THE EXPERT'S VOICE® IN JAVA

Java EE 5 compatible

Pro JSF and Ajax

Building Rich Internet Components

Jonas Jacobi and John R. Fallows

Foreword by Adam Winer, JSF Expert Group Member

Apress®

Pro JSF and Ajax

Building Rich Internet Components

Jonas Jacobi and John R. Fallows

Apress®

Pro JSF and Ajax: Building Rich Internet Components

Copyright © 2006 by Jonas Jacobi and John R. Fallows

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

ISBN-13: 978-1-59059-580-0

ISBN-10: 1-59059-580-7

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Trademarked names may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, we use the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

Lead Editor: Tony Davis
Technical Reviewers: Peter Lubbers, Kito D. Mann, Matthias Wessendorf
Editorial Board: Steve Anglin, Dan Appleman, Ewan Buckingham, Gary Cornell, Jason Gilmore, Jonathan Hassell, James Huddleston, Chris Mills, Matthew Moodie, Dominic Shakeshaft, Jim Sumser, Matt Wade
Project Managers: Beckie Stones, Elizabeth Seymour
Copy Edit Manager: Nicole LeClerc
Copy Editor: Kim Wimpsett
Assistant Production Director: Kari Brooks-Copony
Production Editor: Laura Cheu
Compositor: Molly Sharp, ContentWorks
Proofreader: Elizabeth Berry
Indexer: Carol Burbo
Artist: Kinetic Publishing Services, LLC
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `http://www.springeronline.com`.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

The information in this book is distributed on an “as is” basis, without warranty. Although every precaution has been taken in the preparation of this work, neither the author(s) nor Apress shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

To the love of my life, Marianne,
and our princesses, Emma and Isabelle,
for keeping my spirit up.
—Jonas Jacobi

To my wife, Nan, for her love, support, and patience,
and our son, Jack, for his natural inspiration.
—John R. Fallows

Contents at a Glance

Foreword xiii
About the Authors xv
About the Technical Reviewers xvii
Acknowledgments xix
Introduction xxi

PART 1 Developing Smarter with JavaServer™ Faces

CHAPTER 1 The Foundation of JSF: Components 3
CHAPTER 2 Defining the Date Field Component 49
CHAPTER 3 Defining the Deck Component 105

PART 2 Designing Rich Internet Components

CHAPTER 4 Using Rich Internet Technologies 173
CHAPTER 5 Loading Resources with Weblets 213
CHAPTER 6 Ajax Enabling the Deck Component 223
CHAPTER 7 Ajax Enabling the Date Field Component 267
CHAPTER 8 Providing Mozilla XUL Renderers 303
CHAPTER 9 Providing Microsoft HTC Renderers 361
CHAPTER 10 Switching RenderKits Dynamically 403

INDEX 413

Contents

Foreword xiii
About the Author xv
About the Technical Reviewer xvii
Acknowledgments xix
Introduction xxi

PART 1 ■■■ **Developing Smarter with JavaServer™ Faces**

■ **CHAPTER 1 The Foundation of JSF: Components** 3

Overview of Application Development Technologies 4
One-Tier 4
Two-Tier: Client-Server 5
Multitier: Web Applications 5
Exploring Application Development Today 5
Frameworks 7
Tapestry, Struts, Tiles, TopLink, Hibernate, ADF UIX... 7
Introducing JSF 8
Application Development with JSF 9
JSF Architecture 10
A Component-Based UI Framework 13
UIComponent 15
Converters, Validators, Events, and Listeners 22
Facets 23
Renderers 24
Renderer Types 25
RenderKits 26
Custom Action Tag Handlers 27
Request-Processing Lifecycle 27
Summary 47

CHAPTER 2 Defining the Date Field Component 49

Requirements for the Date Field Component 49
The Input Date Component 51
Designing the Input Date Component Using a Blueprint 52
Step 1: Creating a UI Prototype 53
Step 2: Creating a Client-Specific Renderer 55
Step 3: Creating a Renderer-Specific Subclass 77
Step 4: Registering UIComponent and Renderer 82
Step 5: Creating a JSP Tag Handler and TLD 86
Building an Application with the Input Date Component 103
Summary 104

CHAPTER 3 Defining the Deck Component 105

Requirements for the Deck Component 106
The Deck Component 106
Designing the Deck Component Using a Blueprint 107
Step 1: Creating a UI Prototype 108
Step 2: Creating Events and Listeners 113
Step 3: Creating a Behavioral Superclass 127
Step 4: Creating a Client-Specific Renderer 136
Step 5: Creating a Renderer-Specific Subclass 148
Step 6: Registering a UIComponent and Renderer 150
Step 7: Creating a JSP Tag Handler and TLD 155
Summary 169

PART 2 **Designing Rich Internet Components**

CHAPTER 4 Using Rich Internet Technologies 173

Introducing Ajax 174
The XMLHttpRequest Object 175
Traditional Web Application Development 177
Ajax Web Application Development 179
Building Ajax Applications 181
Ajax Summary 187
Introducing Mozilla XUL 187
Building XUL Applications 188
Creating Custom XUL Components Using XBL 192
XUL Summary 199

Introducing Microsoft Dynamic HTML and HTC 199
HTC Structure 199
Building DHTML Applications 202
HTC Summary 205
Comparing XBL and HTC 206
Defining a Component 206
Adding Content 206
Event Handling 206
Attaching Components 207
JSF—The Greatest Thing Since Sliced Bread! 207
Cross-Platform Support 208
Imagination As the Only Limit 209
A JSF Application Supporting Ajax, XUL, and HTC 209
Summary 211

CHAPTER 5 Loading Resources with Weblets 213

Introducing Resource Loading 213
Using Existing Solutions 214
Using Weblets 215
Exploring the Weblet Architecture 215
Using Weblets in Your Component Library 216
Using Weblets in a JSF Application 221
Summary 222

CHAPTER 6 Ajax Enabling the Deck Component 223

Requirements for the Deck Component's Ajax Implementation 223
The Ajax-Enabled Deck Component 224
Designing the Ajax-Enabled Deck Component Using a Blueprint 226
Step 1: Creating a UI Prototype 227
Step 4: Creating a Client-Specific Renderer 230
Step 6: Registering a UIComponent and Renderer 238
Step 8: Creating a RenderKit and ResponseWriter 238
Step 9: Extending the JSF Implementation 249
Step 10: Registering the RenderKit and JSF Extension 262
Step 11: Registering Resources with Weblets 263
Summary 265

CHAPTER 7 Ajax Enabling the Date Field Component 267

Requirements for the Date Component's Ajax Implementation 267
The Ajax-Enabled Date Component 268
Designing JSF Components Using a Blueprint 269
Step 1: Creating a UI Prototype 270
Step 4: Creating Converters and Validators 276
Step 5: Creating a Client-Specific Renderer 279
Step 7: Registering a UIComponent and Renderer 297
Step 8: Creating a JSP Tag Handler and TLD 297
Step 12: Registering Your Ajax Resources with Weblets 301
Summary 301

CHAPTER 8 Providing Mozilla XUL Renderers 303

Requirements for the Deck and Date Components' XUL Implementations 304
What Mozilla XUL Brings to JSF 304
What JSF Brings to XUL 304
The XUL Implementation of the Deck and Date Components 304
Designing JSF XUL Components Using a Blueprint 306
Step 1: Creating a UI Prototype 307
Step 3: Creating a Behavioral Superclass 322
Step 5: Creating a Client-Specific Renderer 324
Step 6: Creating a Renderer-Specific Subclass 341
Step 7: Registering a UIComponent and Renderer 345
Step 8: Creating a JSP Tag Handler and TLD 347
Step 9: Creating a RenderKit and ResponseWriter 350
Step 11: Registering a RenderKit 354
Step 12: Registering Resources with Weblets 355
Building Applications with JSF XUL Components 357
Summary 359

CHAPTER 9 Providing Microsoft HTC Renderers 361

Requirements for the Deck and Date Components' HTC Implementations 362
What HTC Brings to JSF 362
What JSF Brings to HTC 362

The HTC Implementation of the Deck and Date Components 362
Designing JSF HTC Components Using a Blueprint 363
Step 1: Creating a UI Prototype 363
Step 5: Creating a Client-Specific Renderer 380
Step 7: Registering a UIComponent and Renderer 396
Step 11: Registering a RenderKit and JSF Extension 396
Step 12: Registering Resources with Weblets 398
Building Applications with JSF HTC Components 398
Summary 401

CHAPTER 10 Switching RenderKits Dynamically 403

Requirements for Dynamically Switching RenderKits 404
The Dynamic RenderKit Implementation 405
Syntax for Dynamic RenderKit ID 405
The Dynamic RenderKit Managed Bean 406
The DynamicRenderKitViewHandler Class 409
Registering the Dynamic RenderKit Solution 411
Summary 412

INDEX 413

Foreword

Does the world really and truly need another JavaServer Faces book?

I was fairly well convinced the answer could only be a resounding "no!" After all, there are a good half-dozen books out in stores today, by a whole host of Web luminaries, and I've even personally helped as a technical reviewer on half of those. So what more could really be said on the subject?

But when I thought about this a bit more, it became clear that all of these books go only so far. They'll show you how to use what JSF gives you out of the box, throw you a bone for writing your own components and renderers, and give you maybe even a bit more. But none that I've seen get to the heart of why JSF is really and truly a cool and important technology; they make JSF look like YAMVCF (Yet Another Model-View-Controller Framework) for HTML—more powerful here and there, easier to use in many places, a bit harder to use in others, but really nothing major. And certainly nothing that takes us beyond the dull basics of building ordinary-looking Web applications.

This book goes a lot further. It covers the basics, of course, and shows you how to build components, but then it keeps going: on to Ajax, on to HTC, on to XUL—and how you can wrap up this alphabet soup underneath the heart of JSF, its component model, and how you can leverage it to finally develop Web applications that don't need radical rearchitecting every time the winds of client technologies blow in a different direction. Along the way, you'll learn a wide array of open source toolkits that make Web magic practical even when you're not a JavaScript guru.

So, heck, I'm convinced. The world *does* need another JSF book.

Adam Winer
JSF Expert Group Member and Java Champion

About the Authors

JONAS JACOBI is a J2EE and open source evangelist at Oracle.

A native of Sweden, Jonas has worked in the software industry for more than 15 years. Prior to joining Oracle, Jonas worked at several major Swedish software companies in management, consulting, development, and project management roles.

For the past three years, Jonas has been responsible for the product management of JavaServer Faces, Oracle ADF Faces, and Oracle ADF Faces Rich Client in the Oracle JDeveloper team.

Jonas is a popular speaker at international conferences such as Oracle OpenWorld, EclipseWorld, and JavaPolis, and he has written numerous articles for leading IT magazines such as *Java Developer's Journal*, *JavaPro*, and *Oracle Magazine*. Jonas has also contributed to the online appendix of *JavaServer Faces in Action*, by Kito D. Mann (Manning, 2005), and was a technical reviewer of *Oracle JDeveloper 10g Handbook*, by Avrom Roy-Faderman, Peter Koletzke, and Paul Dorsey (McGraw-Hill Osborne, 2004).

Apart from spending his spare time working on open source projects such as Weblets, Mabon, and D^2, he likes golf, sailing, and fast cars (preferably driving them); he also enjoys spending time with his wife, Marianne, and his daughters, Emma and Isabelle.

JOHN R. FALLOWS is a JavaServer Faces technology architect at Oracle.

Originally from Northern Ireland, John graduated from Cambridge University in the United Kingdom and has worked in the software industry for more than ten years. Prior to joining Oracle, John worked as a research scientist for British Telecommunications Plc.

For the past four years, John has played a leading role in the Oracle ADF Faces team to influence the architecture of the JavaServer Faces standard and to extend the standard to provide Ajax functionality in the ADF Faces project.

John is an active participant in the open source community, contributing to both the Apache MyFaces project and the Apache Maven project. John is also leading three new open source projects on Java.net—Weblets, Mabon, and D^2—all of which evolved while researching the foundational technologies for this book.

Apart from spending his spare time writing articles about new and exciting technologies, John likes to play soccer with his friends and likes to spend time with his beautiful wife, Nan, and their wonderful son, Jack.

33faf4ff068d72f2adcfa053cf4f7274

About the Technical Reviewers

PETER LUBBERS is an information architect at Oracle. A native of the Netherlands, Peter served as a Special Forces commando in the Royal Dutch Green Berets. Prior to joining Oracle, Peter architected and developed the internationalized Microsoft Office User Specialist (MOUS) testing framework. At Oracle, Peter develops automated help-authoring solutions. Three of these solutions are currently patent pending. He is also the author of the award-winning *Oracle Application Server Portal Configuration Guide* (Oracle, 2005).

KITO D. MANN is the editor-in-chief of JSF Central (`http://www.jsfcentral.com`) and the author of *JavaServer Faces in Action* (Manning, 2005). He is also a member of the JSF 1.2 and JSP 2.1 expert groups and principal consultant at Virtua, specializing in enterprise application architecture, development, mentoring, and JSF product strategy. Kito has consulted with several Fortune 500 clients, including Prudential Financial and J.P. Morgan Chase & Company, and he was recently the chief architect of an educational application service provider. He has a bachelor's degree in computer science from Johns Hopkins University.

MATTHIAS WESSENDORF is a PMC member of the Apache MyFaces project, a well-known JavaServer Faces implementation. Matthias is currently working as a Java Web developer in Germany, focusing on Web technologies such as JSF, Struts, Ajax, and XUL.

Matthias is the author of two developer handbooks, *Struts: Websites mit Struts 1.2 & 1.3 und Ajax effizient entwickeln* and *Web Services und mobile Clients: SOAP, WSDL, UDDI, J2ME, MIDlet, WAP & JSF*, and he has written numerous articles about JavaServer Faces for leading IT magazines in Germany.

Matthias is a frequent speaker at international conferences such as ApacheCon and JAX and also lectures in the Department of Computer Science at the University of Applied Sciences in Dortmund, Germany. During his limited spare time, he enjoys listening to electronic dance music and reading a good book.

Acknowledgments

After completing this book, we found ourselves wondering if we would do it again, and sure, we would! However, anyone who believes a book project is a simple single-author or small-team effort has never written a book. No first-time author, or authors, would be anything without guidance and tremendous support from family, friends, and colleagues.

Peter Zadrozny, thank you, thank you, and thank you! You introduced us to Apress, convinced Apress that this would be the book of the year, and then guided us through everything that newbie authors, like ourselves, needed to know. Without you and your guidance, we would never have taken the first steps toward becoming full-feathered authors.

We thank Apress and Tony Davis for giving us the opportunity to write this book and trusting Peter Zadrozny's instincts.

Peter Lubbers worked tirelessly to help us make this a better book. We owe you *big* time! Kito D. Mann, although he has an extremely busy schedule, took time from his family to provide us with his technical knowledge. Matthias Wessendorf was there from the very first draft to the final product, educating us about MyFaces and providing encouragement when it felt like we would never reach the end. Adam Winer, our ADF Faces colleague and a Java Champion, answered our questions on JSF 1.2 and made sure we kept our edge.

Elizabeth Seymour patiently answered all our questions about book-related and non-book-related issues. Kim Wimpsett helped us with grammar, spelling, and consistency throughout the book, and for this we are forever grateful. Laura Cheu patiently let us do last-, last-, and last-minute edits to text, code, and figures and patiently educated and guided us through the Apress process of finalizing our book. Without you, we would probably still be working on Chapter 4.

We would also like to thank our colleagues at Oracle Server Technologies for supporting us during this year and encouraging us to do our very best.

Jonas Jacobi and John R. Fallows

I have a list as long as any Oscar-winning actor or actress, but I only have so much space; if I've missed someone, you have my heartfelt apologies. I would first like to thank my good friend Peter Z'd for letting me in on the "how-to-make-your-family-happy-when-writing-a-book" secret and for always being there whenever I had doubts about this project.

To a true friend—John R. Fallows. I don't think my vocabulary has enough superlatives to describe my coauthor and colleague. I will be forever in his debt for all the knowledge I pulled out of him during long hours and for the patience and dedication he brought to this project; without John this would not have been possible.

To the most important person in my life, my wonderful wife, Marianne, without whom I wouldn't have been able to complete this book! To my beautiful daughters, Emma and Isabelle, for patiently waiting for me to come home and play.

Jonas Jacobi

I would first like to thank my very good friend and coauthor, Jonas Jacobi, for proposing that we work on this book together. Jonas has my deepest respect for his ability to consume highly detailed architectural knowledge and simplify it for the reader in a practical and entertaining way. There is no doubt that without Jonas this book would simply not have been possible.

To my amazing wife, Nan, whose endless patience and support made it possible for me to work on this book while she was pregnant with our son, Jack, and for the first six months of his life.

To my son, Jack, for those lovable deep laughs that made me smile no matter how tired I was.

To my dad, for always encouraging me to reach for the stars.

John R. Fallows

Introduction

Since JavaServer Faces first arrived on the Internet technology stage as the new standard for building Java-based Web applications, it has gained significant attention from the Java EE Web development community. Many developers are excited that they can use the standard JavaServer Faces HTML Basic `RenderKit` to create HTML-based Web applications, much as they did in the past with other technologies, such as Apache Struts. However, this is only the tip of the iceberg—the true power of JavaServer Faces lies in its extensible component model, which allows you to go far beyond those humble HTML beginnings.

Based on the recent surge in demand for improved usability in Web applications, it is understandable that the hottest topic at the moment is Rich Internet Applications (RIAs) and how they offer distributed, server-based Web applications with a rich interface and the interaction capabilities of desktop applications. Although RIAs hold significant promise, they still have issues with compatibility, portability, usability, and reusability. Many Web application developers are struggling to keep up with new RIA frameworks, both open source and vendor specific, as they appear on the market. What is needed is a standard way of defining an RIA regardless of what RIA framework is being used.

The debate over the best way to develop and deploy RIAs will not end with this book, but looking at the software industry today more and more developers are using the Web to deploy their applications.

User interfaces for these Web applications are often built with technologies such as HTML, CSS, JavaScript, and the DOM. These technologies were not developed with enterprise applications in mind, and with an increasing pressure from consumers to provide applications with features not fully described or supported by these technologies, developers are looking for alternative solutions or to extend the standards.

JSF does not just let you pick a single RIA technology such as Ajax, Mozilla XUL, Microsoft HTC, Macromedia Flash, and so on; it lets you pick and combine any RIA technologies you want and use them where they make the most sense. As with any technology, each RIA technology has its own advantages and disadvantages, but as a JSF component writer, you have the opportunity to leverage the best functionality of each RIA technology to provide the application developer with an extremely powerful RIA solution.

We have been very much involved in the development and the use of component-based frameworks over the past five years, starting with Oracle's own UI component framework, ADF UIX, and lately with Oracle's JSF component library, ADF Faces.

One day a very good friend asked us, "Why don't you guys share some of your experience and write a book about it?" What surprised us was that nobody had actually written a book targeting developers who are interested in the same thing we are—how to develop reusable standards-based JSF components for RIAs.

So, here we are, hoping that you will enjoy reading this book as much as we enjoyed writing it.

An Overview of This Book

Pro JSF and Ajax: Building Rich Internet Components is written to give you all the means to provide your Web application developers with easy-to-use Rich Internet Components (RICs). We decided early on that we would focus on establishing a clear blueprint that you as a developer could follow to be successful in your own JSF component development. We also decided that we would not limit this book to "just" JSF components and that we would incorporate everything you would need to know to be successful in developing, packaging, and deploying your own RICs.

This book is not, and we would like to emphasize this, *not* an introductory level book about JSF or about writing simple JSF components. Sure, this book introduces JSF and covers the basics of writing JSF components, but if you have not acquainted yourself with JSF before reading this book, we strongly encourage you to refer to a few excellent books that will introduce you to JSF and give you the foundation needed to fully appreciate this book. We recommend *JSF in Action*, by Kito D. Mann (Manning, 2005), which is an excellent and very complete book on JSF, and *Core JavaServer Faces*, by David Geary and Cay Horstmann (Prentice, 2004). We are also looking forward to seeing *Java Server Faces: The Complete Reference*, by Ed Burns and Chris Schalk (McGraw-Hill Osborne, 2006), in stores.

Pro JSF and Ajax: Building Rich Internet Components contains ten chapters that focus on writing JSF components. The book's examples are fairly extensive, so we recommend you download the example source code from the Apress Web site to give you a better overview of the examples discussed (see the next section for more information). We assume that, as an experienced Web developer and JSF developer, you can extrapolate the demonstrated topic into your own environment; thus, we avoid cluttering the text and examples with information that is of little use.

Chapter 1 gives a fast-paced and in-depth introduction to JSF, its component model, and its lifecycle. You might have followed our recommendation to read up on JSF before buying this book or you are already experienced working with JSF; either way, this chapter contains crucial information about JSF and its lifecycle that is needed in order to successfully build scalable and reusable JSF components.

Chapter 2 introduces the first JSF component: the date field component. We played with the idea of having a component comprised of three input fields representing *day*, *month*, and *year*, but this did not provide us with enough material to work with when moving forward through the book. So instead, we focused this chapter on the essentials of building a component, such as creating prototypes, managing resources, creating renderers, controlling rendering of children, handling conversion, and figuring out what's going on during postback. To be able to keep track of all the tasks associated with creating JSF components, this chapter introduces the first steps in a JSF component design blueprint. The date field component created in this chapter also introduces you to some new concepts and open source projects when you improve its user interactivity in Chapter 7.

Chapter 3 introduces the second component: the deck component. The deck component works like an accordion to show and hide information. This chapter discusses the JSF event model and teaches you how to create new behavioral superclasses and new event types with corresponding listener interfaces. By the time you finish Chapter 3, you will have enough knowledge to start writing your own basic HTML components. During the course of this book, you will be enhancing the deck and date field components, and you will be providing them with extremely rich user interactivity that leverages RITs.

Chapter 4 introduces you to three RITs (Ajax, Mozilla XUL, and Microsoft HTC) and gives you a high-level overview of these technologies. You will use these technologies in Chapters 6, 7, 8, and 9 to build rich interactivity into the date field and deck components. Of course, some simple applications in this chapter will highlight the core features of each technology.

As promised, to be able to successfully build and package JSF components, and especially RICs, you need a solution that can easily package resources, such as JavaScript libraries, CSS, and images, into the same component library as your JSF infrastructure (renderers, behavioral superclasses, and so on) and then serve them out from the same JAR. Chapter 5 introduces a new open source project—Weblets—that makes resource file management and versioning as easy for Web development as it already is for desktop-based Java development.

Chapters 6, 7, 8, and 9 address the need for a smoother and richer user experience when users interact with your components in a JSF Web application. These four chapters leverage everything you have learned so far and guide you through the gotchas of building Ajax-enabled JSF components with HTML, XUL, and HTC. These chapters also introduce you to one established and two new open source projects: the Dojo toolkit, Mabon, and D^2.

Finally, Chapter 10 pulls it all together. In this chapter, you will learn how to leverage all of the aforementioned techniques to provide your Web application developers (and users) with enterprise-class JSF components that support multiple clients.

Obtaining This Book's Source Code

All the examples in this book are freely available from the Source Code section of the Apress Web site. Point your browser to `http://www.apress.com`, click the Source Code link, and find the *Pro JSF and Ajax: Building Rich Internet Components* book. You can download the source as a zip file from this book's home page. All source code is organized by chapter. The zip file contains an application workspace, built with Oracle JDeveloper 10.1.3, and contains one project per chapter. Each project includes a WAR file that is ready to deploy to any J2EE 1.3–compliant application server. For more information about Oracle JDeveloper, please refer to the Oracle Web site at `http://otn.oracle.com/products/jdev/`.

Obtaining Updates for This Book

There are no errors in this book. Just kidding! Despite our best efforts to avoid any errors, you may find one or two scattered throughout the book. We apologize for those potential errors that may be present in the text or source code. A current errata list is available from this book's home page on the Apress Web site (`http://www.apress.com`), along with information about how to notify us of any errors you may encounter.

Contacting Us

Any feedback, questions, and comments regarding this book's content and source examples are extremely appreciated. You can direct your questions and comments to `projsf@gmail.com`. We will try to reply to your questions and comments as soon as we can, but please remember, we (like you!) may not be able to respond immediately.

Lastly, we would like to thank you for buying this book! We hope you will find this book to be a valuable source of information and inspiration and that you enjoy reading it.

PART 1

Developing Smarter with JavaServer™ Faces

JavaServer Faces (JSF) is a user interface (UI) component framework for Java 2 Enterprise Edition (J2EE) Web applications that, once adopted, allows organizations to migrate from old technologies, such as character-based platforms for virtual terminals (VTs), to more up-to-date standard-based platforms and technologies, such as JSF and Java. Over the past 15 years, the software industry has seen many technologies and platforms rise and fall. Usually, the use of a particular technology declines for several reasons, including fashion and competition. Another common reason for the fall of certain technologies is that if they are designed and maintained by one company, then the consumers of these technologies are forced to rely on support provided solely by the creators. Whenever a creator decides to deprecate a technology in favor of a more advanced solution, the consumer is left with an outdated, unsupported platform. JSF allows organizations and consumers to leverage the latest technology as it emerges, with minimal impact on existing JSF applications. JSF also brings extreme reuse of functionality and visual appearance to the software industry. Part 1 of this book will teach you what JSF is all about, describe how to leverage JSF by developing your own components, and open your eyes to a new horizon.

CHAPTER 1

The Foundation of JSF: Components

JavaServer Faces (JSF) is a user interface (UI) framework for Java Web applications. It is designed to significantly ease the burden of writing and maintaining applications that run on a Java application server and render their UIs back to a target client.

—JavaServer Faces specification

For those of you who have not had a chance to get acquainted with JSF before reading this book, this chapter will give you a fast-paced introduction to its core functionality. If you are already familiar with JSF, you may still find some of the discussion of component and lifecycle architecture to be of interest, because these topics are fundamental to your understanding of the rest of this book. This chapter will cover application development, give an overview of JSF and how it relates to other similar frameworks, and provide an in-depth examination of the JSF architecture and its component model. By the end of this chapter, you should understand the JSF architecture, its building blocks, and its request lifecycle.

Before jumping into the architecture of JSF, we'll define the audience for JSF (and ultimately for this book). The JSF specification defines the types of developers who make up the core audience: page authors, application developers, component writers, tools providers, and JSF implementers, as shown in Table 1-1.

Table 1-1. *JSF Developer Types**

Type	Description
Page author	A page author is responsible for creating the UI and has knowledge about markup and scripting languages, as well as the rendering technology such as JavaServer Pages (JSP). According to the JSF specification, this developer type is generally not familiar with programming languages such as Java or Visual Basic.
Application developer	An application developer is, according to the JSF specification, in charge of the server-side functionality of an application that may or may not be related to the UI. The technical skills of an application developer generally include Java, Enterprise JavaBeans (EJBs), or other server technologies.

Continued

Table 1-1. *Continued*

Type	Description
Component writer	A component writer is the main provider of reusable components. This developer is responsible for creating component libraries that can be consumed by others, such as the page author.
Tools provider	A tools provider, as implied by the name, provides tools that can support developers who are building applications with JSF.
JSF implementers	A JSF implementer is a developer who provides the runtime (or implementation of the JSF specification) for all the previously defined developers. Examples of available implementations are the Sun Reference Implementation (RI) (`http://java.sun.com/j2ee/javaserverfaces/`) and Apache MyFaces (`http://myfaces.apache.org`).

** Source: The JavaServer Faces 1.1 specification*

In our experience, page authors and application developers are usually the same person, so they are knowledgeable in both UI design and programming languages, such as Java or Visual Basic. We will focus most of our attention on component writers in this book.

Overview of Application Development Technologies

During the relatively short history of computers and software, application development has undergone several major evolutionary steps, all promising increased developer productivity and flexibility. These technology improvements have progressed exponentially since the computer was first introduced, and it looks like computer and software technologies will continue to evolve at the same tremendous pace well into the future.

No exponential is forever . . . but we can delay "forever."

—Gordon Moore (famous for Moore's law),
Fairchild Camera and Instrument Corporation

During these evolutionary years, the deployment profile for an application, as well as the computer and software technology used to develop such an application, has changed.

One-Tier

At the end of the 1970s and beginning of the 1980s, a fundamental shift occurred from large and centralized computers to personal computers (PCs), which moved the power of control from a few to many (anyone with a PC). Though most of the applications released during this period were more powerful than anything so far developed, they were developed and designed

for single-user tasks and lacked collaboration over common data; at this point, no central databases or email systems existed. Applications deployed or installed this way are referred to as *one-tier* applications.

From a maintenance point of view, this one-tier solution is an application that resides on an individual's machine and that controls interaction with business logic. These one-tier applications all integrate three application layers (presentation, business logic, and data), making it hard to maintain and almost impossible to share and scale information.

Two-Tier: Client-Server

Two-tier, or *client-server*, solutions took center stage in the 1980s and pushed one-tier solutions into the history archives. A two-tier architecture, which enables sharing data, changed the way applications were developed and deployed. Two-tier applications directly interact with the end user; business and presentation logic are stored on the client, and data resides on a remote server. This architecture allows multiple users to access centralized data with applications such as desktop email clients (such as Microsoft Outlook or Mozilla Thunderbird). Although the two-tier solution solves the issue of having multiple users accessing the same data source, it also has its limitations, such as the lack of flexibility of the design to later modification or porting, which in turn increases maintenance costs.

Multitier: Web Applications

The next phase in application development arrived with the Internet and the Web browser and introduced the *three-tier*, or *multitier*, architecture. In the one-tier solution, presentation, business logic, and data are all integrated in one monolithic application. The multitier architecture breaks this type of application into three layers, allowing developers to focus on specific domain areas—model (data access), view (presentation), and controller (logic). This programming paradigm, representing the split between these layers, is known as the Model-View-Controller (MVC) architecture and was first introduced in SmallTalk and spread to the developer community in the 1980s.

