name = "servletOne", urlPatterns = "/servletOne")
public class ServletOne extends HttpServlet
{
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        System.out.println("Entering ServletOne.doGet()");
        response.getWriter().write("Servlet One");
        System.out.println("Leaving ServletOne.doGet()");
    }
}

```

Similarly, the following `FilterA` is identical to `FilterB` and `FilterC`:

```

public class FilterA implements Filter
{
    @Override
    public void doFilter(ServletRequest request, ServletResponse response,
        FilterChain chain) throws IOException, ServletException {
        System.out.println("Entering FilterA.doFilter()");
        chain.doFilter(request, response);
        System.out.println("Leaving FilterA.doFilter()");
    }

    @Override
    public void init(FilterConfig config) throws ServletException { }

    @Override
    public void destroy() { }
}

```

The filters are mapped to requests in the deployment descriptor as follows so that they match the mappings shown in Figure 9-2.

```
<filter>
  <filter-name>filterA</filter-name>
  <filter-class>com.wrox.FilterA</filter-class>
</filter>

<filter-mapping>
  <filter-name>filterA</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>

<filter>
  <filter-name>filterB</filter-name>
  <filter-class>com.wrox.FilterB</filter-class>
</filter>

<filter-mapping>
  <filter-name>filterB</filter-name>
  <url-pattern>/servletTwo</url-pattern>
  <url-pattern>/servletThree</url-pattern>
</filter-mapping>

<filter>
  <filter-name>filterC</filter-name>
  <filter-class>com.wrox.FilterC</filter-class>
</filter>

<filter-mapping>
  <filter-name>filterC</filter-name>
  <url-pattern>/servletTwo</url-pattern>
</filter-mapping>
```

To try this out:

1. Compile the application and start Tomcat from your IDE.
2. Go to `http://localhost:8080/filters/servletOne` in your browser. You can see several messages printed to the standard output from Tomcat in your IDE:

```
Entering FilterA.doFilter().
Entering ServletOne.doGet().
Leaving ServletOne.doGet().
Leaving FilterA.doFilter().
```

3. Change the address in your browser to `/servletTwo` and observe the new output:

```
Entering FilterA.doFilter().
Entering FilterB.doFilter().
Entering FilterC.doFilter().
Entering ServletTwo.doGet().
Leaving ServletTwo.doGet().
Leaving FilterC.doFilter().
Leaving FilterB.doFilter().
Leaving FilterA.doFilter().
```

Notice how the Filter chain progresses from A to C and then the Servlet. Then after the Servlet completes processing the request, the chain exits in reverse order from C to A.

- 4 Change the address in your browser to `/servletThree`. Its output should look like the following code.

```
Entering FilterA.doFilter().
Entering FilterB.doFilter().
Entering ServletThree.doGet().
Leaving ServletThree.doGet().
Leaving FilterB.doFilter().
Leaving FilterA.doFilter().
```

5. Make any changes to the mappings that you can think of to explore how it affects the execution of the filter chain. Try changing one or more of the URL mappings to Servlet name mappings and notice how the filter chain changes again.

Using Filters with Asynchronous Request Handling

As mentioned in the previous section, filters that apply to asynchronous request handling can be tricky to correctly implement and configure. The key problem with asynchronous request handling is that the Servlet's `service` method can return before a response is sent to the client. Request handling can then be delegated to another thread or completed based on some event.

For example, the `service` method (or, by extension, the `doGet`, `doPost`, or other method) could start the `AsyncContext`, register a listener for some type of hypothetical message (such as a chat request received), and then return. Then, when the hypothetical message listener receives a message, it could send the response back to the user. Using this technique, a request thread is not blocked waiting while the request handling is paused. A filter that intercepts such a request would complete before the response is actually sent because when `service` returns, `FilterChain`'s `doChain` method returns.

Filters mapped to the `ASYNC` dispatcher intercept internal requests made as a result of calling one of `AsyncContext`'s `dispatch` methods. To demonstrate this more complex filtering, look at the [Filter-Async](#) on the `wrox.com` code download site. Its `AnyRequestFilter` wraps the request and response (something you explore more in the next section) and can filter any type of request. If it detects that the Servlet started an `AsyncContext`, it prints that information and indicates whether the `AsyncContext` is using the original request and response or the unwrapped request and response.

```
public class AnyRequestFilter implements Filter
{
    private String name;

    @Override
    public void init(FilterConfig config)
    {
        this.name = config.getFilterName();
    }

    @Override
    public void doFilter(ServletRequest request, ServletResponse response,
                        FilterChain chain) throws IOException, ServletException
    {
```

```

        System.out.println("Entering " + this.name + ".doFilter().");
        chain.doFilter(
            new HttpServletRequestWrapper((HttpServletRequest)request),
            new HttpServletResponseWrapper((HttpServletResponse)response)
        );
        if(request.isAsyncSupported() && request.isAsyncStarted())
        {
            AsyncContext context = request.getAsyncContext();
            System.out.println("Leaving " + this.name + ".doFilter(), async " +
                "context holds wrapped request/response = " +
                !context.hasOriginalRequestAndResponse());
        }
        else
            System.out.println("Leaving " + this.name + ".doFilter().");
    }

    @Override
    public void destroy() { }
}

```

This filter is instantiated and mapped three times in `web.xml`. All three mappings can intercept any URL, but the `normalFilter` instance intercepts only normal requests; the `forwardFilter` intercepts only forwarded requests; and the `asyncFilter` intercepts only requests dispatched from the `AsyncContext`. Notice the addition of `<async-supported>true</async-supported>` to each `<filter>` element. This tells the container that the filter is prepared for asynchronous requests. If a filter without `<async-supported>` enabled filters a request, attempting to start an `AsyncContext` in that request will result in an `IllegalStateException`.

The `NonAsyncServlet` is very straightforward: It responds to requests to `/regular` and forwards to the `nonAsync.jsp` view.

```

@.WebServlet(name = "nonAsyncServlet", urlPatterns = "/regular")
public class NonAsyncServlet extends HttpServlet
{
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException
    {
        System.out.println("Entering NonAsyncServlet.doGet().");
        request.getRequestDispatcher("/WEB-INF/jsp/view/nonAsync.jsp")
            .forward(request, response);
        System.out.println("Leaving NonAsyncServlet.doGet().");
    }
}

```

`AsyncServlet` in Listing 9-1 is much more complex. To make logging very clear, it generates a unique ID for the current request. If the `unwrap` parameter is not present, it starts the `AsyncContext` with `startAsync(HttpServletRequest, HttpServletResponse)`. This ensures the `AsyncContext` gets the request and response as passed into `doGet`. If a filter wrapped the request or response, the wrapper is what the `AsyncContext` uses. However, if the `unwrap` is present, the `doGet` starts the `AsyncContext` using the no-argument `startAsync` method. In this case, the `AsyncContext` gets the original request and response, not the wrapped request and response. Notice the call to `AsyncContext`'s

`start(Runnable)` method (using Java 8 method references). Using this tells the container to run the `Runnable` with its internal thread pool. You could also simply start your own thread, but using the container's thread pool is safer and avoids resource exhaustion.

LISTING 9-1: AsyncServlet.java

```

@WebServlet(name = "asyncServlet", urlPatterns = "/async", asyncSupported = true)
public class AsyncServlet extends HttpServlet
{
    private static volatile int ID = 1;

    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException
    {
        final int id;
        synchronized(AsyncServlet.class)
        {
            id = ID++;
        }
        long timeout = request.getParameter("timeout") == null ?
            10_000L : Long.parseLong(request.getParameter("timeout"));

        System.out.println("Entering AsyncServlet.doGet(). Request ID = " + id +
            ", isAsyncStarted = " + request.isAsyncStarted());

        final AsyncContext context = request.getParameter("unwrap") != null ?
            request.startAsync() : request.startAsync(request, response);
        context.setTimeout(timeout);

        System.out.println("Starting asynchronous thread. Request ID = " + id +
            ".");

        AsyncThread thread = new AsyncThread(id, context);
        context.start(thread::doWork);

        System.out.println("Leaving AsyncServlet.doGet(). Request ID = " + id +
            ", isAsyncStarted = " + request.isAsyncStarted());
    }

    private static class AsyncThread
    {
        private final int id;
        private final AsyncContext context;

        public AsyncThread(int id, AsyncContext context) { ... }

        public void doWork()
        {
            System.out.println("Asynchronous thread started. Request ID = " +
                this.id + ".");
        }

        try {

```

continues

LISTING 9-1 *(continued)*

```
        Thread.sleep(5_000L);
    } catch (Exception e) {
        e.printStackTrace();
    }

    HttpServletRequest request =
        (HttpServletRequest) this.context.getRequest();
    System.out.println("Done sleeping. Request ID = " + this.id +
        ", URL = " + request.getRequestURL() + ".");
    this.context.dispatch("/WEB-INF/jsp/view/async.jsp");

    System.out.println("Asynchronous thread completed. Request ID = " +
        this.id + ".");
}
}
```

Now experiment with these Servlets and filters:

1. Compile the application and start Tomcat from your IDE; then go to `http://localhost:8080/filters/regular` in your browser. You should see the following in the output window of the debugger. Notice that `normalFilter` intercepted the request to the Servlet, and `forwardFilter` intercepted the forwarded request to the JSP.

```
Entering normalFilter.doFilter().  
Entering NonAsyncServlet doGet().  
Entering forwardFilter.doFilter().  
In nonAsync.jsp.  
Leaving forwardFilter.doFilter().  
Leaving NonAsyncServlet doGet().  
Leaving normalFilter.doFilter().
```

2. Go to <http://localhost:8080/filters/async>. The following debugger output appears immediately. Notice that `normalFilter` intercepts the request but completes before the response is actually sent.

```
Entering normalFilter.doFilter().  
Entering AsyncServlet.doGet(). Request ID = 1, isAsyncStarted = false  
Starting asynchronous thread. Request ID = 1.  
Leaving AsyncServlet.doGet(). Request ID = 1, isAsyncStarted = true  
Leaving normalFilter.doFilter(), async context holds wrapped request/response=true  
Asynchronous thread started. Request ID = 1.
```

After a 5-second wait, the `AsyncThread` inner class sends the response to the user, and the following debugger output appears. When the request is dispatched to the JSP using the `AsyncContext`'s `dispatch` method, `asyncFilter` intercepts the internal request to that JSP.

Done sleeping. Request ID = 1, URL = http://localhost:8080/filters/async.
Asynchronous thread completed. Request ID = 1.

```
Entering asyncFilter.doFilter().
In async.jsp.
Leaving asyncFilter.doFilter().
```

3. Go to `http://localhost:8080/filters/async?unwrap` and wait for the response to complete. The following debugger output appears (some of it after 5 seconds). It's identical except that, in this case, the `AsyncContext` holds the original request and response instead of the wrapped request and response (the bold output changed).

```
Entering normalFilter.doFilter().
Entering AsyncServlet doGet(). Request ID = 2, isAsyncStarted = false
Starting asynchronous thread. Request ID = 2.
Leaving AsyncServlet doGet(). Request ID = 2, isAsyncStarted = true
Leaving normalFilter.doFilter(), async context holds wrapped request/response=false
Asynchronous thread started. Request ID = 2.
Done sleeping. Request ID = 2, URL = http://localhost:8080/filters/async.
Asynchronous thread completed. Request ID = 2.
Entering asyncFilter.doFilter().
In async.jsp.
Leaving asyncFilter.doFilter().
```

4. Go to `http://localhost:8080/filters/async?timeout=3000`. The following debugger output appears, and then 5 seconds later the sleep completes, and retrieving the request from the `AsyncContext` results in an `IllegalStateException`. This is because the `AsyncContext` `timeout` expired and the response was closed before the `AsyncThread` inner class could complete its work.

```
Entering normalFilter.doFilter().
Entering AsyncServlet doGet(). Request ID = 3, isAsyncStarted = false
Starting asynchronous thread. Request ID = 3.
Leaving AsyncServlet doGet(). Request ID = 3, isAsyncStarted = true
Leaving normalFilter.doFilter(), async context holds wrapped request/response=true
Asynchronous thread started. Request ID = 3.
```

It should be clear by now how complex and powerful asynchronous request handling is. The important point is that if you handle the response using the `AsyncContext` directly, the code executes outside the scope of any filters. However, if you use `AsyncContext`'s `dispatch` method to internally forward the request to a URL, a filter mapped for `ASYNC` requests can intercept the internal forward and apply whatever additional logic is required. You must decide when each approach is appropriate, but in most cases you will have no need for asynchronous request handling. No other parts of this book use asynchronous request handling.

INVESTIGATING PRACTICAL USES FOR FILTERS

The beginning of the chapter discussed many practical uses for filters. The Compression-Filter project on the wrox.com code download site demonstrates two of these uses: a logging filter and a response compression filter. The project contains a simple Servlet mapped to `/servlet` that responds with "This Servlet response may be compressed." It also contains a simple `/index.jsp` file that responds with "This content may be compressed." This project uses the following `ServletContextListener` to programmatically configure the filters for the application.

```

@WebListener
public class Configurator implements ServletContextListener
{
    @Override
    public void contextInitialized(ServletContextEvent event) {
        ServletContext context = event.getServletContext();

        FilterRegistration.Dynamic registration =
            context.addFilter("requestLogFilter", new RequestLogFilter());
        registration.addMappingForUrlPatterns(null, false, "/*");

        registration = context.addFilter("compressionFilter",
            new CompressionFilter());
        registration.setAsyncSupported(true);
        registration.addMappingForUrlPatterns(null, false, "/*");
    }

    @Override
    public void contextDestroyed(ServletContextEvent event) { }
}

```

Adding a Simple Logging Filter

The `RequestLogFilter` class in Listing 9-2 is the first filter in the filter chain for all requests to the application. It times how long a request takes to process and logs information about every request that comes in to the application — the IP address, timestamp, request method, protocol, response status and length, and time to process the request — similar to the Apache HTTP log format. The logging happens in the `finally` block so that any exceptions thrown further down the filter chain do not prevent the logging statement from being written.

LISTING 9-2: RequestLogFilter.java

```

public class RequestLogFilter implements Filter
{
    @Override
    public void doFilter(ServletRequest request, ServletResponse response,
        FilterChain chain) throws IOException, ServletException {
        Instant time = Instant.now();
        StopWatch timer = new StopWatch();
        try {
            timer.start();
            chain.doFilter(request, response);
        } finally {
            timer.stop();
            HttpServletRequest in = (HttpServletRequest) request;
            HttpServletResponse out = (HttpServletResponse) response;
            String length = out.getHeader("Content-Length");
            if(length == null || length.length() == 0)
                length = "-";
        }
    }
}

```

```

        System.out.println(in.getRemoteAddr() + " - - [" + time + "]"
            + " \\" + in.getMethod() + " " + in.getRequestURI() + " " +
            in.getProtocol() + "\\" + out.getStatus() + " " + length +
            " " + timer);
    }
}

@Override
public void init(FilterConfig filterConfig) throws ServletException { }

@Override
public void destroy() { }
}

```

NOTE *The RequestLogFilter will not work properly with asynchronous request handling. If a Servlet starts an AsyncContext, doFilter will return before the response has been sent, meaning the filter will log incomplete or incorrect information. Making this filter work correctly for asynchronous requests is a significant and complex task that is left as an exercise for the reader to consider.*

Compressing Response Content Using a Filter

The `CompressionFilter` in Listing 9-3 is significantly more complex than the `RequestLogFilter`.

When thinking about compressing the response, you might think that you should execute the filter chain and then perform the compression logic on the way back out. Remember, however, that response data can begin flowing back to the client before the Servlet has completed servicing the request. It can also begin flowing back to the client *after* the Servlet has completed servicing the request, in the case of asynchronous request handling. Because of this, if you want to alter the response content, you must wrap the response object passed further down the chain. The `CompressionFilter` does just this.

Take a minute to read over the code and examine what `CompressionFilter` is doing. First, it checks to see if the client has included an `Accept-Encoding` request header containing the “gzip” encoding. This is a very important check because if it hasn’t, this means the client may not understand gzip-compressed responses. If it has, it sets the `Content-Encoding` header to “gzip” and then wraps the response object with an instance of the private inner class `ResponseWrapper`. This class, in turn, wraps the `PrintWriter` or `ServletOutputStream` that sends data back to the client with the private inner class `GZIPOutputStream`. This wrapper contains an internal `java.util.zip.GZIPOutputStream`. Response data is first written to the `GZIPOutputStream`, and when the request completes, it finishes compression and writes the compressed response to the wrapped `ServletOutputStream`. The `ResponseWrapper` also prevents any Servlet code from setting the content length header for the response because the content length cannot be known until after the response is compressed.

LISTING 9-3: CompressionFilter.java

```
public class CompressionFilter implements Filter
{
    @Override
    public void doFilter(ServletRequest request, ServletResponse response,
                         FilterChain chain) throws IOException, ServletException {
        if(((HttpServletRequest)request).getHeader("Accept-Encoding")
           .contains("gzip")) {
            System.out.println("Encoding requested.");
            ((HttpServletResponse)response).setHeader("Content-Encoding", "gzip");
            ResponseWrapper wrapper =
                new ResponseWrapper((HttpServletResponse)response);
            try {
                chain.doFilter(request, wrapper);
            } finally {
                try {
                    wrapper.finish();
                } catch(Exception e) {
                    e.printStackTrace();
                }
            }
        } else {
            System.out.println("Encoding not requested.");
            chain.doFilter(request, response);
        }
    }

    @Override
    public void init(FilterConfig filterConfig) throws ServletException { }

    @Override
    public void destroy() { }

    private static class ResponseWrapper extends HttpServletResponseWrapper
    {
        private GZIPOutputStream outputStream;
        private PrintWriter writer;

        public ResponseWrapper(HttpServletRequest request) {
            super(request);
        }

        @Override
        public synchronized ServletOutputStream getOutputStream()
            throws IOException {
            if(this.writer != null)
                throw new IllegalStateException("getWriter() already called.");
            if(this.outputStream == null)
                this.outputStream =
                    new GZIPOutputStream(super.getOutputStream());
            return this.outputStream;
        }

        @Override
    }
}
```

```
public synchronized PrintWriter getWriter() throws IOException {
    if(this.writer == null && this.outputStream != null)
        throw new IllegalStateException(
            "getOutputStream() already called.");
    if(this.writer == null) {
        this.outputStream =
            new GZIPOutputStream(super.getOutputStream());
        this.writer = new PrintWriter(new OutputStreamWriter(
            this.outputStream, this.getCharacterEncoding()
        ));
    }
    return this.writer;
}

@Override
public void flushBuffer() throws IOException {
    if(this.writer != null)
        this.writer.flush();
    else if(this.outputStream != null)
        this.outputStream.flush();
    super.flushBuffer();
}

@Override
public void setContentLength(int length) { }

@Override
public void setContentLengthLong(long length) { }

@Override
public void setHeader(String name, String value) {
    if(!"content-length".equalsIgnoreCase(name))
        super.setHeader(name, value);
}

@Override
public void addHeader(String name, String value) {
    if(!"content-length".equalsIgnoreCase(name))
        super.setHeader(name, value);
}

@Override
public void setIntHeader(String name, int value) {
    if(!"content-length".equalsIgnoreCase(name))
        super.setIntHeader(name, value);
}

@Override
public void addIntHeader(String name, int value) {
    if(!"content-length".equalsIgnoreCase(name))
        super.setIntHeader(name, value);
}

public void finish() throws IOException {
```

continues

LISTING 9-3 *(continued)*

```
        if(this.writer != null)
            this.writer.close();
        else if(this.outputStream != null)
            this.outputStream.finish();
    }
}

private static class GZIPServletOutputStream extends ServletOutputStream
{
    private final ServletOutputStream servletOutputStream;
    private final GZIPOutputStream gzipStream;

    public GZIPServletOutputStream(ServletOutputStream servletOutputStream)
        throws IOException {
        this.servletOutputStream = servletOutputStream;
        this.gzipStream = new GZIPOutputStream(servletOutputStream);
    }

    @Override
    public boolean isReady() {
        return this.servletOutputStream.isReady();
    }

    @Override
    public void setWriteListener(WriteListener writeListener) {
        this.servletOutputStream.setWriteListener(writeListener);
    }

    @Override
    public void write(int b) throws IOException {
        this.gzipStream.write(b);
    }

    @Override
    public void close() throws IOException {
        this.gzipStream.close();
    }