Splitting the one-tier application into layers—in combination with a standard client (for example, the Web browser) and a standard communication protocol (for example, Hypertext Transfer Protocol [HTTP])—suddenly gave users ubiquitous access to centralized and familiar applications such as email via a browser (for example, Google's browser-based Gmail). Applications are no longer something that only come on a CD or are downloaded. A multitier solution gives the application owner centralized maintenance and administration, which allows the application owner to provide instantaneous upgrades for everyone using the application.

Exploring Application Development Today

In this new world of multitier applications, developers need to keep up-to-date with emerging technologies and standards provided through such organizations as the World Wide Web Consortium (W3C) and the Java Community Process (JCP). The industry is evolving, which is good, but this also adds pressure on the application developer to always be building

competitive multitier applications. If you look at a typical multitier software solution—serving a retail company, for example—it might include support for multiple agents such as Web browsers, mobile devices, and character-based Video Terminals (VT, for example, VT100). Figure 1-1 shows a simplistic schema over the architecture for such a multitier application.

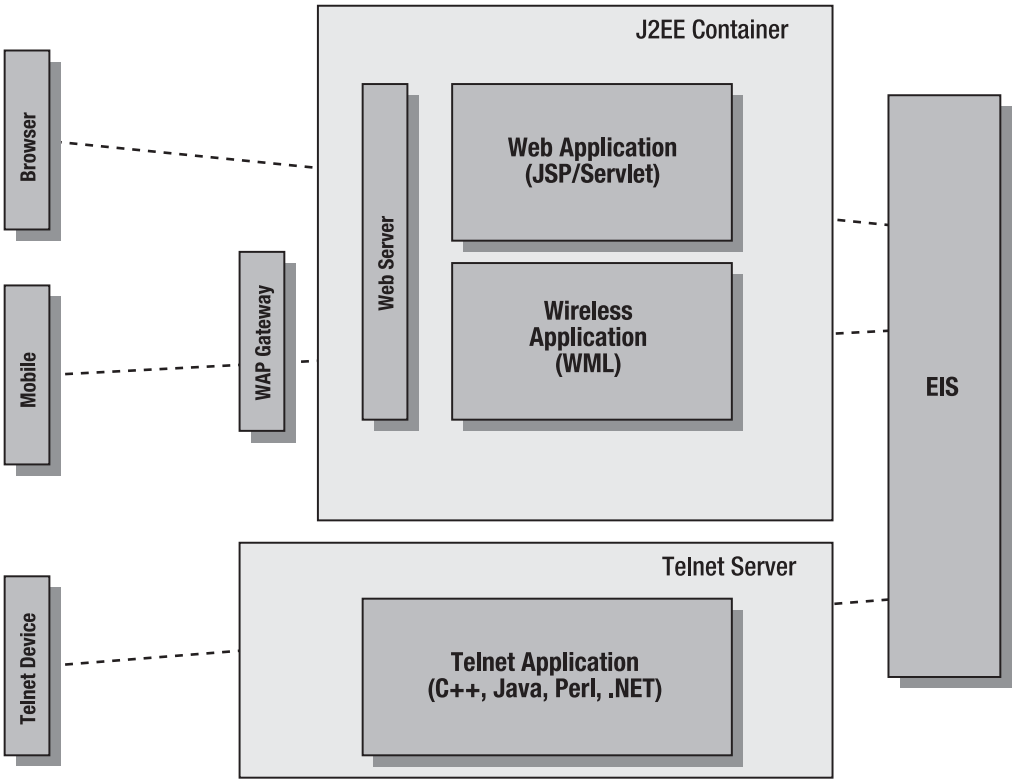

Figure 1-1. *Common J2EE architecture for a typical multitier software solution, serving a retail company*

In this scenario, the application developer is forced to provide not one application but three. This architecture contains one application for the Web interface, one for the mobile device, and finally one for the Telnet device (such as a VT terminal or handheld character-based device). All three applications use their own technology stack, which for the administrator or application developer will be a maintenance nightmare, and may cause issues with security and scalability. For the application developer, it all boils down to one question: "How many technologies do I have to learn in order to successfully build a complete solution for my project?"

Frameworks

Compared to ten years ago, customers today have much higher demands and more specific requirements for new Web application projects. They require richer and more user-friendly Web applications with built-in security, accessibility, internationalization, portability, and so on. Multitier applications must successfully deliver all these features, despite the increased complexity of additional failure scenarios and increased scalability and security requirements.

The growing complexity of building applications creates a need for simplicity. So far, in the J2EE realm, there has not been a clear choice of technology for Web applications. The traditional application programming interfaces (APIs), such as JSP and servlets, do not really provide enough abstraction from the underlying grunt work of implementing a multitier application. To fulfill these requirements and to provide some level of simplicity, the industry has evolved in a direction whereby open source communities and software companies are providing application developers with frameworks to protect them from the complexity introduced by multitier applications.

Tapestry, Struts, Tiles, TopLink, Hibernate, ADF UIX…

Many frameworks have the same underlying ideas but solve a problem a little differently and in different layers of a multitier application (the view layer, the controller layer, and the model layer). Examples of frameworks are Struts (an open source controller framework); TopLink and Hibernate (model frameworks); and Tiles, Tapestry, XUL, and ADF UIX (so-called view frameworks).

The benefits of application frameworks are the modularity, reusability, and inversion of control (IoC) they provide to developers. By encapsulating implementation details, frameworks enhance modularity and improve software quality by centralizing the impact of design and implementation details. Thanks to the stable environment provided by frameworks, they also enhance reusability by allowing developers to create generic components that can be reused in new applications. This reuse of framework components improves application developer productivity and the quality of application software. By leveraging IoC, the framework manages which application-specific methods are called in response to user events.

■ **Note** IoC means you have registered some part of your code with the framework, and the framework will call this code when the client requests it. This is also referred to as the *Hollywood principle*. ("Don't call us. We'll call you.")

In the previous retail software scenario (refer to Figure 1-1), frameworks can help increase developer productivity and ease of maintenance, but the frameworks are also incompatible with each other, which makes integration hard to handle. In contrast, JSF is a standard framework that aims to solve incompatibility.

Introducing JSF

In short, JSF is a UI component framework for J2EE applications. Before we start covering UI components (and by *UI components* we mean building blocks for application developers, not components of the framework itself), it is worthwhile to elaborate on why you need yet another framework. JSF is, after all, attempting to solve the same problems as the aforementioned Apache Tapestry or Oracle ADF UIX, frameworks that have been around for quite some time and have proved to be successful.

The differentiator that JSF brings, which other similar frameworks do not have, is the backing of a standard specification (JSR-127). Because JSF is part of the J2EE standard specification, it is a top priority for every major J2EE tools vendor in the market (including Oracle, IBM, Borland, and Sun) to support it, which in turn will guarantee a wide adoption and good tools support.

Most Web applications are stuck in the 1990s where too much effort was put into basic plumbing and not into high-level components. Basically, when there is limited abstraction or no abstraction over the markup, the development of Web applications becomes cumbersome and hard to maintain. You can invest a lot of time into the application to make it rich and interactive using various technologies from applets, plug-ins (Flex), Dynamic HTML (DHTML), and JavaScript. Used together, these technologies can make up an interactive and powerful Web application, but how do you maintain such an application? How do you reuse what you have built?

Component Model

JSF brings to the table a best-of-breed J2EE framework. JSF is here to simplify life for application developers, making it possible for them to focus on the view without needing to know the underlying markup or scripts. They will see an improvement in productivity with JSF using UI components that hide most of the grunt work of integrating richer functionality into Web applications. The goal is to provide an easy way to construct UIs from a set of reusable UI components.

These reusable components come in various shapes with different functionality, from layout components (such as the layout of an entire page) to simple buttons. Application developers can use these components to construct a page and nest UI components within each other to get the desired effect; for example, nesting text fields and buttons within a row layout component will render the nested UI components in a single row on the client. This structure of nested components is often referred to as a *parent-to-child* relationship and visualized as a UI component hierarchy. This UI component hierarchy represents a JSF page description at runtime.

Navigation Model

JSF provides a declarative navigation model, which allows application developers to set navigation rules to define the navigation from one view to another in a Web application. Navigation rules in JSF are defined inside the JSF configuration file, `faces-config.xml`, and are page-based. Code Sample 1-1 shows a navigation rule configured in `faces-config.xml`.

Code Sample 1-1. *Navigation Rule Configured in* `faces-config.xml`

```
<navigation-rule>
  <from-view-id>/login.jspx</from-view-id>
  <navigation-case>
    <from-outcome>success</from-outcome>
    <to-view-id>/result.jspx</to-view-id>
  </navigation-case>
</navigation-rule>
```

In Code Sample 1-1, a navigation rule is set so that from a view, `login.jspx`, on an outcome of `success`, the user will be sent to a page called `result.jspx`. The outcome is the return value from an action performed in the application such as a button being clicked. In JSF, an action is attached to the `UIComponent`, which allows for fine-grained control on the page. These actions can either have their own navigation rule or share the same navigation rule.

Application Lifecycle

Another benefit that application developers will discover when using JSF is that the framework helps manage UI state across server requests. Instead of having to take care of user selections and passing these selections from page to page, the framework will handle this for you. The JSF framework also has built-in processes in the lifecycle to assist with validation, conversion, and model updates. As a side bonus, JSF provides a simple model for delivering client-generated events to server-side application code.

Application Development with JSF

One of the key differentiators with JSF is that its architecture is designed to be independent of specific protocols and markup, and as such it allows developers to attach any rendering technology to the JSF application. In JSF it is the `RenderKit` that is responsible for the presentation of the JSF application by rendering the markup to the client. You can define a `RenderKit` for any type of markup (HTML, DHTML, Telnet/character mode, and eventually SVG, Flash, XUL, and so on) and use it to display a JSF page.

This separation between the page description (UI component hierarchy) and the rendering of markup is a key differentiator that provides flexibility to the component developer while protecting the application developer from changes isolated at the rendering layer. Instead of having to learn and implement different rendering technologies to solve a common problem, such as portability between different browsers (such as Netscape vs. Internet Explorer), application developers can use custom JSF components to build applications targeted for different browsers, personal digital assistants (PDAs), and so on, with a common programming model—JSF and Java.

Applying this new knowledge about JSF to the previous sample in Figure 1-1, the retail solution, the architecture could look similar to Figure 1-2.

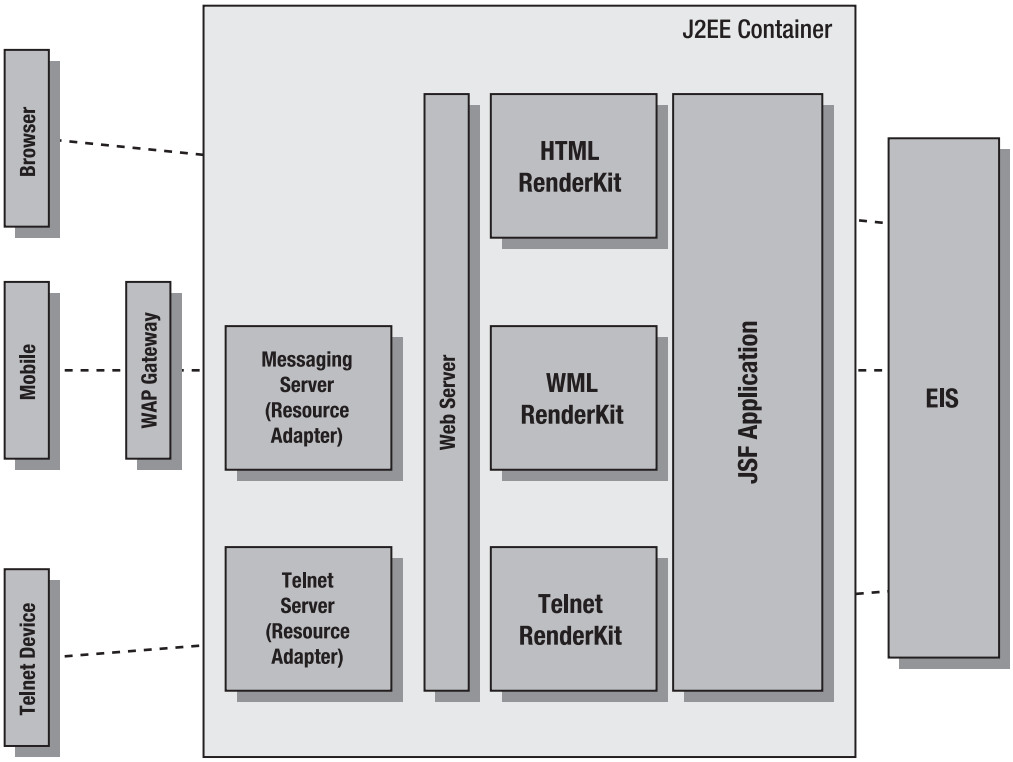

Figure 1-2. *J2EE architecture using JSF for a typical multitier software solution, serving a retail company*

In this architecture, only one application is serving three different agents using three different `RenderKits`—Hypertext Markup Language (HTML), Wireless Markup Language (WML), and Telnet. In practice, the application would probably still be three different pages but with a main difference; they will all be built on the same technology—JSF and Java. This will both save development time and reduce maintenance. Furthermore, and perhaps most important, JSF establishes standards, which are designed to be leveraged by tools (such as Oracle JDeveloper, Sun Studio Creator, and Eclipse plug-ins such as Exadel Studio) to provide developers with the ease of use that has long been sought in the J2EE developer community.

JSF Architecture

From a satellite view, JSF implements what is known as the Model 2 pattern, which is based on the MVC architecture. If you look at how the Model 2 pattern is applied in a JSF application, you can see it consists of three elements—the view, the navigation model, and the application logic, as shown in Figure 1-3.

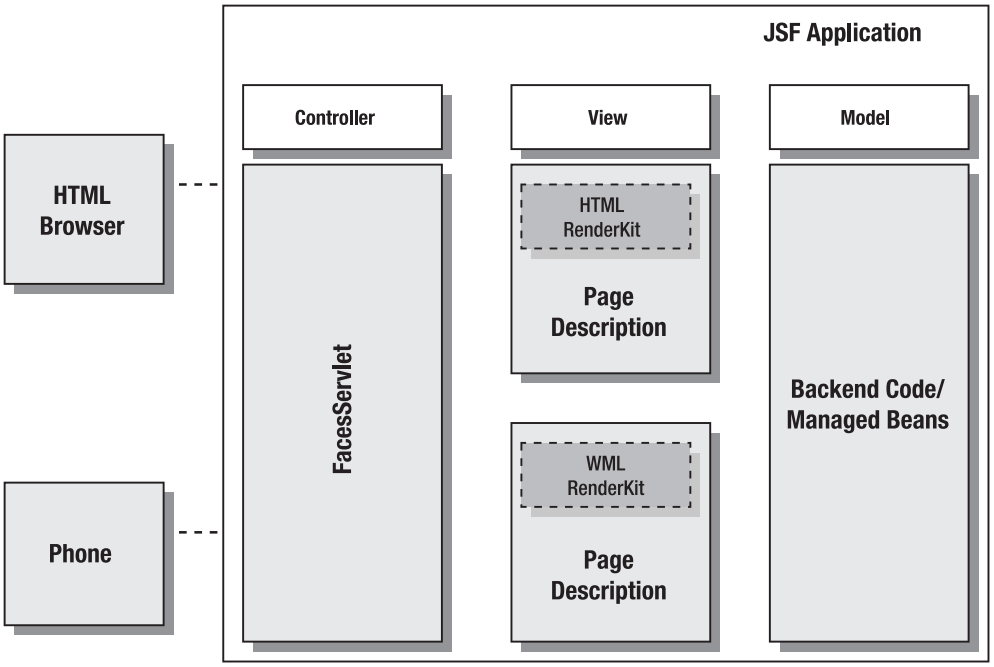

Figure 1-3. *MVC architecture with JSF (Model 2)*

Model

With JSF, the concept of a managed bean has been introduced. The managed bean is the glue to the application logic—backing code or backing bean. Managed beans are defined in the `faces-config.xml` file and give the application developer full access to all the mapped backing bean's methods. This concept of IoC is successfully used in frameworks such as Spring, Hive-Mind, and Oracle ADF model binding (JSR-227). The managed bean facility is responsible for creating the backing beans or other beans such as Data Access Objects (DAO). In JSF, a backing bean is a plain old Java object (POJO) with no dependency on implementation-specific interfaces or classes. The aforementioned JSF controller—the `FacesServlet`—is not aware of what action has been taken; it is aware only of the outcome of a particular action and will use that outcome to decide where to navigate. In JSF it is the component that is aware of which action, or method, to call on a particular user event. Code Sample 1-2 shows a managed bean defined in the `faces-config.xml` file.

Code Sample 1-2. *Managed Bean Defined in the* `faces-config.xml` *File*

```
<managed-bean>
  <managed-bean-name>sample</managed-bean-name>
  <managed-bean-class>
    com.apress.projsf.ch1.application.SampleBean
  </managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
</managed-bean>
```

Code Sample 1-2 defines a backing bean, `sample`, that points to a class called `com.apress.projsf.ch1.applictaion.SampleBean`. The `<managed-bean-scope>` indicates where an instance of this bean will be stored after it has been created—`request`, `session`, or `application` scope. The code sample also has an option to set the scope to `none` for a bean that should not be stored in any scope but instead be instantiated on every access. Table 1-2 lists all the available scopes.

Table 1-2. *Managed Bean Scopes*

Managed Bean Scope	Description
None	Instance created for every method invocation
Request	Instance created for every request
Session	Instance created on initial request and stored in the session
Application	Instance created on initial request and stored in the Web application

View

The JSF view layer describes the intended layout, behavior, and rendering of the application. One of the cornerstones of a JSF application is the `UIComponent`. `UIComponents` are the foundation of the JSF view layer and represent the behavior and structure of the application. A developer would use these `UIComponents` to construct an application by nesting components within each other. This nested structure will at runtime be represented as a component hierarchy, as shown in Figure 1-4, which in turn represents the view or UI, much like developing a Swing-based application.

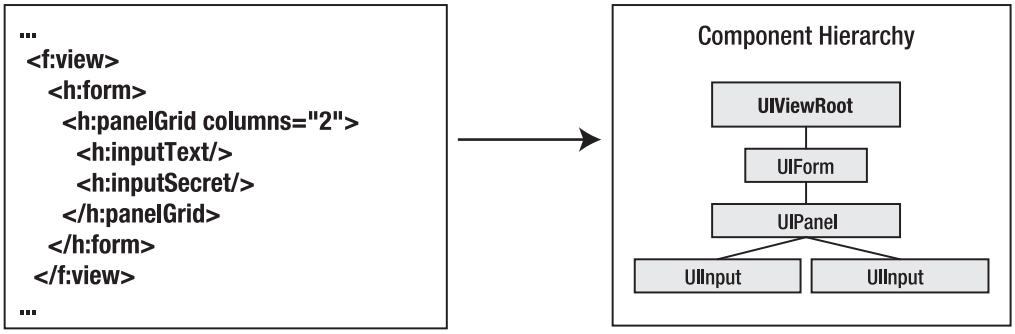

Figure 1-4. *From page description to a JSF component hierarchy*

The default page description defined by the JSF specification is JSP, but there is nothing in the JSF specification preventing an implementer from providing an alternative page description, such as an Extensible Markup Language (XML)–based, WML-based, or plain HTML-based page description. Using JSP as the page description has its good and bad sides. On the plus side, it is a well-known and widespread solution; as such, learning how to build applications with JSF and JSP presents a fairly shallow learning curve for most J2EE developers. In addition,

as a bonus, the adoption of JSF as the view technology of choice for new Web applications is good. The consequence is that JSF has a dependency on JSP, and as such, it needs to work around the different lifecycles of an application that is partially JSP and partially JSF. Later in this chapter (refer to the section "JSF and JSP"), we will cover these differences and the impact they have on applications built with JSP syntax and JSF components.

Controller

JSF comes with a simple controller—the `FacesServlet`. The `FacesServlet` acts as a gatekeeper, controlling navigation flow and dispatching requests to the appropriate JSF page.

A Component-Based UI Framework

We have set the stage for the book, so it is now time to focus on the pieces that are differentiating JSF from other technologies: `UIComponents`. JSF is a component-based UI framework where components, such as `HtmlDataTable` and `HtmlPanelGrid`, can be viewed as prefabricated blocks that allow application developers to productively build complex applications with reusable components. It also allows application developers to focus on the application logic rather than on building the dynamic/rich functionality themselves.

■ **Note** JSF is all about components—and reusable components at that! JSF was first released in March 2004 with a subsequent point release, 1.1, in August 2004. The initial JSR (JSR-127) has been replaced by JSR-252, which delivers the JSF 1.2 release.

A JSF component consists of five building blocks:

- *`UIComponent`*: The `UIComponent` is responsible for the behavior and for accessing the data model.
- *`Renderer`*: The `Renderer` is in charge of the markup rendered to the client for a specific component family.
- *`RenderKit`*: This is a library of `Renderers` with a common rendering technology (for example, HTML).
- *Renderer-specific component subclass*: The renderer-specific component subclass is a convenience class and represents renderer-specific facets and attributes.
- *JSP tag*: The default page description language is JSP, so JSF needs to follow the contract of JSP and provide JSP tags representing each JSF component.

JSF addresses the idea of a clear separation between the application logic and the visual presentation by strongly separating the UI from the underlying data model. The `Renderer` is in charge of the markup rendered to the client, and the `UIComponent` is responsible for the behavior and accessing data model. Figure 1-5 shows the separation of UI, behavior, and data model.

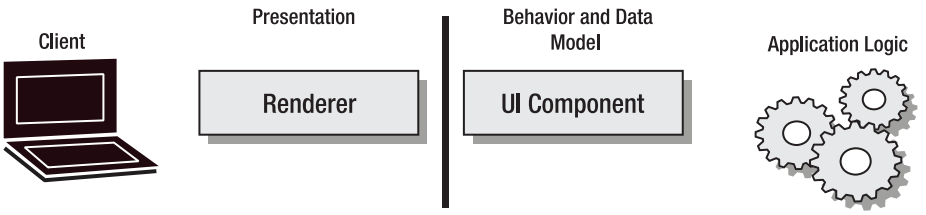

Figure 1-5. *Separation of UI from behavior and data model*

To illustrate the benefit of separating the UI and data models, let's look at an example of the common HTML form element `<select>`. This list element has a `multiple` attribute that changes the behavior from allowing a single-select option to multiple-select options. This model has no separation of rendering and behavior. For an application developer to change the behavior of the element from single select to multiple select, it requires just a minor adjustment—simply setting the attribute `multiple`. However, this will have a bigger impact on the underlying application logic since the values passed from the client are now structured as a list of key-value pairs instead of just a single key-value pair.

The `UISelectOne` and `UISelectMany` UI components provide a good example of clear separation between behavior and appearance. For example, the `UISelectOne` component has a distinct behavior to select a single value from many available options, and the `UISelectMany` component has the behavior of selecting many values from a list of available options. The `UISelectOne` component has three renderer types—`Listbox`, `Radio`, and `Menu`. Changing the appearance from `Radio` to `Menu` will not affect the underlying behavior.

However, if application developers want to change the behavior to a multiple-select component, they have to replace the entire `UISelectOne` JSF component with a `UISelectMany` JSF component, rather than just setting an attribute in the page markup, as they would do when using the `<select>` element directly. This clear separation between changing the behavior of a JSF component and changing its appearance gives application developers a better understanding of the impact of their changes when modifying the page definition. Figure 1-6 illustrates the `UIComponent` and three `Renderers` with different appearances.

Figure 1-6 illustrates a component—`UISelectOne`—from the JSF specification that has three different renderers attached—`Listbox`, `Menu`, and `Radio`. In some cases it might be necessary to create new `UIComponents` or `Renderers`.

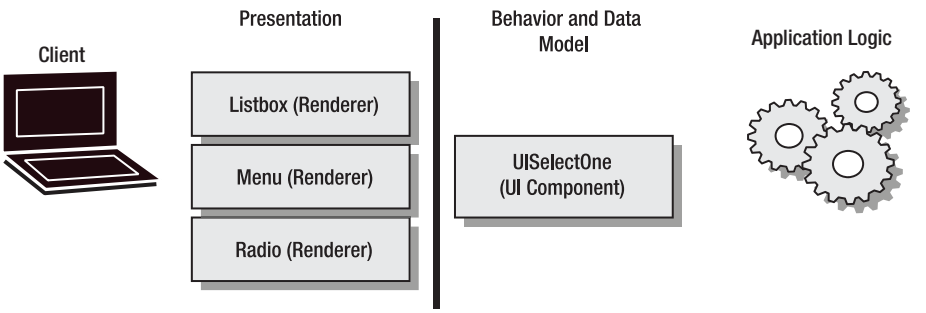

Figure 1-6. `UISelectOne` *and its renderers*

A good rule to follow is before starting a component project, search the Web for already created components. In most cases, you can probably get away with writing a new `Renderer` for an already existing component, and a fair number of components already exist. If you can't find the component you are looking for, then it is time to build your own. To build a new component, you should make sure it introduces a new behavior, functionality, or definition and that the component has a distinct server-side behavior. If the component exists and you just need a new appearance, then you need to create a new `Renderer` (for example, to enable Ajax or an existing input component).

Let's now look at the different pieces making up a JSF component.

UIComponent

The foundations of all JSF components are the abstract `UIComponent` and `UIComponentBase` classes. The `UIComponent` class (`javax.faces.component.UIComponent`) defines the behavioral agreement and state information for all components, and the `UIComponentBase` class (`javax.faces.component.UIComponentBase`) is a convenience subclass that implements almost all methods of the `UIComponent` class. A simplified description of a `UIComponent` is that it is a regular JavaBean with properties, events, and listeners.

The JSF specification defines a set of standard `UIComponent` subclasses, or behavioral *superclasses* (for example, `UISelectOne` and `UISelectMany`), which all extend the `UIComponentBase` class. In most cases, component writers will extend these standard `UIComponent` subclasses. However, they can subclass the `UIComponentBase` class as well. A JSF component consists of a `UIComponent` and one or more `Renderers`. It is important to understand that the standard `UIComponent` subclasses define only non-renderer-specific behaviors, such as `UISelectOne`. Table 1-3 gives an overview of the available standard behavioral `UIComponents` and lists their associated convenience subclasses, renderer types, and JSP tags.

Table 1-3. *Components Provided by the JSF Implementation**

UI Component	Renderer-Specific Class	Renderer Type	Syntax/JSP Tag
UIColumn	null**		<h:column>
UICommand	HtmlCommandButton	Button	<h:commandButton>
	HtmlCommandLink	Link	<h:commandLink>
UIData	HtmlDataTable	Table	<h:dataTable>
UIForm	HtmlForm	Form	<h:form>
UIGraphic	HtmlGraphicImage	Image	<h:graphicImage>
UIInput	HtmlInputHidden	Hidden	<h:inputHidden>
	HtmlInputSecret	Secret	<h:inputSecret>
	HtmlInputText	Text	<h:inputText>
	HtmlInputTextArea	Textarea	<h:inputTextarea>
UIMessage	HtmlMessage	Message	<h:message>
UIMessages	HtmlMessages	Messages	<h:messages>
UIOutput	HtmlOutputFormat	Format	<h:outputFormat>
	HtmlOutputLabel	Label	<h:outputLabel>

Continued

Table 1-3. *Continued*

UI Component	Renderer-Specific Class	Renderer Type	Syntax/JSP Tag
	HtmlOutputLink	Link	<h:outputLink>
	HtmlOutputText	Text	<h:outputText>
UIPanel	HtmlPanelGrid	Grid	<h:panelGrid>
	HtmlPanelGroup	Group	<h:panelGroup>
UIParameter	null*		<h:parameter>
UISelectOneBoolean	HtmlSelectBooleanCheckbox	Checkbox	<h:selectBooleanCheckbox>
UISelectItem	null		<h:selectItem>
UISelectItems	null		<h:selectItems>
UISelectMany	HtmlSelectManyCheckbox	Checkbox	<h:selectManyCheckbox>
	HtmlSelectManyListbox	Listbox	<h:selectManyListbox>
	HtmlSelectManyMenu	Menu	<h:selectManyMenu>
UISelectOne	HtmlSelectOneListbox	Listbox	<h:selectOneListbox>
	HtmlSelectOneMenu	Menu	<h:selectOneMenu>
	HtmlSelectOneRadio	Radio	<h:selectOneRadio>
UIViewRoot	null		<f.view>

** Source: The JavaServer Faces specification 1.1*
*** This component has no associated renderer.*

For each combination of `UIComponent` and `Renderer`, there is a renderer-specific subclass, or convenience class. A standard JSF implementation, such as the Sun RI or the MyFaces runtime, comes with a set of HTML renderers (provided through the standard HTML `RenderKit`) and a set of HTML renderer-specific subclasses, such as `HtmlSelectOneRadio`.

Renderer-Specific Component Subclass

In most cases, this subclass creates an instance of the component at runtime. As defined by its name, this subclass provides access to renderer-specific attributes on a JSF component such as `style`, `disabled`, `tooltip`, and so on—providing property getters and setters for all of these component attributes. In conjunction with the binding attribute on the JSF JSP tag, this subclass allows application developers to use JavaBean property setters to change renderer-specific attributes on the component at runtime.