    @Override
    public void flush() throws IOException {
        this.gzipStream.flush();
    }

    public void finish() throws IOException {
        this.gzipStream.finish();
    }
}
}
```

The wrapper pattern is a very common pattern that you will likely see applied in many filters. Both the request and response objects can be wrapped; however, wrapping the response is usually more

common. Wrapping the response allows you to intercept any method calls on the wrapped response, facilitating the ability to modify the response data. You can also use a very similar filter to that in Listing 9-3 to encrypt response data rather than compress it. The request object could be wrapped to decrypt its contents.

To try out the logging and compression filters:

1. Compile your application and start Tomcat from your IDE.
2. Go to <http://localhost:8080/compression/> and <http://localhost:8080/compression/servlet> in your browser.
3. Using the developer tools built for your browser, start monitoring the headers of requests to and responses from the application. (Microsoft Internet Explorer and Google Chrome have built-in developer tools that can do this, and the Firebug plug-in for Mozilla Firefox has this capability.) You should see a screen like that in Figure 9-4.

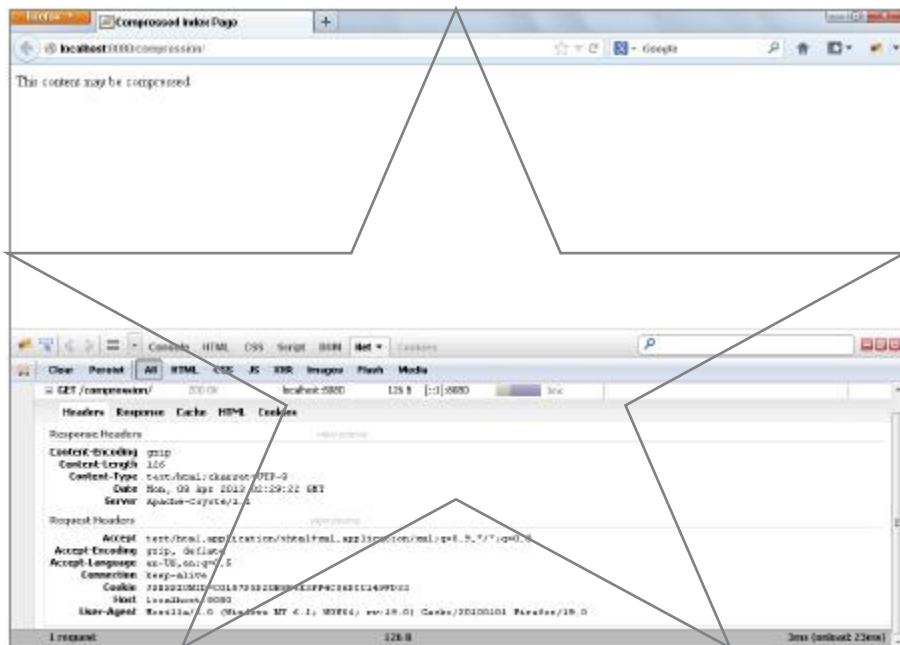


FIGURE 9-4

Notice that the `Accept-Encoding` request header contains the "gzip" encoding, and the `Content-Encoding` response header has the value "gzip." This means that your browser is announcing it can accept gzip-encoded responses, and the compression filter is obliging the request and compressing the response data before sending it to the browser.

SIMPLIFYING AUTHENTICATION WITH A FILTER

One critical use for filters in web applications is for securing applications against unwanted access. The Customer Support application you are making for Multinational Widget Corporation uses a very primitive authentication mechanism for securing its pages. You have probably already noticed that many places in the application contain the same, duplicated code to check for authentication:

```
if(request.getSession().getAttribute("username") == null)
{
    response.sendRedirect("login");
    return;
}
```

At some point you might have thought a better solution is to create a public static method on some class to perform this check and call it everywhere. For sure, this reduces duplicated code, but it still results in performing that method call in multiple places. As the number of servlets in your application increased, so would have the number of calls to that static method.

After what you have learned in this chapter, it should be clear that a filter is a better place to put this code. The Customer-Support-v7 project on the wrox.com code download site demonstrates this by adding the `Configurator` listener class and the `AuthenticationFilter` class. The previous code snippet has been removed from the `doGet` and `doPost` methods in `TicketServlet` and the `doGet` method in `SessionListServlet`. The configurator is simple: It declares the `AuthenticationFilter` and maps it to `/tickets` and `/sessions`:

```
@WebListener
public class Configurator implements ServletContextListener
{
    @Override
    public void contextInitialized(ServletContextEvent event)
    {
        ServletContext context = event.getServletContext();

        FilterRegistration.Dynamic registration = context.addFilter(
            "authenticationFilter", new AuthenticationFilter()
        );
        registration.setAsyncSupported(true);
        registration.addMappingForUrlPatterns(
            null, false, "/sessions", "/tickets"
        );
    }

    @Override
    public void contextDestroyed(ServletContextEvent event) { }
}
```

As you add more Servlets or other protected resources (such as JSPs) to your application, you only need to add their URL patterns to the filter registration to ensure that users log in before accessing those resources. Of course, this filter does not protect the login Servlet because you do not want to protect the login screen. You also do not need to protect any of the CSS, JavaScript, or image resources because they do not contain sensitive data. The `AuthenticationFilter` performs the

authentication check on every request of any HTTP method and redirects users to the login screen if they are not logged in:

```
public class AuthenticationFilter implements Filter
{
    @Override
    public void doFilter(ServletRequest request, ServletResponse response,
                         FilterChain chain) throws IOException, ServletException
    {
        HttpSession session = ((HttpServletRequest)request).getSession(false);
        if(session != null && session.getAttribute("username") == null)
            ((HttpServletResponse) response).sendRedirect("login");
        else
            chain.doFilter(request, response);
    }

    @Override
    public void init(FilterConfig config) throws ServletException { }

    @Override
    public void destroy() { }
}
```

One beautiful thing about this change is that if you alter the authentication algorithm, you only need to change the filter to continue protecting the resources in your application. Previously, you would have had to make changes to every Servlet. Test out these changes by compiling, starting up Tomcat in your IDE, and navigating to <http://localhost:8080/support> in your browser. Even though the authentication check has been removed from all the Servlets, you are still asked to log in before viewing or creating tickets, or viewing the list of sessions.

SUMMARY

In this chapter you explored the purpose of filters and the many reasons you might use them. You learned about the `Filter` interface and how to create, declare, and map filters in your applications. You experimented with the all-important filter chain, and learned how the order in which filters are executed can be unimportant in some scenarios and quite critical in others. You were introduced to the concept of asynchronous request handling and used filters to explore that topic further and gain an understanding of how tricky asynchronous request handling can be. Finally, after exploring the three different ways of declaring and mapping filters — in the deployment descriptor, using annotations, and programmatically — you experimented with a logging filter, a response compression filter, and an authentication filter.

In the next chapter you explore the technology of WebSockets, how they dramatically improve interactive web applications, and how to use them in Java and JavaScript. An interesting point is that the code in Tomcat that makes WebSockets possible actually uses a filter to intercept all WebSocket-bound requests in your application and send them to your WebSocket endpoints (which you learn about more in the next chapter). So, without even doing anything, your application already has filters inspecting and, if necessary, modifying requests.

10

Making Your Application Interactive with WebSockets

IN THIS CHAPTER

- How Ajax evolved to WebSockets
- A discussion on the WebSocket APIs
- Using WebSockets to create multiplayer games
- Communicating in a cluster with WebSockets
- How to add chatting to a web application

WROX.COM CODE DOWNLOADS FOR THIS CHAPTER

You can find the `wrox.com` code downloads for this chapter at <http://www.wrox.com/remtitle.cgi?isbn=1118656464> on the Download Code tab. The code for this chapter is divided into the following major examples:

- Game-Site Project
- Simulated-Cluster Project
- Customer-Support-v8 Project

NEW MAVEN DEPENDENCIES FOR THIS CHAPTER

In addition to the Maven dependencies introduced in previous chapters, you also need the following Maven dependencies:

```
<dependency>
  <groupId>javax.websocket</groupId>
  <artifactId>javax.websocket-api</artifactId>
```

```
<version>1.0</version>
<scope>provided</scope>
</dependency>

<dependency>
<groupId>org.apache.commons</groupId>
<artifactId>commons-lang3</artifactId>
<version>3.1</version>
<scope>compile</scope>
</dependency>

<dependency>
<groupId>com.fasterxml.jackson.core</groupId>
<artifactId>jackson-core</artifactId>
<version>2.3.2</version>
<scope>compile</scope>
</dependency>

<dependency>
<groupId>com.fasterxml.jackson.core</groupId>
<artifactId>jackson-annotations</artifactId>
<version>2.3.2</version>
<scope>compile</scope>
</dependency>

<dependency>
<groupId>com.fasterxml.jackson.core</groupId>
<artifactId>jackson-databind</artifactId>
<version>2.3.2</version>
<scope>compile</scope>
</dependency>

<dependency>
<groupId>com.fasterxml.jackson.datatype</groupId>
<artifactId>jackson-datatype-jsr310</artifactId>
<version>2.3.2</version>
<scope>compile</scope>
</dependency>
```

EVOLUTION: FROM AJAX TO WEB SOCKETS

In the beginning, man created HTML and saw that it was not enough. Users wanted web pages that interacted with them. The answer to this was JavaScript. In the early days of JavaScript, there was almost no standardization between browsers (in the earliest days, only one browser supported it) and JavaScript was extremely slow and insecure. Over the years it has improved significantly in speed, security, and capability. With many JavaScript frameworks available on the web, you can now create extremely rich, single-page web applications with very little JavaScript. The biggest driver of this innovation was the adoption of a technology known as Ajax.

Problem: Getting New Data from the Server to the Browser

Ajax, an acronym for *Asynchronous JavaScript and XML*, has become synonymous with the idea of asynchronously communicating (or synchronously, for that matter) with a remote server using JavaScript. It does not necessitate the use of XML, as the name implies. (JSON is often used instead of XML, and in such cases it can still be called Ajax or may also be called A/J.) With Ajax, web applications in the browser could now communicate with server-side components without the browser page changing or refreshing. This communication could happen completely transparently, without the user knowing, and it could be used to send new data to the server or fetch new data from the server. This, however, was the crux of the problem: The browser could only fetch new data from the server. But the browser doesn't necessarily know when new data is available to fetch. The server knows that. Just like only the browser knows when it has new data to send to the server, only the server knows when it has new data to send to the browser. For example, when two users are chatting in a web application, only the server knows when user A has sent a message to user B. User B's browser won't know until it contacts the server. Even with something as powerful as Ajax, this was a difficult problem to solve.

Over the past 14 years, dozens of solutions have emerged, some of them widely supported, whereas others were extremely browser-specific. This book doesn't cover all of them, nor would that be useful. However, four major approaches have served as the basis for the varying solutions to this problem, and understanding them and their weaknesses is key to understanding the need for something better.

NOTE *The actual details of making Ajax requests and receiving responses using JavaScript is not important to the discussion in this chapter and outside the scope of this book. If you are unfamiliar with this topic, there are thousands of Ajax tutorials available online.*

Solution 1: Frequent Polling

Perhaps one of the most prevalent approaches to this problem is *frequently polling* the server for new data. The concept is fairly simple: At a regular interval, often once per second, the browser sends an Ajax request to the server polling for new data. If the browser has new data to send to the server, that data hitches a ride on the request. If the server has new data for the browser, it replies with the new data. If it does not, it replies with no content. (This may mean an empty JSON object or XML document, or a `Content-Length` header of 0.) This protocol looks something like the four requests represented in Figure 10-1, except there are usually many more requests and responses.

The downside to this protocol is huge and obvious: An enormous number of wasted requests could be made and replied to needlessly. Notice that the second and third requests in the figure achieved nothing! Add to this the overhead of establishing a connection, sending and receiving headers, and tearing down a connection, and a lot of processing and network resources are wasted to find out that the server has no new data for the browser. Though this technique is still in widespread use due to the sheer simplicity of implementing it, it was clear early on that it was not a good solution to this problem.

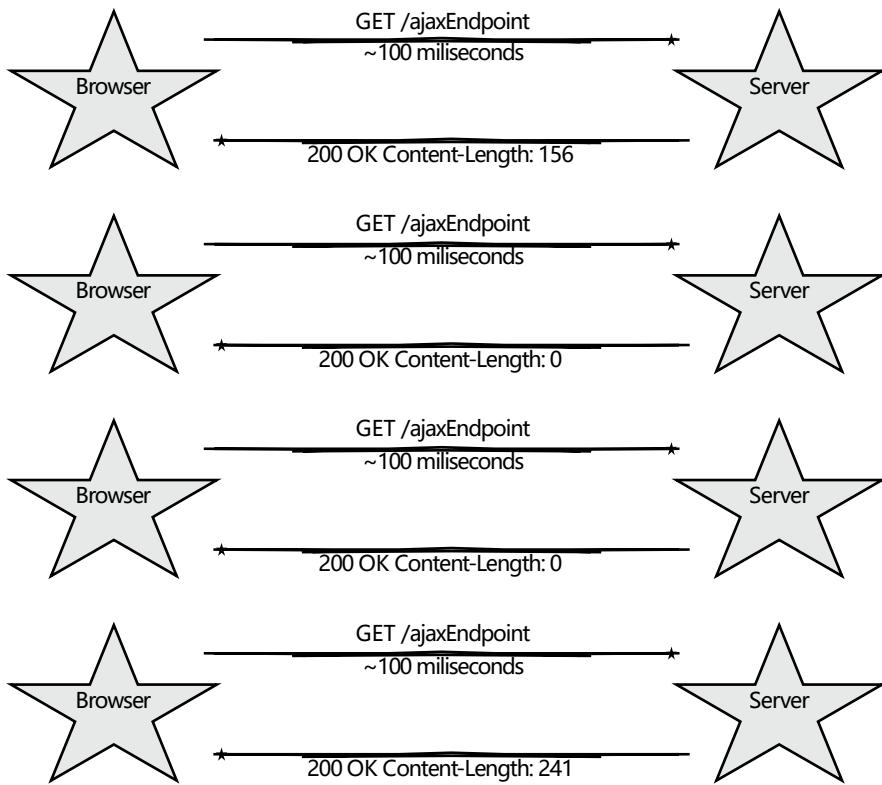


FIGURE 10-1

Solution 2: Long Polling

Long polling, represented in Figure 10-2, is similar to frequent polling, except that the server doesn't respond until it has data to send back to the browser. This is more efficient because fewer compute and network resources are wasted, but it brings some problems of its own to the table:

- **What if the browser has new data to send to the server before the server responds?** The browser must either make a separate, parallel request, or it must terminate the current request (from which the server must recover gracefully) and begin another one.
- **Connection timeouts are built into the TCP and HTTP specifications.** Because of this, the server and client must periodically close and re-establish a connection. Sometimes the connection closes as often as every 60 seconds, although some implementations have been successful at holding a connection for several minutes.
- **A connection limit mandate exists in the HTTP/1.1 specification.** Browsers must permit no more than two simultaneous connections to any given hostname. If one connection is constantly tied up waiting for data to be pushed from the server, it cuts in half the number of connections that can be used to fetch web pages, graphics, and other resources from the server.

Regardless of these issues, this is a popular approach that has gained widespread use over the past several years and is often generically referred to as *Comet* (an ironic play on words with Ajax, as both names can refer to cleaning products).

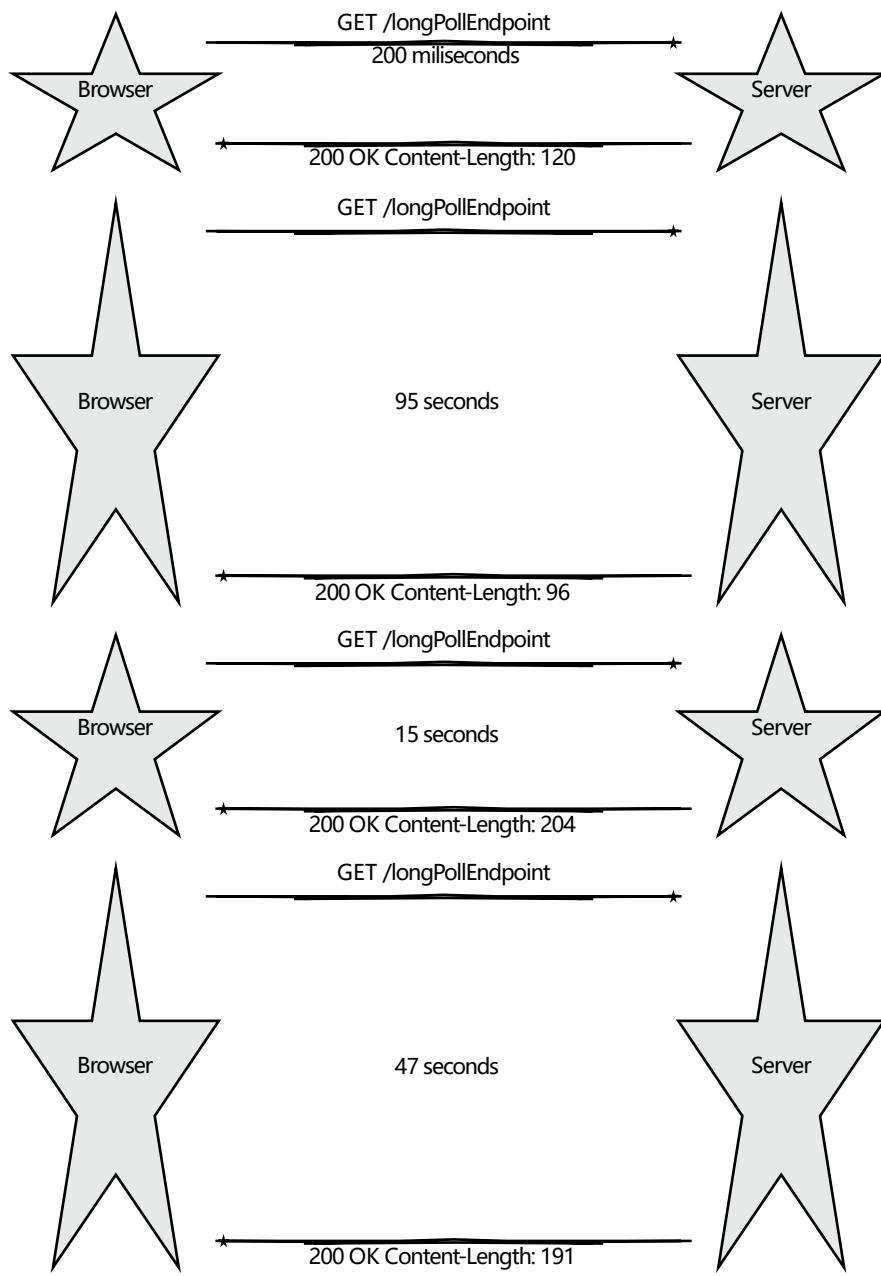


FIGURE 10-2

Solution 3: Chunked Encoding

Very similar to the long polling solution, *chunked encoding* takes advantage of an HTTP/1.1 feature that enables servers to respond to requests without advertising a content length. In place of the `Content-Length: n` header, a response can contain a `Transfer-Encoding: chunked` header. This tells the browser that the response is coming in “chunks.” In the response, each chunk starts with a number indicating the length of the chunk, a series of optional characters indicating chunk extensions (irrelevant to this discussion), and a `CRLF` (carriage return + line feed) sequence. This is then followed by the data contained in that chunk and another `CRLF`. Any number of chunks can follow, and they can be separated, theoretically, by any amount of time, small or great. When a zero-length chunk (a `0` followed by two `CRLFs`) is sent, this indicates the end of the response.

How you typically use chunked encoding to solve the problem at hand is to establish a connection at the beginning for the sole purpose of receiving events sent from the server. Each chunk from the server is a new event and triggers another call to the JavaScript `XMLHttpRequest` object’s `onreadystatechange` event handler. Occasionally, though not nearly as often as with long polling, the connection has to be refreshed. When the browser needs to send new data to the server, it does so with a second short-lived request.

This protocol is represented in Figure 10-3. On the left-hand side, the browser *sends* new data to the “upstream endpoint,” whenever it needs to, using short-lived requests. On the right-hand side, the browser establishes a single, long-lived connection with the “downstream endpoint,” and the server uses that connection to send the browser updates in chunks. Chunked encoding solves the major timeout problem present with long polling (browsers tolerate responses taking a long time to *complete* much better than responses taking a long time to *start*—large file downloads are the perfect example), but the issue with browsers being limited to two connections remains. Additionally, with both long polling and chunked encoding, older browsers tended to indefinitely display a message in the status bar that the page was still loading—though modern browsers have eliminated this behavior.

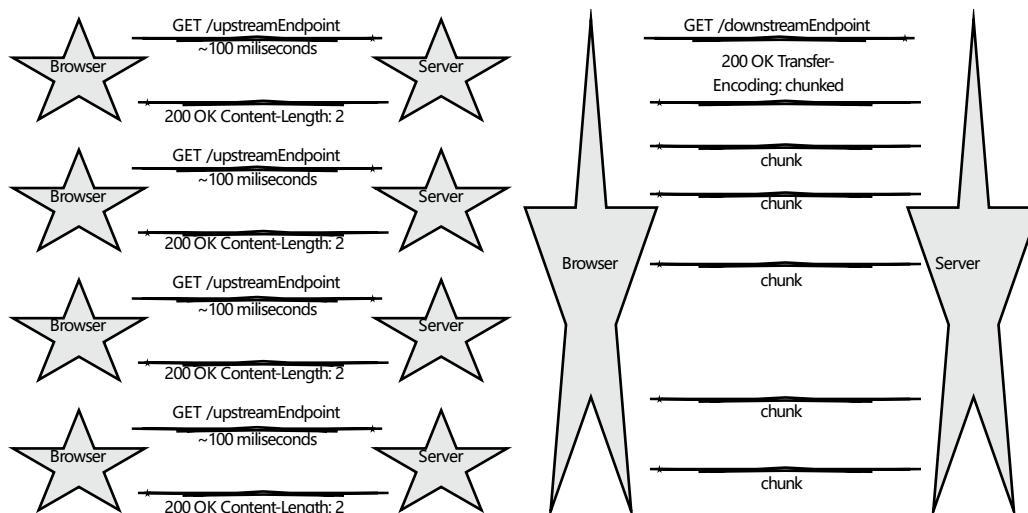


FIGURE 10-3

Solution 4: Applets and Adobe Flash

Early in this evolutionary process, many realized that what all of these solutions were emulating was true full-duplex communications between a browser and server over a single connection. Simply put, this was not going to happen using Ajax and XMLHttpRequest. A popular, though short-lived, approach to this problem was to use Java Applets or Adobe Flash movies, demonstrated in Figure 10-4. Essentially, the developer would create a 1-pixel-square transparent Applet or Flash movie and embed it in the page. This plug-in would then establish an ordinary TCP socket connection (instead of an HTTP connection) to the server. This eliminated all of the restrictions and limitations present in the HTTP protocol. When the server sent messages to the browser, the Applet or Flash movie would call a JavaScript function with the message data. When the browser had new data for the server, it would call a Java or Flash method using a JavaScript DOM function exposed by the browser plug-in, and that method would forward the data on to the server.

This protocol achieved true full-duplex communications over a single connection and eliminated issues such as timeouts and concurrent connection limitations (and even avoided the security constraint placed on Ajax connections that they must originate from a page on the same fully qualified domain name). But it came at a high cost: It required third-party (Java or Flash) plug-ins, which were inherently insecure, slow, and memory-intensive. Because there were no security protocols built in to this solution and each developer was left to his own devices, it also revealed some interesting vulnerabilities.

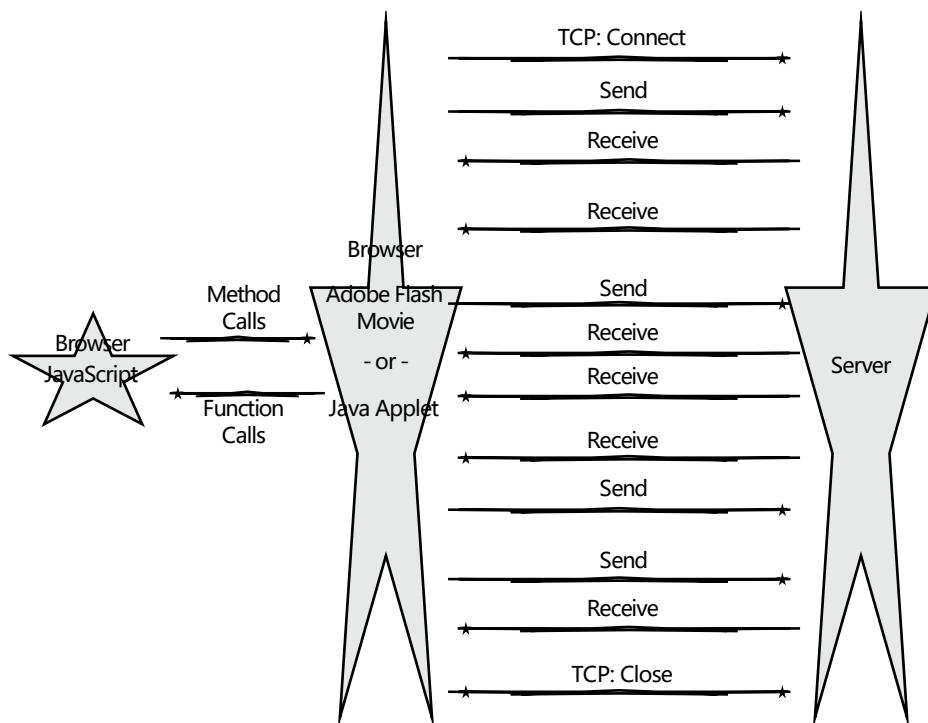


FIGURE 10-4

This technology took off for a while, but not long after that, the mobile web took the technology world by storm. Browsers in most popular mobile device operating systems would not (and to this day still do not) run Java or Flash plug-ins. With an increasing percentage of Internet traffic coming from mobile devices (one-quarter of all traffic as of late 2012), web developers quickly abandoned this approach to getting data from the server. They needed something better. They needed a solution that used raw TCP connections, was secure, was fast, could be easily supported on mobile platforms, and didn't require browser plug-ins to accomplish.

WebSockets: The Solution Nobody Knew Kind of Already Existed

The HTTP/1.1 specification in RFC 2616 was formalized in 1999. It provided the framework for all HTTP communications used for more than a decade to the present day. Section 14.42 included a rarely used, often overlooked feature called *HTTP Upgrade*.

The HTTP/1.1 Upgrade Feature

The premise is simple: Any HTTP client (not just browsers) can include the header name and value `Connection: Upgrade` in a request. To indicate what the client wants to upgrade to, the additional `Upgrade` header must specify a list of one or more protocols. These protocols should be something incompatible with HTTP/1.1, such as IRC or RTA. The server, if it accepts the upgrade request, returns the response code `101 Switching Protocols` along with a response `Upgrade` header with a single value: the first protocol that the server supports from the list of the requested protocols. Originally, this feature was most often used to upgrade from HTTP to HTTPS, but was subject to man-in-the-middle attacks because the entire connection wasn't secured. Thus, the technique was quickly replaced with the `https` URI scheme. Since then, `Connection: Upgrade` has largely fallen out of use.

The most important feature of an HTTP Upgrade is that the resulting protocol can be anything. It ceases to be an HTTP connection after the Upgrade handshake is complete and can even turn into a persistent, full-duplex TCP socket connection. Theoretically speaking, you could use HTTP Upgrade to establish any kind of TCP communications between any two endpoints with a protocol of your own design. However, browsers aren't about to turn JavaScript developers loose on the TCP stack (nor should they), so some protocol needed to be agreed upon. Thus, the WebSockets protocol was born.

NOTE *If a particular resource on a server accepts only HTTP Upgrade requests and a client connects to this resource without requesting an upgrade, the server can respond with 426 Upgrade Required to indicate that an upgrade is mandatory. In this case the response could also include the `Connection: Upgrade` header and the `Upgrade` header containing a list of the upgrade protocols the server supports. If a client requests an upgrade to a protocol the server doesn't support, the server responds with 400 Bad Request and can include the `Upgrade` header containing a list of the upgrade protocols the server supports. Finally, if the server does not accept upgrade requests, it responds with 400 Bad Request.*

WebSocket Protocol Sits on Top of HTTP/1.1 Upgrade

A *WebSocket* connection, represented in Figure 10-5, begins with a not-so-unordinary HTTP request to a URL with a special scheme. The URI schemes `ws` and `wss` correspond to their HTTP counterparts `http` and `https`, respectively. The `Connection: Upgrade` header is present along with the `Upgrade: websocket` header, instructing the server to upgrade the connection to the WebSocket protocol, a persistent, full-duplex communications protocol formalized as RFC 6455 in 2011. After the handshake is completed, text and binary messages are sent in either direction at the same time without closing and re-establishing the connection. At this point, there is essentially no difference between client and server — they have equal capabilities and power over the connection, and are simply *peers*.

NOTE *The `ws` and `wss` schemes aren't strictly part of the HTTP protocol, since HTTP requests and request headers don't actually include URI schemes. Instead, HTTP requests include only the server-relative URL in the first line of the request and the domain name in the `Host` header. The specialized WebSocket schemes are mainly used to inform browsers and APIs as to whether you intend to connect using SSL/TLS (`wss`) or no encryption (`ws`).*

There are many advantages to the way the WebSocket protocol is implemented:

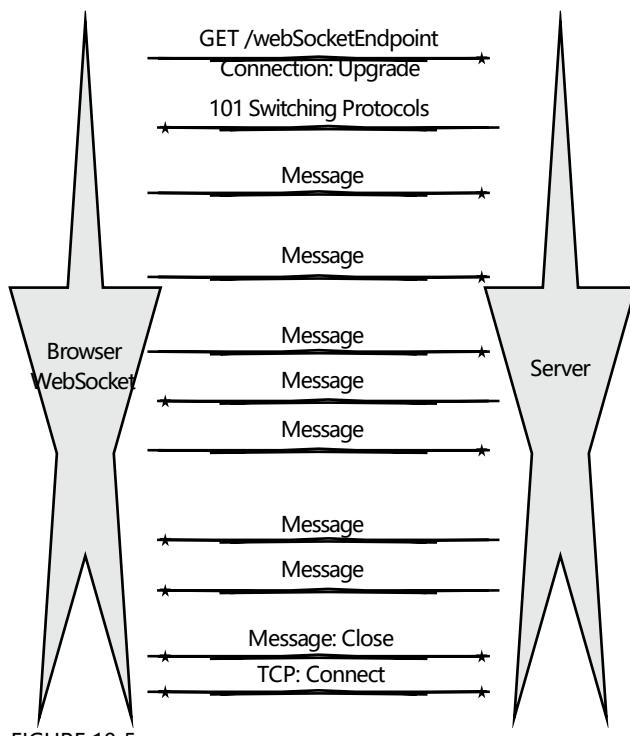


FIGURE 10-5

- Because the connection is established on port 80 (`ws`) or 443 (`wss`), the same ports used for HTTP, almost no firewalls block WebSocket connections.
- The protocol integrates naturally into Internet browsers and HTTP servers because the handshake takes place over HTTP.
- Heartbeat messages called *pings* and *pongs* are sent back and forth to keep WebSocket connection alive nearly indefinitely. Essentially, one peer periodically sends a tiny packet to the other (the ping), and the other peer responds with a packet containing the same data (the pong). This establishes that both peers are still connected.
- Messages are framed on your behalf without any extra code so that the server and client both know when a message starts and when all its content arrives.
- The closing of the WebSocket connection involves a special close message that can contain reason codes and text explaining why the connection was closed.
- The WebSocket protocol can *securely* allow cross-domain connections, eliminating restrictions placed on Ajax and `XMLHttpRequest`.
- The HTTP specification requiring browsers to limit simultaneous connections to two per hostname does not apply after the handshake is complete because the connection ceases to be an HTTP connection.

The handshake request headers in a WebSocket connection are simple. A typical WebSocket upgrade request may appear as follows if studied in a traffic analyzer like Wireshark or Fiddler:

```
GET /WebSocketEndpoint HTTP/1.1
Host: www.example.org
Connection: Upgrade
Upgrade: websocket
Origin: http://example.com
Sec-WebSocket-Key: x3JHMbDL1EzLkh9GBhXDw==
Sec-WebSocket-Version: 13
Sec-WebSocket-Protocol: game
```

You should already be familiar with the HTTP prelude (GET /WebSocketEndpoint HTTP/1.1) and the Host header. Also the Connection and Upgrade headers were previously explained. The Origin header is a security mechanism that protects against unwanted cross-domain requests. The browser sets this header to the domain from which the web page was served, and the server checks that value against a list of "approved" domains.

The Sec-WebSocket-Key header is a specification conformance check: The browser generates a random key, base64 encodes it, and places it in the request header. The server appends 258EAFA5-E914-47DA-95CA-C5AB0DC85B11 to the request header value, SHA-1 hashes it, and returns the hashed value base64 encoded in the Sec-WebSocket-Accept response header. Sec-WebSocket-Version indicates the current version of the protocol that the client implements, and Sec-WebSocket-Protocol is an optional header that further indicates which protocol is used on top of the WebSocket protocol. (This is a protocol you define, such as `chat`, `game`, or `stockticker`.) The following is what the response to the previous request might look like:

```
HTTP/1.1 101 Switching Protocols
Server: Apache 2.4
Connection: Upgrade
Upgrade: websocket
Sec-WebSocket-Accept: HSmr0sM1YUkAGmm50PpG2HaGWk=
Sec-WebSocket-Protocol: game
```

At this point, the HTTP connection goes away and is replaced by a WebSocket connection using the same, underlying TCP connection. The biggest hurdle to the success of this connection is HTTP proxies, which historically don't handle HTTP Upgrade requests well (nor, for that matter, HTTP traffic in general). Browsers typically try to detect if the connection is going over a proxy and issue an HTTP CONNECT before the handshake, but this does not always work. Truly the most reliable way to use WebSockets is to *always* use SSL/TLS (`wss`). Proxies typically leave SSL/TLS connections alone and let them do their own thing, and with this strategy, your WebSocket connections can work in nearly all circumstances. It's also secure: The traffic is encrypted, in both directions, using the same industry-tested security as HTTPS.

Although all these details — upgrades, headers, protocols, framing, and binary and text messages — may sound daunting, the good news is that you don't have to worry about any of it. There are several APIs that cover all the protocol's difficult tasks and leave you only the task of creating your application on top of it.

WARNING *The WebSocket protocol is a very new technology relative to the timeframe that browsers typically adopt technologies. As such, you need to have very modern browsers installed on your client machine to use WebSockets. The examples throughout this chapter require you to have two different browsers (not just two windows — two actual different browsers). These browsers need to be from the following list or newer:*

- *Microsoft Internet Explorer 10.0 (must have Windows 7 SP1 or newer)*
- *Mozilla Firefox 18.0*
- *Google Chrome 24.0*
- *Apple Safari 6.0*
- *Opera 12.1*
- *Apple Safari iOS 6.0*
- *Google Android Browser 4.4*
- *Microsoft Internet Explorer Mobile 10.0*
- *Opera Mobile 12.1*
- *Google Chrome for Android 30.0*
- *Mozilla Firefox for Android 25.0*
- *Blackberry Browser 7.0*

The Many Uses of WebSockets

The WebSocket protocol has virtually unlimited uses, much of which includes browser applications, but some of which exists outside of Internet browsers. You see examples from both categories in this chapter. Though this book cannot list them all, the following is a taste of the many uses for WebSockets:

- JavaScript Chat
- Multiplayer online games (Mozilla hosts a fun, involved MMORPG called BrowserQuest written entirely in HTML5 and JavaScript using WebSockets.)
- Live Stock Ticker
- Live Breaking News Ticker
- HD Video Streaming (Yes, believe it or not, it really is that fast and powerful.)
- Communications between nodes in an application cluster
- Bulk, transactional data transfer between applications across the network
- Real-time monitoring of remote system or software status and performance

UNDERSTANDING THE WEBSOCKET APIs

One key thing you should understand about WebSockets is that they are not just for communication between browsers and servers. Two applications written in any framework supporting WebSockets can, theoretically, establish communications over WebSockets. Therefore, many of the WebSocket implementations available contain both client and server endpoint tools. This is true, for example, in Java and .NET. JavaScript, however, is meant to serve only as a client endpoint of a WebSocket connection. In this section, you learn about using the JavaScript WebSocket client endpoint first, and then move on to the Java client endpoint and finally the Java server endpoint.

NOTE *When this book refers to the JavaScript capabilities, it's referring solely to JavaScript as implemented by Internet browsers. Some JavaScript frameworks, such as Node.js, can run outside of the context of a browser and provide additional capabilities (including a WebSocket server) that are not discussed in this book. Learning about these frameworks is an exercise outside the scope of this book.*

HTML5 (JavaScript) Client API

As noted previously, all modern browsers offer WebSocket support, and that support is standardized across supporting browsers. The World Wide Web Consortium (W3C) formalized the requirements and interface for WebSocket communications within a browser as an extension of HTML5. Although you use JavaScript to perform WebSocket communications, the `WebSocket` interface is actually part of HTML5. All browsers provide WebSocket communications through an

implementation of the `WebSocket` interface. (If you remember the early days of Ajax, when different browsers had different classes and functions for performing Ajax requests, this will be a pleasant surprise for you.)

Creating a WebSocket Object

Creating a `WebSocket` object is straightforward:

```
var connection = new WebSocket('ws://www.example.net/stocks/stream');
var connection = new WebSocket('wss://secure.example.org/games/chess');
var connection = new WebSocket('ws://www.example.com/chat', 'chat');
var connection = new WebSocket('ws://www.example.com/chat', {'chat.v1','chat.v2'});
```

The first parameter to the `WebSocket` constructor is the required URL of the WebSocket server to which you want to connect. The optional second argument can be a string or array of strings defining one or more client-defined protocols that you want to accept. Remember that these protocols are of your own implementation and are not managed by the WebSocket technology. This argument simply provides a mechanism for passing the information along if you need to do so.

Using the WebSocket Object

There are several properties in the `WebSocket` interface. The first, `readyState`, indicates the current state of the `WebSocket` connection. Its value is always either `CONNECTING` (the number 0), `OPEN` (1), `CLOSING` (2) or `CLOSED` (3).

```
if(connection.readyState == WebSocket.OPEN) { /* do something */ }
```

It is primarily used internally; however, it is useful to ensure you do not attempt to send messages when the connection is not open. Unlike `XMLHttpRequest`, `WebSocket` does not have an `onreadystatechange` event that gets called whenever any type of event happens, forcing you to check `readyState` to determine a course of action. Instead, `WebSocket` has four separate events representing the four distinct things that can happen to a `WebSocket`:

```
connection.onopen = function(event) { }
connection.onclose = function(event) { }
connection.onerror = function(event) { }
connection.onmessage = function(event) { }
```

The event names clearly indicate when these events are triggered. Importantly, the `onclose` event is triggered when `readyState` changes from `CLOSING` to `CLOSED`. When the handshake completes and `onopen` is called (readyState changes from `CONNECTING` to `OPEN`), the read-only `url`, `extensions` (server-provided extensions) and `protocol` (server-selected protocol) object properties are set and fixed. The event object passed in to `onopen` is a standard JavaScript `Event` with nothing particularly interesting in it. The `Event` passed in to `onclose`, however, does have three useful properties: `wasClean`, `code`, and `reason`. You can use these to report improper closures to the user:

```
connection.onclose = function(event) {
    if(!event.wasClean)
        alert(event.code + ': ' + event.reason);
}
```

The legal closure codes are defined in RFC 6455 Section 7.4 (<http://tools.ietf.org/html/rfc6455#section-7.4>). Code 1000 is normal and all other codes are abnormal. The `onerror`

event contains a `data` property containing the error object, which could be any number of things (typically it is a string message). This event is triggered only for client-side errors; protocol errors result in closure of the connection. `onmessage` is the event handler you must deal with most carefully. Its event also contains a `data` property. This property is a string if the message is text message, a `Blob` if the message is a binary message and the `WebSocket`'s `binaryType` property is set to "blob" (default), or an `ArrayBuffer` if the message is a binary message and `binaryType` is set to "arraybuffer." You should typically set `binaryType` immediately after instantiating the `WebSocket` object and leave it at that value for the rest of the connection; however, it is legal to change the value whenever needed.