Although this does work and is a useful tool for prototyping, we recommend that, where possible, application developers avoid modifying the renderer-specific attributes directly from the backing bean application logic and instead use the behavioral superclass of the component. If application developers use the parent class instead of the convenience subclass, they have no need to modify the backing bean code when the JSF component changes to use a different renderer-specific component in the page definition, such as from `HtmlSelectOneRadio` to `HtmlSelectOneListbox`. The backing bean code needs to change only when the behavioral superclass also changes, such as changing from `HtmlSelectOneRadio` to `HtmlSelectManyList`.

This subclass is optional, but it is good practice to provide this subclass with the JSF component, since sometimes application developers may like to use it for convenience, and for component writers it is hard to know whether application developers will try to use this.

Since this convenience class extends the `UIComponent` and the behavioral subclass (for example, `UISelectOne`) at runtime, the component instance will not only contain information available in this convenience class but also contain information from the extended `UIComponent` classes. If you look at the inheritance model that is used by JSF to create an instance of a component, it will look something like Figure 1-7.

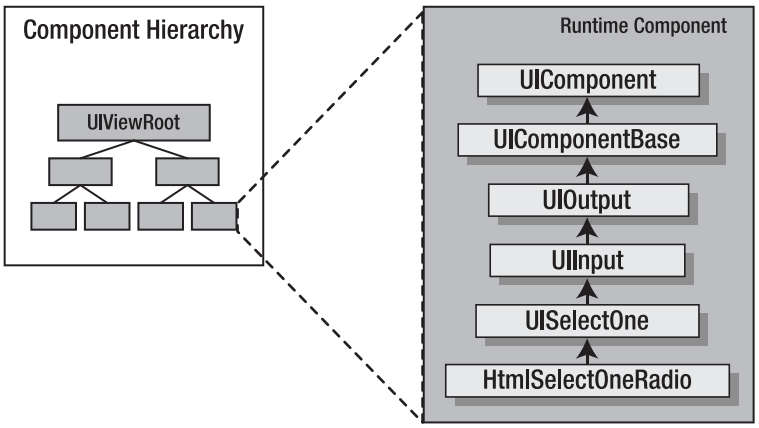

Figure 1-7. `UIComponent` *inheritance*

This model allows programmatic access to all properties and attributes defined by the different classes that build up the component. As mentioned earlier, the `UIComponentBase` class contains behavioral agreements for all components, the `UISelectOne` subclass contains properties and methods specific to its behavior (for example, select one), and the renderer-specific subclass (for example, `HtmlSelectOneListbox`) contains getters and setters for all renderer-specific attributes as well as the `rendererType` for that particular component.

Using a Renderer-Specific Component Subclass

Code Sample 1-3 illustrates the benefit of using the behavioral superclass instead of the convenience class to manipulate the page at runtime. The first bit of code illustrates a page with a simple `selectOneRadio` component with three options and a `commandButton`.

Code Sample 1-3. *JSF* `selectOneRadio` *Bound to a Renderer-Specific Subclass*

```
<h:form>
  <h:selectOneRadio binding="#{sample.selectOneRadio}" >
    <f:selectItem itemLabel="Jonas" itemValue="jonas.jacobi" />
    <f:selectItem itemLabel="John" itemValue="john.fallows" />
    <f:selectItem itemLabel="Duke" itemValue="java.dude" />
  </h:selectOneRadio>
  <h:commandButton value="Select Duke"
                   actionListener="#{sample.onAction}" />
</h:form>
```

In the selectOneRadio JSP tag, or custom action, the binding attribute is set to a value-binding expression—#{sample.selectOneRadio}. This expression points to a backend JavaBean property—selectOneRadio—that in turn is wired to the component instance for the UIComponent created by this JSP tag. Code Sample 1-4 shows the backend JavaBean, or the managed bean, that contains the page logic that at runtime will set the default option on the selectOneRadio component to java.dude at runtime, whenever the user clicks the command button.

Code Sample 1-4. *Backing Bean Using the* HtmlSelectOneRadio *Subclass*

```
package com.apress.projsf.ch1.application;

import javax.faces.event.ActionEvent;
import javax.faces.component.html.HtmlSelectOneRadio;

public class SampleBean
{
  public void onAction(
    ActionEvent event)
  {
    _selectOneRadio.setValue("java.dude");
  }

  public void setSelectOneRadio(
    HtmlSelectOneRadio selectOneRadio)
  {
    _selectOneRadio = selectOneRadio;
  }

  public HtmlSelectOneRadio getSelectOneRadio()
  {
    return _selectOneRadio;
  }

  private HtmlSelectOneRadio _selectOneRadio;
}
```

In Code Sample 1-4, the managed bean is using the renderer-specific subclass HtmlSelectOneRadio. If application developers want to change the UI and replace the selectOneRadio component with a selectOneMenu component in the page, a class cast exception is thrown at runtime. The application developer can avoid this by instead using the parent class of the selectOneRadio component—UISelectOne. Code Sample 1-5 shows how the page and the managed bean source look with the recommended approach.

Code Sample 1-5. *JSF* `selectOneRadio` *Bound to a Behavioral Superclass*

```
<body>
  <h:form>
     <h:selectOneRadio binding="#{sample.selectOne}" >
       <f:selectItem itemLabel="Jonas" itemValue="jonas.jacobi" />
       <f:selectItem itemLabel="John" itemValue="john.fallows" />
       <f:selectItem itemLabel="Duke" itemValue="java.dude" />
     </h:selectOneRadio>
     <h:commandButton value="Select Duke"
                      actionListener="#{sample.onAction}" />
   </h:form>
 </body>
```

Code Sample 1-5 contains the same page description except for one minor adjustment to the value-binding expression. To be more generic, the method name in the managed bean has been changed to `selectOne` instead of `selectOneRadio`, so the expression in the page description has to change to reference the more generic backing bean property name, as shown in Code Sample 1-6.

Code Sample 1-6. *New Backing Bean Using the UISelectOne Class*

```
package com.apress.projsf.ch1.application;

import javax.faces.event.ActionEvent;
import javax.faces.component.UISelectOne;

public class SampleBean
{

  public void onAction(
    ActionEvent event)
  {
    _selectOne.setValue("java.dude");
  }

  public void setSelectOne(
    UISelectOne selectOne)
  {
    _selectOne = selectOne;
  }

  public UISelectOne getSelectOne()
  {
    return _selectOne;
  }

  private UISelectOne _selectOne;
}
```

The new managed bean is now leveraging the inheritance of the components to make it more agnostic to changes in the UI. Instead of the convenience class `HtmlSelectOneRadio`, the behavioral superclass `UISelectOne` is used. Application developers can now change to another component within the same component family without fear of breaking the application logic.

Accessing Renderer-Specific Attributes

In the previous example, we programmatically set the `value` property on the `UISelectOne` component, which is a property defined by the behavioral superclass. But how does an application developer get access to the renderer-specific attributes if a renderer-specific subclass is not provided or (as in Code Sample 1-6) is not used? All attributes and properties are accessed via a centralized `Map` that can be accessible from any of the `UIComponent` classes and subclasses through a property called `attributes`, as shown in Code Sample 1-7.

Code Sample 1-7. *Using the Component Attributes Map to Update a Render-Specific Attribute*

```
// Renderer-specific attribute example
Map attrs = selectOne.getAttributes();
attrs.put("style", "font-face:bold");
```

Code Sample 1-7 shows how a developer can access attributes without using a renderer-specific subclass. Component writers can also introduce an interface for renderer-specific attribute methods, implemented by each renderer-specific subclass.

Saving and Restoring State

One crucial part of building Web applications is state saving. Take the traditional HTML-based shopping cart as an example. Here the application developer has to store the user product selections and persist this information until the user finishes shopping. In most cases, a shopping cart application is built up with multiple pages so the state of each page has to be saved until the buyer has finished shopping. The state is stored in hidden form fields, stored in the session, or passed on as a request to the next page. Those who have dealt with this know this is not a trivial task to accomplish.

State management is one of the primary benefits of using JSF to build applications. JSF provides automatic UI state handling through a class called `StateManager`, which saves and restores state for a particular view (hierarchy of `UIComponents`) between requests on the server. Each `UIComponent` saves and restores its own internal state when requested by the `StateManager`; the `StateManager` itself saves and restores the state associated with the structure of the `UIComponent` hierarchy. If a `UIComponent` is marked as being `transient`, then it is omitted from the structure by the `StateManager`, causing it to be removed from the `UIComponent` tree at the end of the request.

Two alternatives exist for storing the state of a view—on the client side and server side. By default state is saved on the server. The server-side implementation is supported by the JSP and Servlet specifications, but JSF conceals all the details of how this works. A class called `ResponseStateManager`, which is created and managed by a `RenderKit`, manages the client-side state saving. Client-side state saving depends not only on the JSF implementation but also heavily on the markup language rendered to the client and on how state can be managed by that client. With HTML as markup, the state is typically stored in a hidden form field.

■ **Note** Although JSF 1.1 kept the name of the view state in a hidden form field as a private implementation detail, JSF 1.2 now standardizes the name as `javax.faces.ViewState` so that alternative postback mechanisms, such as Ajax, can more easily be integrated with the JSF lifecycle.

One of the drawbacks of saving state in the user session on the server is memory consumption. If scalability is an issue for application developers, the client-side implementation will prevent memory consumption from shooting through the roof and will have an advantage in clustered environments. But, since state will now have to be sent back and forth between the client and the server, response time might increase. You can configure the state-saving method, as shown in Code Sample 1-8, in the application deployment descriptor file—`WEB-INF/web.xml`—by setting the parameter `STATE_SAVING_METHOD` to `client` or `server`.

Code Sample 1-8. *Setting the Method of Saving State to the Server Side in the Deployment Descriptor*

```
<context param>
  <param-name>javax.faces.STATE_SAVING_METHOD</param-name>
  <param-value>server</param-value>
</context param>
```

Component Family and Component Type

The component family is a string that represents the behavior of the component (for example, an input component or command component). The component family is declared in the JSF configuration file—`faces-config.xml`—and used to select a `Renderer` for a particular component. Code Sample 1-9 shows how you associate a `Renderer` with a particular component family.

Code Sample 1-9. *Associating a Renderer to a Particular Component Family*

```
<render-kit>
  <renderer>
    <component-family>
      javax.faces.Input
    </component-family>
    <renderer-type>
      com.apress.projsf.Date
    </renderer-type>
    <renderer-class>
      com.apress.projsf.ch2.render.html.basic.HtmlInputDateRenderer
    </renderer-class>
```

■ **Note** The prefix `javax.faces` is reserved for use by component families defined in the JSF specification. All samples in this book use the `com.apress.projsf` prefix for custom component families.

The component type is a string that is used as an identifier for the `UIComponent` subclass. You can find information about the relationship between the component type and `UIComponent` subclass in the JSF configuration file, as shown in Code Sample 1-10.

Code Sample 1-10. *Mapping of Component Type and* `UIComponent` *Subclass*

```
<component>
  <component-type>
    com.apress.projsf.ProInputDate
  </component-type>
  <component-class>
    com.apress.projsf.ch2.component.pro.ProInputDate
  </component-class>
```

In Code Sample 1-10, a `UIComponent` subclass—`com.apress.projsf.component.pro.ProInputDate`—has been assigned `com.projsf.ProInputDate` as the component type. By convention, the component type is also declared in the `UIComponent` subclass as a constant—`COMPONENT_TYPE`. This simplifies life for developers so they don't need to remember the component type for every component.

■ **Note** The prefix `javax.faces` is reserved for use by component types defined in the JSF specification. All samples in this book use the `com.apress.projsf` prefix for custom component types.

Converters, Validators, Events, and Listeners

Apart from providing `UIComponents`, a JSF implementation also provides helper classes for these `UIComponents`. These helper classes are divided into converters, validators, and an event and listener model. The converters provide a bidirectional type conversion between the submitted value of a component and the corresponding strongly typed object in the model tier. The validators perform validation on the strongly typed object; for example, they can ensure that a date is not in the past. Code Sample 1-11 shows an `inputText` component with attached date converter.

Code Sample 1-11. `inputText` *Component with Attached Date Converter*

```
<h:inputText value="#{sample.date}" >
  <f:convertDateTime pattern="yyyy-MMM-dd" />
</h:inputText>
```

JSF also provides a way to attach listeners to components and broadcast events to those listeners, much the same way it works in AWT and Swing. For example, a `commandButton` is a source of `ActionEvents`. When a `commandButton` is clicked, a postback occurs, and a new `ActionEvent` is stored in an event queue. Any event listeners registered with the `commandButton` are notified of this event. Code Sample 1-12 shows a `commandButton` with an attached `Listener`.

■ **Note** The JSF specification for registering listeners and broadcasting events is based on the design patterns of the JavaBean specification, version 1.0.1.

Code Sample 1-12. `commandButton` *with Attached Listener*

```
<h:commandButton value="Login"
                 action="success"
                 actionListener="#{sample.onLogin}" />
```

In Code Sample 1-12, a `commandButton` component has two properties—`action` and `actionListener`. Both attributes take method-binding expressions, and the differences are that the `action` attribute requires a method that returns a `String` object and the `actionListener` attribute requires a method that accepts an `ActionEvent` that has a `void` return type. The `action` attribute's string value is used for navigation purposes.

When the queued `ActionEvent` is processed, these method-binding expressions will be used to execute the backing bean methods referenced by `action` and `actionListener`.

Facets

A JSF view is comprised of a component hierarchy, providing access to each parent component's children by index. Sometimes it is also necessary to provide an alternative way of adding subordinate components that are not part of this ordered list.

One example is the `dataTable` component, where the children represent the columns to be rendered in the table. In some cases it might be useful to identify a component that represents the header and/or footer of the entire table, separate from the usual child collection that represents the individual columns. These `header` and `footer` child components are called *facets*, referenced only by name, with no specific order. The name of a facet represents the role that the nested component will play in the parent component. It is important to note that a parent component can contain only one child component per named facet, but the same parent component can contain many indexed child components.

Code Sample 1-13 shows how to add a `header` facet to both the `dataTable` component and the `column` component.

Code Sample 1-13. *Facets Within a* `dataTable` *Component*

```
<h:dataTable value="#{sample.tableList}" var="rows" >
  <f:facet name="header" >
    <h:outputText value="Contact Information" />
  </f:facet>
```

```
  <h:column>
    <f:facet name="header" >
      <h:outputText value="Firstname" />
    </f:facet>
    <h:outputText value="#{rows.firstname}" />
  </h:column>
  …
</h:dataTable>
```

Renderers

The JSF specification outlines two models for how a JSF component can handle values from incoming requests (*decode*) and outgoing response (*encode*). These two models—direct implementation and delegated implementation—have two distinct approaches; the direct implementation relies on the `UIComponent` instance to handle decode and encode, and the delegate implementation delegates these responsibilities to a `Renderer`. As you have seen in Figure 1-5, the delegate approach allows application developers to work with the `UIComponent` independently from what will be rendered on the client. In this book, we will discuss only the delegate implementation approach, since our goal is to provide multiple `Renderers` for each behavioral component. This approach is also what makes JSF such a powerful UI framework.

■ **Note** The direct implementation approach provides slightly better performance since there is no need to delegate to a `Renderer`, but it also severely limits extensibility and portability across clients.

`Renderers` are responsible for the presentation of a JSF component and must generate the appropriate client-side markup, such as HTML and JavaScript, or XUL. `Renderers` are also in charge of converting information coming from the client to something understandable for the component (for example, a string value from an HTML form `POST` converted to a strongly typed `Date` object).

Although a `Renderer` introduces client-side attributes such as `style`, `disabled`, `tooltip`, and so on, these attributes are actually exposed in the renderer-specific component subclass (for example, `HtmlSelectOneRadio`).

One major difference between `UIComponents` and `Renderers` is the way they are defined at runtime. `Renderers` are defined as singletons, so there is only one `Renderer` for all instances of a `UIComponent` for each particular renderer type.

■ **Caution** Since individual `Renderer` instances will be instantiated as requested during the rendering process and used throughout the life of a Web application, it is important to understand that each instance may be invoked from more than one request-processing thread simultaneously. This requires that `Renderers` are programmed in a thread-safe manner.

Renderer Types

The renderer type is an identifier that is defined by the component, and in combination with the component family, it uniquely identifies which `Renderer` class to use with the component. Combining the renderer type and the component family is extremely powerful since it allows the reuse of the renderer type for multiple behavioral components.

Code Sample 1-14 illustrates how a component family is associated with a specific `Renderer` and renderer type.

Code Sample 1-14. *Renderer Type As Defined in the JSF Configuration File*

```
<render-kit>
  <renderer>
    <component-family>
      javax.faces.Input
    </component-family>
    <renderer-type>
      com.apress.projsf.Date
    </renderer-type>
    <renderer-class>
      com.apress.projsf.ch2.renderer.html.HtmlInputDateRenderer
    </renderer-class>
```

Table 1-4 contains a subset of the standard component families with their associated components and component and renderer types.

Table 1-4. *A Subset of All Standard Component Families and Their Components, Component Types, and Renderer Types*

Component Family*	Component	Component Type	Renderer Type**
Command	UICommand	Command	
	HtmlCommandButton	HtmlCommandButton	Button
	HtmlCommandLink	HtmlCommandLink	Link
Data	UIData	Data	
	HtmlDataTable	HtmlDataTable	Table
Form	UIForm	Form	
	HtmlForm	Form	Form
Graphic	UIGraphic	Graphic	
	HtmlGraphicImage	HtmlGraphicImage	Image
Input	UIInput	Input	
	HtmlInputHidden	HtmlInputHidden	Hidden
	HtmlInputSecret	HtmlInputSecret	Secret
	HtmlInputText	HtmlInputText	Text
	HtmlInputTextArea	HtmlInputTextArea	Textarea

Continued

Table 1-4. *Continued*

Component Family*	Component	Component Type	Renderer Type**
Output	UIOutput	Output	
	HtmlOutputFormat	HtmlOutputFormat	Format
	HtmlOutputLabel	HtmlOutputLabel	Label
	HtmlOutputLink	HtmlOutputLink	Link
	HtmlOutputText	HtmlOutputText	Text
Panel	UIPanel	Panel	
	HtmlPanelGrid	HtmlPanelGrid	Grid
	HtmlPanelGroup	HtmlPanelGroup	Group

** The fully qualified name of the component family is* `javax.faces.<name in table>`.
*** The fully qualified name for renderer type is* `javax.faces.<name in table>`.

Table 1-4 shows that the renderer type `Text` is used in several places—for both the `HtmlInputText` component and the `HtmlOutputText` component. The combination of the component family `Output` and the renderer type `Link` uses the `Renderer` class that would generate a regular HTML link element—`<a href="...">some text</a>`—to the client.

RenderKits

The functionality of a `RenderKit` is to support `UIComponents` that use the delegate implementation approach with the delegation of `Renderers` to the `UIComponent`. `RenderKits` group instances of `Renderers` with similar markup types, and the default `RenderKit` provided by all JSF implementations is the HTML Basic `RenderKit` containing `Renderers` that output HTML 4.0.1. Other possible `RenderKits` can have `Renderers` supporting view technologies such as SVG, WML, Ajax, XUL, and so on. In this book, you'll look at additional `RenderKits` for Microsoft's DHTML/HTML Components (HTC) and Mozilla's XUL/XML Binding Language (XBL) technologies.

The `RenderKit` is not responsible for creating the `Renderer` because it will store only a single instance of each renderer type. Each `RenderKit` is associated to a view (component hierarchy) at runtime as a `UIViewRoot` property. If no `RenderKit` has been set, the default `RenderKit` will be used. When it comes to `RenderKits`, many times you have no need to create a new `RenderKit`. Adding a custom `Renderer` to an already existing `RenderKit` is just a configuration operation. If a `RenderKit` identifier is omitted, the custom `Renderer` is automatically added to the default HTML Basic `RenderKit`. If you would like to add a `RenderKit` with custom `Renderers`, you can do the same thing—update the JSF configuration file. Code Sample 1-15 shows how you can add a new `Renderer` to the JSF configuration file.

Code Sample 1-15. *New Renderer Added to the Default HTML Basic RenderKit*

```
<render-kit>
  <!-- no render-kit-id, so add this Renderer to the HTML_BASIC RenderKit -->
  <renderer>
```

```
    <component-family>
      javax.faces.Input
    </component-family>
    <renderer-type>
      com.apress..projsf.Date
    </renderer-type>
    <renderer-class>
      com.apress.projsf.ch2.renderer.html.HtmlInputDateRenderer
    </renderer-class>
    ...
</render-kit>
```

By not adding a `RenderKit` identifier to the `RenderKit` configuration, the `Renderer` sample in Code Sample 1-15—`com.apress.projsf.ch2.renderer.html.HtmlInputDateRenderer`—is automatically added to the standard default HTML Basic `RenderKit`.

Custom Action Tag Handlers

Since the default page description language is JSP, most JSF components will have a JSP custom action. When the JSP container encounters a custom action, it asks for the JSF tag handler associated with this action. The main purpose of the JSF tag handler is to create an instance of the component, using the renderer-specific subclass, and associate the component with a `Renderer` at the first page request.

Request-Processing Lifecycle

As a component writer, it is essential you have a clear understanding about the lifecycle of JSF. A page constructed with JSF components will go through a well-defined request-processing lifecycle. This lifecycle consists of six phases—Restore View, Apply Request Values, Process Validations, Update Model Values, Invoke Application, and Render Response, as shown in Figure 1-8.

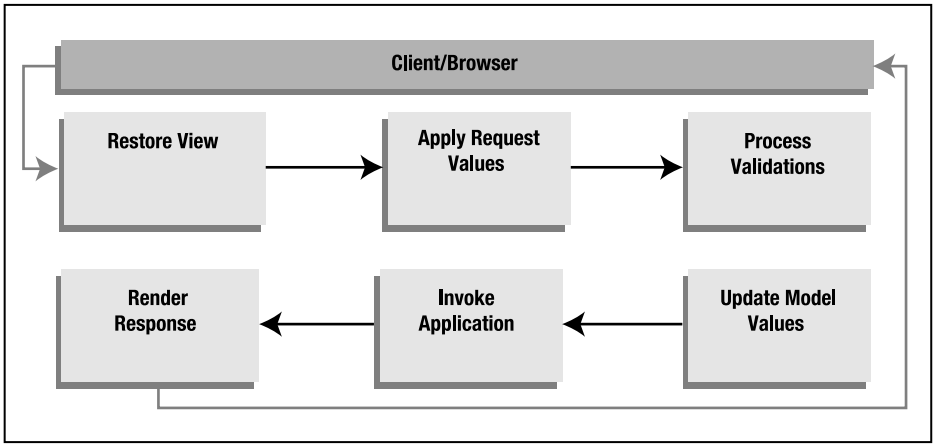

Figure 1-8. *Formal lifecycle of JSF*

Here's the process broken down:

1. *Restore View*: This phase is responsible for restoring the component hierarchy from the previous request and attaching it to the `FacesContext`. If no saved state is available, then the Restore View phase is responsible for creating a new `UIViewRoot`, which is the root node in the component hierarchy, and storing it on the `FacesContext`.

2. *Apply Request Values*: In this phase, each component has the opportunity to update its current state with information included in the current request.

3. *Process Validations*: This phase is in charge of processing any validators or converters attached to components in the component hierarchy.

4. *Update Model Values*: During this phase, all suitable model data objects will have their values updated to match the local value of the matching component, and the component local values will be cleared.

5. *Invoke Application*: At this phase, any remaining events broadcast to the application need to be performed (for example, actions performed by an `HtmlCommandButton`).

6. *Render Response*: This phase is responsible for rendering the response to the client and storing the new state for processing of any subsequent requests.

To put these phases into a real-life context, we'll use a simple example where the user will access an application built with JSF and JSP. This application contains a simple login page with some input fields for a username and password and a button to log in. On successful login, the user is redirected to a second page that will display the user's username.

Building an Application Using JSF

This application contains three essential pieces—the application description (JSP), a JSF configuration file, and a managed bean. This application has two JSP pages—`login.jspx` and `result.jspx`. Code Sample 1-16 shows the login page.

Code Sample 1-16. *The Login Page*

```
<?xml version="1.0" encoding="utf-8"?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" version="2.0"
          xmlns:f="http://java.sun.com/jsf/core"
          xmlns:h="http://java.sun.com/jsf/html">
  <jsp:output omit-xml-declaration="true" doctype-root-element="HTML"
              doctype-system="http://www.w3.org/TR/html4/loose.dtd"
              doctype-public="-//W3C//DTD HTML 4.01 Transitional//EN"/>
  <jsp:directive.page contentType="text/html;charset=utf-8"/>
  <f:view>
    <html>
      <body>
        <h:form>
          <h:outputText value="Application Login" />
          <h:inputText value="#{credentials.username}" />
```

```
          <h:inputText value="#{credentials.password}" />
          <h:commandButton value="Login"
                           action="success"
                           actionListener="#{credentials.onLogin}" />
        </h:form>
      </body>
    </html>
  </f:view>
</jsp:root>
```

The structure of the page is simple and describes a page containing two input fields for a username and password and a login button. Figure 1-9 shows what the page looks like when rendered.

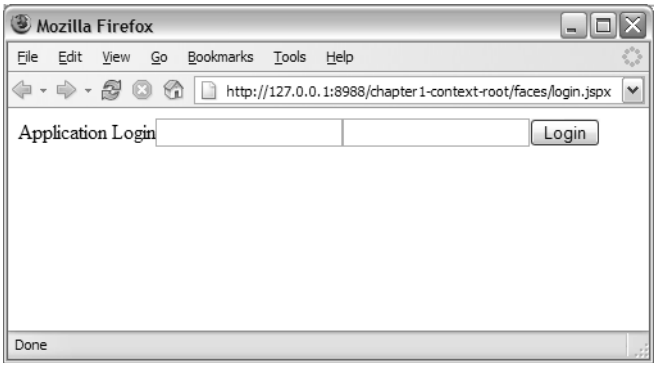

Figure 1-9. *The login page*

The second page in the application, shown in Code Sample 1-17, is simple and merely illustrates navigation and completion of the lifecycle. The page contains an `<h:outputText>` component that will render the entered value from the username `<h:inputText>` component on the initial page on successful login.

Code Sample 1-17. *Navigation Rules and Managed Beans for the Application*

```
<?xml version="1.0" encoding="utf-8"?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" version="2.0"
          xmlns:f="http://java.sun.com/jsf/core"
          xmlns:h="http://java.sun.com/jsf/html">
  <jsp:output omit-xml-declaration="true" doctype-root-element="HTML"
              doctype-system="http://www.w3.org/TR/html4/loose.dtd"
              doctype-public="-//W3C//DTD HTML 4.01 Transitional//EN"/>
  <jsp:directive.page contentType="text/html;charset=utf-8"/>
  <f:view>
    <html>
```

```
      <body>
        <h:form>
            <h:outputText value="#{credentials.username}" />
        </h:form>
      </body>
    </html>
  </f:view>
</jsp:root>
```

To be able to navigate from one page to another, you have to define a navigation case in the JSF configuration file—`faces-config.xml`. You also need to create a mapping to the back-end code using a managed bean. Code Sample 1-18 shows how to do this.

Code Sample 1-18. *Navigation Rules and Managed Beans for the Application*

```
<navigation-rule>
  <from-view-id>/login.jspx</from-view-id>
  <navigation-case>
    <from-outcome>success</from-outcome>
    <to-view-id>/result.jspx</to-view-id>
  </navigation-case>
</navigation-rule>
<managed-bean>
  <managed-bean-name>credentials</managed-bean-name>
  <managed-bean-class>
    com.apress.projsf.ch1.application.CredentialsBean
  </managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
</managed-bean>
```

As you can see, Code Sample 1-18 defines that from the `login.jspx` page, on an outcome of `success`, the user of the application will be sent to the `result.jspx` page. It also defines a managed bean that points to a class—`CredentialsBean`—containing some simple application logic. Code Sample 1-19 shows the application logic.

Code Sample 1-19. *The Application Logic*

```
package com.apress.projsf.ch1.application;

import javax.faces.event.AbortProcessingException;
import javax.faces.event.ActionEvent;

public class CredentialsBean
{
  public void onLogin(
    ActionEvent event)
```

```
  {
    If (!"duke".equalsIgnoreCase(_username))
       throw new AbortProcessingException("Unrecognized username!");
    // clear out the password, for good measure!
    _password = null;
  }

  public void setUsername(
    String username)
  {
    _username = username;
  }

  public String getUsername()
  {
    return _username;
  }

  public void setPassword(
    String password)
  {
    _password = password;
  }

  public String getPassword()
  {
    return _password;
  }

  private String _username;
  private String _password;
}
```

Web Application Start-Up

Upon receiving a JSF request, the JSF implementation must *launch*, or acquire, references to several processes/services that must be available to a JSF Web application running in a servlet or portlet environment. To get access to these references, the JSF implementation will call several factories that are responsible for creating instances needed to launch the JSF application.

When a JSF Web application starts, four factories are instantiated; each of these factories is responsible for different areas within a JSF Web application:

ApplicationFactory: The `ApplicationFactory` class is responsible for the creation of the `Application` instance, which can be seen as a service that allows, for example, the `Lifecycle` instance to create and restore JSF views (component hierarchies) on incoming requests and to store the state of the JSF view.

LifecycleFactory: The `LifecycleFactory` is in charge of returning a `Lifecycle` instance for a lifecycle identifier. The default `Lifecycle` instance is in charge of invoking processing logic to implement the required functionality for each phase of the JSF request-processing lifecycle.

RenderKitFactory: The `RenderKitFactory` is responsible for returning a `RenderKit` for the JSF Web application. A `RenderKit` is a library of `Renderers` with a common rendering technology.

FacesContextFactory: The `FacesContextFactory` provides the JSF implementation with a way to create an instance of `FacesContext` that is used to represent contextual information associated with the incoming request and eventually with the response.