```
var connection = new WebSocket('ws://www.example.net/chat');
connection.binaryType = 'arraybuffer';
...
```

The `WebSocket` object has two methods: `send` and `close`. The `close` method accepts an optional close code as its first argument (default 1000) and an optional string reason as its second argument (default blank). The `send` method, which accepts a string, `Blob`, `ArrayBuffer`, or `ArrayBufferView` as its sole argument, is the only place you are likely to use the `WebSocket` interface's `bufferedAmount` property. `bufferedAmount` indicates how much data from previous `send` calls is still waiting to be sent to the server. Although you may continue to send data even if data is still waiting to be sent, sometimes you may want to push new data to the server only if no data is still waiting:

```
connection.onopen = function() {
  var intervalId = window.setInterval(function() {
    if(connection.readyState != WebSocket.OPEN) {
      window.clearInterval(intervalId);
      return;
    }
    if(connection.bufferedAmount == 0)
      connection.send(updatedModelData);
  }, 50);
}
```

The previous example sends fresh data at most every 50 milliseconds but, if the buffer has outgoing data in it still, it waits another 50 milliseconds and tries again. If the connection is not open, it stops sending data and clears the interval.

Java WebSocket APIs

The Java API for `WebSocket` was formalized in the JCP as JSR 356 and included in Java EE 7. It contains both a client and a server API. The client API is the foundational API: It specifies a set of classes and interfaces in the `javax.websocket` package that include all the necessary common functionality for a `WebSocket` peer. The server API contains `javax.websocket.server` classes and interfaces that use and/or extend client classes to provide additional functionality. As such, there are two artifacts for this API: the client-only artifact and the full artifact (which includes client and server classes and interfaces). Both APIs contain many classes and interfaces, and not all of them are covered here. That's what the API documentation is for (which you can find at <http://docs.oracle.com/javaee/7/api/> with the rest of the Java EE documentation). The rest of this section highlights the important details of the two APIs. The example code throughout the chapter can give you a better idea of exactly how to use the APIs.

The Client API

The client API is built on the `ContainerProvider` class and the `WebSocketContainer`, `RemoteEndpoint`, and `Session` interfaces. `WebSocketContainer` provides access to all the WebSocket client features, and the `ContainerProvider` class provides a static `getWebSocketContainer` method for obtaining the underlying WebSocket client implementation. `WebSocketContainer` provides four overloaded `connectToServer` methods that all accept a `URI` to connect to a remote endpoint and initiate a handshake. These methods accept either a POJO instance of any type annotated with `@ClientEndpoint`, a `Class<?>` for a POJO of any type annotated with `@ClientEndpoint`, an instance of the `Endpoint` class, or a `Class<? extends Endpoint>`. If you use either of the `Class`-variety methods, the class you supply must have a zero-argument constructor and will be instantiated on your behalf.

If you use the `Endpoint` or `Class<? extends Endpoint>` methods, you must also supply a `ClientEndpointConfig`. When the handshake is complete, the `connectToServer` method returns a `Session`. You can do many things with the `Session` object, most notably close the `Session` (which closes the WebSocket connection) or send messages to the remote endpoint.

NOTE *The Java API for WebSocket specifies an API, not an implementation. You can program against this API, but at run time you need an implementation to back it. If your application runs in a Java EE application server or a web container supporting WebSockets, client and server implementations are provided for you already. If it runs standalone, you need to find a standalone client or server implementation (depending on your needs) to deploy with your application.*

The WebSocket `Endpoint` has `onOpen`, `onClose`, and `onError` methods that are called on those events, whereas `@ClientEndpoint` classes can have (optional) methods annotated `@OnOpen`, `@OnClose`, and `@OnError`. `@ClientEndpoint` classes and classes extending `Endpoint` can specify one or more methods annotated with `@OnMessage` to handle receiving messages from the remote endpoint. Using annotated classes and methods, you have a lot of flexibility in what method parameters you can require.

`@OnOpen` methods can have:

- One optional `Session` parameter
- One optional `EndpointConfig` parameter

`@OnClose` methods can have:

- One optional `Session` parameter
- One optional `CloseReason` parameter

`@OnError` methods can have:

- One optional `Session` parameter
- One required `Throwable` parameter

`@OnMessage` methods are more complex. In addition to the standard optional `Session` parameter, they must have exactly one of the following combinations of parameters:

- One `String` to receive an entire text message
- One `String` plus one `boolean` to receive a text message in chunks, with the `boolean` set to `true` on the last chunk
- One Java primitive or primitive wrapper to receive an entire text message converted to that type
- One `java.io.Reader` to receive a text message as a blocking stream
- One `byte[]` or one `java.nio.ByteBuffer` to receive an entire binary message
- One `byte[]` or one `ByteBuffer`, plus one `boolean` to receive a binary message in chunks
- One `java.io.InputStream` to receive a binary message as a blocking stream
- One `PongMessage` for custom handling of heartbeat responses
- Any Java object if the endpoint has a `Decoder.Text`, `Decoder.Binary`, `Decoder.TextStream`, or `Decoder.BinaryStream` registered to convert that type. The message type of text or binary must match the registered decoder.

An endpoint can have only one method each to handle open, close, and error events; however, it may have up to three message-handling methods: no more than one for text messages, no more than one for binary messages, and no more than one for pong messages. One final note about the client API: The WebSocket Maven dependency used in this chapter is for the server API, which includes a dependency on the client API. If you ever need to write a Java application that is *only* a WebSocket client, you need only the client API Maven dependency:

```
<dependency>
  <groupId>javax.websocket</groupId>
  <artifactId>javax.websocket-client-api</artifactId>
  <version>1.0</version>
  <scope>provided</scope>
</dependency>
```

The Server API

The server API depends on the entire client API and adds only a handful of classes and interfaces. `ServerContainer` extends `WebSocketContainer` and adds methods to programmatically register `ServerEndpointConfig` instances or classes annotated with `@ServerEndpoint`. In a Servlet environment, you obtain a `ServerContainer` instance by calling `ServletContext.getAttribute("javax.websocket.server.ServerContainer")`. In a standalone application, you need to follow the instructions of the particular WebSocket implementation you are using to get a `ServerContainer` instance.

However, in nearly all use cases (including all uses within Java EE web containers) you never actually need to obtain a `ServerContainer`. Instead, simply annotate your server endpoint class or classes with `@ServerEndpoint`; the WebSocket implementation can scan your classes for this annotation and automatically pick up and register your server endpoints for you. The container

creates a new instance of your endpoint class each time it receives a new WebSocket connection for that endpoint and disposes of each instance after its connection closes.

When using `@ServerEndpoint`, you specify at least the required `value` attribute, which indicates the application-relative URL that this endpoint responds to. The URL path, which must start with a leading slash, can contain template parameters. So, for example, consider the following annotation:

```
@ServerEndpoint ("/game/{gameType}")
```

If your application were deployed at `http[s]://www.example.org/app`, this server endpoint would respond to `ws[s]://www.example.org/app/game/chess`, `ws[s]://www.example.org/app/game/checkers`, and so on. Then, any `@OnOpen`, `@OnClose`, `@OnError`, or `@OnMessage` method in the server endpoint could have an optional additional parameter annotated with `@PathParam ("gameType")`, and the container would provide "chess" and "checkers," respectively, as the values to that parameter. The event handling methods in your server endpoints work just like the event handling methods in your client endpoints. The difference between server and client only matters at the time of handshake. After the handshake is complete and the connection is established, the server and client become peers and are completely equal endpoints with the same capabilities and responsibilities.

NOTE *Because the primary purpose of the WebSocket protocol is communications between the server and browser, this chapter contains a lot of JavaScript code. Some of this code includes extensive use of the jQuery JavaScript library. This book would have to be hundreds of pages longer to teach you about JavaScript programming for the web, so it is assumed that you have a basic working knowledge of both JavaScript and jQuery (used to make easier, shorter examples). If you are not familiar with JavaScript, you are encouraged to take a break now and go find a JavaScript and jQuery book or tutorial. Otherwise, some of the code could be hard to follow.*

CREATING MULTIPLAYER GAMES WITH WEB SOCKETS

You saw earlier that one of the many things WebSockets can do is facilitate communication for multiplayer online games, even Massively Multiplayer Online Role Playing Games (MMORPGs). In this section, you create a simple multiplayer game to demonstrate the power of WebSockets. The key for multiplayer games is responsiveness: When a player takes some action, his opponents must see that action as soon as possible. This is especially critical for battle and action sequences, where opponents must compete in real time.

It would take many thousands of lines of code to create an action-packed game where responsiveness was critical. Instead, you can implement a simple, two-player Tic-tac-toe game. Tic-tac-toe (also known as Xs and Os or Noughts and Crosses) is a simplistic strategy game where the player's goal is to place three Xs or Os (whichever is his piece) in a row vertically, horizontally, or diagonally. Although Tic-tac-toe certainly does not require responsiveness, this example still demonstrates just how responsive WebSockets can be. You should follow along in the Game-Site project available on the wrox.com code download site. Only some of the code is printed in this book, and there are graphics and style sheets that are necessary for the game.

Implementing the Basic Tic-Tac-Toe Algorithm

Online Tic-tac-toe can be played against either a human opponent or a computer. The algorithm gets very complex when you include computer play because you must implement some artificial intelligence algorithms that are beyond the scope of this book. Besides, that wouldn't be as good a demonstration of WebSockets. Instead, your Tic-tac-toe game requires two human opponents. The core algorithm is contained within the `TicTacToeGame` class. This class is important to the project but doesn't perform any WebSocket operations. This is an example of separation of concerns—the game is abstracted into its own class so that any user interface, including a WebSocket interface, can use the game logic.

Because the class contains nothing new, it is not printed in this book. Open the Game-Site project and review `TicTacToeGame`. It contains some simple methods for retrieving a player's name—whose ever turn it is—including whether the game is a tie or over, and who the winner is (if anyone). The `move` method checks whether the intended move is legal, executes the move, and then calls other methods to calculate if the game is over and who the winner is. `calculateWinner` implements the algorithm for determining the winner of the game. Finally, several static methods serve as a mechanism for starting, joining, and coordinating games.

Creating the Server Endpoint

Enough with game logic, what you really came here to see is the Java WebSocket code. This is found in the `TicTacToeServer` class in Listing 10-1. The `TicTacToeGame` could be used by any user interface code to play Tic-tac-toe, but `TicTacToeServer` acts as a gateway for two WebSocket Sessions to interact with a `TicTacToeGame`. This requires some careful machinations. (Imagine if this were an unlimited-player game!)

The `onOpen` method gets the game ID and username from the path parameters (this game site is obviously not worried about security) and then, depending on whether the user is starting a new game or joining an existing game, creates or completes a `Game` object. The inner `Game` class associates the `Session` objects for the two players with their `TicTacToeGame` instance. When both users have joined, the `GameStartedMessage` is sent to both endpoints using the internal `sendJsonMessage` helper method.

LISTING 10-1: `TicTacToeServer.java`

```
@ServerEndpoint("/ticTacToe/{gameId}/{username}")
public class TicTacToeServer
{
    private static Map<Long, Game> games = new Hashtable<>();
    private static ObjectMapper mapper = new ObjectMapper();

    @OnOpen
    public void onOpen(Session session, @PathParam("gameId") long gameId,
                      @PathParam("username") String username) {
        try {
            TicTacToeGame ticTacToeGame = TicTacToeGame.getActiveGame(gameId);
            if(ticTacToeGame != null) {
                session.close(new CloseReason(
                    CloseReason.CloseCodes.UNEXPECTED_CONDITION,
```

```

        "This game has already started."
    )));
}
List<String> actions = session.getRequestParameterMap().get("action");
if(actions != null && actions.size() == 1) {
    String action = actions.get(0);
    if("start".equalsIgnoreCase(action)) {
        Game game = new Game();
        game.gameId = gameId;
        game.player1 = session;
        TicTacToeServer.games.put(gameId, game);
    } else if("join".equalsIgnoreCase(action)) {
        Game game = TicTacToeServer.games.get(gameId);
        game.player2 = session;
        game.ticTacToeGame = TicTacToeGame.startGame(gameId, username);
        this.sendJsonMessage(game.player1, game,
            new GameStartedMessage(game.ticTacToeGame));
        this.sendJsonMessage(game.player2, game,
            new GameStartedMessage(game.ticTacToeGame));
    }
}
} catch(IOException e) {
    e.printStackTrace();
    try {
        session.close(new CloseReason(
            CloseReason.CloseCodes.UNEXPECTED_CONDITION, e.toString()
        ));
    } catch(IOException ignore) { }
}
}

@OnMessage
public void onMessage(Session session, String message,
    @PathParam("gameId") long gameId) {
    Game game = TicTacToeServer.games.get(gameId);
    boolean isPlayer1 = session == game.player1;
    try {
        Move move = TicTacToeServer.mapper.readValue(message, Move.class);
        game.ticTacToeGame.move(
            isPlayer1 ? TicTacToeGame.Player.PLAYER1 :
            TicTacToeGame.Player.PLAYER2,
            move.getRow(), move.getColumn()
        );
        this.sendJsonMessage((isPlayer1 ? game.player2 : game.player1), game,
            new OpponentMadeMoveMessage(move));
        if(game.ticTacToeGame.isOver()) {
            if(game.ticTacToeGame.isDraw()) {
                this.sendJsonMessage(game.player1, game,
                    new GameIsDrawMessage());
                this.sendJsonMessage(game.player2, game,
                    new GameIsDrawMessage());
            } else {
                boolean wasPlayer1 = game.ticTacToeGame.getWinner() ==
                    TicTacToeGame.Player.PLAYER1;
                this.sendJsonMessage(game.player1, game,

```

continues

LISTING 10-1 *(continued)*

```

        new GameOverMessage(wasPlayer1));
    this.sendJsonMessage(game.player2, game,
        new GameOverMessage(!wasPlayer1));
    }
    game.player1.close();
    game.player2.close();
}
} catch(IOException e) {
    this.handleException(e, game);
}
}

@OnClose
public void onClose(Session session, @PathParam("gameId") long gameId) {
    Game game = TicTacToeServer.games.get(gameId);
    if(game == null)
        return;
    boolean isPlayer1 = session == game.player1;
    if(game.ticTacToeGame == null) {
        TicTacToeGame.removeQueuedGame(game.gameId);
    } else if(!game.ticTacToeGame.isOver()) {
        game.ticTacToeGame.forfeit(isPlayer1 ? TicTacToeGame.Player.PLAYER1 :
            TicTacToeGame.Player.PLAYER2);
        Session opponent = (isPlayer1 ? game.player2 : game.player1);
        this.sendJsonMessage(opponent, game, new GameForfeitedMessage());
        try {
            opponent.close();
        } catch(IOException e) {
            e.printStackTrace();
        }
    }
}

private void sendJsonMessage(Session session, Game game, Message message) {
    try {
        session.getBasicRemote()
            .sendText(TicTacToeServer.mapper.writeValueAsString(message));
    } catch(IOException e) {
        this.handleException(e, game);
    }
}

private void handleException(Throwable t, Game game) {
    t.printStackTrace();
    String message = t.toString();
    try {
        game.player1.close(new CloseReason(
            CloseReason.CloseCodes.UNEXPECTED_CONDITION, message
        ));
    } catch(IOException ignore) { }
    try {
        game.player2.close(new CloseReason(

```

```

        CloseReason.CloseCodes.UNEXPECTED_CONDITION, message
    );
} catch(IOException ignore) { }
}

private static class Game {
    public long gameId;
    public Session player1;
    public Session player2;
    public TicTacToeGame ticTacToeGame;
}

public static class Move {
    private int row;
    private int column;
    // accessor and mutator methods
}

public static abstract class Message {
    private final String action;
    public Message(String action) {
        this.action = action;
    }
    public String getAction() { ... }
}

public static class GameStartedMessage extends Message {
    private final TicTacToeGame game;
    public GameStartedMessage(TicTacToeGame game) {
        super("gameStarted");
        this.game = game;
    }
    public TicTacToeGame getGame() { ... }
}

public static class OpponentMadeMoveMessage extends Message {
    private final Move move;
    public OpponentMadeMoveMessage(Move move) {
        super("opponentMadeMove");
        this.move = move;
    }
    public Move getMove() { ... }
}

public static class GameOverMessage extends Message {
    private final boolean winner;
    public GameOverMessage(boolean winner) {
        super("gameOver");
        this.winner = winner;
    }
    public boolean isWinner() { ... }
}

public static class GameIsDrawMessage extends Message {
    public GameIsDrawMessage() {
        super("gameIsDraw");
    }
}

public static class GameForfeitedMessage extends Message {

```

continues

LISTING 10-1 (continued)

```
    public GameForfeitedMessage() {
        super("gameForfeited");
    }
}
```

In this system, messages sent from the browser to the server are always `Moves` (an inner class), whereas messages sent from the server to the browser are always `Messages` (another inner class). WebSocket messages here are exchanged in text format, and the Jackson Data Processor library serializes `Messages` into outgoing messages and deserializes incoming messages into `Moves`. When `onMessage` is called, that means a player has made a move and sent that move to the endpoint. The server endpoint registers the move with the `TicTacToeGame` and then notifies the opponent of the move by sending a WebSocket message over the opposing `Session`. If the game is over, it sends both players the `GameOverMessage` or `GameIsDrawMessage` as appropriate and then closes both `Sessions`. If a connection is closed, `onClose` makes sure that the game was over or never started. If the game is in progress, the user who closed their `Session` forfeited the game. (Likely they closed their browser or navigated away from the page.)

Of course, all this is useless without a user interface, and you might wonder how that `TicTacToeGame` got created in the first place. This is all coming up next.

Writing the JavaScript Game Console

To create the user interface, the first thing you need is a Servlet for starting or joining games, getting usernames for players, and forwarding requests to the appropriate JSPs. `TicTacToeServlet` accomplishes exactly that. First, a user starts a game by simply entering a username for that game session. Next, the game is added to a list of pending games. When another user comes to the site, he sees the list of pending games and chooses one to join. When he joins that game by providing his username, the game will move from pending to in-progress. The `list.jsp` page is where pending games are listed and contains the UI code for starting a new game as well. There shouldn't be much new or exciting in either the Servlet or this JSP, so they aren't printed here.

The `game.jsp` page in Listing 10-2 is where all the fun JavaScript code is happening. You'll notice at the top the inclusion of the `ticTacToe.css` style sheet, the jQuery JavaScript library, and the Bootstrap JavaScript library and CSS file. Without these doing some of the boring non-WebSockets work, you would have to write many more lines of code. The game surface is a basic `div` layout with three rows and three columns. To simplify the code, the local player is always Os and the opponent is always Xs. (Although typically the first player to move is Xs.) When it's your turn, you can hover over each game square and, if it's a legal move, the square displays a faded O in it. Clicking a square commits to that move and cannot be undone.

When the document has loaded, the code checks that the browser supports WebSockets and displays an error if it does not. It then connects to the server and, when the connection is established, it displays a message about waiting for the opponent to join. An `onbeforeunload` event is added to the `window` object to ensure that the user forfeits if he closes the browser or leaves the page. The `onclose` event ensures that the closure of the connection was clean and the `onerror` event handles errors. `onmessage` handles the five different types of messages the server could send (GameStartedMessage, OpponentMadeMoveMessage, GameOverMessage, GameIsDrawMessage, and

`GameForfeitedMessage`), clears squares that the opponent claims, and notifies the user when the game is over. Finally, the `move` function, which is called when you click a game square, sends your moves to the server.

LISTING 10-2: game.jsp

```

<%--@elvariable id="action" type="java.lang.String"--%>
<%--@elvariable id="gameId" type="long"--%>
<%--@elvariable id="username" type="java.lang.String"--%>
<!DOCTYPE html>
<html>
    <head>
        <title>Game Site :: Tic Tac Toe</title>
        <link rel="stylesheet" href="http://cdnjs.cloudflare.com/ajax/libs/twitter-
bootstrap/2.3.1/css/bootstrap.min.css" />
        <link rel="stylesheet"
            href="

```

continues

LISTING 10-2 (continued)

```
</div>
<div class="modal-body" id="modalErrorBody">A blah error occurred.
</div>
<div class="modal-footer">
    <button class="btn btn-primary" data-dismiss="modal">OK</button>
</div>
</div>
<div id="modalGameOver" class="modal hide fade">
    <div class="modal-header">
        <button type="button" class="close" data-dismiss="modal">&times;
        </button>
        <h3>Game Over</h3>
    </div>
    <div class="modal-body" id="modalGameOverBody">&nbsp;</div>
    <div class="modal-footer">
        <button class="btn btn-primary" data-dismiss="modal">OK</button>
    </div>
</div>
<script type="text/javascript" language="javascript">
    var move;
    $(document).ready(function() {
        var modalError = $("#modalError");
        var modalErrorBody = $("#modalErrorBody");
        var modalWaiting = $("#modalWaiting");
        var modalWaitingBody = $("#modalWaitingBody");
        var modalGameOver = $("#modalGameOver");
        var modalGameOverBody = $("#modalGameOverBody");
        var opponent = $("#opponent");
        var status = $("#status");
        var opponentUsername;
        var username = '<c:out value="${username}" />';
        var myTurn = false;

        $('.game-cell').addClass('span1');

        if(!("WebSocket" in window))
        {
            modalErrorBody.text('WebSockets are not supported in this ' +
                'browser. Try Internet Explorer 10 or the latest ' +
                'versions of Mozilla Firefox or Google Chrome.');
            modalError.modal('show');
            return;
        }

        modalWaitingBody.text('Connecting to the server.');
        modalWaiting.modal({ keyboard: false, show: true });

        var server;
        try {
            server = new WebSocket('ws://' + window.location.host +
                '<c:url value="/ticTacToe/${gameId}/${username}">
                <c:param name="action" value="${action}" />
```

```

        </c:url>');
    } catch(error) {
        modalWaiting.modal('hide');
        modalErrorBody.text(error);
        modalError.modal('show');
        return;
    }

    server.onopen = function(event) {
        modalWaitingBody
            .text('Waiting on your opponent to join the game.');
        modalWaiting.modal({ keyboard: false, show: true });
    };

    window.onbeforeunload = function() {
        server.close();
    };

    server.onclose = function(event) {
        if(!event.wasClean || event.code != 1000) {
            toggleTurn(false, 'Game over due to error!');
            modalWaiting.modal('hide');
            modalErrorBody.text('Code ' + event.code + ': ' +
                event.reason);
            modalError.modal('show');
        }
    };
}

server.onerror = function(event) {
    modalWaiting.modal('hide');
    modalErrorBody.text(event.data);
    modalError.modal('show');
};

server.onmessage = function(event) {
    var message = JSON.parse(event.data);
    if(message.action == 'gameStarted') {
        if(message.game.player1 == username)
            opponentUsername = message.game.player2;
        else
            opponentUsername = message.game.player1;
        opponent.text(opponentUsername);
        toggleTurn(message.game.nextMoveBy == username);
        modalWaiting.modal('hide');
    } else if(message.action == 'opponentMadeMove') {
        $('#r' + message.move.row + 'c' + message.move.column)
            .unbind('click')
            .removeClass('game-cell-selectable')
            .addClass('game-cell-opponent game-cell-taken');
        toggleTurn(true);
    } else if(message.action == 'gameOver') {
        toggleTurn(false, 'Game Over!');
        if(message.winner) {
            modalGameOverBody.text('Congratulations, you won!');
        } else {
    