Figure 1-10 shows the players involved at application start-up.

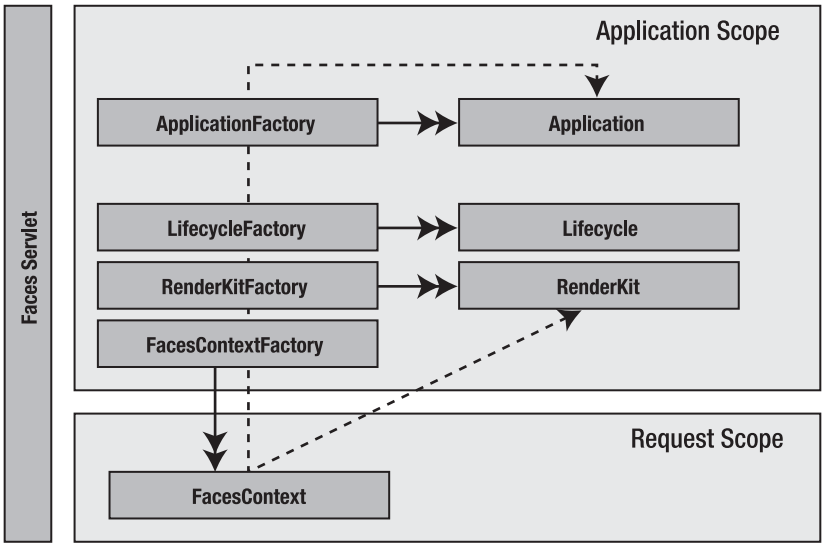

Figure 1-10. *Application creation*

Each JSF Web application has one `ApplicationFactory`. This factory class is responsible for creating and replacing the `Application` instance that is required by all applications utilizing JSF. The `Application` instance will then serve other processes with services supported by this instance. Likewise, the JSF configuration file—`faces-config.xml`—is read once during the creation of the Web application and stored in the `Application` instance.

The `RenderKitFactory` is responsible for returning a `RenderKit` instance based on the `RenderKit` identifier for this JSF Web application. For each JSF implementation, there has to be one default `RenderKit`—the HTML `RenderKit` that is identified by a string constant—`RenderKitFactory.HTML_BASIC_RENDER_KIT`. The `LifecycleFactory` is in charge of creating (if needed) and returning a `Lifecycle`. This `Lifecycle` instance is in charge of invoking processing logic to implement the required functionality for each phase (refer to Figure 1-8) of the

request-processing lifecycle. The last factory class—`FacesContextFactory`—is providing the JSF implementation with a way to create an instance of `FacesContext` that is used to represent contextual information associated with the incoming request and eventually creating the response.

Initial Request

When the user accesses the application for the first time, an initial request is sent to the `FacesServlet`, which dispatches the request to the JSF `Lifecycle` instance (refer to Figure 1-10).

Restore View Phase

The first phase of the JSF lifecycle is the Restore View phase (see Figure 1-11) whose responsibility it is to check whether this page has been requested earlier or if this is a new request.

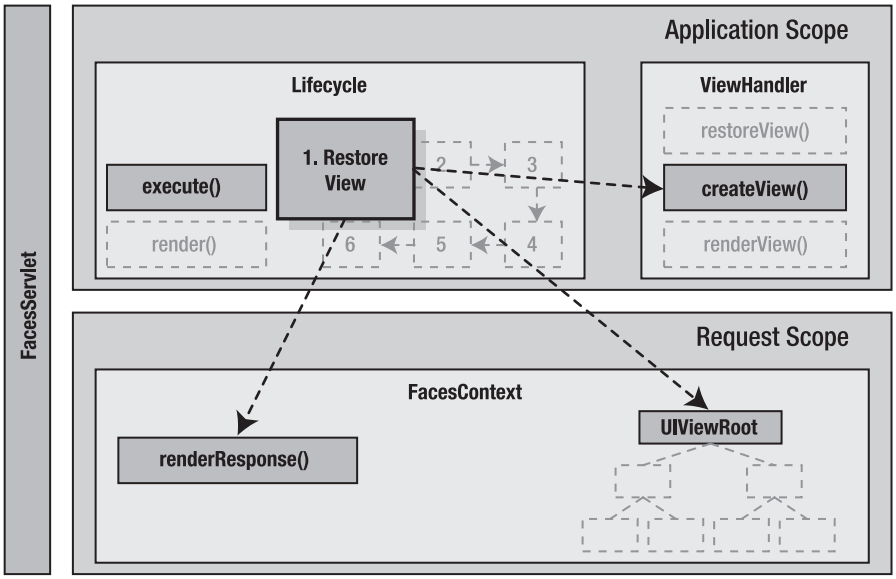

Figure 1-11. *Restore View phase during initial request*

In Figure 1-11, you are looking at the process for an incoming request and how the first phase—Restore View—in the JSF lifecycle is responsible for restoring a view from the server and client state. During the first request for this view, the `ViewHandler.restoreView()` method will return `null`, since there is no stored state.

■ **Note** The JSF lifecycle phase identifiers are part of the JSF public API in the `PhaseId` class.

If the return value is `null`, the Restore View phase will call `renderResponse()` on the `FacesContext` for this request. The `renderResponse()` method will indicate that when this phase is done, the `render()` method is called to execute phase 6—Render Response—without proceeding with phases 2 through 5. Subsequently, the Restore View phase will call the `ViewHandler.createView()` method to create the component hierarchy root—`UIViewRoot`—and attach it to the `FacesContext`. The `UIViewRoot` component performs no rendering but plays an important role in event delivery during a postback request.

Render Response Phase

When the `renderResponse()` method is called during the Restore View phase, the lifecycle skips directly to the `render()` method, which is responsible for performing the Render Response phase, as shown in Figure 1-12.

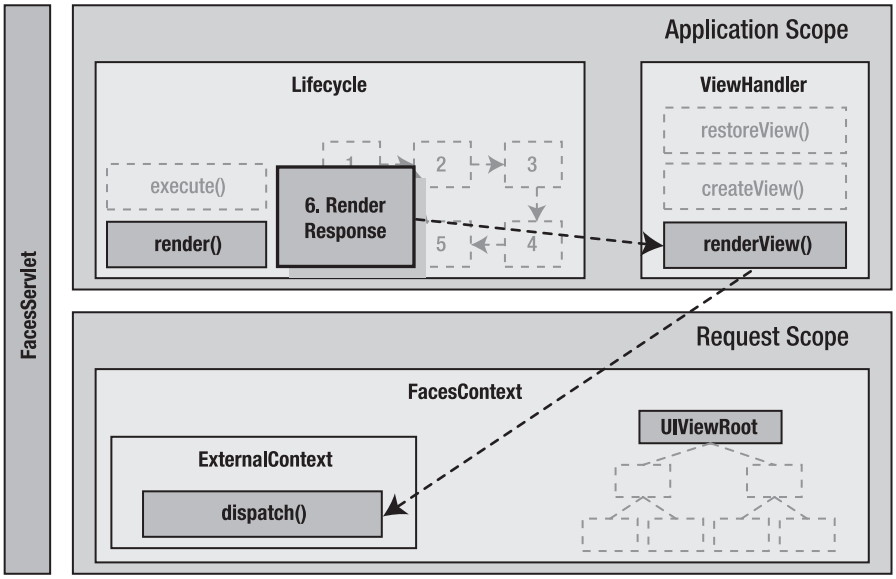

Figure 1-12. *Render Response phase during initial request*

During this phase, the `ViewHandler.renderView()` method is called to execute the JSP document. The `renderView()` method will pass the value of the `viewId` property acquired from the `UIViewRoot` node as a context-relative path to the `dispatch()` method of the `ExternalContext` associated with this request. The `dispatch()` method will forward the value of `viewId` property (for example, `/login.jspx` as a context-relative path) to the Web container.

Since the JSF-specific mapping is not part of the forwarded request, the request is ignored by the `FacesServlet` and passed to the JSP container, which in turn will locate the JSP based on the context-relative path and execute the JSP page matching the `viewId` (for example, `/login.jspx`). Figure 1-13 shows the processing of the JSF JSP document.

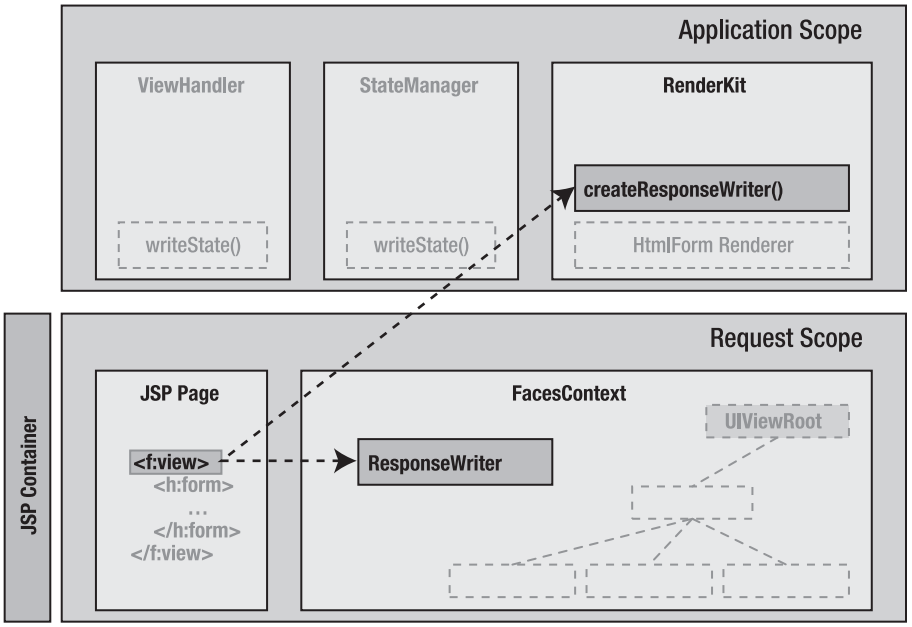

Figure 1-13. *Setting the* `ResponseWriter` *on the* `FacesContext`

THE JSF VIEW IDENTIFIER: VIEW ID

Depending on which mapping is used—prefix or suffix—the `UIViewRoot` view identifier is derived slightly differently from the request uniform resource identifier (URI). If prefix mapping is used, such as `/faces/*` (which is the most common) for the `FacesServlet`, the `viewId` property is set from the path information coming after the mapping; for example, `/context-rootfaces/login.jspx` will set a view identifier equal to `/login.jspx`. If suffix mapping is used, such as `*.jsf`, the `viewId` property is set from the servlet path information of the request URI, after replacing the suffix with the value of the context initialization parameter named by the symbolic constant `ViewHandler.DEFAULT_SUFFIX_NAME`. For example, `/context-root/login.jsf` will set a view identifier equal to `/login.jsp` by default but can leverage the context initialization parameter to use `.jspx` as the default suffix instead.

Before processing and executing the JSF JSP document, the JSP runtime first determines the content type and character encoding to use. For JSF to work in harmony with the JSP lifecycle, the `<f:view>` tag needs to be present. The `<f:view>` tag is a JSP body tag that buffers all the rendered output from the nested JSF components. Simply put, the `<f:view>` tag serves as a container for all other JSF components. The `<f:view>` tag is responsible for creating and storing an instance of the `ResponseWriter` on the `FacesContext`.

The `createResponseWriter()` method creates a new instance of the `ResponseWriter` for the specified content type and character encoding. The `ResponseWriter` is responsible for writing the generated markup to the requesting client, in this case the `<f:view>` body content buffer.

CONTENT TYPE AND CHARACTER ENCODING

When a server sends a document to an HTTP browser client, it also passes information in the Content-Type HTTP header about the Multipurpose Internet Mail Extensions (MIME) type, such as `text/html`, and the character set, such as `UTF-8` or `ISO-8859-1`. The client uses this information to correctly process the incoming bytes from the server.

A list of acceptable content types and character encodings is sent in the Accept HTTP header from the client to the server. This can be used to dynamically select an appropriate content type for the response, or the application developer can specify a static content type for the document.

In JSF, the `<f:view>` tag passes `null` to the `RenderKitFactory` as the list of acceptable content types, even though the JSP container is aware of the complete list accepted by the requesting browser. So, the default `RenderKit`—the standard HTML Basic `RenderKit`—must assume the content type has already been set to `text/html` since it is rendering only HTML. The `RenderKit` uses information about the content type and character encoding to create a `ResponseWriter` that can produce correctly formatted markup to the client.

For each JSF JSP tag within `<f:view>` during the initial render, a JSF component is created and attached to the component hierarchy. As you remember, the `UIViewRoot` was created in the first phase and attached to the `FacesContext`, so you can safely assume that the components will be attached to the component hierarchy. Figure 1-14 shows execution of the `<h:form>` start tag.

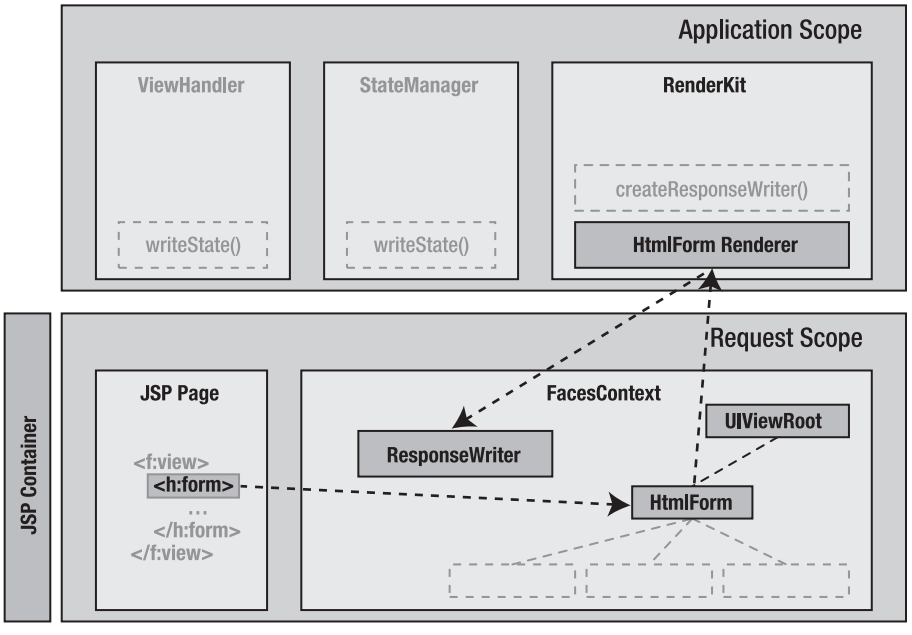

Figure 1-14. *Writing* `<form>` *start element to the* `<f:view>` *body content buffer*

In the login JSP document, the next JSF JSP tag to be executed is the `<h:form>` tag. The JSF JSP tag calls the `Application.createComponent()` method that takes a string representing the component type, for example `javax.faces.HtmlForm` (see the section "Component Family and Component Type"). The component type is mapped to a class defined in the `faces-config.xml` file, and an instance of the `HtmlForm` component is created and attached to the `UIViewRoot`. Next, a `Renderer` for the newly created component needs to be found. A `Renderer` is located by component family and renderer type, which together define a unique identifier for the `Renderer` (see the section "Renderer Types").

■ **Note** Let's use the `HtmlInputText` component to illustrate the relationship between the component family and renderer type. The `HtmlInputText` component has the component family `javax.faces.Input` and the renderer type `javax.faces.Text`. Together, they uniquely identify the appropriate `Renderer` class within the HTML Basic `RenderKit`—`javax.faces.renderer.html.HtmlInputText`.

The renderer type is already known by the `<h:form>` tag, and the component family can be located in the component's superclass, `UIForm`. The tag then calls a method called `encodeBegin()` on the component, which in turn calls the `encodeBegin()` method on the `HtmlForm` renderer. The `encodeBegin()` method on the `Renderer` calls methods on the `ResponseWriter` to write the markup for the HTML form element—`<form method="" action="">`. All markup output from the `ResponseWriter` ends up in the `<f:view>` body content buffer. Figure 1-15 shows the closing process of the `<h:form>` tag.

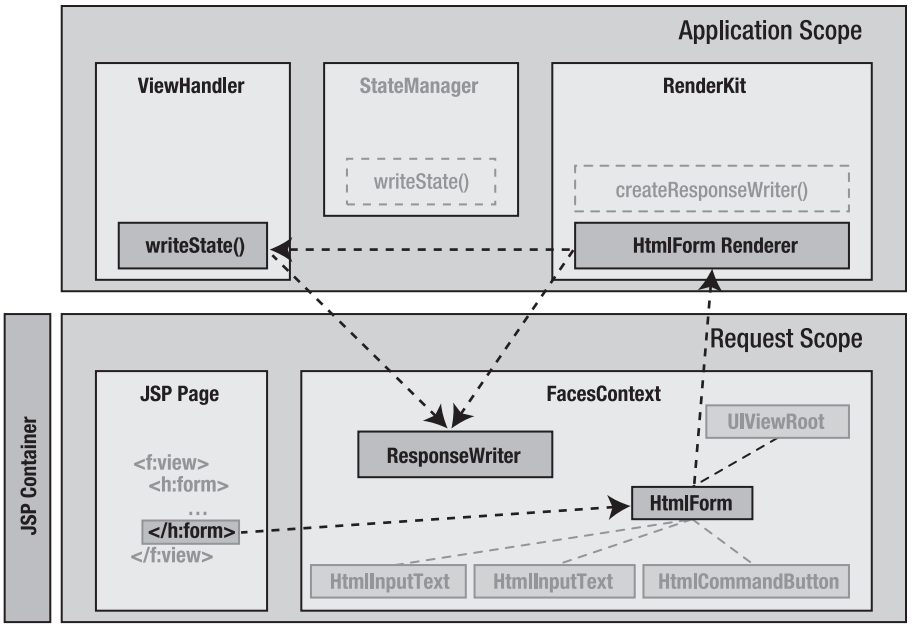

Figure 1-15. *Output token and closing the* `</form>` *element*

The process continues, and all nested components within the `HtmlForm` component are rendered and added to the `<f:view>` body content buffer. Then, the closing tag for the `<h:form>` tag is executed. The `<h:form>` tag calls the `encodeEnd()` method on the `Renderer`, `HtmlFormRenderer`, which in turn calls the `writeState()` method on the `ViewHandler`. The `writeState()` method passes a token to the `ResponseWriter`, which is added to the `<f:view>` body content buffer. The `encodeEnd()` method then calls methods on the `ResponseWriter` to write the closing tag for the HTML form element—`</form>`. Figure 1-16 shows the closing of the `<f:view>` tag.

■ **Note** The `ViewHandler` represents the view technology, and in this case the view technology is JSP. Nothing in the JSF specification prevents anyone from implementing an alternate `ViewHandler` for another view technology, such as XML.

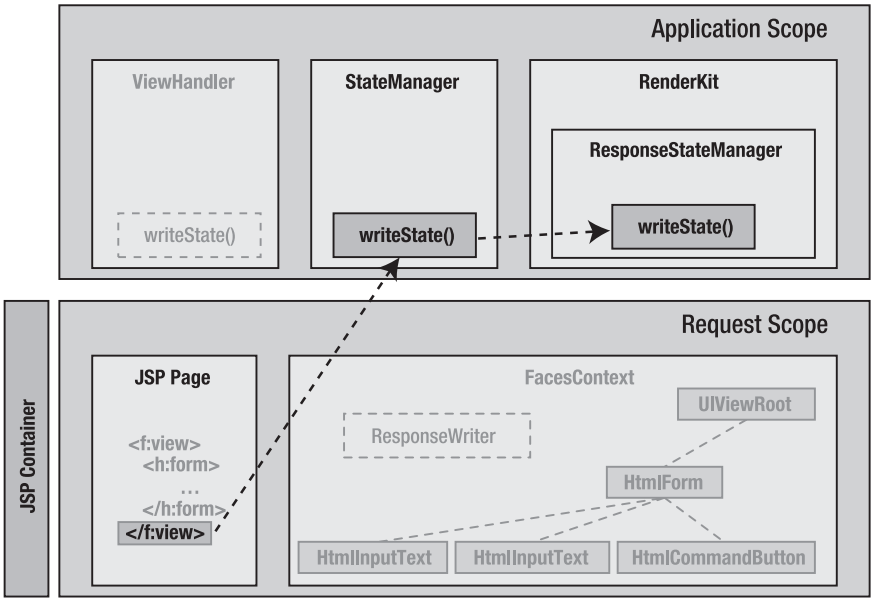

Figure 1-16. *Replacing token with serialized state and closing* `</f:view>`

By the time you get to the `</f:view>` closing tag, the entire component hierarchy is available. It is not until you have the complete tree that you can store the state of the component hierarchy representing this page of the application. The `</f:view>` end tag calls the `writeState()` method on the `StateManager`. Depending on the init parameter—`STATE_SAVING_METHOD`—for state saving (see the section "Saving and Restoring State"), the `StateManager` stores the state in the session on the server or delegates to the `ResponseStateManager` to save state on the client replacing the token with the serialized state. After the state has been saved, the buffer is flushed out to the client, and execution of any remaining non-JSF JSP tags will take place. The login page is now rendered in the browser.

Note In JSF 1.2, the `<f:view>` tag is no longer responsible for buffering the output. Instead, buffering is achieved by using a `ServletResponse` wrapper. In addition, the component hierarchy is no longer created inline during rendering. Instead, during the Render Response phase of JSF 1.2, the component hierarchy is created first and rendered next. Therefore, during rendering, the full component hierarchy is available, so the state is written directly into the buffered response, rather than needing to use a placeholder token to be replaced by the real state in `</f:view>`.

Postback Request

So far, the only thing the user has seen is the initial rendering of the first requested page. After receiving the page, the user enters a username and password and clicks the login button. A postback is performed, and you will now look at how JSF handles postback. Some parts are similar to what we have already been covering in the initial request, but there are obviously differences and additions, especially in the JSF request lifecycle. At postback, all six phases of the JSF request lifecycle get called (unless somewhere in the process the `FacesContext.renderResponse()` method is called causing the lifecycle to jump directly to the Render Response phase). This is different from the initial request where only the first and last phases are called.

Restore View Phase

The first part of a postback is the same as for the initial request; the Restore View phase executes and calls the `restoreView()` method on the `ViewHandler` to restore any state available from the previous request. Figure 1-17 shows restoring the saved state of the component hierarchy.

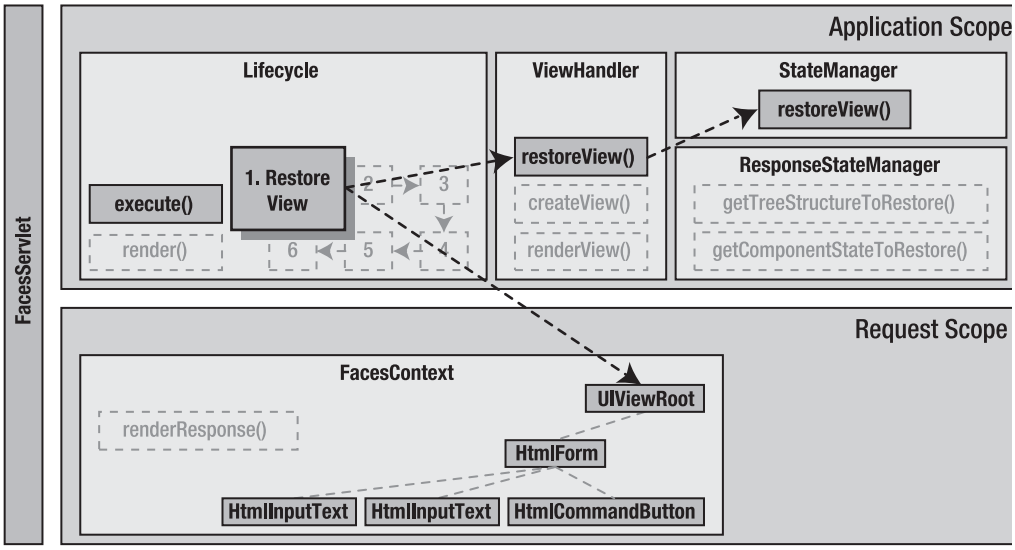

Figure 1-17. *Restoring the saved state of the component hierarchy*

Here is where the similarities end; instead of returning `null`, the `restoreView()` method will return the current state of the component hierarchy associated with a particular `viewId` and `FacesContext` from the `StateManager`, and if the init parameter—`STATE_SAVING_METHOD`—is set to client-side state-saving, call the `ResponseStateManager` to retrieve the state from the current request. The restored component hierarchy is then passed to the `FacesContext` by the Restore View phase.

Apply Request Values Phase

In the Apply Request Values phase, each input component establishes the submitted value from the request parameters, and each command component queues an event to be delivered in the Invoke Application phase. Figure 1-18 shows how the Apply Request Values phase passes new values to the components.

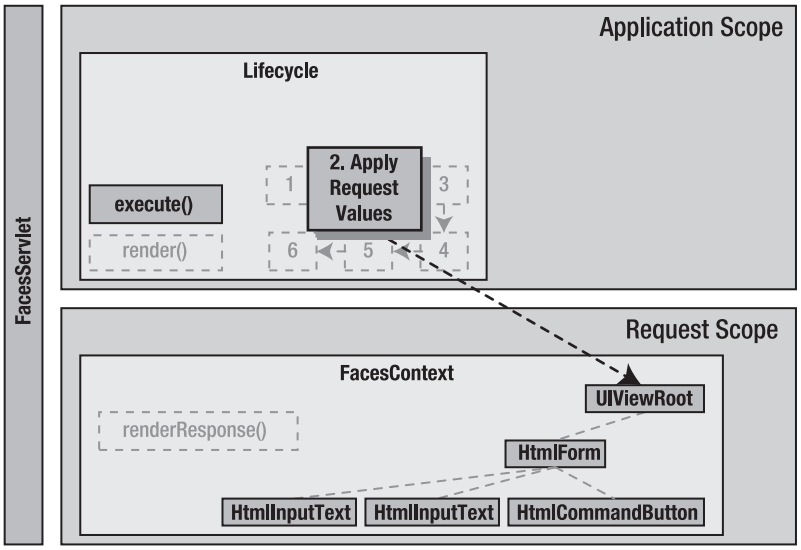

Figure 1-18. *Applying new values passed on the request to the components*

The submitted value is at this point stored only as "submitted" on the component, and no value has been pushed into the underlying model yet. By the time the Apply Request phase is completed, the `Renderers` no longer need to observe the request parameters, since all values have been updated on each component.

Process Validation Phase

In the Process Validations phase, conversion and validation are performed by calling the `processValidators()` method on the `UIViewRoot`. Figure 1-19 shows conversion and validation.

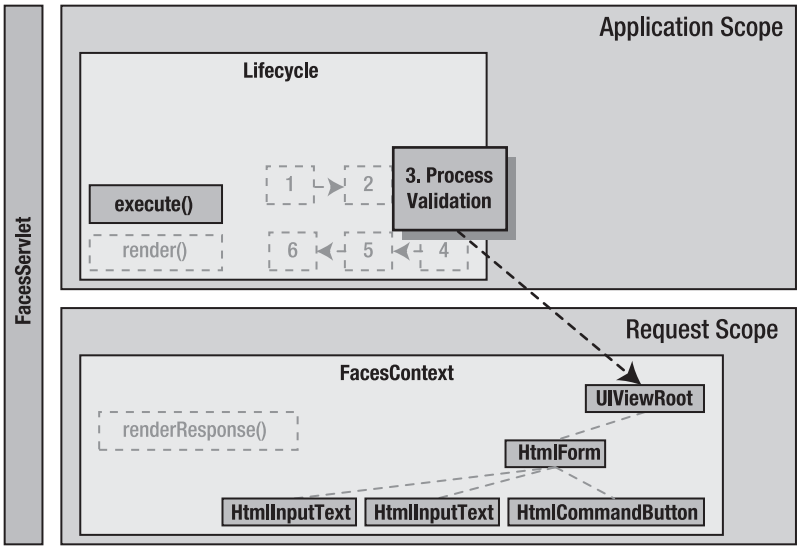

Figure 1-19. *Performing conversion and validation*

This process will continue calling the `processValidators()` method recursively on each component in the component hierarchy. During validation of each `HtmlInputText` component, type conversion will occur first on the component's submitted value (for example, a string to a strongly typed object). The new object is set as a local value on the component, and the submitted value is cleared. The new strongly typed object is then validated. If there are no errors, then the next step is to queue a `ValueChangeEvent` that will be delivered at the end of the Apply Request Values phase.

If a conversion or validation error occurs, a corresponding JSF message is attached to the `FacesContext` using the component `clientId`, and then the `renderResponse()` method is called to indicate that the lifecycle should skip directly to the Render Response phase after the Process Validations phase is complete.

Update Model Phase

At this point in the lifecycle, all submitted values have been successfully converted and validated, so it is safe to push them into the underlying data model. During the Update Model phase, the JSF lifecycle walks over the component hierarchy, calling the `processUpdates()` method on each component. Figure 1-20 shows the Update Model phase updating the underlying model.

To determine where to store the new value, the `processUpdates()` method will use the value binding, which is defined in the `value` attribute on the component (for example, `#{credentials.username}`). The value binding points to a property on a managed bean (for example, `username`). Using the value binding, the locally stored value on the component is pushed into the data model, and the locally stored value on the component is cleared.

Any JSF messages and errors on the model—for example, validations implemented by the model—are attached to the `FacesContext` with the component's client ID. The `renderResponse()` method is then called to indicate that the lifecycle should skip directly to the Render Response phase after the Update Model phase is complete.

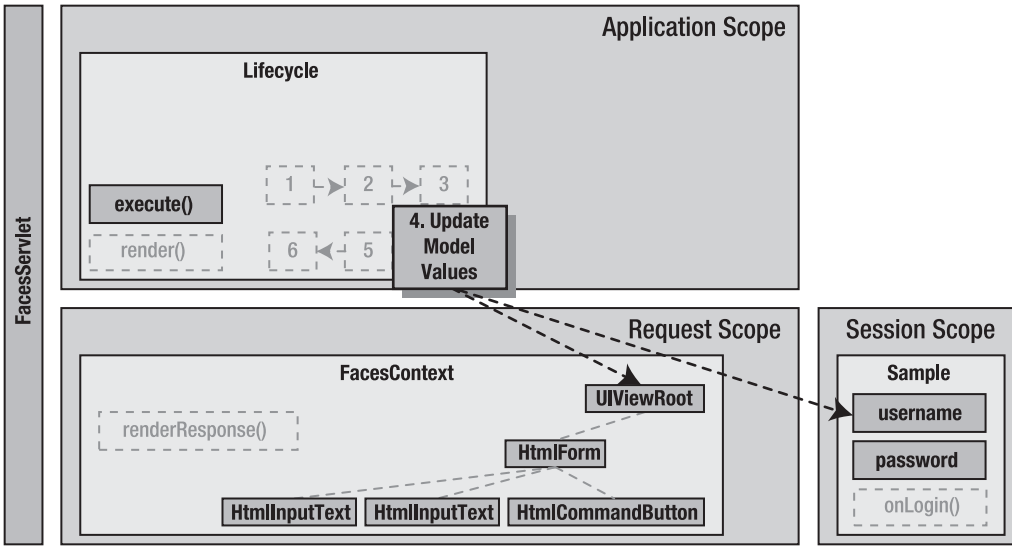

Figure 1-20. *Updating underlying model*

Invoke Application Phase

In the Invoke Application phase, you have no need to walk the component hierarchy, since this phase will handle only the queued events from previous phases and, depending on the outcome, will either continue to the last phase—Render Response—or redirect to another page. Figure 1-21 shows the broadcasting of events queued for this phase and the processing of action method bindings.

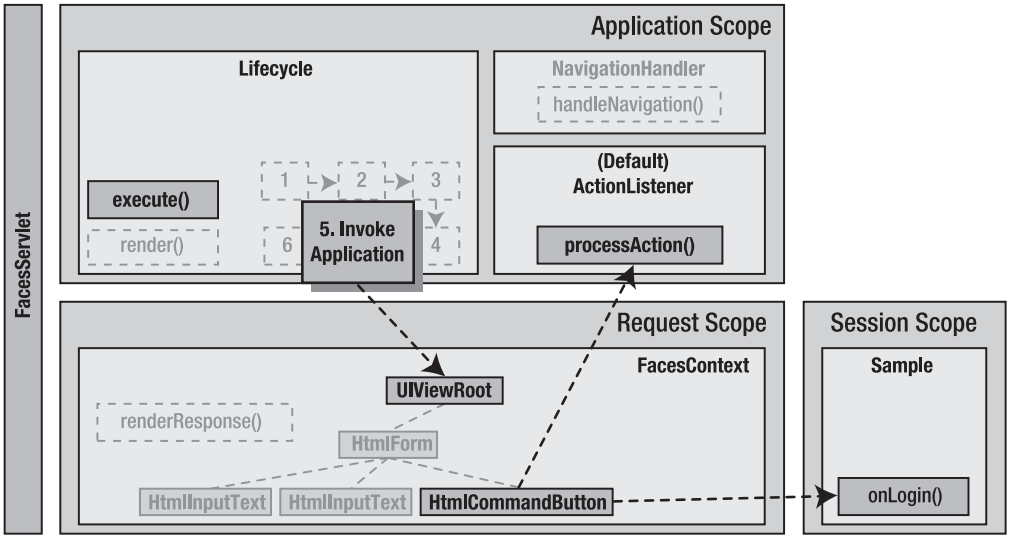

Figure 1-21. *Performing application logic*

As mentioned in the earlier section "Converters, Validators, Events, and Listeners," you have two methods that will be processed when an `ActionEvent` occurs. The first thing that happens is a call to the `processApplication()` method on the `UIViewRoot` that takes each queued event and broadcasts to the target component for the event (for example, `commandButton.broadcast(FacesEvent)`). The `UICommand` component knows about the `action` and `actionListener` attributes, as well as the default `ActionListener` attached to the `Application` object.

First all previously registered `ActionListeners` are called, then the `actionListener` method binding is executed (for example, `#{credentials.onLogin}`), and finally the component calls the `processAction()` method on the default `ActionListener` to process the action method binding and handle navigation. It is important that the action method binding is called at the end of this process, since it defines possible navigation, and you don't want to navigate before you have processed all events.

Postback with Navigation

When the default `ActionListener` is processing an `ActionEvent`, it invokes the action method binding and gets the `outcome`, which is a `String` object. If the `outcome` returns `null`, then the default `ActionListener` will continue with the next queued event. After all events have been broadcast, the Invoke Application phase is complete and lifecycle processing continues to the last phase—Render Response. If the `outcome` is not `null`, then the default `ActionListener` passes `FacesContext`, `fromAction` (which is the method-binding expression text—for example, `credentials.doLogin`), and `outcome` to the `NavigationHandler`. Figure 1-22 shows navigation on postback.

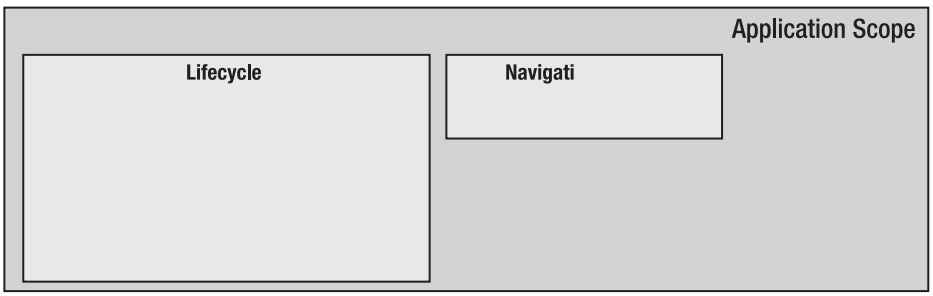

Figure 1-22. *Navigation on postback*

```
  _scroll(0);

  if (parsedDate != null)
    _select(parsedDate.getDate());
}
```

The _scroll() Function

The `_scroll()` function, as shown in Code Sample 9-7, allows the users to navigate plus or minus one month using arrow controls in the calendar. It is also here you use Mabon to determine the availability of dates defined by the managed bean attached to the `HtcAjaxInputDate` component. Figure 9-4 shows the sequence of function calls in the `_scroll()` function.

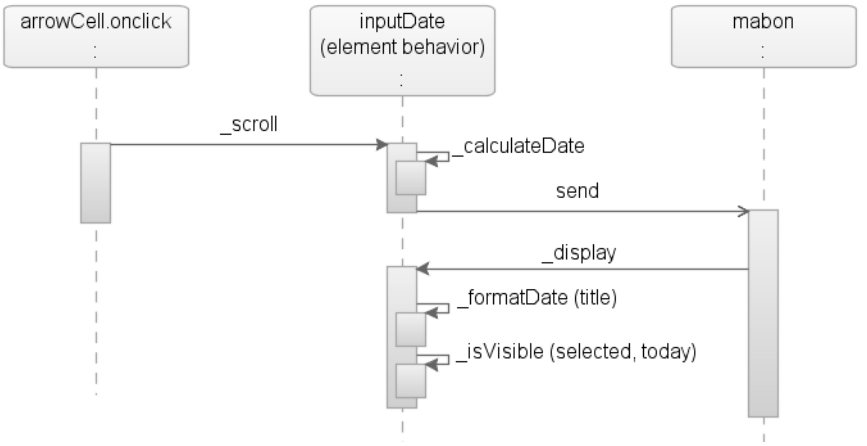

Figure 9-4. *HTC* `<pro:inputDate> _scroll()` *function*

Code Sample 9-7. *The* `_scroll` *Function in the HTC File*

```
function _scroll(offset)
{
  // scroll months, updating year as necessary
  internalState._currentMonth = internalState._currentMonth + offset;
  internalState._currentYear += Math.floor(internalState._currentMonth / 12);
  internalState._currentMonth = (internalState._currentMonth + 12) % 12;

  // use Mabon to retrieve availability
  if (element.targetURL)
  {
    var startDate = _calculateDate(1);
    var endDate = _calculateDate(31);
```

```
    var millisPerDay = 1000 * 60 * 60 * 24;
    var startDay = Math.floor(startDate.getTime() / millisPerDay);
    var endDay = Math.floor(endDate.getTime() / millisPerDay);

    // use Mabon to determine availability
    mabon.send(
      {
        url: element.targetURL,
        args: [startDay, endDay],
        callback: function(result) { _display(result); }
      });
  }
  else
  {
    var available = [];
    for (var i=0; i < 32; i++)
    {
      available.push(true);
    }
    _display(available);
  }
}
```

The _clickCell() Function

The `_clickCell()` function, as shown in Code Sample 9-8, is called when the user clicks a cell representing a date in the calendar. Figure 9-5 shows the sequence of function calls in the `_clickCell()` function.

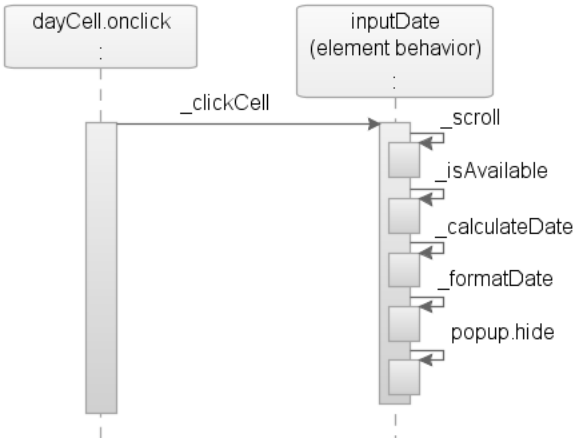

Figure 9-5. *HTC* `<pro:inputDate>` `_clickCell` *function*

You can obtain the target node invoking the event by calling `event.srcElement` (see Code Sample 9-8). When you have the target node, you can check to see whether the user clicked a cell that is outside the range of the displayed month and, if so, navigate to the month for that selected date: `_scrollNext()` or `_scrollPrev()`. If the selection is within the boundaries of the month, you need to see whether this date is available; if it is, add the selected date to the input element.

Code Sample 9-8. *The* `_clickCell()` *Function*

```
function _clickCell()
{
  var event = popup.document.parentWindow.event
  var cellNode = event.srcElement;
  var rowNode = cellNode.parentNode;

  var row = rowNode.sectionRowIndex;
  var col = cellNode.cellIndex;
  var day = Number(cellNode.innerText);

  if (row == -1)
  {
    return;
  }
  else if (row == 0 && day > 7)
  {
    _scrollPrev();
  }
  else if (row > 3 && day < 15)
  {
    _scrollNext();
  }
  else
  {
    if (_isAvailable(day))
    {
      var selectedDate = _calculateDate(day);
      input.value = _formatDate(selectedDate, element.pattern);

      // flush the changes for next postback
      _flushChanges();

      popup.hide();
    }
  }
}
```

The HTML Deck Implementation Prototype

Figure 9-6 shows a page that includes the <pro:showOneDeck> prototype implemented in HTC.

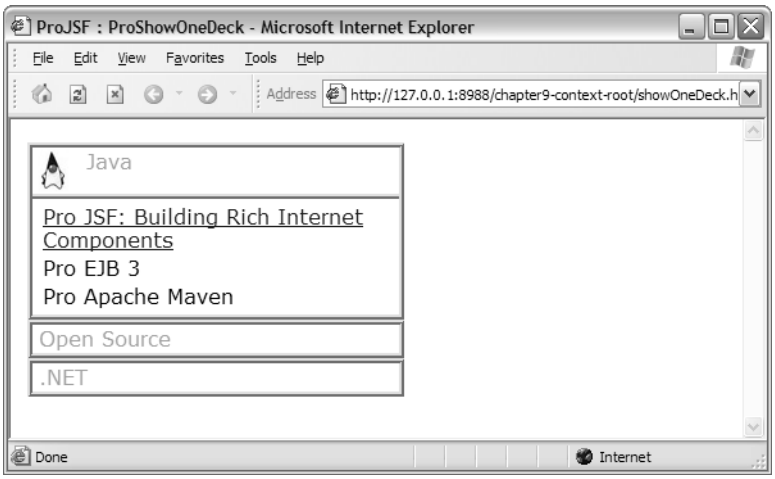

Figure 9-6. *The* <pro:showOneDeck> *component implemented in HTML and HTC*

Code Sample 9-9 shows the markup needed to create a page using the HTC <pro:showOneDeck> prototype shown in Figure 9-6.

Code Sample 9-9. *Markup to Create a Page Using the* <pro:showOneDeck> *HTC Prototype*

```
<html xmlns:pro="http://projsf.apress.com/tags" >
  <head>
    <title>ProJSF : ProShowOneDeck</title>
    <link rel="stylesheet" href="/.../resources/stylesheet.css"/>  </head>
  <body>
    <form id="form" method="post" >
      ...
        <?import namespace="pro"
                implementation="/.../projsf-ch9/showItem.htc" ?>
        <pro:showOneDeck id="form:showOneDeck" style="display:block;">
          <pro:showItem itemId="first"
                        active="true"
                        style="display:block;">
            <pro:headerFacet>
              <img src="/.../resources/java_small.jpg"
                   alt="The Duke"
                   style="margin-right: 8px; vertical-align:bottom;" />
              Java
            </pro:headerFacet>
```

```
            <table>
              <tbody>
                <tr>
                  <td>
                    <a href="http://apress.com/book/bookDisplay.html?bID=10044">
                    Pro JSF: Building Rich Internet Components
                    </a>
                  </td>
                </tr>
      ...
    </form>
  </body>
</html>
```

First you need to define the namespace prefix `pro` in the `<html>` element, which allows you to import and bind the element behavior (`showItem.htc`) to a specific tag name in the `pro` namespace. The tag name is defined inside the HTC component; once the HTC component is imported, you can use the element behavior in the page with the prefix `pro` (for example, `<pro:showItem>`).

The HTC Deck Element Behavior

The structure of the `<pro:showOneDeck>` component is slightly different from the `<pro:inputDate>` component, since it is of a composite nature. If you look at what is needed to create the deck component in a JSP page, you need three JSP tag handlers: `<pro:showOneDeck>`, `<pro:showItem>`, and `<f:facet name="header">`. `<pro:showOneDeck>` is a container for `<pro:showItem>` and defines which `<pro:showItem>` should be expanded by default. `<pro:showItem>` is also a container and defines what should be displayed when interacted with. `<f:facet name="header">` defines the clickable header of `<pro:showItem>`.

The problem you are facing is that HTC does not recognize JSF facets, and therefore you will have to come up with a way to define facets using HTC syntax. Also, HTC components are treated as encapsulated documents; for example, an HTC file is basically an HTML file parsed by Internet Explorer.

The first obvious approach is to define one HTC element behavior for each JSF tag (that is, `showOneDeck.htc`, `showItem.htc`, and `headerFacet.htc`), but from an HTC view only one will include actual behavior—`showItem`. The `<pro:showItem>` component is the one expanding and collapsing, not the `<pro:showOneDeck>` or the `<pro:headerFacet>`; thus, you should create only one HTC element behavior: `showItem.htc`.

Code Sample 9-10 shows the `<body>` section of the HTC file, `showItem.htc`, since that will give you an understanding of how this component is constructed.

Code Sample 9-10. *The* `<body>` *of the* `<pro:showItem>` *Component*

```
<html>
  ...
  <body style="display:block" class="showItem" >
    <div id="header" ></div>
    <div id="content" style="display:none;" ></div>
```

```
  </body>
</html>
```

This is a simple component in its structure. It contains only two elements: one for the header of the <pro:showItem> and one for the actual content of the <pro:showItem>. You leverage CSS to make sure that elements outside the <pro:showItem> will wrap properly by setting the style attribute on the <body> element to display:block; in other words, other <pro:showItem> components will be stacked either above or below. You also set the content element's style to display:none by default. This value causes the element to take no space at all in the browser.

■ **Note** For more information about visual formatting using block boxes, please visit the W3C Web site at http://www.w3.org/TR/REC-CSS2/visuren.html#initial-containing-block.

In the <head> section of the HTC file, as shown in Code Sample 9-11, you define the element behavior prototype using the HTC-specific <public:component> element.

Code Sample 9-11. *The* <head> *Element in the* <pro:showItem> *HTC Component*

```
<head>
  <public:component tagName="showItem" >
    <public:property name="itemId" />
    <public:property name="styleClass" />
    <public:property name="headerStyleClass" />
    <public:property name="contentStyleClass" />
    <public:property name="active" />

    <public:attach event="oncontentready" handler="_constructor" />
  </public:component>
```

The enclosing <public:component> element has the tagName attribute set to showItem. The tagName attribute specifies the name of the custom tag.

Also, five public properties (itemId, styleClass, headerStyleClass, contentStyleClass, and active) are defined on the element behavior. The previous code sample uses the <public:attach> element to bind an event handler, _constructor, to the HTC-specific event oncontentready. The _constructor() function will initialize the internal state of the element behavior.

The _constructor() Function

The _constructor() function, as shown in Code Sample 9-12, is the core piece of the HTC element behavior. The oncontentready event will fire when the content of the <pro:showItem> element, to which the behavior is attached, has been parsed completely. This will invoke the _constructor() function, which will set the internal state of the <pro:showItem> tag based on the content written to the browser.

Code Sample 9-12. *The* `_constructor()` *Function*

```
<script type="text/javascript" >
      function _constructor()
      {
        header.className = (element.headerStyleClass || 'showItemHeader');
        header.onclick = _expand;

        for (var i=0; i < childNodes.length; i++)
        {
          var childNode = childNodes[i];
          if (childNode.scopeName == 'pro' &&
              childNode.nodeName == 'headerFacet')
          {
            // set header on showItem
            header.innerHTML = childNode.innerHTML;
          }
          else
          {
            // set content inside showItem
            switch (childNode.nodeType)
            {
              case 1: // Element
                content.insertAdjacentHTML("beforeEnd", childNode.outerHTML);
                break;
              case 3: // Text
                content.insertAdjacentHTML("beforeEnd", childNode.nodeValue);
                break;
            }
          }
        }

        // show the contents if active
        if (element.active == 'true')
        {
          content.className = (element.contentStyleClass || 'showItemContent');
          content.style.display = 'block';
        }

        defaults.viewLink = document;
      }
```

The source of the `_constructor()` function is simple. You first set the `onclick` event handler on the HTC component's header to use the `_expand` function so that this function will be invoked whenever the header is clicked. You then loop over all children of the `<pro:showItem>` tag listed in the parent document. If a child node is a `<pro:headerFacet>`, you then add its `innerHTML` to the header element in the HTC component body; otherwise, you add it to the

content element. If the child node is being added to the content element, you have to check to see whether it is another element or just plain text.

If the `active` attribute on the `<pro:showItem>` tag is set to `true`, the content of the `<pro:showItem>` element behavior will be displayed. Before the `_constructor()` is done executing and the content of the HTC element behavior is displayed in the browser, you have to create a `viewLink` between the root element of the document fragment in the HTC file to the master element, `<pro:showItem>`, in the primary document. You can define a `viewLink` using a script shown previously or by inserting the appropriate declaration in the component section of the HTC file (see Code Sample 9-4).

■ **Note** The `insertAdjacentHTML` method is specific to Internet Explorer and appends the given HTML to the HTML content of the DOM element. The first argument on the `insertAdjacentHTML` method takes one of four string values: `beforeBegin`, `afterBegin`, `beforeEnd`, and `afterEnd`. The `beforeEnd` string tells the method to insert the HTML markup immediately before the end of the DOM element, after all the other content in the DOM element.

The _expand() Function

As shown in Figure 9-7, when the element behavior is bound to the `pro` namespace and attached to the `<pro:showItem ...>` tag, an `_expand()` function, as shown in Code Sample 9-13, is added as the event handler and will be invoked when the header is clicked.

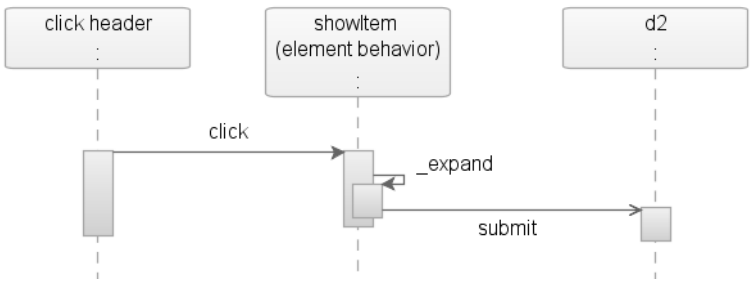

Figure 9-7. *HTC* `<pro:showItem>` `_expand` *function*

Code Sample 9-13. *The* `_expand()` *Function*

```
function _expand()
{
  var showOneNode = element.parentNode;
  var showOneClientId = showOneNode.id;

  var currentNode = element;
  while (currentNode != null)
```

```
    {
      var method = currentNode.method;

      if (method != null &&
          (method.toLowerCase() == 'get' ||
           method.toLowerCase() == 'post'))
      {

        //The following function call to d2 is needed to perform
        //an Ajax postback when implemented in the JSF HTC
        //ProShowOneDeck component.
        var formNode = currentNode;
        var content = new Object();
        content[showOneClientId] = element.itemId;
        d2.submit(formNode, content);
        break;
      }

      currentNode = currentNode.parentNode;
    }
  }
</script>
```

In the prototype you are constructing only the UI, but for Code Sample 9-13 the `d2.submit()` function has been added, passing the activated form `id` and the `id` of the selected node to the `d2.submit()` function. The `d2.submit()` function calls the `dojo.io.bind()` method, passing information about what form to submit, content (that is, the ID of the selected component), the accepted request header (`'X-D2-Content-Type': <contentType>`), and the MIME type (`text/plain`) for this request.

This information will determine which node to expand and which `ResponseWriter` to use for this request in your JSF implementation of the JSF HTC deck component.

The D^2 library also defines a callback function, `d2._loadtext`, that is used to get the response data from the server. The `d2._loadtext` function will replace the target document with the document returned on the response. This will cause the HTC `<pro:showItem>` component to invoke the `_constructor()` again and cause the `<pro:showItem>` to be updated with new content sent from the server.

To be able to asynchronously communicate with the server when a deck is activated, you also have to download a set of JavaScript libraries (`dojo.js` and `d2.js`) to the page. These libraries need to be part of your component and should be downloaded automatically to the client on initial request.

Step 5: Creating a Client-Specific Renderer

Your HTC solution contains three new `Renderer` classes: `HtmlDocumentRenderer`, `HtcAjaxInputDateRenderer`, and `HtcAjaxShowOneDeckRenderer`. Let's start by looking at the `HtmlDocumentRenderer` class.

The HtmlDocumentRenderer Class

The `HtmlDocumentRenderer` class (see Figure 9-8) is basically a port of the `XulDocumentRenderer` you created in Chapter 8. By porting the `XulDocumentRenderer` code to an HTML version, you can now provide application developers with one complete solution, allowing the component writer to switch the `RenderKit` without any changes to the actual application page description.

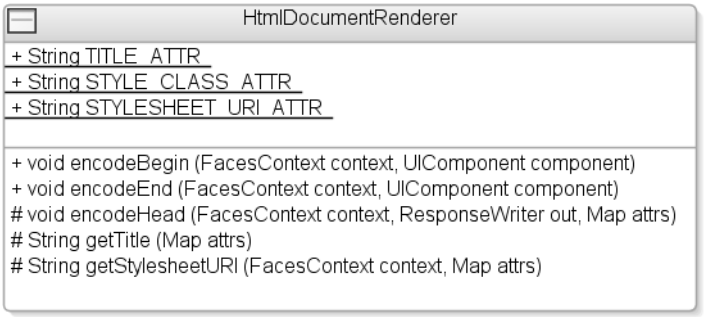

Figure 9-8. *Class diagram showing the* `HtmlDocumentRenderer` *class*

Another freebie is the "at-most-once" semantics for script resources you get when extending the `HtmlRenderer`.

The only requirement that the HTML document `Renderer` has is to support running applications on different clients, without forcing application developers to provide different solutions for each client. The `HtmlDocumentRenderer`, a top-level component that controls the root element rendered to the client, provides enormous possibilities and provides total control over the markup for the component writer.

You can use `HtmlDocumentRenderer` as the root component by any `Renderer` (for example, `HtmlInputDateRenderer`, `HtmlAjaxInputDateRenderer`, and `HtcAjaxInputDateRenderer`) that is targeting HTML as the default markup. Code Sample 9-14 shows the `encodeBegin()` method of the `HtmlDocumentRenderer`.

Code Sample 9-14. *The* `HtmlDocumentRenderer` `encodeBegin()` *Method*

```
package com.apress.projsf.ch9.render.html.basic;

import java.io.IOException;
import java.util.Map;

import javax.faces.application.Application;
import javax.faces.application.ViewHandler;
import javax.faces.component.UIComponent;
import javax.faces.context.FacesContext;
import javax.faces.context.ResponseWriter;

import com.apress.projsf.ch2.render.html.HtmlRenderer;
```

```
public class HtmlDocumentRenderer extends HtmlRenderer
{
  /**
   * The title attribute.
   /
  public static String TITLE_ATTR = "title";

  /*
   * The styleClass attribute.
   /
  public static String STYLE_CLASS_ATTR = "styleClass";

  /*
   * The stylesheetURI attribute.
   */
  public static String STYLESHEET_URI_ATTR = "stylesheetURI";

  public void encodeBegin(
    FacesContext context,
    UIComponent  component) throws IOException
  {
    ResponseWriter out = context.getResponseWriter();
    Map attrs = component.getAttributes();
    String styleClass = getStyleClass(attrs);

    out.startElement("html", component);
    out.startElement("head", null);
    encodeHead(context, out, attrs);
    out.endElement("head");
    out.startElement("body", null);
    if (styleClass != null)
      out.writeAttribute("class", styleClass, STYLE_CLASS_ATTR);
  }
```

The `encodeBegin()` method takes two arguments: `FacesContext context` and `UIComponent component`. From the component, you can obtain a `Map` containing all the available attributes. In this case, the application developer can set three attributes in his JSP document or backing bean (the `title`, the `styleClass` and the `stylesheetURI`), and the attribute map is passed as an argument to the `encodeHead()` method.

The `startElement()` method takes the following arguments: `name` and `component`. The `name` argument is the name of the element generated (for example, `html`), and the `component` argument is the `UIComponent` that this element represents. In Code Sample 9-14, this is represented with the component `UIDocument`.

The `encodeEnd()` method, as shown in Code Sample 9-15, is basically just closing the HTML `<body>` and `<html>` tags.

Code Sample 9-15. *The* `encodeEnd()` *Method*

```
public void encodeEnd(
  FacesContext context,
  UIComponent  component) throws IOException
{
  ResponseWriter out = context.getResponseWriter();
  out.endElement("body");
  out.endElement("html");
}
```

The `encodeHead()` method, as shown in Code Sample 9-16, is responsible for writing out the <head> element. The <head> element contains information about the document, and in this case it is the title and style sheet.

Code Sample 9-16. *The* `encodeHead()` *Method*

```
protected void encodeHead(
  FacesContext   context,
  ResponseWriter out,
  Map            attrs) throws IOException
{
  String title = getTitle(attrs);
  String stylesheetURI = getStylesheetURI(context, attrs);

  if (title != null)
  {
    out.startElement("title", null);
    out.writeText(title, TITLE_ATTR);
    out.endElement("title");
  }
  if (stylesheetURI != null)
  {
    out.startElement("link", null);
    out.writeAttribute("rel", "stylesheet", null);
    out.writeAttribute("href", stylesheetURI, STYLESHEET_URI_ATTR);
    out.endElement("link");
  }
}
```

The `getTitle()` and `getStylesheetURI()` methods, as shown in Code Sample 9-17, return the values of the `title` and `stylesheetURI` attributes.

Code Sample 9-17. *The Getters for the* `UIDocument` *Attributes*

```
  protected String getTitle(
    Map attrs)
```

```
  {
    return (String)attrs.get(TITLE_ATTR);
  }

  protected String getStyleClass(
    Map attrs)
  {
    return (String)attrs.get(STYLE_CLASS_ATTR);
  }

  protected String getStylesheetURI(
    FacesContext context,
    Map          attrs)
  {
    String stylesheetURI = (String)attrs.get(STYLESHEET_URI_ATTR);

    if (stylesheetURI != null)
    {
      Application application = context.getApplication();
      ViewHandler handler = application.getViewHandler();
      stylesheetURI = handler.getResourceURL(context, stylesheetURI);
    }

    return stylesheetURI;
  }
}
```

The HtcAjaxInputDateRenderer Class

You already know that XUL can make a JSF component writer's life easier, but how about HTC? Microsoft's HTC components provide a similar level of abstraction as Mozilla's XUL/XBL. So, without further ado, let's look at the JSF HTC implementation (see Figure 9-9).

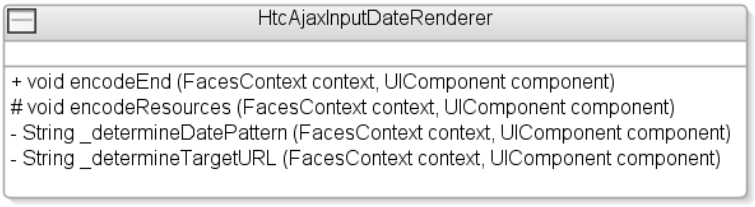

Figure 9-9. *Class diagram showing the* `HtcAjaxInputDateRenderer` *class*

Even in the HtcAjaxInputDateRenderer, you will recognize most of the code from previous chapters, except that the actual output to the client is a mix of regular HTML and HTC. As with XUL, HTC allows you to reuse the UI prototype. By adding the element behavior—the inputDate.htc prototype file—to your resources, the only element you need to write out for this JSF HTC component is <pro:inputDate ...> and its attributes.