```

continues

LISTING 10-2 (continued)

```

        modalGameOverBody.text('User "' + opponentUsername +
            '" won the game.');
    }
    modalGameOver.modal('show');
} else if(message.action == 'gameIsDraw') {
    toggleTurn(false, 'The game is a draw. ' +
        'There is no winner.');
    modalGameOverBody.text('The game ended in a draw. ' +
        'Nobody wins!');
    modalGameOver.modal('show');
} else if(message.action == 'gameForfeited') {
    toggleTurn(false, 'Your opponent forfeited!');
    modalGameOverBody.text('User "' + opponentUsername +
        '" forfeited the game. You win!');
    modalGameOver.modal('show');
}
};

var toggleTurn = function(isMyTurn, message) {
    myTurn = isMyTurn;
    if(myTurn) {
        status.text(message || 'It\'s your move!');
        $('.game-cell:not(.game-cell-taken)')
            .addClass('game-cell-selectable');
    } else {
        status.text(message || 'Waiting on your opponent to move.');
        $('.game-cell-selectable')
            .removeClass('game-cell-selectable');
    }
};

move = function(row, column) {
    if(!myTurn) {
        modalErrorBody.text('It is not your turn yet!');
        modalError.modal('show');
        return;
    }
    if(server != null) {
        server.send(JSON.stringify({ row: row, column: column }));
        $('#r' + row + 'c' + column).unbind('click')
            .removeClass('game-cell-selectable')
            .addClass('game-cell-player game-cell-taken');
        toggleTurn(false);
    } else {
        modalErrorBody.text('Not connected to came server.');
        modalError.modal('show');
    }
};
});
</script>
</body>
</html>

```

Playing WebSocket Tic-Tac-Toe

Now that you have reviewed the code and see how all the pieces fit together, you can do the following:

1. Compile the Game-Site project.
2. Start up Tomcat in your debugger, and open up two different browsers (such as Firefox and Safari, or Internet Explorer 10 and Chrome). Make them small enough so that you can place them side-by-side or one on top of the other in your screen.
3. Go to `http://localhost:8080/games/ticTacToe` in both browsers. There should be no games listed yet.
4. Click Start a Game in one browser and enter your name or favorite username in the prompt, and then click OK. You will be taken to the game page and presented with the message about waiting for your opponent.
5. In the other browser, reload the listing page and you should see the pending game listed. Click that game and enter in a *different* name or username in the prompt. As soon as you land on the game page, the waiting message should disappear in the first browser. You'll notice that it is nearly instantaneous. One of the browsers should have a message that says, It's Your Move!, whereas the message in the other says, Waiting on Your Opponent to Move.
6. Go back and forth between browsers making moves. Before long your screen should look something like Figure 10-6 (where John is about to beat Scott with a move in the lower-left corner). Notice how quickly the move from one player shows up in the other player's browser. The delay is imperceptible. The speed and scalability of WebSockets make it an extraordinarily powerful technology.

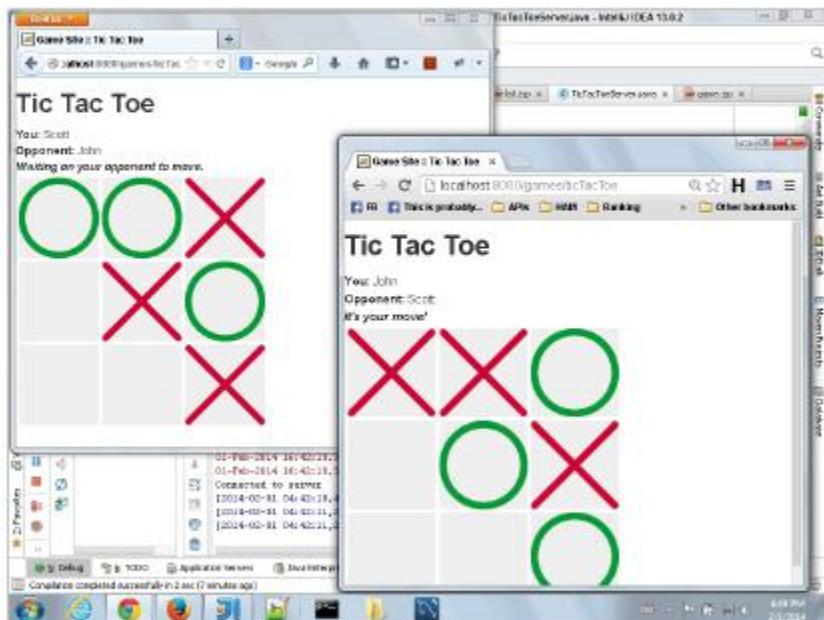


FIGURE 10-6

USING WEB SOCKETS TO COMMUNICATE IN A CLUSTER

Now that you've seen how Java server endpoints work, and how to communicate with server endpoints using JavaScript, it's time to explore the Java client endpoint. Because the client endpoint can't be used to connect to a browser, you need to connect to some other application. There are many possible uses for this, from data transfers to coordination of distributed activities across multiple servers. There are endless possibilities. As a software developer, chances are you will have to deal with a cluster of servers at some point to scale a web application to handle large numbers of users, and WebSockets might be a way to help application nodes communicate with each other.

Simulating a Simple Cluster Using Two Servlet Instances

In a standard cluster scenario, nodes would notify each other of their existence through some means, often by sending a packet to an agreed-upon multicast IP address and port. They would then establish communications through some other channel, such as a TCP socket. This is a complicated scenario to replicate in a small example, but you can easily simulate it by using multiple servlets in a single application. On the wrox.com code download site, this example is the Simulated-Cluster project. Take a look at `web.xml` first, and you'll notice two Servlet mappings. The first Servlet mapping is in the following code. The second Servlet mapping is identical except that the name, init parameter value, and URL pattern have a 2 in them instead of a 1. If you're wondering why you need this in the deployment descriptor, remember that you can't map the same Servlet twice using annotations. You must use either the deployment descriptor or programmatic configuration to accomplish this.

```
<servlet>
  <servlet-name>clusterNode1</servlet-name>
  <servlet-class>com.wrox.ClusterNodeServlet</servlet-class>
  <init-param>
    <param-name>nodeId</param-name>
    <param-value>1</param-value>
  </init-param>
</servlet>
<servlet-mapping>
  <servlet-name>clusterNode1</servlet-name>
  <url-pattern>/clusterNode1</url-pattern>
</servlet-mapping>
```

The `ClusterNodeServlet` class in Listing 10-3, which extends `HttpServlet`, is annotated with `@ClientEndpoint`. Unlike `@ServletEndpoint`, `@ClientEndpoint` does not mean the class will be instantiated automatically. `@ClientEndpoint` is a marker to tell the container that this is a valid endpoint.

Alternatively, you could implement the `Endpoint` abstract class. *Any* class can be an endpoint. A Servlet is used here only because it's easy and convenient. The `init` method, which is called on the first request, connects to the server endpoint, and the `destroy` method closes the connection. Every time a request comes in, the Servlet sends a message to the cluster about it. The `onMessage` method (annotated with `@OnMessage`) accepts messages echoed from other cluster nodes, and `onClose` (annotated with `@OnClose`) prints an error message if the connection is closed abnormally.

LISTING 10-3: ClusterNodeServlet.java

```

@ClientEndpoint
public class ClusterNodeServlet extends HttpServlet
{
    private Session session;
    private String nodeId;

    @Override
    public void init() throws ServletException {
        this.nodeId = this.getInitParameter("nodeId");
        String path = this.getServletContext().getContextPath() +
                      "/clusterNodeSocket/" + this.nodeId;
        try {
            URI uri = new URI("ws", "localhost:8080", path, null, null);
            this.session = ContainerProvider.getWebSocketContainer()
                .connectToServer(this, uri);
        } catch(URISyntaxException | IOException | DeploymentException e) {
            throw new ServletException("Cannot connect to " + path + ".", e);
        }
    }

    @Override
    public void destroy() {
        try {
            this.session.close();
        } catch(IOException e) {
            e.printStackTrace();
        }
    }

    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        ClusterMessage message = new ClusterMessage(this.nodeId,
            "request:{ip:\\" + request.getRemoteAddr() +
            "\",queryString:\\" + request.getQueryString() + "\")");
        try(OutputStream output = this.session.getBasicRemote().getSendStream();
            ObjectOutputStream stream = new ObjectOutputStream(output)) {
            stream.writeObject(message);
        }
        response.getWriter().append("OK");
    }

    @OnMessage
    public void onMessage(InputStream input) {
        try(ObjectInputStream stream = new ObjectInputStream(input)) {
            ClusterMessage message = (ClusterMessage)stream.readObject();
            System.out.println("INFO (Node " + this.nodeId +
                "): Message received from cluster; node = " +
                message.getNodeId() + ", message = " + message.getMessage());
        } catch(IOException | ClassNotFoundException e) {
            e.printStackTrace();
        }
    }
}

```

continues

LISTING 10-3 *(continued)*

```

    }

    @OnClose
    public void onClose(CloseReason reason) {
        CloseReason.CloseCode code = reason.getCloseCode();
        if(code != CloseReason.CloseCodes.NORMAL_CLOSURE) {
            System.err.println("ERROR: WebSocket connection closed unexpectedly;" +
                " code = " + code + ", reason = " + reason.getReasonPhrase());
        }
    }
}

```

Transmitting and Receiving Binary Messages

You probably noticed that the `ClusterNodeServlet` used Java serialization to send and receive the `ClusterMessage` implements `Serializable` messages over the WebSocket connection. To do this, the WebSocket messages have to be binary. This is different from the Tic-tac-toe server endpoint, which translates messages to and from JSON and sends and receives them as text messages. Java serialization is faster than JSON, so when possible it's better to use Java serialization. However, you can only do so when both peers are Java. If only one peer is Java, you have to use a different serialization technology, such as JSON. Although `ClusterNodeServlet` sends and receives binary messages using `OutputStreams` and `InputStreams`, `ClusterNodeEndpoint` in Listing 10-4 uses `byte` arrays. In this case, this takes slightly more code because `ObjectOutputStreams` and `ObjectInputStreams` are already needed to perform the serialization and deserialization. In some cases using `byte` arrays will be easier, and in others using `ByteBuffers` or streams will be easier, so it's valuable to learn both approaches. It all depends on where your data comes from or what you want to do with it. If you need to send data that already exists as a `byte` array, it's easier to just use that `byte` array. On the other hand, if you're already dealing with streams, it's usually easier to just stick to streams.

The endpoint's only responsibilities are echoing messages sent from one node to all the other nodes and notifying connected nodes when other nodes join or leave the cluster. In another cluster scenario, you might not have a central endpoint that collects and echoes all messages. Instead, each node might connect directly to all other nodes. It all depends on your use case and the needs of your application.

LISTING 10-4: `ClusterNodeEndpoint.java`

```

@ServerEndpoint("/clusterNodeSocket/{nodeId}")
public class ClusterNodeEndpoint
{
    private static final List<Session> nodes = new ArrayList<>(2);

    @OnOpen
    public void onOpen(Session session, @PathParam("nodeId") String nodeId) {
        System.out.println("INFO: Node [" + nodeId + "] connected to cluster.");
        ClusterMessage message = new ClusterMessage(nodeId, "Joined the cluster.");
        try {
            byte[] bytes = ClusterNodeEndpoint.toByteArray(message);
            for(Session node : ClusterNodeEndpoint.nodes)

```

```

        node.getBasicRemote().sendBinary(ByteBuffer.wrap(bytes));
    } catch(IOException e) {
        System.err.println("ERROR: Exception when notifying of new node");
        e.printStackTrace();
    }
    ClusterNodeEndpoint.nodes.add(session);
}

@OnMessage
public void onMessage(Session session, byte[] message) {
    try {
        for(Session node : ClusterNodeEndpoint.nodes) {
            if(node != session)
                node.getBasicRemote().sendBinary(ByteBuffer.wrap(message));
        }
    } catch(IOException e) {
        System.err.println("ERROR: Exception when handling message on server");
        e.printStackTrace();
    }
}

@OnClose
public void onClose(Session session, @PathParam("nodeId") String nodeId) {
    System.out.println("INFO: Node [" + nodeId + "] disconnected.");
    ClusterNodeEndpoint.nodes.remove(session);
    ClusterMessage message = new ClusterMessage(nodeId, "Left the cluster.");
    try {
        byte[] bytes = ClusterNodeEndpoint.toByteArray(message);
        for(Session node : ClusterNodeEndpoint.nodes)
            node.getBasicRemote().sendBinary(ByteBuffer.wrap(bytes));
    } catch(IOException e) {
        System.err.println("ERROR: Exception when notifying of left node");
        e.printStackTrace();
    }
}

private static byte[] toByteArray(ClusterMessage message) throws IOException {
    try (ByteArrayOutputStream output = new ByteArrayOutputStream();
        ObjectOutputStream stream = new ObjectOutputStream(output)) {
        stream.writeObject(message);
        return output.toByteArray();
    }
}
}

```

Testing the Simulated Cluster Application

Testing the simulated cluster application is fairly straightforward. Just take the following steps:

1. Compile the application and start Tomcat from your IDE.
 2. Go to <http://localhost:8080/cluster/clusterNode1> in your favorite browser, and you should see the following message in the debugger output. This is the result of the first Servlet instance's `init` method being called and connecting to the WebSocket server endpoint.

INFO: Node [1] connected to cluster.

3. Now go to `http://localhost:8080/cluster/clusterNode2`. You should see a couple of messages this time. In this case the second Servlet instance connected to the endpoint, and when it sent a message, the endpoint echoed that message back to the first Servlet.

```
INFO: Node [2] connected to cluster.
INFO (Node 1): Message received from cluster; node = 2, message =
Joined the cluster.
INFO (Node 1): Message received from cluster; node = 2, message =
request:{ip:"127.0.0.1",queryString:""}
```

4. Try the first URL again, but this time add a query string to it: `http://localhost:8080/cluster/clusterNode1?hello=world&foo=bar`. Notice this time there is no connection message because both Servlets are started now.

```
INFO (Node 2): Message received from cluster; node = 1, message =
request:{ip:"127.0.0.1",queryString:"hello=world&foo=bar"}
```

5. One last time, try the second URL again, also with a query string: `http://localhost:8080/cluster/clusterNode2?baz=qux&animal=dog`.

```
INFO (Node 1): Message received from cluster; node = 2, message =
request:{ip:"127.0.0.1",queryString:"baz=qux&animal=dog"}
```

6. If you want to experiment further, create another instance of the Servlet and map it to `/clusterNode3`; then compile and start the application again. Try going to all three URLs, and you'll see that whenever a Servlet responds to a `GET` request, the other two Servlets receive a WebSocket message about it.

This may seem like a juvenile example, but it's a simple way to demonstrate how the Java WebSocket client and server APIs can work together. It is also a good opportunity for you to learn about using `byte arrays`, `ByteBuffers`, and streams for binary messages. This approach might be very useful for you in some clustered applications, and you may need a different set of technologies in other clustered applications. You explore more options in Chapter 18.

NOTE *You may have noticed that the Servlet instances in this project were set to initialize on the first request instead of on application start-up. There is a very important reason for this: The `ClusterNodeServlet` connects to the WebSocket endpoint in the `init` method. If the application is in the process of starting when the Servlets initialize, they won't connect to the endpoint. Thus, the Servlets must initialize after application start-up.*

ADDING “CHAT WITH SUPPORT” TO THE CUSTOMER SUPPORT APPLICATION

Perhaps the most ubiquitous example of the usefulness of WebSockets is Internet chatting. Many applications provide chatting capabilities from your desktop and most use some kind of service provider through which chat messages are routed. Chat is also common on websites, with many social networks, forums, and online communities offering chat capabilities. Typically, chat works one of two ways:

- **Chat room** — This has more than two, and usually unlimited, participants. Chat rooms are also usually public, requiring only membership on the site to join.
- **Private chat** — This usually has only two participants. Nobody else can see the contents of the chat.

Whether a chat is private or in a chat room, the server-side implementation is essentially the same: The server accepts connections, associates all the related connections, and echoes incoming messages to all associated connections. It also publishes interesting events, such as when somebody connects to or disconnects from the chat. The only big difference is how many connections are associated with each other.

Multinational Widget Corporation needs support chat in its Customer Support application. Support chat is a basic concept: In an urgent situation, a customer might need live help. The customer could log on to the support site and enter a private chat with a customer support representative. Typically, this would be offered only during certain hours. Also, usually customers have the ability to download or have the chat log e-mailed to them at the end of the chat session. In this section, you use WebSockets to add "Chat with Support" to the Customer Support application. You should follow along in the Customer-Support-v8 project available on the wrox.com code download site because there is not enough room in this book to print it all.

There are some changes to the `main.css` style sheet and a new `chat.css` style sheet to correctly style the chat pages. The `/WEB-INF/tags/template/main.tag` file now includes some third-party CSS and JavaScript libraries and defines a couple of JavaScript functions for use on any pages. Also, `basic.tag` now has links for creating a new chat with support or viewing pending chat requests.

Normally, only customer support representatives could view and respond to pending chat requests, but the customer support application doesn't have full security with user permissions yet. You'll add that in Part IV of this book. The `ChatServlet` has a fairly simple task: It manages the listing, creating, and joining of chat sessions. The `doPost` method sets the `Expires` and `Cache-Control` headers to ensure the browser doesn't cache the chat page. `/WEB-INF/jsp/view/chat/list.jsp` is responsible for listing pending chats for customer support personnel to respond to. Finally, the `Configurator` class now also maps the `AuthenticationFilter` to the `/chat` URL so that the `ChatServlet` is protected.

Using Encoders and Decoders to Translate Messages

Earlier in the chapter you learned about all the possible parameters you could specify for `@OnMessage` methods. Remember:

- You can specify any Java object as a parameter as long as you provide a decoder capable of translating incoming text or binary messages into that object.
- You can send any object using the `sendObject` methods of `RemoteEndpoint.Basic` or `RemoteEndpoint.Async` as long as you provide an encoder capable of translating that object into a text or binary message.
- You provide encoders by implementing `Encoder.Binary`, `Encoder.BinaryStream`, `Encoder.Text`, or `Encoder.TextStream` and specifying their classes in the `encoders` attribute of `@ClientEndpoint` or `@ServerEndpoint`.

- You can implement `Decoder.Binary`, `Decoder.BinaryStream`, `Decoder.Text`, or `Decoder.TextStream` and use the `decoders` attribute of the endpoint annotations to provide decoders for your messages.

```
public class ChatMessage
{
    private OffsetDateTime timestamp;
    private Type type;
    private String user;
    private String content;

    // accessor and mutator methods

    public static enum Type
    {
        STARTED, JOINED, ERROR, LEFT, TEXT
    }
}
```

The previous code for the `ChatMessage` shows that it is a simple POJO. The WebSocket API needs both an encoder and a decoder so that your chat application can send and receive messages. Listing 10-5 contains a simple class that encodes and decodes `ChatMessages`. It uses the Jackson Data Processor to encode and decode the messages. The `encode` method takes a `ChatMessage` and an `OutputStream`, encodes the message by converting it to JSON, and writes it to the `OutputStream`. The `decode` method does the opposite: Given an `InputStream`, it reads and deserializes the JSON `ChatMessage`. The `init` and `destroy` methods are specified in both the `Encoder` and `Decoder` interfaces. They are not used here, but could come in handy if you ever need to initialize or release resources that your encoders and decoders use.

LISTING 10-5: ChatMessageCodec.java

```
public class ChatMessageCodec
    implements Encoder.BinaryStream<ChatMessage>,
               Decoder.BinaryStream<ChatMessage>
{
    private static final ObjectMapper MAPPER = new ObjectMapper();

    static {
        MAPPER.findAndRegisterModules();
        MAPPER.configure(JsonGenerator.Feature.AUTO_CLOSE_TARGET, false);
    }

    @Override
    public void encode(ChatMessage chatMessage, OutputStream outputStream)
        throws EncodeException, IOException {
        try {
            ChatMessageCodec.MAPPER.writeValue(outputStream, chatMessage);
        } catch(JsonGenerationException | JsonMappingException e) {
            throw new EncodeException(chatMessage, e.getMessage(), e);
        }
    }

    @Override
```

```
public ChatMessage decode(InputStream inputStream)
    throws DecodeException, IOException {
    try {
        return ChatMessageCodec.MAPPER.readValue(
            inputStream, ChatMessage.class
        );
    } catch (JsonParseException | JsonMappingException e) {
        throw new DecodeException((ByteBuffer)null, e.getMessage(), e);
    }
}

@Override
public void init(EndpointConfig endpointConfig) { }

@Override
public void destroy() { }
}
```

Creating the Chat Server Endpoint

The server endpoint uses the following `ChatSession` class to associate a user requesting a chat with the support representative who responds. It includes the opening message and a log of all messages sent during the chat.