By extending the HtmlInputDateRenderer, you get access to the writeScriptInline() method, as shown in Code Sample 9-18. The writeScriptInline() method provides the same "at-most-once" semantics for inline scripts as the writeScriptResource() method does for external resources. An application developer might add two or more ProInputDate components to the page, but the semantics behind the writeScriptInline() method, provided by the Renderer implementation, will make sure this inline script is written only once.

The writeScriptInline() method writes out a script that will add a namespace (http://projsf.apress.com/tags) and will set the namespace prefix to pro. When the namespace is added, you can import the element behavior (inputDate.htc) and attach it to the namespace prefix.

Code Sample 9-18. *The* HtcAjaxInputDateRenderer

```
package com.apress.projsf.ch9.render.htc.ajax;

import java.io.IOException;

import java.text.DateFormat;
import java.text.SimpleDateFormat;

import javax.faces.application.Application;
import javax.faces.application.ViewHandler;
import javax.faces.component.UIComponent;
import javax.faces.component.UIInput;
import javax.faces.context.FacesContext;
import javax.faces.context.ResponseWriter;
import javax.faces.convert.Converter;
import javax.faces.convert.DateTimeConverter;
import javax.faces.el.MethodBinding;
import javax.faces.validator.Validator;

import com.apress.projsf.ch5.render.html.basic.HtmlInputDateRenderer;
import com.apress.projsf.ch7.validate.DateValidator;

public class HtcAjaxInputDateRenderer extends HtmlInputDateRenderer
{
  protected void encodeResources(FacesContext context,
                                 UIComponent component) throws IOException
  {
    super.encodeResources(context, component);
```

```
    ViewHandler handler = context.getApplication().getViewHandler();
    String behaviorURL = handler.getResourceURL(context,
                              "weblet://com.apress.projsf.ch9/inputDate.htc");
    writeScriptInline(context,
                      "document.namespaces
                               .add('pro','http://projsf.apress.com/tags');\n" +
                      "document.namespaces
                               .item('pro').doImport('" + behaviorURL + "');
                      ");
  }
```

By design, the <pro:inputDate> component can have a Converter added by a JSP tag. At initial render, during the creation of the component hierarchy, a custom JSP converter tag has not yet been executed, so the Converter is not yet attached to the component inside the encodeBegin() method. Instead, the Renderer is using the encodeEnd() method, as shown in Code Sample 9-19, to write out the markup and to obtain the Converter.

Code Sample 9-19. *The* encodeEnd() *Method*

```
public void encodeEnd(
  FacesContext context,
  UIComponent  component) throws IOException
{
  String pattern = _determineDatePattern(context, component);
  String targetURL = _determineTargetURL(context, component);

  UIInput input = (UIInput)component;
  String valueString = (String)input.getSubmittedValue();

  if (valueString == null)
  {
    Object value = input.getValue();
    if (value != null)
    {
      Converter converter = getConverter(context, input);
      valueString = converter.getAsString(context, component, value);
    }
  }

  String clientId = input.getClientId(context);

  ResponseWriter out = context.getResponseWriter();
  out.startElement("pro:inputDate", component);
  out.writeAttribute("id", clientId, null);
  if (valueString != null)
    out.writeAttribute("value", valueString, null);
  if (pattern != null)
```

```
    out.writeAttribute("pattern", pattern, null);
  if (targetURL != null)
    out.writeAttribute("targetURL", targetURL, null);
  out.endElement("pro:inputDate");
}
```

In `encodeEnd()`, you call two methods—`_determineDatePattern()` and `_determineTargetURL()`. These methods obtain the date format pattern, and the target URL for the managed bean is bound to the `Validator`. Finally, you write out the `<pro:inputDate>` component with its attribute to the client.

The `_determineDatePattern()` method, as shown in Code Sample 9-20, is identical to the one you used in both the XUL Ajax and HTML Ajax solutions, and you could have created a base class, or utility class, for any custom `Renderer` that might need this method. But for educational purposes we decided that it is easier to understand when it is explained this way.

For the HTC implementation to work, you need to know what date pattern has been set on the `DateTimeConverter` by the application developer. This date pattern will be used in two places. First, it parses the date entered by the user in the `<input>` element. This parsed date will then be used to set the selected date in the calendar. Second, it makes sure the date selected in the calendar follows the correct date format when added to the `<input>` element.

Code Sample 9-20. *The* `_determineDatePattern()` *Method*

```
private String _determineDatePattern(
  FacesContext context,
  UIComponent  component)
{
  UIInput input = (UIInput)component;
  Converter converter = getConverter(context, input);

  if (converter instanceof DateTimeConverter)
  {
    DateTimeConverter dateTime = (DateTimeConverter)converter;
    return dateTime.getPattern();
  }
  else
  {
    SimpleDateFormat dateFormat = (SimpleDateFormat)
                                  DateFormat.getDateInstance(DateFormat.SHORT);
    return dateFormat.toPattern();
  }
}
```

You may have seen the `_determineTargetURL()` method, as shown in Code Sample 9-21, in previous chapters. It provides you with the needed binding reference to the managed bean. You first get all the validators attached to this input component. You then check to see whether one or many of these validators are an instance of the `DateValidator`. (The `DateValidator` was created in Chapter 7.)

Code Sample 9-21. *The* `_determineTargetURL()` *Method*

```
  private String _determineTargetURL(
    FacesContext context,
    UIComponent  component)
  {
    UIInput input = (UIInput)component;
    Validator[] validators = input.getValidators();

    for (int i=0; i < validators.length; i++)
    {
      if (validators[i] instanceof DateValidator)
      {
        DateValidator validateDate = (DateValidator)validators[i];
        MethodBinding binding = validateDate.getAvailability();
        if (binding != null)
        {
          String expression = binding.getExpressionString();
          // #{backingBean.methodName} -> backingBean.methodName
          String bindingRef = expression.substring(2, expression.length(): 1);

          Application application = context.getApplication();
          ViewHandler handler = application.getViewHandler();
          return handler.getResourceURL(context, "mabon:/" + bindingRef);
        }
      }
    }

    return null;
  }
}
```

If it is an instance of the `DateValidator`, you check to see whether you have a `MethodBinding`. If a `MethodBinding` exists, you get the expression (for example, `#{managedBean.methodName}`) and strip off the `#{}`. This leaves you with `managedBean.methodName`, which you concatenate with `mabon:/`. The `MabonViewHandler` will recognize the string and return a resource URL that will be written to the client (for example, `/context-root/mabon-servlet-mapping/managedBean.methodName`).

The HtcAjaxShowOneDeckRenderer Class

You are getting close! Since the `UIShowOne` component is a container component, it needs to render its children, and as such you have to implement `encodeBegin()`, `encodeChildren()`, and `encodeEnd()` in the new renderer (see Figure 9-10).

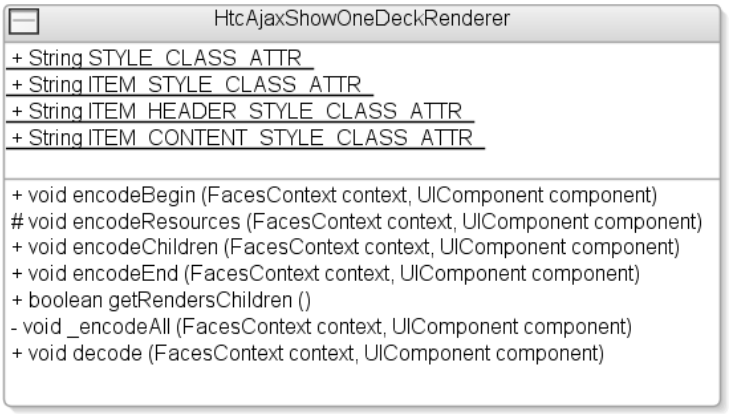

Figure 9-10. *Class diagram showing the* `HtcAjaxShowOneDeckRenderer` *class*

We'll first cover the `encodeBegin()` method for the `HtcAjaxShowOneDeckRenderer`, as shown in Code Sample 9-22. The `encodeBegin()` method takes two arguments: `FacesContext` and `UIComponent`. The Render Response phase will call `encodeBegin()` on the `UIShowOne` component, which in turn will delegate to the `encodeBegin()` method on the `HtcAjaxShowOneDeckRenderer`, passing the `FacesContext` and the `UIShowOne` component instance.

Code Sample 9-22. *The* `HtcAjaxShowOneDeckRenderer` `encodeBegin()` *Method*

```
package com.apress.projsf.ch9.render.htc.ajax;

import java.io.IOException;
import java.util.Iterator;
import java.util.List;
import java.util.Map;

import javax.faces.application.ViewHandler;
import javax.faces.component.UIComponent;
import javax.faces.context.ExternalContext;
import javax.faces.context.FacesContext;
import javax.faces.context.ResponseWriter;

import com.apress.projsf.ch2.render.html.HtmlRenderer;
import com.apress.projsf.ch3.component.UIShowItem;
import com.apress.projsf.ch3.component.UIShowOne;
import com.apress.projsf.ch3.event.ShowEvent;

public class HtcAjaxShowOneDeckRenderer extends HtmlRenderer
{
```

```
  /**
   * The styleClass attribute.
   */
  public static String STYLE_CLASS_ATTR = "styleClass";

  /**
   * The itemStyleClass attribute.
   */
  public static String ITEM_STYLE_CLASS_ATTR = "itemStyleClass";

  /**
   * The itemHeaderStyleClass attribute.
   */
  public static String ITEM_HEADER_STYLE_CLASS_ATTR = "itemHeaderStyleClass";

  /**
   * The itemContentStyleClass attribute.
   */
  public static String ITEM_CONTENT_STYLE_CLASS_ATTR = "itemContentStyleClass";

  public void encodeBegin(
    FacesContext context,
    UIComponent component) throws IOException
  {
    super.encodeBegin(context, component);

    UIShowOne showOne = (UIShowOne)component;
    String clientId = showOne.getClientId(context);

    ViewHandler handler = context.getApplication().getViewHandler();
    String showItemURL = handler.getResourceURL(context,
                            "weblet://com.apress.projsf.ch9/showItem.htc");

    ResponseWriter out = context.getResponseWriter();
    writeScriptInline(context, "document.namespaces.add('pro',
                                      'http://projsf.apress.com/tags');");
    out.write("<?import namespace=\"pro\"
                        implementation=\"" + showItemURL + "\" ?>");

    out.startElement("pro:showOneDeck", component);
    out.writeAttribute("id", clientId, null);
    out.writeAttribute("style", "display:block;", null);
  }
```

Before you write anything to the client, you need to obtain the component's unique identifier: `clientId`. You do this by calling the `getClientId()` method on the `UIShowOne` instance

passed as an argument to the `Renderer`. You include this unique identifier in the generated markup to ensure that you will be able to decode the request and apply any values or events to the right component on postback. For more information about the `clientId`, see Chapter 2.

You then use weblets to obtain the resource URL of the `<pro:showItem>` HTC component that will be used to set the `implementation` attribute on the `<?import>` processing instruction. This is an alternative solution to the `doImport()` method used in the `HtcAjaxInputDateRenderer` (see Code Sample 9-18) to import an element behavior.

You get the `ResponseWriter` and write out the first element (`<pro:showOneDeck>`) representing the component.

The `<pro:showOneDeck>` component relies on the Dojo toolkit and D^2 project to be able to asynchronously communicate with the server. To ensure that these resources are loaded to the client and written only once, you will use the semantics behind the `writeScriptResource()` method, as shown in Code Sample 9-23.

Code Sample 9-23. *The* `HtcAjaxShowOneDeckRenderer` `encodeResources()` *Method*

```
protected void encodeResources(
  FacesContext context,
  UIComponent  component) throws IOException
{
  super.encodeResources(context, component);

  writeScriptResource(context, "weblet://org.dojotoolkit.browserio/dojo.js");
  writeScriptResource(context, "weblet://net.java.dev.d2/d2.js");
}
```

In the `encodeChildren()` method, as shown in Code Sample 9-24, you check to see whether this `UIShowOne` component has any children at all. If the application developer has not added any children, you do not need to render this instance of the `UIShowOne` component to the client. You then collect information about which default `UIShowItem` `id` to display and which style classes to use for the child items.

Code Sample 9-24. *The* `encodeChildren()` *Method*

```
public void encodeChildren(
  FacesContext context,
  UIComponent  component) throws IOException
{
  if (component.getChildCount() > 0)
  {
    UIShowOne showOne = (UIShowOne)component;
    String showItemId = showOne.getShowItemId();

    Map attrs = showOne.getAttributes();
    String styleClass = getItemStyleClass(attrs);
    String headerStyleClass = getItemHeaderStyleClass(attrs);
    String contentStyleClass = getItemContentStyleClass(attrs);
```

After that, you collect all children of the UIShowOne component, iterate over the list of children, and check whether each child is an instance of UIShowItem, as shown in Code Sample 9-25. If not, the child will not be rendered. If the child is a UIShowItem component instance, you gather the clientId and all attributes available on the UIShowItem component. The showItemId is then compared with the id of the current UIShowItem component, and based on the result, the active variable will be used as a true or false flag. This flag will later be used to set the active attribute on the <pro:showItem> tag to indicate whether this UIShowItem component should render its children.

Code Sample 9-25. *The* encodeChildren() *Method*

```
List children = component.getChildren();
for (Iterator iter = children.iterator(); iter.hasNext();)
{
  UIComponent child = (UIComponent) iter.next();
  if (child instanceof UIShowItem)
  {
    UIShowItem showItem = (UIShowItem)child;
    Map attrs = showItem.getAttributes();

    String id = showItem.getId();
    boolean active = (id.equals(showItemId));

    ResponseWriter out = context.getResponseWriter();
    out.startElement("pro:showItem", showItem);
    out.writeAttribute("itemId", id, null);
     if (styleClass != null)
       out.writeAttribute("styleClass", styleClass, ITEM_STYLE_CLASS_ATTR);
     if (headerStyleClass != null)
       out.writeAttribute("headerStyleClass", headerStyleClass,
                          ITEM_HEADER_STYLE_CLASS_ATTR);
     if (contentStyleClass != null)
       out.writeAttribute("contentStyleClass", contentStyleClass,
                          ITEM_CONTENT_STYLE_CLASS_ATTR);
    if (active)
      out.writeAttribute("active", Boolean.toString(active), null);
    out.writeAttribute("style", "display:block;", null);
```

In Code Sample 9-26, you then get the header facet from the UIShowItem component by calling the getHeader() method. This convenience method returns the named facet, header, if it exists; otherwise, it returns null. If the getHeader() method returns a facet, you call the _encodeAll() method to process any children of this facet. After the facet control, you use the active flag to determine whether this is the "active" UIShowItem component. If it is, you call the _encodeAll() method to start the encode process of any children to the UIShowItem component.

Code Sample 9-26. *The* `encodeChildren()` *Method*

```
        // the header facet
        UIComponent header = showItem.getHeader();
        if (header != null)
        {
          out.startElement("pro:headerFacet", null);
          _encodeAll(context, header);
          out.endElement("pro:headerFacet");
        }

        // the expanded item contents
        if (active)
        {
          _encodeAll(context, showItem);
        }

        out.endElement("pro:showItem");
      }
    }
  }
}
```

If you take a close look at the actual output required by the deck component, any children added will be at the end of the generated markup. This way, the `UIShowOne` component's renderer can quickly close the generated markup, as shown in Code Sample 9-27.

Code Sample 9-27. *The* `encodeEnd()` *Method*

```
public void encodeEnd(
  FacesContext context,
  UIComponent  component) throws IOException
{
  ResponseWriter out = context.getResponseWriter();
  out.endElement("pro:showOneDeck");
}
```

For the `UIShowOne` component, the `Renderer` is responsible for rendering its children, and thus this flag needs to be set to `true`, as shown in Code Sample 9-28.

Code Sample 9-28. *The* `getRendersChildren()` *Method*

```
public boolean getRendersChildren()
{
  return true;
}
```

The requirement has not changed since we first introduced the deck component. It has to be flexible enough to handle any type of child component that the application developer adds to the UIShowItem component. The UIShowItem component itself is not responsible for rendering its children, but an application developer may add a child container component in charge of rendering its children (for example, an HtmlPanelGroup component).

To be able to achieve this, you first render the beginning of the current state of this UIComponent to the ResponseWriter attached to the specified FacesContext. You then check whether the component is responsible for rendering its children. If it is, you call encodeChildren() on the component to start rendering its children. If the component is not responsible for rendering its children, you call getChildren() on the component. The getChildren() method returns a List of all children of the UIComponent. If this component has no children, an empty List is returned, and you close the generated markup by calling the encodeEnd() method on the component. If it has children, you recursively call the _encodeAll(), as shown in Code Sample 9-29, until all children have been rendered, and then you close the generated markup by calling the encodeEnd() method on the component.

Code Sample 9-29. *The* _encodeAll() *Method*

```
private void _encodeAll(
  FacesContext context,
  UIComponent  component) throws IOException
{
  component.encodeBegin(context);
  if (component.getRendersChildren())
  {
    component.encodeChildren(context);
  }
  else
  {
    List kids = component.getChildren();
    Iterator it = kids.iterator();
    while (it.hasNext())
    {
      UIComponent kid = (UIComponent)it.next();
      _encodeAll(context, kid);
    }
  }
  component.encodeEnd(context);
}
```

Remember, during the Apply Request Values phase, a method, processDecodes(), will be called on the UIViewRoot at the top of the component hierarchy. This processDecodes() method on the UIViewRoot will recursively call processDecodes() of each UIComponent in the component hierarchy. If a Renderer is present for any of these components, the UIComponent will delegate the responsibility of decoding to the Renderer. Code Sample 9-30 is identical to the encode() method in the first HtmlShowOneDeckRenderer introduced in Chapter 3. For more information about processDecodes(), please refer to Chapter 2.

Code Sample 9-30. *The* `decode()` *Method*

```
public void decode(
  FacesContext context,
  UIComponent component)
{
  ExternalContext external = context.getExternalContext();
  Map requestParams = external.getRequestParameterMap();
  String clientId = component.getClientId(context);
  String newShowItemId = (String)requestParams.get(clientId);
  if (newShowItemId != null && newShowItemId.length() > 0)
  {
    UIShowOne showOne = (UIShowOne)component;
    String oldShowItemId = showOne.getShowItemId();
    if (!newShowItemId.equals(oldShowItemId))
    {
      showOne.setShowItemId(newShowItemId);
      ShowEvent event = new ShowEvent(showOne, oldShowItemId, newShowItemId);
      event.queue();
    }
  }
}
```

Code Sample 9-31 shows all the getters for the different style classes supported by the `HtcAjaxShowOneDeckRenderer`.

Code Sample 9-31. *Getters for the* `HtcAjaxShowOneDeckRenderer` *Attributes*

```
  protected String getStyleClass(
    Map attrs)
  {
    return (String)attrs.get(STYLE_CLASS_ATTR);
  }

  protected String getItemStyleClass(
    Map attrs)
  {
    return (String)attrs.get(ITEM_STYLE_CLASS_ATTR);
  }

  protected String getItemHeaderStyleClass(
    Map attrs)
  {
    return (String)attrs.get(ITEM_HEADER_STYLE_CLASS_ATTR);
  }

  protected String getItemContentStyleClass(
    Map attrs)
  {
```

```
    return (String)attrs.get(ITEM_CONTENT_STYLE_CLASS_ATTR);
  }
}
```

Step 7: Registering a UIComponent and Renderer

For the HTC-Ajax implementation to work, you need to register the custom `Renderers`, as shown in Code Sample 9-32.

Code Sample 9-32. *The HTC Registration in the* `faces-config.xml` *File*

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE faces-config
   PUBLIC "-//Sun Microsystems, Inc.//DTD JavaServer Faces Config 1.1//EN"
          "http://java.sun.com/dtd/web-facesconfig_1_1.dtd">
<faces-config xmlns="http://java.sun.com/JSF/Configuration" >
  <render-kit>
    <renderer>
      <component-family>com.apress.projsf.Document</component-family>
      <renderer-type>com.apress.projsf.Document</renderer-type>
      <renderer-class>
        com.apress.projsf.ch9.render.html.basic.HtmlDocumentRenderer
      </renderer-class>
  </renderer>
</render-kit>

<render-kit>
  ...
  <renderer>
      <component-family>javax.faces.Input</component-family>
      <renderer-type>com.apress.projsf.Date</renderer-type>
      <renderer-class>
        com.apress.projsf.ch9.render.htc.ajax.HtcAjaxInputDateRenderer
      </renderer-class>
    </renderer>
    <renderer>
      <component-family>com.apress.projsf.ShowOne</component-family>
      <renderer-type>com.apress.projsf.Deck</renderer-type>
      <renderer-class>
        com.apress.projsf.ch9.render.htc.ajax.HtcAjaxShowOneDeckRenderer
      </renderer-class>
    </renderer>
  </render-kit>
</faces-config>
```

Step 11: Registering a RenderKit and JSF Extension

Although you did not need to create a new `RenderKit` for the HTC solution, you still need to register a `RenderKit` with a unique `RenderKit` ID, as shown in Code Sample 9-33. You need this

to ensure that you don't mix the HTC components with regular HTML components that work across multiple browsers.

Code Sample 9-33. *The HTC Registration in the* `faces-config.xml` *File*

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE faces-config
   PUBLIC "-//Sun Microsystems, Inc.//DTD JavaServer Faces Config 1.1//EN"
          "http://java.sun.com/dtd/web-facesconfig_1_1.dtd">
<faces-config xmlns="http://java.sun.com/JSF/Configuration" >

  <render-kit>
    <!-- no renderkit-id, so these renderers are added to
         the default renderkit -->
    <renderer>
      <component-family>com.apress.projsf.Document</component-family>
      <renderer-type>com.apress.projsf.Document</renderer-type>
      <renderer-class>
        com.apress.projsf.ch9.render.html.basic.HtmlDocumentRenderer
      </renderer-class>
    </renderer>
  </render-kit>

  <render-kit>
    <render-kit-id>com.apress.projsf.htc.ajax[HTML_BASIC]</render-kit-id>
    <render-kit-class>
      com.apress.projsf.ch6.render.html.ajax.HtmlAjaxRenderKit
    </render-kit-class>
    <renderer>
      <component-family>javax.faces.Input</component-family>
      <renderer-type>com.apress.projsf.Date</renderer-type>
      <renderer-class>
        com.apress.projsf.ch9.render.htc.ajax.HtcAjaxInputDateRenderer
      </renderer-class>
    </renderer>
    <renderer>
      <component-family>com.apress.projsf.ShowOne</component-family>
      <renderer-type>com.apress.projsf.Deck</renderer-type>
      <renderer-class>
        com.apress.projsf.ch9.render.htc.ajax.HtcAjaxShowOneDeckRenderer
      </renderer-class>
    </renderer>
  </render-kit>
</faces-config>
```

As you can see, the `HtmlDocumentRenderer` is defaulted to use the basic HTML `RenderKit` since it is a basic HTML `Renderer`, whereas the `HtcAjaxInputDateRenderer` and `HtcAjaxShowOneDeckRenderer` are added to the `HtmlAjaxRenderKit`. Notice that you are reusing the `HtmlAjaxRenderKit` created in Chapter 6 but assigning it a new `RenderKit` ID (that is, `com.apress.projsf.htc.ajax[HTML_BASIC]`) to ensure that you are not mixing HTC-specific renderers with plain HTML renderers.

Step 12: Registering Resources with Weblets

You need to register the HTC resources (`inputDate.css`, `inputDate.htc`, `showOneDeck.css`, and `showOneDeck.htc`) as weblets, as shown in Code Sample 9-34, which will enable you to package these resources as part of the custom JSF component library.

Code Sample 9-34. *The Weblets Configuration File*

```
<?xml version="1.0" encoding="UTF-8" ?>
<weblets-config xmlns="http://weblets.dev.java.net/config" >

  <weblet>
    <weblet-name>com.apress.projsf.ch9</weblet-name>
    <weblet-class>net.java.dev.weblets.packaged.PackagedWeblet</weblet-class>
    <init-param>
      <param-name>package</param-name>
      <param-value>com.apress.projsf.ch9.render.htc.ajax.resources</param-value>
    </init-param>
    <mime-mapping>
      <extension>htc</extension>
      <mime-type>text/x-component</mime-type>
    </mime-mapping>
  </weblet>

  <weblet-mapping>
    <weblet-name>com.apress.projsf.ch9</weblet-name>
    <url-pattern>/projsf-ch9/*</url-pattern>
  </weblet-mapping>

</weblets-config>
```

Building Applications with JSF HTC Components

Figure 9-11 shows the end result of the JSF HTC `ProInputDate` implementation. As you can see, the page looks the same as the one you created in previous chapters, except that this page uses an HTC `Renderer`. This HTC Ajax implementation provides the same functionality as both the HTML Ajax and XUL Ajax implementations, where dates that are not selectable are marked red and dates outside the scope of the current month are gray. When the user enters a date and clicks a submit button, a full postback will occur, and the attached validator, if any, will be invoked.

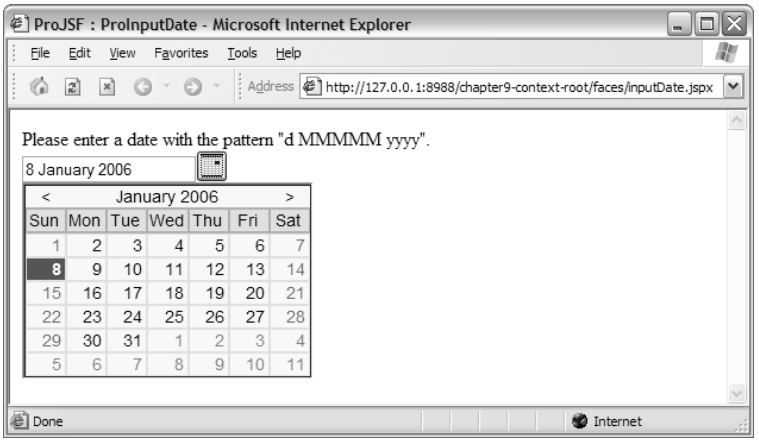

Figure 9-11. *The* `<pro:inputDate>` *component implemented in HTC*

Code Sample 9-35 shows the actual code behind this JSF page.

Code Sample 9-35. *JSF Page Source for HTC Implementation*

```
<?xml version="1.0" encoding="UTF-8" ?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" version="1.2"
          xmlns:pro="http://projsf.apress.com/tags"
          xmlns:f="http://java.sun.com/jsf/core"
          xmlns:h="http://java.sun.com/jsf/html" >
  <jsp:directive.page contentType="application/x-javaserver-faces"/>
  <f:view>
    <pro:document title="Pro JSF : ProInputDate" >
      <h:form>
        <pro:inputDate id="dateField"
                       title="Date Field Component"
                       value="#{inputDateBean.date}" >
          <f:convertDateTime pattern="d MMMMM yyyy" />
          <pro:validateDate availability="#{inputDateBean.getAvailability}" />
        </pro:inputDate>
        <br/>
        <h:message for="dateField" />
        <br/>
        <h:commandButton value="Submit" />
        <br/>
        <h:outputText value="#{inputDateBean.date}" >
          <f:convertDateTime pattern="d MMMMM yyyy" />
        </h:outputText>
      </h:form>
    </pro:document>
  </f:view>
</jsp:root>
```

No changes to the application logic are required whatsoever! This is the same page you used for the XUL solution.

Figure 9-12 shows the end result of the JSF HTC `<pro:showOneDeck>` implementation. It looks the same as the previous implementations of the `<pro:showOneDeck>` component, except that this page uses an HTC `Renderer`.

Figure 9-12. `<pro:showOneDeck>` *implemented in HTC*

Code Sample 9-36 shows the actual code behind this JSF page.

Code Sample 9-36. *JSF Page Source for HTC Implementation*

```
<?xml version="1.0" encoding="UTF-8" ?>
<jsp:root ...>
  <jsp:directive.page contentType="application/x-javaserver-faces"/>
  <f:view>
      ...
      <pro:showOneDeck showItemId="first"
                       showListener="#{backingBean.doShow}">
        <pro:showItem id="first" >
          <f:facet name="header">
            <h:panelGroup>
              <h:graphicImage url="/resources/java_small.jpg" alt="The Duke"
                              style="margin-right: 8px; vertical-align:bottom;" />
              <h:outputText value="Java"/>
            </h:panelGroup>
          </f:facet>
          <h:panelGrid columns="1">
            <h:outputLink value="http://apress.com/book/bookDisplay.html?bID=10044">
              <h:outputText value="Pro JSF: Building Rich Internet Components"/>
      ...
  </f:view>
</jsp:root>
```

You are probably now telling yourself, "This is way cool! I have the same page source in three other solutions!" You are right, and that's the beauty of JSF! As we have said on multiple occasions, without impacting the application developer, you can create Rich Internet Components that support client-specific markup for optimized performance and responsiveness.

Summary

You have now completed four different solutions for the `ProInputDate` and `ProShowOneDeck` components. You used traditional HTML `Renderer`s, HTML Ajax `Renderer`s, XUL Ajax `Renderer`s, and HTC Ajax `Renderer`s. Who said JSF is not a rich client development platform?

With the experience you have gained so far, it is important to keep in mind that an application developer might be using your component in combination with other technologies that you did not even consider. You need to keep the abstraction for the application developer, and although tempting, you should not design your component with a dependency on the client-side rendered markup, since you do not have control over other components' generated markup.

One of things you want you to take from this chapter is that you should stay open to new and controversial suggestions; do not get locked into one technology stack because it is what you know or is what others tell you is the latest and greatest. Always ask yourself, "How can this solve my problems?"

After reading this chapter, you should have a clear understanding of what HTC is and how you can leverage it in your component design.

CHAPTER 10

Switching RenderKits Dynamically

Get used to working in components and only components, and you're future-proofed. Stick to JSF plus HTML hybrids, and someone is going to hate you in five years' time . . . .

—Duncan Mills, Java Evangelist, Oracle

Welcome to the last chapter of *Pro JSF and Ajax: Building Rich Internet Components*. This book has covered how JSF lets component writers mix and match technologies to streamline packaging and increase richness and user interactivity for their components. We have proven that JSF's component model can provide an abstraction layer on top of the underlying client-specific markup, which increases an application developer's productivity. We have also shown you that component writers can *manually* switch `RenderKit`s without impacting the application developer or the actual application logic.

Now that you have a set of rich Internet components that support multiple client technologies at your disposal, only one question is left: how can you automatically select a `RenderKit` to deliver the proper markup to any user agent?

The main technology covered in this chapter is Oracle ADF Faces, which is a rich set of standard JSF components introduced in the fall of 2004. The Oracle ADF Faces component library provides various user interface components with built-in functionality, such as data tables, hierarchical tables, and color and date pickers. ADF Faces also includes many of the framework features most needed by JSF developers today.

After reading this chapter, you should be able to dynamically switch `RenderKit`s and know how to set them up to detect different user agents, such as Mozilla GRE and Microsoft Internet Explorer.

Note At the time of writing this chapter, Oracle has completed the first step of donating the ADF Faces source code to the Apache Software Foundation. By the time this book hits the shelves, the Apache MyFaces community should be actively evolving the Oracle ADF Faces source code donation. For more information about the Apache MyFaces open source project, please visit `http://myfaces.apache.org`.

Requirements for Dynamically Switching RenderKits

The requirement is clear—the application developer wants to be able to dynamically change RenderKits, at runtime, based on the user agent. For example, if it is the Firefox browser requesting the page, the solution should serve XUL markup to the client.

A RenderKit's function is to help out with the delegation of Renderer to the UIComponent. A RenderKit groups instances of renderers of similar markup types, and in this book, you created RenderKits for HTML Ajax, Microsoft's DHTML/HTC, and Mozilla's XUL/XBL technologies. Each RenderKit is associated with a view (component hierarchy) as a UIViewRoot property at runtime. If an application developer wants to add a RenderKit with custom Renderers to the application, a RenderKit ID must be added to the application's JSF configuration file, as shown in Code Sample 10-1.

Code Sample 10-1. *Setting the Default* RenderKit *ID*

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE faces-config PUBLIC
  "-//Sun Microsystems, Inc.//DTD JavaServer Faces Config 1.1//EN"
  "http://java.sun.com/dtd/web-facesconfig_1_1.dtd">
<faces-config xmlns="http://java.sun.com/JSF/Configuration">

  <application>
    <default-render-kit-id>com.apress.projsf.xul.ajax</default-render-kit-id>
  </application>
...
</faces-config>
```

This code sample shows the faces-config.xml file with the <default-render-kit-id> set to your custom XUL RenderKit. The faces-config.xml file is read once when the Web application is created and stored in the Application instance. The ViewHandler is responsible for returning the renderKitId for the current and subsequent requests from the client. It is important to understand that there can be only one default RenderKit per Web application, which is identified by a string (for example, com.apress.projsf.xul.ajax).

To solve the requirement of enabling access to the application with any browser and to provide a different RenderKit implementation for each browser, you have three tasks to complete in this chapter. First, you need to define the default RenderKit ID in the faces-config.xml file in such a way that you can dynamically set it at runtime. Second, you need to detect the user agent requesting the application. Third, you need to set the RenderKit ID using a ViewHandler. The custom ViewHandler is required if you want to have multiple RenderKit instances for the same application.

■ **Note** JSF 1.1 applications require a custom javax.faces.application.ViewHandler instance to dynamically select a RenderKit. However, JSF 1.2 adds support for directly specifying the RenderKit ID on the <f:view> tag of individual pages in a Web application.

The Dynamic RenderKit Implementation

Figure 10-1 shows the dynamic `RenderKit` solution.

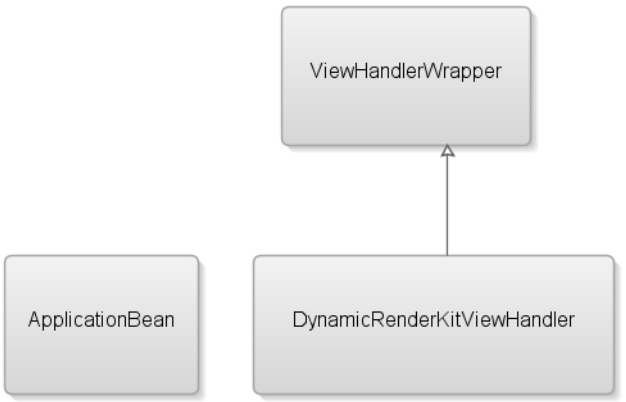

Figure 10-1. *Structure of dynamic* `RenderKit` *implementation*

The dynamic `RenderKit` solution contains three classes:

- `ViewHandlerWrapper` is a wrapper class that provides a loose coupling between the solution and the JSF implementation.
- `ApplicationBean` is a managed bean that contains logic to detect what agent has been used to request the application and contains information about what `renderKitId` to use.
- `DynamicRenderKitViewHandler` overrides the default `ViewHandler`'s `calculateRenderKitId()` method in order to get the correct ID from the `ApplicationBean`.

Syntax for Dynamic RenderKit ID

A feature in JSF that is often underutilized is the managed bean facility. This facility is not only useful for providing application logic, but you can also use it to initialize settings before launching the actual application. In this case, you will use the JSF EL syntax in the `faces-config.xml` file to set a pointer to the managed bean (for example, the `ApplicationBean`), which will be invoked and will return the correct `renderKitId` to the `ViewHandler` (see Code Sample 10-2).

Code Sample 10-2. *Setting the Default* `RenderKit` *ID*

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE faces-config PUBLIC
  "-//Sun Microsystems, Inc.//DTD JavaServer Faces Config 1.1//EN"
  "http://java.sun.com/dtd/web-facesconfig_1_1.dtd">
<faces-config xmlns="http://java.sun.com/JSF/Configuration">
```

```
  <application>
    <default-render-kit-id>#{[managedBean].[property]}</default-render-kit-id>
  </application>
   ...
</faces-config>
```

With an explicit syntax shown in Code Sample 10-2, you can use the `ViewHandler` to first check the pattern of the string and then use the string to create a `ValueBinding` for the managed bean defined by the expression. In this case, the completed configuration would look something like Code Sample 10-3.

Code Sample 10-3. *Setting the Default* `RenderKit` *ID Using a Managed Bean*

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE faces-config PUBLIC
  "-//Sun Microsystems, Inc.//DTD JavaServer Faces Config 1.1//EN"
  "http://java.sun.com/dtd/web-facesconfig_1_1.dtd">
<faces-config xmlns="http://java.sun.com/JSF/Configuration">

  <application>
    <default-render-kit-id>#{projsf.renderKitId}</default-render-kit-id>
  </application>
...
</faces-config>
```

In this case, the `renderKitId` is a JavaBean property of the `ApplicationBean` that returns the correct `RenderKit` identifier for the requesting user agent.

The Dynamic RenderKit Managed Bean

Let's look at the actual `ApplicationBean` class. Figure 10-2 shows the `ApplicationBean` in a class diagram, and in Code Sample 10-4, you can observe the `User-Agent` request header for choosing an appropriate `RenderKit`.

Figure 10-2. *Class diagram showing the* `ApplicationBean` *class*

Code Sample 10-4. *The* `getRenderKitId()` *Method with* `User-Agent` *Request Header*

```
package com.apress.projsf.ch10.application;

import java.util.Map;
```

```
import javax.faces.render.RenderKitFactory;
import javax.faces.context.FacesContext;
import javax.faces.context.ExternalContext;

/**
 * The ApplicationBean returns a dynamic RenderKit identifier, based on
 * the value of the User-Agent request header.
 */
public class ApplicationBean
{
  public String getRenderKitId()
  {
    FacesContext context = FacesContext.getCurrentInstance();
    ExternalContext external = context.getExternalContext();
    Map requestHeaders = getRequestHeaderMap();
    String userAgent = (String) requestHeaders.get("User-Agent");

    // Mozilla Firefox 1.0.7
    // Mozilla/5.0 (Windows; U; Windows NT 5.0; en-US; rv:1.7.12)
    // Gecko/20050915 Firefox/1.0.7
    if (userAgent.indexOf("Gecko/") != -1)
    {
      return "com.apress.projsf.xul.ajax";
    }
    // MS Internet Explorer 6.0
    // Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0)
    else if (userAgent.startsWith("Mozilla") &&
             userAgent.indexOf("MSIE") != -1)
    {
      return "com.apress.projsf.htc.ajax";
    }
    // Safari
    // Mozilla/5.0 (Macintosh; U; PPC Mac OS X; en-us)
    //             AppleWebKit/XX (KHTML, like Gecko) Safari/YY
    else if ((userAgent.indexOf("AppleWebKit") != -1) ||
             (userAgent.indexOf("Safari") != -1))
    {
      return "com.apress.projsf.html.ajax";
    }
    else
    {
      // default to standard HTML Basic for PDAs, etc.
      return RenderKitFactory.HTML_BASIC_RENDER_KIT;
    }
  }
}
```

In Code Sample 10-4, you are testing the `User-Agent` request header directly against known user agent identifiers to decide which `RenderKit` is appropriate to use in the response. Notice that some of the syntax for user agents can overlap, such as `Mozilla` appearing in the user agent header for Firefox, Internet Explorer, and Safari. Given the complexity of accurately parsing the wide range of possible `User-Agent` headers, it is best to reuse a common implementation rather than repeating the agent detection code each time it is needed.

Oracle ADF Faces provides a `User-Agent` abstraction to handle this case, and in Code Sample 10-5 we have simplified the `ApplicationBean` by leveraging some of the Oracle ADF Faces public APIs to obtain the user agent.

Code Sample 10-5. *The* `getRenderKitId()` *Method*

```
package com.apress.projsf.ch10.application;

import javax.faces.render.RenderKitFactory;

import oracle.adf.view.faces.context.AdfFacesContext;
import oracle.adf.view.faces.context.Agent;

/**
 * The ApplicationBean returns a dynamic RenderKit identifier, based on
 * the ADF Faces Agent name.
 */
public class ApplicationBean
{
  public String getRenderKitId()
  {
    AdfFacesContext afc = AdfFacesContext.getCurrentInstance();
    Agent agent = afc.getAgent();

    if (Agent.AGENT_GECKO.equals(agent.getAgentName()))
    {
      return "com.apress.projsf.xul.ajax";
    }
    else if (Agent.AGENT_IE.equals(agent.getAgentName()) &&
             Agent.TYPE_DESKTOP.equals(agent.getType()))
    {
      return "com.apress.projsf.htc.ajax";
    }
    else if (Agent.AGENT_WEBKIT.equals(agent.getAgentName()))
    {
      return "com.apress.projsf.html.ajax";
    }
    else
    {
      // default to standard HTML Basic for PDAs, etc.
```

```
      return RenderKitFactory.HTML_BASIC_RENDER_KIT;
    }
  }
}
```

From the `AdfFacesContext`, you can obtain the user agent by calling the `getAgent()` method. ADF Faces also comes with a set of predefined keys for each available Web client (for example, Microsoft Internet Explorer, Mozilla GRE, and so on). By comparing the agent name to these keys, you can determine which `renderKitId` to return.

The DynamicRenderKitViewHandler Class

Let's now look at the `DynamicRenderKitViewHandler` class. Figure 10-3 shows the `DynamicRenderKitViewHandler` in a class diagram, and in Code Sample 10-6, you can see how it uses the default `RenderKit` identifier as a base to locate agent-specific `RenderKits` for the incoming request.

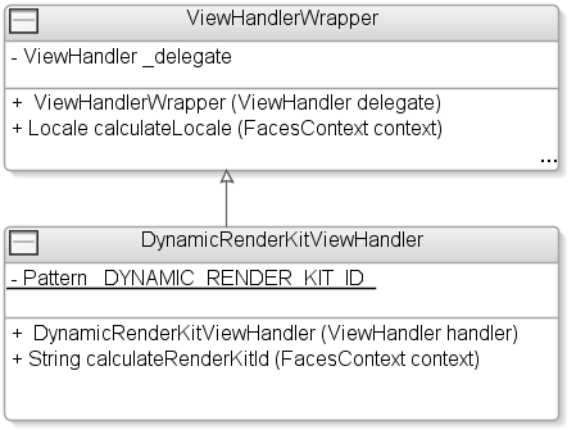

Figure 10-3. *Class diagram showing the* `DynamicRenderKitViewHandler` *implementation*

Code Sample 10-6. *The* `DynamicRenderKitViewHandler` *Class*

```
package com.apress.projsf.ch10.application;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

import javax.faces.application.Application;
import javax.faces.application.ViewHandler;
import javax.faces.context.FacesContext;
import javax.faces.el.ValueBinding;
```

```
/**
 * The DynamicRenderKitViewHandler provides EL support
 * for the <default-render-kit-id> element in faces-config.xml.
 */
public class DynamicRenderKitViewHandler extends ViewHandlerWrapper
{
  public DynamicRenderKitViewHandler(
    ViewHandler handler)
  {
    super(handler);
  }

  public String calculateRenderKitId(
    FacesContext context)
  {
    String renderKitId = super.calculateRenderKitId(context);

    Matcher matcher = _DYNAMIC_RENDER_KIT_ID.matcher(renderKitId);
    if (matcher.matches())
    {
      String expression = matcher.group(1);
      Application application = context.getApplication();
      ValueBinding binding = application.createValueBinding(expression);
      if (binding.getType(context) == String.class)
        renderKitId = (String)binding.getValue(context);
    }

    // return either the calculated or dynamic RenderKit ID
    return renderKitId;
  }

  // Matches RenderKit identifier of the form "#{...}"
  static private final Pattern _DYNAMIC_RENDER_KIT_ID =
                                  Pattern.compile("(\\Q#{\\E[^\\}]+\\Q}\\E)");
}
```

The `DynamicRenderKitViewHandler` overrides only one method, `calculateRenderKitId()`, which is used to calculate the `RenderKit` identifier to use for this request. You first calculate the `RenderKit` identifier by calling `super`. Then you detect whether the identifier is an expression that can be used to evaluate the dynamic `RenderKit` identifier. If it matches the EL-like syntax, you use the expression to create a `ValueBinding` that returns the value representing the `renderKitId` for this request. In practice, this will pick the right `RenderKit` for the browser accessing the application by following the dynamic `RenderKit` selection logic in your `ApplicationBean`.

Registering the Dynamic RenderKit Solution

You need to register the DynamicRenderKitViewHandler and the managed bean ApplicationBean with the component library in order for the dynamic switching to work (see Code Sample 10-7).

Code Sample 10-7. *Registering the Dynamic* RenderKit *Implementation*

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE faces-config
   PUBLIC "-//Sun Microsystems, Inc.//DTD JavaServer Faces Config 1.1//EN"
          "http://java.sun.com/dtd/web-facesconfig_1_1.dtd">

<faces-config xmlns="http://java.sun.com/JSF/Configuration" >
  <factory>
    ...
  </factory>

  <application>
    <view-handler>
      com.apress.projsf.ch10.application.DynamicRenderKitViewHandler
    </view-handler>
  </application>

  <managed-bean>
    <managed-bean-name>projsf</managed-bean-name>
    <managed-bean-class>
       com.apress.projsf.ch10.application.ApplicationBean
    </managed-bean-class>
    <managed-bean-scope>application</managed-bean-scope>
  </managed-bean>
  ...
<faces-config>
```

First you set the <view-handler> to point to the custom ViewHandler—DynamicRenderKitViewHandler. Then you define the ApplicationBean in the same way most application developers would define their own managed beans. Notice that you set the managed bean on the application scope so that there will be only one instance for all Web applications.

You have now reached the end of this chapter—and the end of the book. You should now be able to dynamically switch RenderKits at runtime. This solution to switch RenderKits is not specific to the components created in this book; any component library can use the same technique with multiple RenderKits. Figure 10-4 shows how the deck component created in this book would look in three different devices.

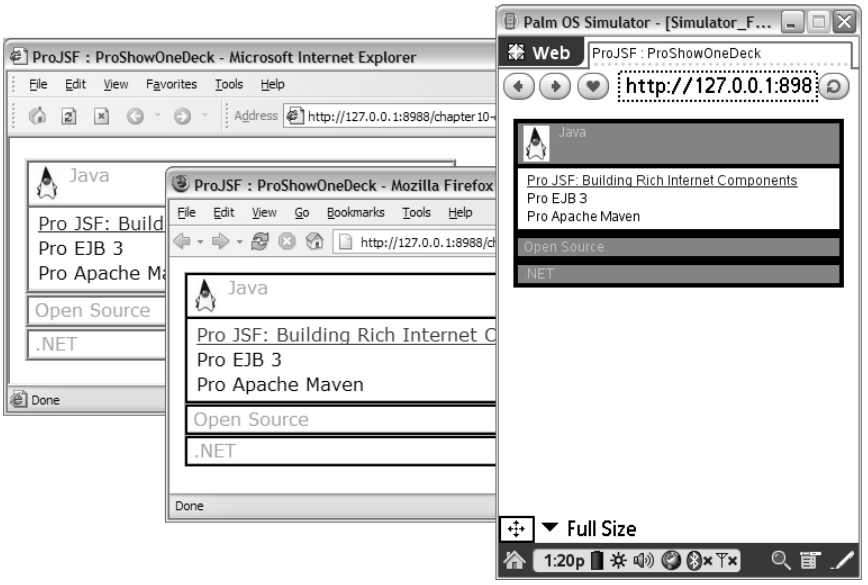

Figure 10-4. `ProShowOneDeck` *running in multiple clients using client-specific markup*

Summary

In this chapter, we showed you how easy it is to provide dynamic `RenderKit` switching with JSF. By using a component-driven design, application developers can build applications for any type of user agent without being impacted by the underlying client markup.

In the previous chapters, we demonstrated how you can write `Renderers` that support regular HTML, Ajax, XUL, and HTC. Some component writers are already looking at even more client technologies to provide application developers with a common programming model regardless of the user agent. A good example of this is the Oracle ADF Faces component library. It has built-in support for HTML, RIAs, character-based solutions, instant messenger clients such as Gaim and Yahoo, PDAs, and so on. The MyFaces open source project also provides an alternative `RenderKit` to HTML—the WML `RenderKit`.

Now that you know how to create reusable rich Internet components with JSF and how to use multiple `RenderKits`, we hope you will apply the techniques you have learned in this book to create your own custom components and build RIAs with JSF.

Index

A

"A Modular Way of Defining Behavior for XML and HTML" (Netscape), Web site address for, 193
abort() method, XMLHttpRequest object, 176
action parameter, function of with <form> tag, 177
addShowListener() method, for associating a listener to the deck component, 132
ADF UIX, as JSF component, 7
Ajax (Asynchronous JavaScript and XML)
  fetching data with, 273–274
  formerly known as XMLHTTP, 173–174
  overview of, 174–187
  providing file upload functionality in, 233
  question of cross-platform support in, 209
  resources, 233–235
  sequence diagram over your postback implementation, 232
  Web development technique, 224
Ajax and JSF architectures, architectural solutions for applying changes, 230–232
Ajax applications
  book filter implementation, 184–187
  building, 181–187
  vs. traditional Web applications, 181
Ajax architecture, selecting, 231–232
Ajax book filter, implementation of, 184–187
Ajax implementation, requirements for deck component's, 223–224
Ajax postback, overview of, 180–181
Ajax Renderer, creating, 279–297
Ajax resources
  introducing the Dojo toolkit, 234
  object-orented JavaScript, 291–292
  registering with Weblets, 301
Ajax resources and Weblets, registering, 263–264
Ajax Web application development, 179–181
Ajax-enabled deck component, designing using a blueprint, 226–227
Apache MyFaces, Web site address for, 128
Apache MyFaces open source project, Web site address for information about, 403
application developer, 3
application development, exploring today, 5–7
application development technologies, overview of, 4–5
application lifecycle, when using JSF, 9
application managed bean scope, 12
ApplicationBean class
  class diagram showing, 406
  function of, 405
Application.createComponent() method, function of, 37
ApplicationFactory class, function of, 31–32
Apply Request Values phase
  applying new values passed on the request to the components, 115
  event handling in, 125–127
  function of, 71–72
  how it passes new values to the components, 40
  the processDecodes() and decode() methods, 72
  in request-processing lifecycle, 27–28
ApplyRequestValuesPhase class, in Mabon, 281
arrays, function of in JSON, 280
Asynchronous JavaScript and XML. *See* Ajax (Asynchronous JavaScript and XML)
ATTACH element, in HTC, 201
attached behavior, released with Internet Explorer 5.0, 203
attribute values, looking up, 62–63

B

backing bean, as plain old Java object (POJO) in JSF, 11–12
"Behavioral Extensions to CSS", Web site address for, 200
behavioral superclass, creating, 127–136, 322–324
behavioral superclasses, available in JSF specification, 50–51
binding element, defining using a style sheet, 321–322
<bindings> element, containing one <binding> element, 192–193
bindings.xml file
  with a <vbox> element as a parent container and <hbox> element to hold the toolbar, 311
  code sample for XBL date component prototype, 309
  code sample of the <implementation> element in, 312
  code sample of the clickCell method in, 316–317
  code sample of the popup method in, 314
  code sample of the scroll method in, 315
  structure of as shown in the Oracle JDeveloper 10.1.3 structure window, 321

block boxes, Web site address for information about visual formatting using, 377
blueprint
  needed to successfully implement a custom JSF component, 53
  steps in for creating a new JSF component, 269–270
<body> HTC element, function of, 366
<body> XBL element, function of, 312
<box> component, Mozilla XUL, 189
broadcast events, function of, 117
broadcast() method signature, code sample, 117
<button> component, Mozilla XUL, 189

C

calendar pop-up, code sample for input and button markup needed for, 271–272
Change Detection in Hierarchically Structured Information research project, function of, 234
<checkbox> component, Mozilla XUL, 189
child components, controlling rendering of, 70
<children> XBL element, function of, 312
chrome system, Mozilla XUL's, 188
chrome URL vs. HTTP URL, 188
class diagram
  creating the FixedContentTypeResponseWriter during Ajax postback, 248
  of the DecoratingRenderKitFactory class, 241
  for DynamicRenderKitViewHandler class, 409
  of the HtmlAjaxRenderKit, 243
  of Mabon, 281
  over the HtmlInputDateRenderer created in Chapter 2, 56
  over the ProInputDate tag handler and its support class, 87
  show the HtcAjaxShowOneDeckRenderer class, 389
  showing all classes created in "Providing Microsoft HTC Renderers" chapter, 363
  showing all classes created in "Providing Mozilla XUL Renderers" chapter, 305
  showing all classes needed for the event and listener implementation, 114
  showing classes created for the Ajax-enabled date component, 268
  showing classes created for the Ajax-enabled deck component, 225
  showing date field component classes created in Chapter 2, 52
  showing deck component classes created in Chapter 3, 106
  showing the ApplicationBean class, 406
  showing the DateValidator, 277
  showing the HtcAjaxInputDateRenderer class, 384
  showing the HtmlAjaxInputDateRenderer, 279
  showing the HtmlDocumentRenderer class, 381
  showing the HtmlShowOneDeckRenderer extending the HtmlRenderer, 137
  showing the ProDocument class, 341
  showing the ProDocumentTag class, 347
  showing the ProShowOneDeck class, 148
  showing the UIDocument class, 324
  showing the UIShowOne implementation, 127
  showing the XMLResponseWriter class, 353
  showing the XulAjaxInputDateRenderer class, 329
  showing the XulAjaxRenderKit class, 351
  showing the XulDocumentRenderer class, 325
  showing the XUL Renderers, 325
  showing three tag handlers for the deck component, 156
  of the UIShowItem implementation, 135
  for the ValidateDateTag class, 298
  for the XulAjaxShowOneDeckRenderer class, 334
classless programming. *See* prototype-based programming
_clickCell() function, function of, 373–374
clientId
  calling the createUniqueId() method to create, 64
  obtaining for the HtcAjaxShowOneDeckRenderer class, 390–391
  using the _findFormClientId method to locate, 145
client-side attributes, provided by a Renderer, 55
client-specific renderer
  creating for Ajax-enabled deck components, 230–238
  creating the HtmlInputDate, 55–77
  creating to provide some additional markup to support the pop-up calendar, 279–297
  creating XulAjaxInput, D
  creating XulAjaxShowOneDeckRenderer classes, 324–341
  providing Microsoft HTC, 380–396
code sample
  the _clickCell() function, 374
  the _constructor() function, 369–370, 378
  the _determineDatePattern() method, 290
  the _determineTargetURL() method, 290–291
  the _expand() function, 379–380
  the _popup() function in the HTC file, 371–372
  the _scroll() function in the HTC file, 372–373
  the <body> element of the <pro:inputDate> HTC component, 366

the <body> of the <pro:showItem> component of the HTC file, 376–377
of the <constructor> element in the bindings.xml file, 313
the <head> element in the <pro:inputDate> HTC component, 368
the <head> element in the <pro:showItem> HTC component, 377
the <implementation> element in the bindings.xml file, 312
for accessor and mutator for the showItemId behavioral attributes, 131
actual code behind the ValidateDateTag class, 298–299
for adapting a JSF 1.1 MethodBinding into a ShowListener instance, 121–123
adding a header facet within a dataTable component, 23–24
adding event handlers—projsf-bindings.xml into and XBL component, 197–198
adding properties and methods—pro-bindings.xml, 196
for adding the decode() method to the XulAjaxInputDateRenderer, 331
the Ajax version of the showOneDeck.js library, 237
alternative RenderKit registration, 240
the application logic for the JSF application, 30–31
assigning a call to a function with HTC, 202
associating a function with an event in HTC, 202
associating a renderer to a particular component family, 21
for attaching the onColor() function to the mouseover event, 201
backing bean using the HtmlSelectOneRadio subclass, 18
behavioral and renderer-specific attributes for ProInputDateTag class, 94–96
the bindings.xml file, 311
the broadcast() method signature, 117
building the Login page for a JSF application, 28–29
for building the ProDocument class, 342–344
calling the encodeAll() method until all children have been rendered, 394
changes to UIComponentTagSupport setStringProperty() method in for JSF 1.2, 89
for checking whether each child is an instance of UIShowItem, 337–338
of the clickCell method in the bindings.xml file, 316–317
collecting data for rendering, 77
combining all rendered output into a single encodeEnd() method, 144
commandButton with attached listener, 23
constructing the pop-up calendar with a <vbox> element as a parent container, 310–311
for controlling rendering of child components, 70
for coupling an event raised on the client with an underlying function, 201
the createHttpRequest() function that creates the XMLHttpRequest object, 185–186
for the createShowListener method, 164
creating an instance of the XMLHttpRequest object, 175
creating an instance of the XMLHttpRequest object using ActiveXObject, 175
creating properties and accessors for client-side attributes, 79–80
creating the ShowListener interface, 120
data-bound table component, 276
deck component as it would be used in a JSF JSP document, 112–113
deck HTML implementation, 228–229
the decode() method, 395
the decode() method in the HtmlInputDateRenderer class, 72–73
decoding the request, 147
of the DeferredContentTypeResponse, 255–257
of the DeferredContentTypeResponse class, 257–258
for the DeferredPrintWriter class, 261–262
the DeferredServletOutputStream class, 259–260
defining a navigation case in JSF configuration file, 30
defining element behavior, 203–204
defining the component type and component class, 83–84
for defining your ProInputDate renderer-specific attributes, 102
to determine date pattern and launch calendar pop-up, 287–289
the _determineDatePattern() method, 387
for the doShow() ShowListener method, 124
for doStartTag() method, 162–163
the DynamicRenderKitViewHandler class, 409–410
the encodeChildren() method for getting the header facet from the UIShowItem component, 392–393
the encodeChildren() method to iterate over the list of children, 392
the encodeChildren() method to see if UIShowOne has any children, 391
the encodeEnd() method for closing the HTML <body> and <html> tags, 383
the encodeEnd() method for closing the UIShowOne component's renderer, 393