```
public class ChatSession
{
    private long sessionId;
    private String customerUsername;
    private Session customer;
    private String representativeUsername;
    private Session representative;
    private ChatMessage creationMessage;
    private final List<ChatMessage> chatLog = new ArrayList<>();

    // accessor and mutator methods

    @JsonIgnore
    public void log(ChatMessage message) { ... }

    @JsonIgnore
    public void writeChatLog(File file) throws IOException {
        ObjectMapper mapper = new ObjectMapper();
        mapper.findAndRegisterModules();
        mapper.configure(JsonGenerator.Feature.AUTO_CLOSE_TARGET, false);
        mapper.configure(SerializationFeature.WRITE_DATES_AS_TIMESTAMPS, false);

        try(FileOutputStream stream = new FileOutputStream(file))
        {
            mapper.writeValue(stream, this.chatLog);
        }
    }
}
```

The `ChatEndpoint` class in Listing 10-6 receives chat connections and coordinates them appropriately. Some of the code has been omitted due to its length, but concepts that are new to you have been included. Nested within the class is the `EndpointConfigurator` class, which overrides the `modifyHandshake` method. At handshake time, this method is called and exposes the underlying HTTP request.

You can get the `HttpSession` object from this request, and in this case you want the HTTP session to make sure the user is logged in and also close the WebSocket session if the user logs out. This is also why the endpoint is an `HttpSessionListener`. When a session gets invalidated, the `sessionDestroyed` method is called, and the endpoint ends the chat session. One thing to keep in mind is that a new instance of this class is created at startup as a web listener, and a new instance is created each time a client endpoint connects to the server endpoint. That's why all the fields are static: so that the information about the `Sessions`, `HttpSessions`, and `ChatSessions` can be coordinated between instances.

The `onOpen` method, called when a new handshake has completed, first checks to make sure an `HttpSession` was associated with the `Session` (in the `modifyHandshake` method) and that the user is logged in. If the chat session ID is 0 (a new session has been requested), a new chat session is created and added to the list of pending sessions. If it is greater than 0, a representative is joining a requested session and messages are sent to both clients. When `onMessage` receives a message from either client, it echoes that message back to both clients. When a `Session` is closed, an error occurs, or an `HttpSession` is destroyed, a message is sent to the other user notifying her that the chat is over, and both connections are closed.

LISTING 10-6: ChatEndpoint.java

```

@ServerEndpoint(value = "/chat/{sessionId}",
    encoders = ChatMessageCodec.class,
    decoders = ChatMessageCodec.class,
    configurator = ChatEndpoint.EndpointConfigurator.class)
@WebListener
public class ChatEndpoint implements HttpSessionListener
{
    ...
    private static final Map<Long, ChatSession> chatSessions = new Hashtable<>();
    private static final Map<Session, ChatSession> sessions = new Hashtable<>();
    private static final Map<Session, HttpSession> httpSessions =
        new Hashtable<>();
    public static final List<ChatSession> pendingSessions = new ArrayList<>();

    @OnOpen
    public void onOpen(Session session, @PathParam("sessionId") long sessionId) {
        HttpSession httpSession = (HttpSession)session.getUserProperties()
            .get(ChatEndpoint.HTTP_SESSION_PROPERTY);
        try {
            if(httpSession==null || httpSession.getAttribute("username")==null) {
                session.close(new CloseReason(
                    CloseReason.CloseCodes.VIOLATED_POLICY,
                    "You are not logged in!"
                ));
            }
        }
    }
}

```

```

        return;
    }
    String username = (String)httpSession.getAttribute("username");
    session.getUserProperties().put("username", username);
    ChatMessage message = new ChatMessage();
    message.setTimestamp(OffsetDateTime.now());
    message.setUser(username);
    ChatSession chatSession;
    if(sessionId < 1) {
        message.setType(ChatMessage.Type.STARTED);
        message.setContent(username + " started the chat session.");
        chatSession = new ChatSession();
        synchronized(ChatEndpoint.sessionIdSequenceLock) {
            chatSession.setSessionId(ChatEndpoint.sessionIdSequence++);
        }
        chatSession.setCustomer(session);
        chatSession.setCustomerUsername(username);
        chatSession.setCreationMessage(message);
        ChatEndpoint.pendingSessions.add(chatSession);
        ChatEndpoint.chatSessions.put(chatSession.getSessionId(),
            chatSession);
    } else {
        message.setType(ChatMessage.Type.JOINED);
        message.setContent(username + " joined the chat session.");
        chatSession = ChatEndpoint.chatSessions.get(sessionId);
        chatSession.setRepresentative(session);
        chatSession.setRepresentativeUsername(username);
        ChatEndpoint.pendingSessions.remove(chatSession);
        session.getBasicRemote()
            .sendObject(chatSession.getCreationMessage());
        session.getBasicRemote().sendObject(message);
    }
    ChatEndpoint.sessions.put(session, chatSession);
    ChatEndpoint.httpSessions.put(session, httpSession);
    this.getSessionsFor(httpSession).add(session);
    chatSession.log(message);
    chatSession.getCustomer().getBasicRemote().sendObject(message);
} catch(IOException | EncodeException e) {
    this.onError(session, e);
}
}

@OnMessage
public void onMessage(Session session, ChatMessage message) {
    ChatSession c = ChatEndpoint.sessions.get(session);
    Session other = this.getOtherSession(c, session);
    if(c != null && other != null) {
        c.log(message);
        try {
            session.getBasicRemote().sendObject(message);
            other.getBasicRemote().sendObject(message);
        } catch(IOException | EncodeException e) {
            this.onError(session, e);
        }
    }
}

```

continues

LISTING 10-6 (continued)

```

    }

    @OnClose
    public void onClose(Session session, CloseReason reason) { ... }

    @OnError
    public void onError(Session session, Throwable e) { ... }

    @Override
    public void sessionDestroyed(HttpSessionEvent event) { ... }

    @Override
    public void sessionCreated(HttpSessionEvent event) { /* do nothing */ }

    @SuppressWarnings("unchecked")
    private synchronized ArrayList<Session> getSessionsFor(HttpSession session) {
        try {
            if(session.getAttribute(WS_SESSION_PROPERTY) == null)
                session.setAttribute(WS_SESSION_PROPERTY, new ArrayList<>());
            return (ArrayList<Session>)session.getAttribute(WS_SESSION_PROPERTY);
        } catch(IllegalStateException e) {
            return new ArrayList<>();
        }
    }

    private Session close(Session s, ChatMessage message) { ... }

    private Session getOtherSession(ChatSession c, Session s) { ... }

    public static class EndpointConfigurator
        extends ServerEndpointConfig.Configurator {
        @Override
        public void modifyHandshake(ServerEndpointConfig config,
                                   HandshakeRequest request,
                                   HandshakeResponse response) {
            super.modifyHandshake(config, request, response);
            config.getUserProperties().put(
                ChatEndpoint.HTTP_SESSION_PROPERTY, request.getHttpSession()
            );
        }
    }
}
}

```

Writing the JavaScript Chat Application

The `/WEB-INF/jsp/view/chat/chat.jsp` file contains the user interface for the support chat. Much of the code is presentation, connection, and error handling, which you have seen before with the Tic-tac-toe game. Of particular importance are the `onmessage` event and the `send` function, which deal with receiving and sending binary messages, respectively. Because this application uses JSON to send messages between the browser and the server, it would have been easier to use text messages instead of binary messages. However, you have not yet seen how to handle binary WebSocket messages in JavaScript, so this example demonstrates that.

```

server.onmessage = function(event) {
    if(event.data instanceof ArrayBuffer) {
        var message = JSON.parse(String.fromCharCode.apply(
            null, new Uint8Array(event.data)
        ));
        objectMessage(message);
        if(message.type == 'JOINED') {
            otherJoined = true;
            if(username != message.user)
                infoMessage('You are now chatting with ' +
                    message.user + '.');
        }
    } else {
        modalErrorBody.text('Unexpected data type [' +
            typeof(event.data) + '].');
        modalError.modal('show');
    }
};

send = function() {
    if(server == null) {
        modalErrorBody.text('You are not connected!');
        modalError.modal('show');
    } else if(!otherJoined) {
        modalErrorBody.text(
            'The other user has not joined the chat yet.');
        modalError.modal('show');
    } else if(messageArea.get(0).value.trim().length > 0) {
        var message = {
            timestamp: new Date(), type: 'TEXT', user: username,
            content: messageArea.get(0).value
        };
        try {
            var json = JSON.stringify(message);
            var length = json.length;
            var buffer = new ArrayBuffer(length);
            var array = new Uint8Array(buffer);
            for(var i = 0; i < length; i++) {
                array[i] = json.charCodeAt(i);
            }
            server.send(buffer);
            messageArea.get(0).value = '';
        } catch(error) {
            modalErrorBody.text(error);
            modalError.modal('show');
        }
    }
};

```

Now compile and start the Customer-Support-v8 application, and open two different Internet browsers to <http://localhost:8080/support>. Log in to the support application in both browsers as *different users*. (Remember, you can view the available usernames and passwords in the `LoginServlet` code.) In one browser, click Chat with Support to request a chat session. In the other browser click View Chat Requests and then click the chat session you previously opened. Type

messages in each browser and click Send. You should soon see something like Figure 10-7. You are now chatting with yourself through both browsers.

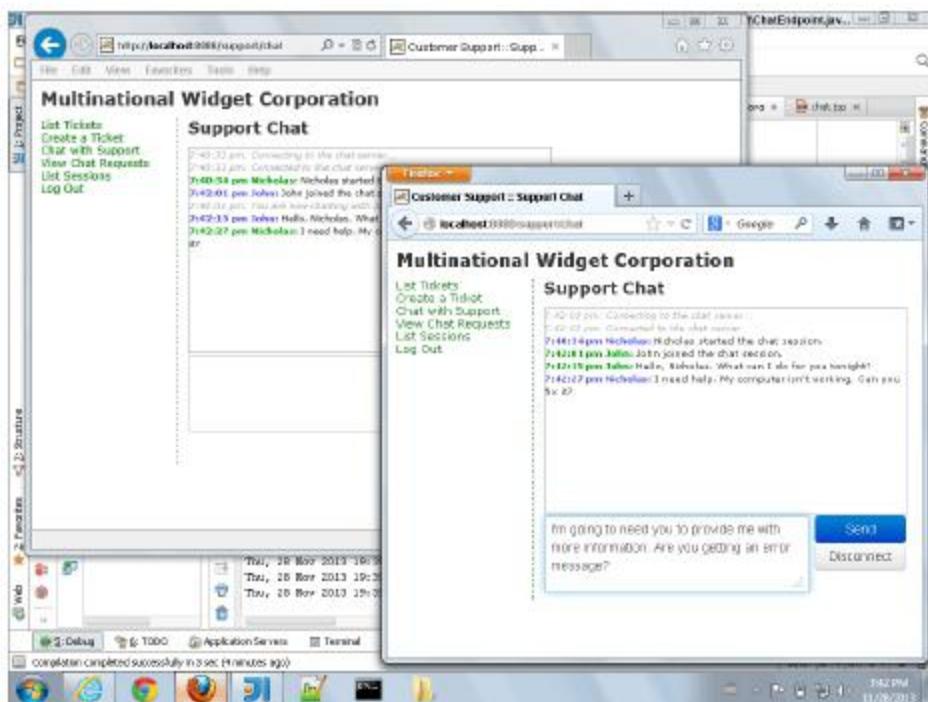


FIGURE 10-7

SUMMARY

WebSockets are an extremely useful, powerful new technology in the Internet world. Like the HTTP protocol, which underwent many changes in its early days, the WebSockets protocol is likely to undergo many changes in the coming years as well. Currently, it has a framework for extensions, but no existing extensions. That is likely to change as more and more developers begin using the technology. There are endless uses for WebSockets, and you explored several of those uses in this chapter. You created a multiplayer game of Tic-tac-toe, used WebSockets to communicate within nodes in an application cluster, and added support chat to the Customer Support application. You learned about the technologies that led to WebSockets and how WebSockets solved many of the problems those technologies couldn't. By now you should be familiar with the protocol and the Java and JavaScript APIs necessary to use it.

In the next chapter you learn about application logging principles and the technologies you can use to facilitate debugging and tracing your application and identifying errors that occur.

11

Using Logging to Monitor Your Application

IN THIS CHAPTER

- All about logging
- What are the different logging levels?
- What logging facility is right for you?
- How to integrate logging into your application
- Include logging in the Customer Support application

WROX.COM CODE DOWNLOADS FOR THIS CHAPTER

You can find the `wrox.com` code downloads for this chapter at <http://www.wrox.com/go/projavaforwebapps> on the Download Code tab. The code for this chapter is divided into the following major examples:

- Logging-Integration Project
- Customer-Support-v9 Project

NEW MAVEN DEPENDENCIES FOR THIS CHAPTER

In addition to the Maven dependencies introduced in previous chapters, you also need the following Maven dependencies:

```
<dependency>
  <groupId>org.apache.logging.log4j</groupId>
  <artifactId>log4j-api</artifactId>
  <version>2.0</version>
  <scope>compile</scope>
```

```
</dependency>

<dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-core</artifactId>
    <version>2.0</version>
    <scope>runtime</scope>
</dependency>

<dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-jcl</artifactId>
    <version>2.0</version>
    <scope>runtime</scope>
</dependency>

<dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-slf4j-impl</artifactId>
    <version>2.0</version>
    <scope>runtime</scope>
</dependency>

<dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-taglib</artifactId>
    <version>2.0</version>
    <scope>runtime</scope>
</dependency>
```

UNDERSTANDING THE CONCEPTS OF LOGGING

Imagine a scenario: You have a report of a bug in your application. Given precise replication steps from customer support, you can duplicate the bug every time. The problem is, as soon as you hit a breakpoint in the debugger, the problem goes away. Vanishes without an explanation. This likely (though not certainly) means the problem is related to multithreading, and hitting a breakpoint slows the execution down enough to mask the problem. So how are you supposed to figure out what the problem is? Application logging is critical to successful debugging, troubleshooting, monitoring, and error reporting in your application. In this chapter you explore the concepts of logging; learn about logging façades, APIs, and implementations; and integrate logging into an application.

Why You Should Log

As a personal choice, we tend not to use debuggers beyond getting a stack trace or the value of a variable or two. One reason is that it is easy to get lost in details of complicated data structures and control flow; we find stepping through a program less productive than thinking harder and adding output statements and self-checking code at critical places. Clicking over statements takes longer than scanning the output of judiciously-placed displays. It takes less time to decide

where to put print statements than to single-step to the critical section of code, even assuming we know where that is. More important, debugging statements stay with the program; debugger sessions are transient.

Kernighan, Brian W. and Rob Pike. *THE PRACTICE OF PROGRAMMING*. Reading: Addison Wesley Longman, Inc., 1999. 119.

There are many reasons you should log. The preceding quote highlights some important points about logging—and about the drawbacks of debuggers. It should be noted that much time has passed since that book was published, and debuggers have come a long way. The ease of firing up a debugger in an IDE and stepping through an application's execution cannot be understated in today's world. In many ways the Java debugger is the best around, and debugger integration in today's Java IDEs makes inspection and even modification of static, instance, and stack variables a simple matter. In most Java IDEs, you can evaluate an arbitrary expression at run time when stopped at a breakpoint, and you can change, recompile, and reload code without restarting the Java virtual machine.

But even with all that, debugging is not enough. The hypothetical presented earlier is one of many examples in which a debugger gets in the way. In addition, sometimes a debugger is simply not available. Consider a situation in which a customer issue cannot be replicated in a development environment, or even a quality assurance environment. In most cases, attaching a debugger to your customer's running production application is simply not an option.

Let's face it: Bugs are not planned. You write your application to be perfect, but bugs surface anyway. They always will. There are many techniques you can employ and tools you can use to detect and minimize bugs in your code, but you will never avoid them completely. So, it makes sense that you should plan to troubleshoot a bug without a debugger.

Debugging your application is not the only reason you should log, though. Often you need to be notified of important events such as errors, changes in application configuration, unexpected behavior, component startup or shutdown, user logins, data changes, and more. You may also want to simply monitor your application to make sure it is working and undergoing activity of some sort. All of these needs—and more—demonstrate why you should log.

In the earliest days of Java (`java.util.logging` wasn't part of the framework until Java 1.4), many developers used `System.err.println()` and `System.out.println()` to "log" errors and interesting information. Even today, many applications and even libraries are still filled with statements such as these. These streams can be redirected to separate files, which does have some usefulness, but their lack of granularity and "always on" disposition causes logs to fill up quickly and hinders the performance of the application. Ideally, you want something that is easily customizable, can be configured to display a little or a lot of information, can display different information depending on the package or class, and (most important) does not hinder performance.

What Content You Might Want to See in Logs

Given the ideal logging system described previously—customizable, configurable, variable by package or class, and performs well—think about all the information you would want to put in a log file. You can probably think of a number of things, and really the possibilities are technically endless, but here are some basic ideas to get you started:

- **When an application crashes or exits unexpectedly, it is essential to log in as much detail as possible, everything that led up to the crash.** This may include messages, stack traces, thread dumps, and even heap dumps. JVM crashes are not easy events to log. Most of the time the very nature of the crash prevents logging facilities from working. Because of this, most JVMs have tools built in to note when these events occur. For example, the Oracle and OpenJDK JVMs have the command line option `-XX:+HeapDumpOnOutOfMemoryError` that makes Java generate a heap dump when the JVM runs out of memory. Consult your JVM manufacturer's documentation for more information.
- **When errors occur, you should log any information you have about that error (type, message, stack trace, and what the application was doing when the error occurred).** Then, with all that important data stored in a log somewhere, you can display to users only a minimal error message telling them that something went wrong without divulging details that could compromise your system (such as file paths or SQL queries).
- **Sometimes issues arise that aren't quite errors and don't stop an action from continuing but should probably be brought to somebody's attention at a later time.** These issues indicate a potential, but not definite, problem with your application. You should log these types of situations, including a detailed message and what the application was doing. Sometimes, a stack trace is also applicable to these situations. You might consider these "warnings."
- **There are certain important events that you might want to see in your logs.** Logging things such as the creation of entities, the starting of components, a successful user login, or the execution of tasks help you know that your system is working properly.
- **When you want to track down a problem, you want to see detailed information about the execution of methods and routines, often including the value of applicable variables.** However, you probably don't want to see this data all the time, and creating that much data could hinder performance.
- **In the toughest debugging scenarios, you may need to see when any method is entered or exited, as well as the parameter values and return value for each method.** In any loops that are executing, you might require data from each iteration. This, of course, is a ton of information in a high-load scenario, so you definitely don't want it on all the time, and probably not for all methods, even when you do turn it on. In fact, sometimes you may even want to limit this type of logging detail to a specific user or other criteria.
- **Often you need to audit activities in your application.** If your field is security, healthcare, or legal, the law sometimes mandates keeping a record of who did what and when! At some point, you will likely need a mechanism for keeping track of these things. Make no mistake: This is just another form of logging. There may be unique features to this type of logging, and it may have to be separate from any other logging, but it is still just logging.

This is by no means an exhaustive list. The needs of every developer and project are different, and it's likely you have already thought of something not on this list. However, these key concepts are common to almost all applications, and you should keep them in mind both when writing your application and when designing your logging system.

How Logs Are Written

An important thing to keep in mind is that a “log” is not necessarily a file in and of itself. Often when you think of logging, you naturally think of a file. After all, at some point your log contents will probably end up in a file somewhere. But this does not mean that your log contents are written directly to a flat file to later be read manually in eye-bleeding sessions. (Although you could certainly do that if you prefer.) Logs can be written any number of ways to any number of media. Sometimes, logs are written to multiple locations simultaneously, or written to a primary media falling over to a backup media if the primary media fails. Sometimes logs are written asynchronously so that your application can continue processing while the logged information is queued waiting to be written. In some circumstances, logging must even be transaction-safe so that the action stops if logging fails. All these things, and your application’s business needs, affect how logs are written.

Generally speaking, there are some common mechanisms that people use for storing application logs. You may want to use one or more of these, or come up with something on your own.

The Console

A time-tested mechanism, console logging is effectively equivalent to using `System.out.println()`. Although you can gain many advantages using a logging system, the output of the log is the same place: `stdout`. The thing about console logging is that, unless the process’s standard output redirects using operating system facilities (which often happens), the contents of the console are gone after the application exits or the data falls off the end of the console buffer. However, console logging can be extraordinarily useful during development cycles and when other mechanisms fail. It is nearly always configured as at least a failover logging mechanism for most applications.

Flat Files

Another traditional logging method, flat files are perhaps the easiest media to write logs to and the most difficult media to read logs from. The primary advantage flat files have over console logging is that they are persistent by nature (until you delete them), but that’s about where the advantages end.

Typically, flat-file logging consists of a single line in the file per log entry, but because log messages sometimes take up multiple lines (like stack traces), it can be difficult to apply this pattern programmatically. Standard flat-file logs are decently easy to read with the human eye, but nearly impossible to filter or sift through. There are some variations of flat-file logging that, in some cases, improve on this problem slightly:

- **XML logging persists each log event using a standardized XML syntax.** The primary advantage of this is that it’s both human readable AND machine readable, meaning programs can be written to read log contents and display them programmatically with filtering capabilities. However, XML is an extremely bloated format and can often triple or quadruple the size of your logs (or worse!).
- **JSON logging is similar to XML except that it stores log contents in the JavaScript Object Notation format.** This means much less bloat than XML, but on the other hand makes it slightly harder for humans to read.

- On Linux operating systems, **syslog** is a facility that all processes can write to. The specifics of this are outside the scope of this book, but essentially log events are written to flat files managed by the operating system logging tool and stored in a central location in a standardized format. Windows offers a similar functionality through the Windows Event Log, and that data is stored in XML and comes with a log viewer built in to the operating system.

The concept of rolling files can be applied to all these patterns (and, in the case of syslog and Event Log, is applied automatically). When a logging system is set up to use rolling files, it periodically changes the file it is logging to. Sometimes this rolling is time-based — Tomcat, for example, gets a fresh set of log files every day — whereas other times, when a log file reaches a certain size, log files get backed up to an indexed filename and a new log file starts.

Sockets

A less common method for logging, sockets are nonetheless quite useful in some situations. Log events are translated into some network-communicable form and then sent over the network to some other location. Sometimes the receiving application is on the same machine; sometimes it is on a different machine on the same network; and sometimes it is on a different network (even the other side of the world). In nearly all cases, the other end of the socket connection is a dedicated logging server whose sole purpose is receiving and persisting logs in some way.

SMTP and SMS

Although sending an e-mail for logging events may sound strange, think about how badly you want to know when an error occurs in your application. E-mail and SMS log notifications are rarely sent for typical run-of-the-mill logging events. In every case that this author has seen, e-mail and SMS were reserved for severe application errors only, such as exceptions that caused the abortion of a system action or caused a component to fail to start. This notifies the system administrator that a problem needs to be looked at right away.

Databases

It is very common today for applications to log to a database of some type and for obvious reasons. Databases are efficient, transaction safe means of storage that are extremely easy to query and filter data from. If you are logging audit data, chances are you will have to use a database to store those logs. The disadvantages to logging to a database may also be obvious. The first is speed: Writing data to an indexed table with the overhead of the network stack and a database is much slower (on a small scale) than writing to a flat file. But then there's the issue of database size: Many databases have limits on how much data they can hold and, even when those limits are lifted, performance suffers drastically. Inserts can take large fractions of seconds or more. For many, this is an unreasonable price to pay for logging that can be easily filtered.

Enter NoSQL databases, which lack many of the problems that relational databases present to logging systems. These databases store data in a nonrelational manner, which removes much of the overhead and enables well-performing storage of many hundreds of gigabytes or even terabytes of data. One of the most popular databases to use for such a task is MongoDB, a NoSQL document database. This database stores data in the Binary JSON (BSON) format, which is an extremely compressed and efficient means of storage.

MongoDB sacrifices read performance — it can be quite slow when filtering large datasets — and exchanges it for write performance. Inserting data into MongoDB can be orders of magnitude faster than popular relational database systems when database size exceeds several gigabytes. This is a huge plus for logging systems, which you want to impact application performance as little as possible. MongoDB, like most other document databases, is not fully ACID-compliant, but if you structure your documents correctly, you can reduce the chances of losing data to about the same as with an ACID RDBMS.

USING LOGGING LEVELS AND CATEGORIES

In the previous section, you explored some of the concepts of logging, including an overview of the various types of information you might want to see in your application logs. As you explored this topic, it probably became clear to you very quickly that there are different types or severity of logging events, and that not all types are logged all the time. This is commonly known as the concept of logging *levels*. Sometimes, levels are not enough. Often you need the finest logging details from only specific parts of your application, not from the entire thing. If you could just *categorize* your logging events, your logging data would become even more useful. In this section, you investigate logging levels and categories. You also look at the concept of log sifting and how it relates to and differs from log categorization.

Why Are There Different Logging Levels?

Chances are you have already answered this question. So far in this chapter, you've identified a lot of information that should be included in logs; some of it indicative of serious errors, and other types represent extremely fine and numerous debugging detail. If you had all this detail turned on all the time, your logs could grow by megabytes per second. (Trust me, I've seen it happen.) Not only would this seriously hinder the performance of the application, but also your logs would become essentially useless. When you have multiple megabytes of log data to sort through for a single second of application execution, you might as well not have any logs.

Levels indicate the severity or relative importance of a logged event. The word "relative" is used here because that's exactly what it is: One logging level is either more or less important than another logging level. The name of a logging level itself does not define the actual importance of the message it conveys; it just defines the importance of the message as compared to some other message. In some systems, logging levels are merely integers and nothing more. In other systems, logging levels are assigned a name to provide some semblance of semantic usefulness. There is no agreed-upon standard for the number, order, or name of logging levels. Nearly every system you use can define these differently, and it's up to you to determine what works best for you.

Logging Levels Defined

Although there is not an agreed-upon standard, some common patterns appear in most logging systems written for Java applications. Logging levels are typically named to give them some high-level meaning, and in most systems anywhere from six to ten distinct levels exist. Table 11-1 lists the most common types in order from most severe to most trivial and the equivalent constant as defined in `java.util.logging.Level`.

TABLE 11-1: Common Logging Levels

GENERIC NAME	LEVEL CONSTANT	SEMANTIC MEANING
Fatal Error / Crash	No equivalent	Indicates an error of the severest form. Usually these errors result in a crash or at least premature termination of the application.
Error	SEVERE	Indicates that a serious problem has occurred
Warning	WARNING	Indicates that some event has occurred that may or may not be a problem and probably needs to be looked at
Information	INFO	As the name implies, this indicates informational items that could be useful for application monitoring and debugging.
Configuration Details	CONFIG	Events with this level typically contain details about configuration information. You often see this at application or component startup.
Debug	FINE	Indicates debugging information and often includes the values of variables
Trace Level 1 Trace Level 2	FINER FINEST	These levels represent different levels of application tracing. Many logging frameworks define only a single trace level. One possible way to use these levels separately would be to utilize <code>FINER</code> for logging executed SQL statements and <code>FINEST</code> for logging entry into and exiting from method calls.

It's important to note that these examples are based off of one specific logging system: `java.util.logging` built in to the Java SE APIs. But there are many different logging systems both inside and outside of Java programming, and each defines its own slightly different set of levels. Some systems (such as `java.util.logging`) enable you to extend and define more levels. Other systems do not.

As an example of a different set of levels, consider the logging levels available in the Apache HTTPD 2.2 server logs: `emerg`, `alert`, `crit`, `error`, `warn`, `notice`, `info`, and `debug`. (Version 2.4 adds 8 more levels: `trace1` through `trace8`.) The disproportionate number of levels for unusual activity as opposed to debugging highlights a fundamental difference in this system: It was considered more important here to differentiate many different types of errors than many different types of debug information. (Though, that has changed in later versions.) Likewise, your needs may vary based on the use case for your application.

How Logging Categories Work

The concept of a logging category is slightly more abstract than logging level. In nearly all cases in Java (including all of the examples you see in this chapter), logging categories are represented by named logger instances, and each logger can have a different level assigned to it. Through this

pattern, two different classes could have two different loggers, and you could set one to log trace data and the other to log warnings only. In fact, that's exactly how categories are used in most cases: At development time, each class gets its own logger, usually named after the fully qualified class name. Depending on the logging system, a logger hierarchy is usually established so that loggers with undefined level assignments inherit the level from some parent logger.

How Log Sifting Works

Log sifting is conceptually similar to log categorization but serves a slightly different purpose. Using log sifting, different types or originations of events get logged to different locations (different files, different database tables or documents, and so on). In some ways this is the same idea as categories, except that categories typically define a different logging level as opposed to a different logging destination. Many logging frameworks enable you to sift logs by actually using the category, thus knocking out two problems with one feature. Others provide you with the ability to sift using categories *and* other attributes of a log event (usually determined in a filter of some type). In Java, all major logging frameworks permit sifting based on at least the logging category, if not more attributes of each log event. You explore the topic of sifting more in both of the next two sections.

CHOOSING A LOGGING FRAMEWORK

After you understand the logging needs of your application, your next task is to choose a logging framework. This might not mean using a third-party framework—it might mean creating your own. However, in almost all cases you can find existing, industry-tested logging frameworks that you can configure to meet your needs, which means creating your own is simply an unnecessary task. In this section you examine two important guidelines to keep in mind and then take a look at some existing logging frameworks.

API versus Implementation

Consider this scenario: You've written a large, enterprise application with many thousands of classes. Most of these classes (the ones that aren't POJOs) use logging. After careful research, you chose and integrate an industry-standard logging system into your application. A week after going live, your system's engineering team informs you that the syslog output you've chosen is not adequate, and that it needs database logging. Unfortunately, the logging system you've chosen doesn't support database logging. What can you do? You could extend the logging system, but there's a perfectly good logging system out there that already supports database logging, and you'd rather use that. However, if you did that you'd need to change thousands of classes to use a different logging system. The effort could take weeks and cost your company tens of thousands of dollars.

The underlying problem in this scenario is that the logging API you are using is tightly coupled with an underlying implementation. You can't swap out the implementation without changing everywhere in your code where logging is used. You could have avoided this problem if you had simply written a simple logging API to hide the underlying framework from your uses of logging in the code. Then, all you'd have had to do to swap out the framework is change a handful of classes in your API; the rest of your application could continue using your API for logging as it always had. Thankfully, this

is not a new concept in the world of logging frameworks, and chances are you won't even have to write the API yourself.

The standard `java.util.logging` framework built in to the Java platform is one example of the separation of logging API from implementation. The `java.util.logging.LogManager` class is responsible for creating and returning `Logger` instances when requested and delegates to a default implementation. Developers can extend `LogManager` and specify a system property to provide an alternative implementation of the standard logging API.

The standard implementation has handlers that can write to the files, streams, sockets, console, and even memory. You might wonder, then, why you can't just use the standard logging facility. The simple answer is, "You can." In many cases and for many applications, the built-in logging system is completely adequate. However, it has a key drawback that makes it downright difficult if not impossible to use in web applications: It loads its configuration from a system property instead of the classpath. Because of this, two web applications deployed in the same container instance cannot have different logging configurations unless the container has extended the base logging implementation. Tomcat does do this, but not all containers do, so relying on this isn't portable. Instead, it's better to find some other solution that combines separate APIs and implementations.

Performance

Obviously, performance is a huge consideration no matter what part of an application you work on, and logging is no different. Think for just a second, however, about the process of writing to files, sockets, or databases, or sending e-mails. These tasks spend considerable time blocking while input/output operations complete, and blocking is a time-intensive situation. There is simply no way around it: Logging messages takes time, and the vast majority of that time is out of your hands and in that of the operating system — meaning you can't speed it up through software alone.

But, really, if you have a message that you feel is important enough to log, chances are you won't care that it adds a few milliseconds to an activity. So why does performance matter? The key is it doesn't matter (much) when you *are* logging; it matters (drastically) when you *are not* logging. You are likely to fill up your code with trace, debug, and informational statements that you won't want to see most of the time. You essentially want these operations to do absolutely nothing unless you have that level of detail enabled.

The following are the critical situations in which performance can make or break a logging framework:

- Calling a debug method when debug logging is disabled shouldn't add *milliseconds* to an action — it should add *nanoseconds or less*.
- With a well-performing logging system, you should fill your code to the brim with logging statements, turn all logging off, and notice no perceivable difference in application response time. This is the key performance indicator that you should look for when choosing a logging system.

Sure, good performance when you turn on logging is admirable and desirable, but it is meaningless if performance is bad when logging is turned off.

A Quick Look at Apache Commons Logging and SLF4J

Apache Commons Logging (<http://commons.apache.org/proper/commons-logging/>) and Simple Logging Façade for Java (<http://www.slf4j.org/>) are both Open Source, ultra-thin logging APIs. Neither API contains any implementation code. Instead, developers are expected to pick a logging framework to sit under the API. In a Maven project, the Commons Logging or SLF4J APIs are often given `compile` scope, whereas the implementation is given `runtime` scope, thereby completely preventing the application from using the underlying implementation directly.

Though both APIs have several classes, there are only two classes in each API that you will ever use directly. In Commons Logging, the `LogFactory` class has two static `getLog` methods that return named `Logs`. The `Log` class has methods for logging debug, info, warning, error, and other messages. SLF4J is very similar. The static `getLogger` methods on its `LoggerFactory` class return `Loggers`, which have methods for logging various message levels. The SLF4J API is a bit more flexible because its `Logger` class supports markers, format-syntax `Strings`, and arguments for each logging method. You learn more about this later in this section.

Both of these APIs are heavily used in libraries and are the two most popular for this purpose. Application code is not the only code that needs logging. Often libraries that applications use, especially complex libraries such as Spring Framework or Hibernate, need to log messages as well. This logging helps you, the consumer of the library, understand what is happening and track down problems when things go wrong. However, because libraries do not run but instead are coded against, they have no way of knowing (nor do they care) how logging actually takes place.

Enter the Commons Logging and SLF4J APIs. Both logging APIs ship with absolutely no implementation; instead, they come with adapters to multiple logging frameworks. At run time, they discover the logging framework in use (by convention or configuration) and activate the appropriate adapter to translate logging events from the API to the implementation. Commons Logging provides adapters for Avalon, `java.util.logging`, Log4j 1.2, and LogKit logging. SLF4J provides adapters for (ironically) Commons Logging, `java.util.logging`, and Log4j 1.2 logging. When the APIs do not provide adapters to support a particular logging framework, often the logging frameworks provide adapters. Logback, for example, is a popular implementation to use with SLF4J and provides the SLF4J adapter binding. In the worst-case scenario, both APIs make writing custom adapters easy.

For years, Log4j was the most popular logging framework to use either standalone or underneath Commons Logging or SLF4J. (It was most commonly used with Commons Logging.) However, Log4j had some drawbacks, and in particular suffered from a very narrow interface (no markers, no message formatting, no message arguments) and performance issues. Dissatisfaction with various APIs and frameworks led to the creation of SLF4J (an improvement on Commons Logging), and later Logback (an improvement on Log4j). Logback boasted an extremely useful interface, a strongly performing implementation, and a battery of tests to demonstrate its stability and performance. It didn't take long for work to begin on Log4j 2, the successor to Log4j whose goal was to improve the interface and performance of the original Log4j framework.

Introducing Log4j 2

Log4j 2 (<http://logging.apache.org/log4j/2.x/>), which was released in February 2014, is a vast improvement over Log4j. It includes significant performance improvements, a greatly expanded interface, and plenty of tests to back it all up. In addition, the API and implementation have been

completely separated. The API specifies an interface for developers to log to and also provides hooks for creating an underlying implementation.

Of course, there's also the default implementation, but developers of applications and other logging frameworks are free to use the API hooks to adapt the API to sit on top of other logging implementations. Because there are so many logging frameworks out there and Log4j 2 is still very new, you should still use Commons Logging or SLF4J as your logging API when writing a new *library*. This is because you have no way of knowing what logging frameworks consumers of your library will use, and there is a better chance those logging frameworks will have adapters for Commons Logging or SLF4J than for Log4j 2. However, when writing an application you call the shots. Writing directly against the Log4j 2 API provides more flexibility and could also help performance somewhat, depending on your use of it, and you won't lose the ability to swap out the underlying implementation if you ever need to.

NOTE *Log4j 2 includes several performance metrics on its website. The most important metric—performance when logging is turned off—is impressive: on midline hardware, 3 nanoseconds for calls to `isDebugEnabled`, and 4 nanoseconds for calls to `debug` when debug is turned off. That's 4 billionths of a second sacrificed to put helpful message logging in your application. To put this in perspective, with all debug logging disabled, your application would have to execute 250,000 debug methods to add 1 millisecond in execution time, or 250,000,000 debug methods per second to double execution time.*

All this performance is actually roughly the same as Logback performance. Where Log4j 2 really excels is with log filters. In a multithreaded application, tests show Log4j 2 beats Logback by an order of magnitude or more when filter processing is active.

For the rest of this book, you will log against the Log4j 2 API with the Log4j 2 core implementation underneath it. (In a real-world scenario, you should evaluate the needs of your project and determine which API and implementation work best for you.) You have undoubtedly noticed that this chapter has added five Log4j 2 Maven dependencies. Here is an explanation of each:

- `log4j-api` provides the API for logging. This is the only Log4j dependency that is in `compile` scope in your application because it contains the only classes you should code against.
- `log4j-core` contains the standard Log4j 2 implementation. As you explore Log4j 2 configuration throughout the rest of this chapter, it is the implementation you are configuring, not the API. The API requires no configuration.
- `log4j-jcl` is an adapter to support the Commons Logging API. Several libraries you use in the rest of this book log against the Commons Logging API, and this adapter causes Commons Logging to use Log4j 2 as its implementation.
- `log4j-slf4j-impl` is an SLF4J implementation adapter. Several libraries used throughout the rest of this book log against the SLF4J API, and this adapter causes SLF4J to use Log4j 2 as its implementation.

- `log4j-taglib` is an adapter that includes a JSP Tag Library for logging within your JSPs. Like the previous three dependencies, this dependency is in `runtime` scope for the purposes of writing your application code because you do not need to compile against it. However, if you were to configure a build to compile your JSPs, you would need this dependency to have `compile` scope for the JSP compilation process.

The Log4j 2 implementation has several key concepts that you should understand. Although the API exposes some of these concepts in a generic sense (this is noted where applicable), it is the integration with the implementation that makes the features possible.

WHY NOT USE COMMONS LOGGING OR SLF4J?

At this point, you may be thinking back to the scenario presented earlier where you need to quickly swap out your logging implementation. While the Log4j 2 API certainly makes this *possible*, it does not make it *easy*. The API is, undoubtedly, written with the implementation in mind. So why don't you use Commons Logging or SLF4J in front of Log4j 2?

Neither the Commons Logging API nor the SLF4J API is as complete and feature-rich as the Log4j 2 API. Choosing Commons Logging or SLF4J means giving up some features, such as fatal logging, any-level logging, custom level logging, easy method entry and exit logging, and message formatting with arguments. Using a logging façade is a much more important thing to do in an independent library than in an application. If you are willing to sacrifice some Log4j 2 features, you can certainly choose SLF4J or Commons Logging over the Log4j 2 API. On the other hand, if you want to utilize all Log4j 2 features and also use a logging façade, you could create your own façade to match the Log4j 2 API. That choice is left up to you. For this book, you will use the Log4j 2 API directly.

Configuration

The Log4j 2 implementation is completely self-configuring. At the most, the only thing you must do to use Log4j 2 is place logging statements in your code. By default, Log4j 2 can configure itself to log errors and higher and to write log messages to the console. It does this only if all the following steps have failed to locate an explicit configuration:

1. Inspect the `log4j.configurationFile` system property, and load the configuration from that file if it exists.
2. Look for a file named either `log4j2-test.json` or `log4j2-test.json` on the classpath, and load the configuration from that file if it exists.
3. Look for a file named `log4j2-test.xml` on the classpath and load the configuration from that file if it exists.
4. Look for a file named either `log4j2.json` or `log4j2.json` on the classpath, and load the configuration from that file if it exists.
5. Look for a file named `log4j2.xml` on the classpath, and load the configuration from that file if it exists.

The purpose of the separate `-test` files having higher priority is simple: This covers the cases in which unit (or other) tests are executing, and both files are on the classpath. In the next section you explore Log4j 2 configuration by integrating it into a web application.

Levels

In the `org.apache.logging.log4j.Level` class, Log4j 2 defines six logging levels and two special-meaning levels, listed in order of their priority (most severe to least severe): `OFF`, `FATAL`, `ERROR`, `WARN`, `INFO`, `DEBUG`, `TRACE`, `ALL`. Note that `OFF` and `ALL` are not levels you would actually log against; instead, you would assign these levels to a particular `org.apache.logging.log4j.Logger` or `org.apache.logging.log4j.core.Appender`, `OFF` meaning “do not log anything” and `ALL` meaning “log everything.” `Level` is, by necessity, exposed through the API, but how it is used to control the logging of messages is configured through the implementation.

Originally, `Level` was an enum, meaning it could not be extended. Shortly before Log4j 2 was released, `Level` was changed from an enum to a final class with a private constructor and a factory method `forName` (`String name, int intLevel`). This method returns the existing `Level` if it exists and creates a new `Level` if it does not. The enum `org.apache.logging.log4j.spi.StandardLevel` helps map custom `Level`s you create to standard levels when logging to bridge APIs like SLF4J. There is always exactly one instance of a `Level` of any given name. This means you can easily create your own custom `Level`s using the following technique and you can compare a value to a `Level` using the `==` operator instead of `equals`.

```
public final class CustomLevels
{
    public static final Level CONFIG = Level.forName("CONFIG", 350);
    public static final Level NOTICE = Level.forName("NOTICE", 450);
    public static final Level DIAG      = Level.forName("DIAG", 550);
}
```

Loggers

Loggers and categories are synonymous in Log4j 2, and you must deal with the `Logger` class in every class you log in. You obtain a `Logger` by calling one of the `getLogger` methods of `org.apache.logging.log4j.LogManager`. When you have a `Logger` instance you can log errors, warnings, and other messages. Loggers have names in Log4j 2, just like they do in most APIs and implementations. The `Logger` name defines the logging category, and by convention that name is the fully qualified class name of the class using the `Logger` (each class has its own `Logger` instance).

Multiple calls to `getLogger` with the same name or class result in the exact same instance of `Logger`, not in multiple instances with the same name (in other words, Loggers are cached). Any two Loggers can have different `Level` settings and be assigned to zero or more Appenders. Logger names in Log4j 2 follow a dot-separated hierarchy, with more specific Loggers inheriting the `Level` and Appenders of ancestor Loggers. This is demonstrated in Table 11-2. This table deals only with Levels, but the inheritance rules are essentially the same for Appenders. The Assigned Level column represents Levels that you might assign in your Log4j configuration.

TABLE 11-2: Inheritance Hierarchy for Log4j 2 Loggers

LOGGER	ASSIGNED LEVEL	EFFECTIVE LEVEL	LEVEL INHERITED FROM
root (special)	WARN	WARN	n/a
com	—	WARN	root
com.wrox	INFO	INFO	—
com.wrox.chat	DEBUG	DEBUG	—
com.wrox.shop	—	INFO	com.wrox
com.example.test	—	WARN	root

Appenders

Appenders are responsible for actually writing log contents to their destination. Appenders have essentially the same inheritance rules as Levels when assigned to Loggers, with the exception that the inheritance also has an additivity property that determines whether an Appender adds to or overrides other inherited Appenders.

For example, if the root logger is assigned to a console Appender and com.wrox is assigned to a file Appender, log messages written to com.wrox, com.wrox.chat, and others go to *both* the console and the file, whereas log messages written to root or com.example.test go only to the console. However, if you set the additivity property to false for com.wrox, messages written to com.wrox, com.wrox.chat, and others go only to the file. (They stop at that Appender.) If com.wrox's additivity property is still false and com.wrox.shop has a syslog Appender with additivity set to true (default), messages written to com.wrox.shop go to syslog and the file, but not to the console.

Layouts

Appenders often use Layouts to determine how to format their output. The most common type of Layout is a pattern-based Layout that defines a series of tokens replaced at write-time with data from a logging message. There are also HTML, XML, syslog, and serialized layouts, to name a few examples. You explore the PatternLayout more in the next section.

Filters

Log4j 2 Filters, not to be confused with Servlet filters, provide a mechanism whereby log messages can be examined to determine if or how they should be written. The result of a Filter evaluation is either ACCEPT, DENY, or NEUTRAL, just like with network firewalls. ACCEPT indicates that the message should be written and all other filters should be ignored. If a Filter evaluation results in ACCEPT, even the message Level is ignored, and the message is logged even if its Level is not high enough. DENY is the opposite: This means that the message is immediately rejected, and the following filters do not get the opportunity to evaluate the message. NEUTRAL indicates that the Filter neither accepts nor denies the message, and other filters may evaluate it further.

In a Log4j 2 configuration, `Filters` can be attached to four different stages of the architecture: the context configuration, the `Logger` configuration, the `Appender` reference, or the `Appender` configuration. When a message is logged, its first stop is any context-wide `Filters` in the order they were declared. This happens before the `Level` is evaluated. Assuming the context `Filters` say `NEUTRAL`, the message then goes to have its `Level` evaluated. If it passes the `Level` test, its next stop is the `Logger` configuration `Filters`. After this come the `Appender` reference `Filters`, which determine whether the `Logger` should route the message to a particular `Appender`. (This can be used to sift messages into different files or database tables, for example.) Finally, `Filters` on individual `Appenders` determine whether the `Appender` should write the message.

`Filters` can act on all sorts of information. A common pattern is to include `Marker` objects with logging messages and have `Filters` examine those `Marker` objects. (This is especially useful for sifting.) `Filters` can also look at the contents of a message, the message type, exceptions attached to a message, and data stored in the currently running `Thread`, just to name a few things. For that matter, `Filters` can use something as unrelated to the current message as the system time to determine how to act on a message. You'll see an example for configuring a `Filter` in the next section.

INTEGRATING LOGGING INTO YOUR APPLICATION

Using and configuring Log4j is a fairly simple task. As described in the last section, you can get away with as little as simply putting logging statements in your application:

```
private static final Logger log = LogManager.getLogger();  
  
    @Override  
    protected void doGet(HttpServletRequest request, HttpServletResponse response)  
        throws ServletException, IOException  
    {  
        if(request.getParameter("action") == null)  
            log.error("No action specified.");  
    }
```

Log4j 2 can configure itself with the default configuration and start writing logging messages to the console. Of course, this default configuration is likely not sufficient for you. Chances are you're going to need more than just errors logged (you'll probably want warnings and possibly informational messages), and you'll likely want the logs to go somewhere other than the console. You can easily achieve both with a configuration file.

NOTE *If you are used to configuring previous versions of Log4j, you should keep two important things in mind. First, Log4j 2 does not support configuration through Java properties files anymore. You must use either XML or JSON. Second, Log4j 2's XML schema is not the same as Log4j's XML schema. If you have existing log4j.xml files in any of your applications, you cannot simply rename those files to log4j2.xml. You must adopt the new configuration schema.*

Creating the Log4j 2 Configuration Files

Remember that Log4j 2 looks for two different configuration files (`log4j2` and `log4j2-test`), and that it supports three different extensions for those files (`.xml`, `.json`, and `.jsn`). The `.json` and `.jsn` extensions are both for JSON-format configurations. JSON configurations are much more easily read and written with code than XML configurations, so if you want to programmatically create Log4j 2 configurations, JSON is probably the way to go. If you're creating your configuration by hand, XML is an easier choice, so that's what you use here. You can follow along in the Logging-Integration project available for download on the wrox.com code download site.

The project is simple, containing a single servlet with some methods that execute correctly and others that do not. The first thing you must do is create the appropriate configuration files. When you create the standard configuration file, this also affects how logging behaves when unit tests run. Because the standard configuration file instructs Log4j 2 to log to a file and you probably don't want that when running unit tests, you should first create a `log4j2-test.xml` file in the `resources` directory of your project's `test` folder:

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration status="WARN">
    <appenders>
        <Console name="Console" target="SYSTEM_OUT">
            <PatternLayout
                pattern="%d{HH:mm:ss.SSS} [%t] %-5level %logger{36} - %msg%n"/>
        </Console>
    </appenders>
    <loggers>
        <root level="debug">
            <appender-ref ref="Console"/>
        </root>
    </loggers>
</configuration>
```

NOTE *There is no official XML schema for the Log4j configuration files. The XML that you can use is flexible and also varies depending on which extensions and plugins are on the classpath, thus the configuration file cannot be strictly validated against a schema. You can learn more about which XML elements and attributes are valid in the Log4j Configuration Documentation.*

This configuration file is nearly identical to the implicit default configuration. The only differences here are that the `configuration status` `Logger Level` changed from `OFF` to `WARN` and the `root` `Logger Level` changed from `ERROR` to `DEBUG`, both of which are more appropriate for testing situations.

You are already familiar with the `root` `Logger` (the ancestor of all `Loggers` in Log4j 2), but you may wonder what the `configuration status` `Logger` is. Log4j, too, needs to log messages when things aren't quite working right. It does this through a special `Logger` called the `StatusLogger`. This `Logger`'s sole purpose is to log events occurring within the logging system itself. As an example, say

you create a `socket Appender` that cannot connect to its destination server. In this case it would log this failure using the `StatusLogger`. The default setting for the `StatusLogger`, `OFF`, suppresses all these messages from the logging system. Here, that `Level` has been changed to `WARN` (by changing the `<configuration>` element's `status` attribute).

Now that you have an adequate test configuration, create a `log4j2.xml` file in the `resources` directory of your project's production source folder:

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration status="WARN">
    <appenders>
        <Console name="Console" target="SYSTEM_OUT">
            <PatternLayout>
                pattern="%d{HH:mm:ss.SSS} [%t] %-5level %logger{36} - %msg%n"
            </PatternLayout>
        <RollingFile name="WroxFileAppender" fileName="../logs/application.log"
            filePattern="../logs/application-%d{MM-dd-yyyy}-%i.log">
            <PatternLayout>
                pattern>%d{HH:mm:ss.SSS} [%t] %X{id} %X{username} %-5level
                %c{36} %l: %msg%n
            </PatternLayout>
            <Policies>
                <SizeBasedTriggeringPolicy size="10 MB" />
            </Policies>
            <DefaultRolloverStrategy min="1" max="4" />
        </RollingFile>
    </appenders>
    <loggers>
        <root level="warn">
            <appender-ref ref="Console" />
        </root>
        <logger name="com.wrox" level="info" additivity="false">
            <appender-ref ref="WroxFileAppender" />
            <appender-ref ref="Console">
                <MarkerFilter marker="WROX_CONSOLE" onMatch="NEUTRAL"
                    onMismatch="DENY" />
            </appender-ref>
        </logger>
        <logger name="org.apache" level="info">
            <appender-ref ref="WroxFileAppender" />
        </logger>
    </loggers>
</configuration>
```

There's some new and interesting stuff here. First, the rolling file `Appender` is configured to write to `application.log` in the Tomcat `logs` directory. (This assumes that your IDE starts Tomcat from the `bin` directory, which is typical; in a real-world scenario, you want to make these paths configurable.) Although Log4j 2 has a simpler file `Appender` that writes to a file, this `Appender` can roll the log in one or more scenarios, such as the log reaching a certain size, the date changing, the application starting, or any combination of those three. In this case, the `Appender` is configured to roll the log file every time it reaches 10 megabytes and to keep no more than four backed up logs.

NOTE *If, when running the application, you can't find the application.log file in Tomcat's logs directory, try changing the relative path to an absolute path of a directory on your system and then restart the application.*

The `PatternLayout` has many patterns that can log event information, and you can learn about them all (and other layouts) in the Log4j Layout Documentation. In this case, the file `Appender` pattern has substituted `%c` for `%logger` (the former is shorthand for the latter) and added `%l`, which prints the class, method, file, and line number where the logging message occurred. It has also added `%X{id}` (a fish tag) and `%X{username}`, which are properties on the `ThreadContext` — you learn about these next. You'll also notice that the pattern for the file `Appender`, in addition to actually being a different pattern, is configured using a different XML format than the console `Appender`. You can specify properties of `Appenders`, `Filters`, `Loggers`, and so on as tag attributes *or* as nested tags. The two approaches are interchangeable.

Finally, the new `Logger` configurations in this file say that all `Loggers` in the `com.wrox` and `org.apache` hierarchies have the level of `INFO` and that the `Loggers` in `com.wrox` are not additive — they do not inherit the console `Appender` and only log to the file `Appender`. However, notice the `com.wrox <appender-ref>` element for the console `Appender` has a nested `<MarkerFilter>` element with it. This filter says that `Loggers` in the `com.wrox` hierarchy *can* log to the console `Appender`, but only for events containing a `Marker` named `WROX_CONSOLE`. You can learn about the various `Filters` available in the Log4j Filter Documentation. The following incomplete configuration shows all of the valid locations for `Filter` configuration elements (shown in bold):

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
  ...
  <FilterName ... /><!-- This is a context-wide filter -->
  <appenders>
    <AppenderName name="someAppender">
      <FilterName ... /><!-- This is an appender filter -->
      ...
    </AppenderName>
    ...
  </appenders>
  <loggers>
    <logger name="someLogger" level="info">
      <FilterName ... /><!-- This is a logger filter -->
      ...
      <appender-ref ref="someAppender">
        <FilterName ... /><!-- This is an appender reference filter -->
        ...
      </appender-ref>
      ...
    </logger>
    ...
  </loggers>
</configuration>
```

This book cannot show you all the possible ways to configure Log4j, as there are many. If you want more information, a good place to start is the Log4j Manual.

Utilizing Fish Tagging with a Web Filter

When using any logging framework, you should *fish tag* requests so that you can group together logging messages belonging to the same request and analyze them. The `org.apache.logging.log4j.ThreadContext` stores properties in the current thread until the `ThreadContext` is cleared. Any events logged in the same thread between when a property is added to the `ThreadContext` and when it is removed can be associated with that property. If there are many concurrent web requests executing, unique fish tags for each request enable you to identify all the messages logged during that particular request.

A fish tag is typically something strongly unique, such as a UUID. The `ThreadContext` can store anything else useful for distinguishing logging events, such as the username of the user who is logged in. The following `LoggingFilter` adds the fish tag (`id`) and session username (`username`) to the `ThreadContext` at the beginning of the request and clears the `ThreadContext` as the request completes. The pattern discussed previously then prints these properties with `%X{id}` and `%X{username}`. Because the filter supports multiple dispatcher types and may execute multiple times in a single request, it only sets the fish tag and username properties if they have not already been set, and it only clears the `ThreadContext` for the same invocation that set the fish tag and username properties:

```
@WebFilter(urlPatterns = "/*", dispatcherTypes = {
    DispatcherType.REQUEST, DispatcherType.ERROR, DispatcherType.FORWARD,
    DispatcherType.INCLUDE, DispatcherType.ASYNC
})
public class LoggingFilter implements Filter {
    @Override
    public void doFilter(ServletRequest request, ServletResponse response,
        FilterChain chain) throws IOException, ServletException {
        boolean clear = false;
        if(!ThreadContext.containsKey("id")) {
            clear = true;
            ThreadContext.put("id", UUID.randomUUID().toString());
            HttpSession session = ((HttpServletRequest)request).getSession(false);
            if(session != null)
                ThreadContext.put("username",
                    (String)session.getAttribute("username"));
        }
        try {
            chain.doFilter(request, response);
        } finally {
            if(clear)
                ThreadContext.clear();
        }
    }
    ...
}
```

WARNING Recall the lessons you learned in Chapter 9 about asynchronous request handling. If you set values on the `ThreadContext` at the beginning of an asynchronous request, those values will be cleared before the request completes. For this reason, you must be very careful whenever you use the `ThreadContext` with asynchronous requests. The Web Applications and JSPs section of the Log4j 2 manual online includes helpful information about using all aspects of Log4j with asynchronous contexts.

Writing Logging Statements in Java Code

Using Log4j 2 `Logger` instances is extremely straightforward. Overloaded methods exist for each of the logging `Levels`, and these enable you to specify `Strings`, `Objects`, or `Messages` as messages, provide one or more parameters to substitute in a `String` message, provide `Throwables` whose stack traces should be logged, and provide `Markers` to flag an event. Take a look at Listing 11-1, which uses a `Logger` to log activity in the `ActionServlet`. A singleton `Logger` is created for all instances of the `Servlet`. Then methods like `info`, `warn`, and `error` write log messages. The special methods `entry` and `exit` log at the `TRACE` level and are shorthand for tracing program execution through method calls and returns. Many more methods are available on this `Logger` interface, and you should read its API documentation to learn more.

NOTE `LogManager`'s no-argument `getLogger` method returns a `Logger` whose name is equal to the fully-qualified class name of the class calling `getLogger`. In the case of Listing 11-1, that name is `com.wrox.ActionServlet`. You could also use the `getLogger` method that accepts a `Class` argument, which would return a `Logger` named after that `Class`. If you want to use something other than class names as `Logger` names, you could use the `getLogger` method that accepts a `String` argument—the name of the `Logger`.

LISTING 11-1: `ActionServlet.java`

```
@WebServlet(name = "actionServlet", urlPatterns = "/files")
public class ActionServlet extends HttpServlet
{
    private static final Logger log = LogManager.getLogger();

    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        String action = request.getParameter("action");
        if(action != null) {
            log.info("Received request with action {}.", action);
            String contents = null;
            switch(action) {
```

continues

LISTING 11-1 (continued)

```

        case "readFoo":
            contents = this.readFile("../foo.bar", true);
            break;
        case "readLicense":
            contents = this.readFile("../LICENSE", false);
            break;
        default:
            contents = "Bad action " + action + " specified.";
            log.warn("Action {} not supported.", action);
    }
    if(contents != null)
        response.getWriter().write(contents);
} else {
    log.error("No action specified.");
    response.getWriter().write("No action specified.");
}
}

protected String readFile(String fileName, boolean deleteWhenDone) {
    log.entry(fileName, deleteWhenDone);
    try {
        byte[] data = Files.readAllBytes(new File(fileName).toPath());
        log.info("Successfully read file {}.", fileName);
        return log.exit(new String(data));
    } catch(IOException e) {
        log.error(MarkerManager.getMarker("WROX_CONSOLE"),
                  "Failed to read file {}.", fileName, e);
        return null;
    }
}
}
}

```

Now compile and start up the application and go to `http://localhost:8080/logging-integration/files` in your browser. The `application.log` file should appear in the Tomcat logs directory and have an error in it. Adding `?action=badAction` to the URL and reloading the log file reveals a new informational message and warning in the file. Changing the action to `readLicense` reads the Tomcat license file back to the browser. Only informational messages appear in the log this time. Finally, change the action to `readFoo` and an error, complete with exception stack trace, appears in the log file. This time the error also appeared in the console (debugger) output because it included the `Marker` named `WROX_CONSOLE`. Notice the `fish` tag for each message logged and how it is the same for multiple messages logged in the same request. You should experiment with the `Log4j` configuration file and change the `level` and `additivity` attributes to see how the data logged changes.

NOTE *The relative file name used to access the Tomcat license file only works if your IDE starts Tomcat from Tomcat's bin directory, which is typical. If your IDE starts Tomcat from some other directory, you may need to change the path to this file to get it to work.*

Using the Log Tag Library in JSPs

As mentioned earlier, Log4j 2 comes with a tag library that enables you to log messages in JSPs without using scripting. This book won't go into a lot of detail on this subject because, generally speaking, there is much less need for logging in the presentation layer. In fact, as long as your JSPs don't have any business logic in them, you'll never need to log in them. With that said, you can include the tag library with the following `taglib` directive:

```
<%@ taglib prefix="log" uri="http://logging.apache.org/log4j/tld/log" %>
```

The Logging-Integration project has a `logging.jsp` file in the web root that demonstrates how to use these logging tags:

```
<log:entry />
<!DOCTYPE html>
<html>
  <head>
    <title>Test Logging</title>
  </head>
  <body>
    <log:info message="JSP body displaying." />
    Messages have been logged.
    <log:info>JSP body complete.</log:info>
  </body>
</html>
<log:exit />
```

Each JSP in your application that uses one or more `log` tags automatically has a unique `Logger` created for it. You can override this `Logger` using the `<log:setLogger>` tag or the `logger` attribute on most of the other tags. Try out the JSP logging by starting the application and going to `http://localhost:8080/logging-integration/logging.jsp` in your browser. Depending on the logging level you have configured, you see different messages appear in the log.

Logging in the Customer Support Application

The Customer-Support-v9 project on the `wrox.com` code download site has had all uses of `System.out`, `System.err`, and `Throwable.printStackTrace()` removed from its code and replaced with Log4j 2 logging. In addition, more logging statements were added throughout the application. Finally, the `LoggingFilter` was copied from the previous example and added to the beginning of the filter chain in the `Configurator` so that all requests are fish tagged.

You should download and review these changes, but they are not detailed here because it's redundant to what you've already learned. The default logging level for all `Loggers` in the `com.wrox` hierarchy is set to `INFO` and the only `INFO` event so far is when a user logs in. Most other logging events are `DEBUG`, `TRACE`, `WARN`, or `ERROR`. Because of this, you won't see much in the `support.log` file when you run the Customer Support application. Rest assured, this will change in Part II when you start using Spring Framework.

SUMMARY

In this chapter you learned the fundamentals of application logging and why logging in your applications is so important. You investigated several common logging patterns and explored the concepts of categories and logging levels. You also learned about the importance of separating a logging API from its underlying implementation, saving you headaches down the road. You were introduced to several popular logging APIs and implementations, such as Apache Commons Logging, SLF4J, Logback, Log4j 1.x, and Log4j 2. Finally, you learned details for using and configuring Log4j 2 and experimented with integrating it into a web application. Throughout the rest of the book, logging is quietly present in every chapter. Each project you work on contains a Log4j 2 configuration and logging statements. This way, you can get into the habit of consistently logging whenever you program.

This wraps up Part I, where you learned about various aspects of web application development using Java SE, Java EE, Servlets, JSPs, filters, WebSockets, application servers, web containers, and more. By now, you should have a firm grasp on the basics, and you should be able to write fairly complex applications. In Part II you start growing your enterprise development skillset as you learn about Spring Framework and how it supplements, enhances, and—in some cases—supplants parts of the Java Platform, Enterprise Edition.