Find it faster at http://superindex.apress.com/

code sample *(continued)*
the encodeEnd() method to write out the markup and obtain the Converter, 386–387
the encodeHead() method, 383
the encodeResources() method for Ajax enabling the date field component, 289–290
the encodeResources() method for the deck component, 139
the encodeResources() method in the HtmlShowOneDeckRenderer code, 214
encodeResources() method to write a reference to the CSS style sheet, 68
excerpt from the d2.js library, 235
excerpt from the inputDate.css file, 292
excerpt of backing bean following the contract of your Validator, 299
of the ExtendedRenderKit class, 244–245
of the ExtendedRenderKitFactory class, 241–243
extending the HtmlShowOneDeckRenderer class, 235–236
extending the UIComponentBase class, 129–130
the FacesContextFactoryImpl class, 252–253
the _findFormClientId method, 145
the FixedContentTypeResponseWriter, 248–249
the getConverter() method, 69–70
the getRenderKitId() method for getting the dynamic RenderKit ID, 408–409
the getRenderKitId() method with User-Agent request header, 406–407
getters for the HTCAjaxShowOneDeckRenderer attributes, 395–396
the getters for the UIDocument attributes, 328–329, 383–384
getters for the XulAjaxShowOneDeckRenderer attributes, 340–341
for getting attribute values from the UIComponent, 63
getting the IDs of the UIForm and UIShowOne components, 141
getValueAsString() method, 68–69
a globally assigned event handler in HTC, 202
for handling child components added to UIShowOne component, 339
HTC file structure, 200
HTC file with viewlink set, 205
the HTC registration in the faces-config.xml file, 396, 397
the HtcAjaxInputDateRenderer class, 385–386
HTML markup needed for the <pro:inputDate> HTC implementation, 364–365
an HTML page leveraging Ajax to update a <select> element, 183
HTML prototype for the input date component, 53–54
the HTML version of the showOneDeck.js library, 237
HtmlAjaxRenderKit class, 246–247
HtmlInputDate.prototype._clickCell method, 296–297
HtmlInputDate.prototype._scroll method, 295
the HtmlInputDate.prototype.showPopup method, 293–294
HtmlRenderer superclass providing convenience methods for other HTML Renderers, 56–57
for implementation of the ShowListener interface, 124–125
for implementing at-most-one semantics for each script resource, 60
for implementing at-most-one semantics for each style resource, 59
implementing the ShowSource interface, 132
import statements for the renderer package, 60–61
input and button markup needed for calendar pop-up, 271–272
inputText component with attached date converter, 22
introducing the UIShowOne behavioral superclass, 129–130
JSF document using the <pro:inputDate> tag, 103
JSF page matching the XUL and HTC samples, 210–211
JSF page source, 272–273
JSF page source for HTC implementation, 399, 400
JSF page source for the Ajax-enabling deck component, 229
JSF page source for XUL implementation, 357–358, 358–359
JSF selectOneRadio bound to a behavioral superclass, 19
JSF selectOneRadio bound to a renderer-specific subclass, 17–18
for the JSP tag handler, 347–349
the Login page modified with some JSP tags, 45–46
login page with <f:verbatim> tag wrapped around non-JSF content, 46–47
the Mabon protocol, 282
Mabon servlet configuration, 285
the mabon.js library, 286
managed bean defined in the faces-config.xml file, 11
managing state saving, 133
mapping of component type and UIComponent subclass, 22

the markup needed to create the XUL date implementation prototype page, 307–308
the markup needed to create the XUL deck implementation prototype page, 317–318
markup to create a page using the <pro:showOneDeck> HTC prototype, 375–376
method handling boolean attributes and properties, 90
method handling MethodBinding attributes and properties, 91
method handling string attributes and properties, 88–89
method handling ValueBinding attributes and properties, 90–91
of method to close the generated markup for the UIShowOne components Renderer, 339
navigation rule configured in faces-config.xml, 9
navigation rules and managed beans for the JSF application, 29–30
new backing bean using the UISelectOne class, 19
new renderer added to the default HTML basic RenderKit, 26–27
overriding weblets mapping, 222
page source with the showListener tag, 123
parameterized HTML for the input date component, 54
for parameterized HTML for the showOneDeck renderer, 110–111
passing arguments to the Mabon send() function, 286
the populateBookList() function, 184–185
the popup method in the bindings.xml file, 314
processing decodes, 133–134
processing facet and children of the UIShowItem component, 142–143
of the pro.css file for defining the binding element, 322
ProInputDate attributes, 86
ProInputDate component with attached date Validator, 277
ProInputDateTag class, 93–94
the ProShowOneDeck client-specific subclass, 148–150
the ProShowOneDeckTag class, 156–159
for referring to a remote server using the http:// URL, 191
register the Ajax-enabled Renderer and RenderKit, 238
to register the Ajax-enabled Renderer and RenderKit, 262–263
registering ProShowOneDeck renderer-specific class, 154–155
registering the dynamic RenderKit implementation, 411
registering the HtmlAjaxInputDateRenderer, 297
for registering the JSF XUL implementation of the ProDocument class, 345–346
registering UIShowOne and UIShowItem, 150–152
registration of the ProInputDateRenderer in a faces-config.xml file, 82–83
of a regular HTML document structure, 323
the release() method, 98
the Renderer getConvertedValue() method, 75
renderer type as defined in the JSF configuration file, 25
rendering the start of each UIShowItem child component, 142
restoring state in the ProInputDate component, 81–82
a sample CSS file that has the -moz-binding property set—projsf.css, 194
a sample HTML file with XUL components—prototype-ch4.xul, 194
sample page with the date field component, 55
saving state in the ProInputDate component, 81
script reference using http://, 191
of the scroll method in the bindings.xml file, 315
for the setAvailability() and getAvailability() methods, 278
setProperties() method, 97
setting rendersChildren property flag to true, 144
for setting the CONTENT_TYPE variable to the accepted XUL contentType, 352
for setting the default RenderKit ID, 404
setting the default RenderKit ID, 405–406
setting the default RenderKit ID using the managed bean, 406
setting the method of saving state to the server side in the deployment descriptor, 21
the ShowEvent subclass, 118–119
showing how the name of a custom action element is defined, 99
showing the createValueBinding() and createMethodBinding() methods, 92–93
showing the deck HTML prototype implementation, 109–110
showing the encodeBegin() method for the HtmlShowOneDeckRenderer, 137–138
showing XBL file (bindings.xml) and the first binding (inputDate), 309
the showItem binding and the expand method, 319–320

Find it faster at http://superindex.apress.com/

code sample *(continued)*
the showItemActive binding, 320–321
the showItemHeader binding, 321
for the ShowItemTag class, 160–161
for the ShowListenerTag class, 161–162
the showOneDeck binding component, 319
the ShowSource interface, 128–129
a simple HTC file, 201
of a simple HTML file with attached behavior, 203
a simple use case of an event and predefined event handler, 190–191
a simple XUL file with embedded HTML elements, 190
a simple XUL page using an XBL binding with attached event handler, 198
source for your Ajax titles<em dash>ajax.json, 187
the source of the showOneDeck.js file, 140
string returned after Mabon has evaluated the Mabon protocol, 282
tag attributes to support the behavioral UIInput attributes, 100–101
Tag Library Descriptor (TLD), 165–168, 300
the _determineTargetURL() method, 388
for TLD defining just one custom action, 98–99
the transferListItems() function that populates the <select> element, 186
turning an HTC file into a custom tag, 203–204
UIComponent attributes, 99–100
UIComponent clientId lookup, 64
UIComponent-inherited attributes, 84
the UIComponentTagSupport class, 88
the UIDocument class, 324
UIInput-inherited attributes, 85–86
UIShowItem component, 135–136
for the unique keys used to identify resources, 60
using <deferred-method> syntax, 102
using <deferred-value> syntax, 102
using a managed bean for creating and mapping to backend code, 30
using getRendersChildren() method for the UIShowOne component, 393
using relative weblet:/ protocol syntax, 356
using the _determineDatePattern() method, 333
using the _determineTargetURL() method, 333–334
using the component attributes map to update a renderer-specific attribute, 20
using the decode() method, 340
using the encodeChildren() method, 337
using the encodeChildren() method for writing out the start element and any attributes, 338
using the encodeEnd() method to get hold of the Converter, 330–331
using the getConvertedValue() method, 332
using the getConverter() method, 332
using the getRendersChildren() method for the UIShowOne component, 339
using the input date component with the Ajax Renderer, 272–273
using the input date component with the XUL Renderer, 357–358
using the mabon:/ syntax, 286
using the release() method to release stored state, 164
using the weblet protocol to serve up resources, 220
using weblets to serve resources from the JAR file, 356
of a valid XUL document structure, 323
the validate() method, 277–278
Weblet configuration for the D^2 library, 264
Weblet configuration for the Dojo toolkit, 263–264
Weblet configuration for the HtmlAjaxInputDateRenderer resources, 301
weblet container configuration in the web.xml file, 221–222
for the weblet protocol syntax, 219
weblets configuration file defining a custom MIME type, 217
of the weblets configuration file for HTC resources, 398
the weblets configuration file for packaging XUL resources, 355–356
weblets configuration file using 1.0 versioning for production, 218
weblets configuration file using SNAPSHOT versioning for development, 219
weblets configuration file, weblets-config.xml, 216
writing output to the JSP buffered body tag, 67
for writing style resources to the client, 58
an XBL file containing one binding—projsf-bindings.xml, 193
the XML data island for the <pro:inputDate> calendar, 367–368
the XmlResponseWriter, 353–354
a XUL file with XBL components—prototype-ch4.xul, 197
XUL registration in faces-config.xml, 354–355
the XulAjaxInputDateRenderer extending XulRenderer, 329–330
for the XulAjaxShowOneDeckRenderer class encodeBegin() method, 335–336
for the XulDocumentRenderer encodeBegin() method, 326–327
for the XulDocumentRenderer encodeEnd() method, 328
com.apress.projsf prefix, use of in samples in book, 22

commandButton component, properties of, 23
component, defining in XBL vs. HTC, 206
COMPONENT element, in HTC, 201
component encapsulation, in HTC, 204–205
component families, a subset of all standard and their components, component types, and rendered types, 25–26
component family, associating a renderer to a particular one, 21
component library, weblets built-in support for versioning of, 217–219
component model, JSF (JavaServer Faces), 8
component type, mapping of it and UIComponent subclass, 22
component type and component class, code sample defining, 85–86
component writer, 4
component-based UI framework, JSF as, 13–15
componentForElement parameter, function of, 67
"Componentizing Web Applications", Web site address for proposal sent to W3C, 200
components
  attaching in XBL vs. in HTC, 207
  as the foundation of JSF, 3–48
  overview of those provided by the JSF implementation, 15–16
_constructor() function
  as core piece of the HTC element behavior, 377–379
  local variables and, 369–370
constructor method, supported by XBL, 196
<constructor> XBL element
  code sample in the bindings.xml file, 313
  function of, 312
content type
  dealing with, 250–251
  providing a custom via the JSP page directive, 250–251
content type and character encoding, 36
controller, FacesServlet that comes with JSF, 13
conversion errors, ConverterException thrown by getConvertedValue() method, 75
converters, function of, 22
converters and validators, creating, 276–279
converters, validators, events, and listeners, helper classes provided by JSF implementations, 22–23
createHttpRequest() function, that creates the XMLHttpRequest object, 185–186
createMethodBinding() method
  code sample for, 92
  function of, 93
createResponseWriter() method, function of, 35
createShowListener method, code sample for, 164
createUniqueId() method, calling to create a clientId, 64
createValueBinding() method, code sample for, 92
cross-platform support, importance of in developing RIAs, 208–209
CSS (Cascading Style Sheets), attaching an XBL component or behavior to a XUL application with, 193–195
CSS (Cascading Style Sheets) file
  a sample that has the -moz-binding property set, 194
  using to define the binding element, 321–322
custom action element, code showing how the name is defined, 99
custom action tag handlers, function of, 27

D

D² (D-squared) framework, function of in the Ajax-enabled deck component, 224
D² library, registering, 264
D² open source project
  introducing, 234–235
  using to process a response and modify the DOM in the target document, 247–248
d2_loadtext callback function, defined by D² library, 235
d2.js library
  code showing excerpt from, 235
  functions in, 235
d2.submit() function, dojo.io.bind() method called by, 235
data fetch request, sequence diagram of Mabon/Ajax, 284
data islands. *See* XML data islands
data-bound table component, code sample for, 276
dataTable component, adding a header facet to, 23–24
date component
  requirements for HTC implementations, 362
  XUL implementation of, 304–359
date field component
  Ajax enabling, 267–302
  defining, 49–104
  requirements for, 49–51
  sample page with, 55
DateValidator class
  function of, 269, 277–279
  function of in HTC Renderers, 387–388
deck component
  Ajax enabling, 223–265
  creating a UI prototype, 108–113
  creating a UI prototype for an Ajax-enabled, 227–230
  defining, 105–169
  designing an Ajax-enabled using a blueprint, 226–227
  designing using a blueprint, 107–169
  diagram showing classes created in Chapter 3, 106
  event handling overview, 114–115
  implemented in HTML, 227

Find it faster at http://superindex.apress.com/

deck component *(continued)*
implemented in HTML showing the Java item expanded, 108
as it would be used in a JSF JSP document, 112–113
list of classes, 107
requirements for, 106
requirements for Ajax implementation, 223–224
requirements for HTC implementations, 362
ShowItemTag class that represents leaf nodes of, 107
use of the alert() function attached to each item, 111
XUL implementation of, 304–359
decode() method
adding to the XulAjaxInputDateRenderer, 331
code sample for HtcAjaxShowOneDeckRenderer class, 395
code sample of, 340
decodes, processing of, 133–134
DecoratingRenderKitFactory class, class diagram of, 241
DEFAULT element, in HTC, 201
default page description, defined by the JSF specification, 12–13
DeferredContentType implementation, diagram over, 255
DeferredContentTypeResponse
code sample showing, 255–257
creating, 252
DeferredContentTypeResponse class
code sample showing, 257–258
decoration of the JSP HttpServletResponse with, 255–258
function of, 225
<deferred-method> syntax, function of, 102
DeferredPrintWriter class
code sample for, 261–262
function of, 225, 261–262
DeferredServletOutputStream class
code sample for, 259–260
function of, 225, 258–260
initial processing of the request to set the content type on HttpServletResponse, 259
<deferred-value> syntax, function of, 102
Delta DOM (D^2) framework, function of in the Ajax-enabled deck component, 224
Delta DOM Rendering (D^2R), as architectural solution for applying changes in Ajax, 231
descendant selectors, function of, 292–293
_determineDatePattern() method
code sample for, 290
code showing use of for XUL implementation, 333
for obtaining the date format pattern for the managed bean bound to the Validator, 387
_determinePattern() method, calling in encodeEnd() method, 331
_determineTargetURL() method
calling in encodeEnd() method, 331
code sample for, 290–291
code showing use of for XUL solution, 333–334
for obtaining the target URL for the managed bean bound to the Validator, 387
DHTML (dynamic HTML). *See also* Microsoft Dynamic HTML and HTC
for building rich Internet applications, 199
DHTML applications, building, 202–205
DHTML behaviors, 361
DHTML behaviors and HTC, Web site address for information about, 364–365
DHTML (dynamic HTML) toolkit, Dojo toolkit as, 234
direct and delegated implementation, for handling decode and encode, 24
dispatch() method, function of in Restore View phase in JSF lifecycle, 34
Dojo toolkit
code sample of Weblet configuration for, 263–264
providing file upload functionality in Ajax with, 233
registering, 263–264
Web site address for, 224
dojo.io.bind() method, function of, 235
DOM Inspector, a page's DOM tree with and XBL component, 195
DOM mutation, loss of changes made since last form POST in, 231
doShow() method, for ShowListener interface, 124
doStartTag() method, code sample for, 162–163
dynamic HTML (DHTML). *See* Microsoft Dynamic HTML and HTC
dynamic RenderKit ID, syntax for, 405–406
dynamic RenderKit implementation, structure of, 405
dynamic RenderKit managed bean, function of, 406–409
dynamic RenderKit solution, registering, 411–412
DynamicRenderKitViewHandler class
class diagram for, 409
function of, 405

E

element behavior, released with Internet Explorer 5.5, 203
_encodeAll() method
calling to render the header facet, 145–146

for handling any kind of child component added to the UIShowOne component, 339
encodeAll() method, recursively calling until all children have been rendered, 394
encodeBegin() method
arguments taken by, 138
arguments taken by for the XulDocumentRenderer, 326–327
function of, 37, 45
function of in client-specific renderer, 61
for the HtcAjaxShowOneDeckRenderer class, 389–391
of the HtmlDocumentRenderer class, 381–382
for the HtmlShowOneDeckRenderer, 137–138
obtaining a Map containing all available attributes from UIComponent, 382
for the XulAjaxShowOneDeckRenderer class, 335–336
encodeChildren() method
calling the _encodeAll() method to render the header facet, 145–146
for checking whether the UIShowOne component has any children, 337
code for checking whether each child is an instance of UIShowItem, 337–338
code for writing out the start element and any attributes, 338
function of in client-specific renderer, 61
for the HtcAjaxShowOneDeckRenderer class, 391–393
rendering the start of each UIShowItem child component, 142
of UIShowOne component, 141–144
encodeEnd() method
arguments, 62
for closing the generated markup for the UIShowOne components Renderer, 338–339
for closing the HTML <body> and <html> tags, 382–383
code sample using to get hold of the Converter, 330–331
combining all rendered output into a single, 144
function of, 38
function of in client-specific renderer, 61
of the HtmlInputDateRenderer class, 77
methods called to retrieve the date format pattern and the target URL, 331
encodeHead() method, responsible for writing out the <head> element, 383
encodeResources() method
automatic calling of during encodeBegin(), 57
code sample, 68
code sample for, 139, 289–290
code showing use of in XulAjaxShowOneDeckRenderer class, 336
for the HtcAjaxShowOneDeckRenderer class, 391
for the HtmlShowOneDeckRenderer class, 214
overriding to write a reference to the CSS style sheet, 67
enctype parameter, function of with <form> tag, 177
endDocument() method, provided by the JSF ResponseWriter class, 65
endElement() method
provided by the JSF ResponseWriter class, 65–66
using ResponseWriter's to improve performance, 139
eval() function, function of, 280
event and listener implementations, class diagram showing all classes needed for, 114
event delivery, in practice, 123–125
EVENT element, in HTC, 201
event handlers, predefined provided by the GRE DOM implementation, 192
event handling
comparing in XBL and HTC, 206–207
and HTC, 201–202
overview, 114–115
and XBL bindings, 197–198
event listener adapter, function of, 120–123
event listener interface, function of, 120
event subclass, creating, 117–119
EventListener registration, following the standard JavaBeans design pattern for, 129
events, using to be notified about changes to the UI or underlying model, 116
events and listeners, creating for deck components, 113–127
_expand() function, function of, 379–380
ExtendedRenderKit class
code sample of, 244–245
function of, 225
ExtendedRenderKitFactory class
code sample of, 241–243
function of, 226, 241–246

F

faces-config.xml file
code sample for the HTC registration in, 396, 397
managed bean defined in, 11
XUL registration in for XUL Ajax implementation, 354–355
FacesContext context argument, for encodeEnd() method, 62
FacesContextFactory
extending, 251–252
function of, 32–33

Find it faster at http://superindex.apress.com/

FacesContextFactory implementation, diagram over the FacesContext implementation, 251
FacesContextFactoryImpl class
  function of, 226
  support for additional processing of the servlet response object, 252–253
FacesContextFactoryWrapper class, function of, 226
FacesEvent base class, structure and method summary of, 116
FacesLifecycleServlet class, in Mabon, 281
FacesListener interface, as base interface for all default and custom listener interfaces in JSF, 119–120
FacesServlet
  controller for JSF, 13
  function of in JSF lifecycle, 33–34
facets
  within a dataTable component, 23–24
  processing facet and children of the UIShowItem component, 142–143
factories. *See* ApplicationFactory class; FacesContextFactory; LifecycleFactory; RenderKitFactory
field item, that can be added to the binding, 195
<field> XBL element, function of, 312
file upload, providing functionality for in Ajax, 233
_findFormClientId method, code sample, 145
Firefox, DOM mutation support in, 230
FixedContentTypeResponseWriter class
  class diagram for creating during Ajax postback, 248
  diagram illustrating the structure and dependencies of, 248
  function of, 225
  responsibility of, 247
<form> elements, illegality of nested in an HTML document, 367
<form> tag, parameters in a traditional Web application, 177
formClientId argument, to showOneDeck(), 139
frameworks, as JSF components, 7
<f:verbatim> tag
  using to render non-JSF content, 47
  wrapping around non-JSF content, 46–47
<f:view> tag
  function of, 35–36
  illustration showing the closing of, 38

G

GET method
  length restriction for, 177
  using when submitting a form, 177–178
getAgent() method, calling from the AdfFacesContext to obtain the user agent, 408–409
getAllResponseHeaders() method, XMLHttpRequest object, 176
getAttributes() method, for getting attribute values from the UIComponent, 63
getAvailability() method, code sample for with setAvailability() method, 278
getCharacterEncoding() method, provided by the JSF ResponseWriter class, 65
getClientId() method
  for getting a component's unique identifier, 336
  for obtaining clientId from the UIComponent that implements NamingContainer, 64
  for obtaining the clientId for the HtcAjaxShowOneDeckRenderer class, 390–391
getComponent() method, of the FacesEvent base class, 116
getContentTpe() method, provided by the JSF ResponseWriter class, 65
getConvertedValue() method
  for converting the submitted value to a strongly typed object, 332
  exception thrown if there are conversion errors, 75
  function of, 74–75
getConverter() method
  adding to the HtmlInputDateRenderer class to control value conversion, 69–70
  function of, 332
getCreated() method, for evaluating if parent UIComponentTag has a matching UIComponent, 163
getFacet() method
  adding getter and setter methods for the header facet with, 136
  calling to get the header facet from the UIShowItem component, 338
getHeader() method
  for getting the header facet from the UIShowItem component, 392–393
  getting the header facet from the UIShowItem component with, 143
getOutputStream() method, in the DeferredContentTypeResponse class, 258
getPhaseId() method, of the FacesEvent base class, 116
getRenderKitId() method, code sample with Use-Agent request header, 406–407
getRendersChildren() method
  code sample using, 393
  function of in client-specific renderer, 61
  using for rendering children for the UIShowOne component, 339
getResponseHeader("headerLabel") method, XMLHttpRequest object, 176

_getScriptResourceAlreadyWritten() method, for implementing at-most-one semantics for each script resource, 60
_getStyleResourceAlreadyWritten() method, for implementing at-most-one semantics for each style resource, 59
getStylesheetURI() method, for returning the value of the stylesheetURI attribute, 383–384
getTitle() method, for returning the value of the title attribute, 383–384
getValueAsString() method, for returning the string representation of the value to be encoded, 68–69
getWriter() method, in the DeferredContentTypeResponse class, 258

H

handleCommit() method, function of, 259–260
handleNavigation() method, use of in Invoke Application phase, 44–45
<handler> element, as child of the <handlers> element, 197–198
<handlers> element, function of, 197–198
<head> element, defining the element behavior prototype using <public:component> element, 368
header facet
  adding to both a dataTable component and a column component, 23–24
  getting from the UIShowItem component, 338
  using rather than a headerText attribute, 135
<h:form> tag
  execution of, 36–37
  output token and closing process of, 37–38
Hibernate, as JSF component, 7
Hollywood principle, meaning of, 7
<h:panelGrid> tag, use of in Invoke Application phase, 46–47
HTC. *See also* Microsoft HTC
  adding content in vs. XBL, 206
  assigning a call to a function with, 202
  associating a function with an event in, 202
  attaching components in vs. XBL, 207
  comparing XBL and, 206–207
  defining a component in vs. XBL, 206
  and event handling, 201–202
  event handling in vs. XBL, 207
  a globally assigned event handler in, 202
  page in Internet Explorer using HTC as a rendering technology, 210
  scripting languages supported by, 201
  summary, 205
  what it brings to JSF, 362
  vs. XAML, 199
HTC <pro:inputDate> _clickCell() function, sequence of function calls in, 373
HTC <pro:inputDate> _popup() function, responsible for launching the calendar when user clicks the button, 371–372
HTC <pro:inputDate> _scroll() function, sequence of function calls in, 372
HTC <pro:showItem> _expand() function, diagram of, 379
HTC components, using oncontentready vs. ondocumentready to initialize, 368
HTC date element behavior, 365–374
HTC deck element behavior, 376–380
HTC file structure
  and elements, 200–201
  typical, 366
HTC implementation
  of the deck and date components, 362–401
  JSF page source for, 399
HTC implementations, deck and date components' requirements for, 362
HTC public elements, used by HTC to define components, 201
HTC Renderers, providing, 361–401
HTC resources, registering with weblets, 398
HTC structure, 199–202
HtcAjaxDateRenderer class, function of, 384–388
HtcAjaxInputDateRenderer class
  class diagram for, 384
  function of, 363
HtcAjaxShowOneDeckRenderer class
  class diagram showing, 389
  the encodeBegin() method for, 389–391
  encodeResources() method, 391
  function of, 363, 388–396
HTC-specific elements, table of, 366
HTC-specific events, table of, 369
HTML date implementation prototype, implemented in HTC, 364–365
HTML deck implementation prototype, implemented in HTC, 375–376
HTML file, sample with XUL components, 194
HTML pages, using Ajax to filter a list of books based on category in, 182
HtmlAjaxInputDateRenderer
  class diagram showing, 279
  registering, 297
HtmlAjaxInputDateRenderer class
  ending the HtmlInputDateRenderer to add a pop-up calendar, 287–289
  function of, 269
HtmlAjaxInputDateRenderer resources, code for Weblet configuration for, 301
HtmlAjaxRenderKit class
  class diagram of, 243
  code sample of, 246–247
  function of, 225, 246–247
HtmlAjaxShowOneDeckRenderer class
  class diagram showing it extending the HtmlShowOneDeckRenderer, 235
  function of, 225

Find it faster at http://superindex.apress.com/

HtmlDocumentRenderer class
encodeBegin() method of, 381–382
function of, 363
a port of the XulDocumentRenderer, 381–382
HtmlInputDate.prototype._clickCell method
diagram of, 296
function of, 295–297
HtmlInputDate.prototype._scroll method, allowing the user to navigate plus or minus one month in the calendar, 294–295
HtmlInputDate.prototype.showPopup method, function of, 293–294
HtmlInputDateRenderer, using weblets in, 220
HtmlInputDateRenderer class
in charge of markup rendered to the client, 52
code sample showing the encodeEnd() method of, 77
import statements for, 60–61
registering as a renderer for JSF, 82–83
HtmlInputText component, illustrating the relationship between the component family and renderer type, 37
HtmlRenderer class
convenience methods provided by, 56–60
creating for encoding resources, 52
diagram showing HtmlShowOneDeckRenderer extending, 137
HtmlShowOneDeckRenderer class
in charge of the markup rendered to the client, 107
class diagram showing HtmlAjaxShowOneDeckRenderer class extending, 235
code sample extending, 235–236
encodeBegin() method for, 137–138
extending the HtmlRenderer, 136–148
registering in faces-config.xml, 153
using weblets in, 220–221
HTTP GET method. *See* GET method
HTTP POST method. *See* POST method
HttpServletResponse, overriding, 253–262

I

IllegalArgumentException, thrown if value passed doesn't conform to EL expression syntax, 90–91
<image> component, Mozilla XUL, 189
immediate attribute, setting on UICommand components, 97
<implementation> XBL element
code sample of in the bindings.xml file, 312
function of, 312
@import rule, function of, 272
input date component
building an application with, 103
code using with the Ajax Renderer, 272–273
designing using a blueprint, 52–53
HTML prototype for, 53–54
intent of, 51–52
using with the XUL Renderer, 357–358
InputDate component, controlling the decoding process of, 73
inputDate.css file, code excerpt from, 292
inputDate.css resource, 292–293
insertAdjacentHTML method, specific to Internet Explorer, 379
instance-based programming. *See* prototype-based programming
Internet Technologies. *See* Rich Internet Technologies (RITs)
Invoke Application phase
event handling in the application, 126
performing application logic in, 42–43
in request-processing lifecycle, 27–28
InvokeApplicationPhase class, in Mabon, 281
IoC, meaning of, 7
isActive flag, setting, 142
isAppropriateListener() method, of the FacesEvent base class, 116
isValueReference() method, for checking if a values is a JSF EL expression, 89
itemId argument, to showOneDeck(), 139

J

J2EE 1.4 API specification, for more information about the TagSupport and Tag classes, 162
J2EE architecture, common for a typical multitier software solution serving a retail company, 6, 10
Java Community Process (JCP), keeping up-to-date with emerging technologies and standards through, 5–7
Java Web applications, JavaServer Faces as user interface framework for, 3
JavaScript and the DOM, Web site address for information about, 141
JavaScript debugger, Mozilla's Venkman, 179
JavaServer Faces (JSF). *See* JSF (JavaServer Faces)
JavaServer Faces 1.1 specification, list of developer types from, 3–4
javax.faces, reserved for use by component families and component types, 22
JSF (JavaServer Faces)
application development with, 9–10
backing bean as plain old Java object (POJO) in, 11–12
component model, 8
declarative navigation model, 8–9
developer types, 3–4
developing smarter, 1–48
FacesServlet controller for, 13
formal lifecycle of, 27
the foundation of, 3–48

the greatest thing since sliced bread, 207–211
introducing, 8–47
page built with components using XUL as the rendering technology, 209
vs. Swing framework, 114–115
using to build an application, 28–31
what HTC brings to, 362
what it brings to HTC, 362
what Mozilla XUL brings to, 304
JSF 1.1 #{} expressions and JSF 2.0 ${} expressions, 99
JSF 1.1 specification, for more information about the UIComponentTag, 163
JSF 1.2
binding attribute added to all standard converter, validator, and listener tags by, 163
changes in to UIComponentTagSupport setStringProperty() method in, 89
changes in specification, 282
UIComponent.encodeAll(FacesContext) method added to, 146
JSF 1.2 tag handlers, use of JSP ValueExpression and MethodExpression types as parameters, 93
JSF Ajax
different approaches, 274–275
selecting an approach, 275–276
JSF and JSP, 45–47
JSF applications
including Ajax support in, 238–249
using weblets in, 221–222
JSF (JavaServer Faces) architecture, Model 2 pattern in, 10–11
JSF Central, Web site address for, 128
JSF community, resources available in, 128
JSF component
blueprint for creating a new date field component, 53
blueprint for creating a new deck component, 107–108
building blocks, 13
designing using a blueprint, 269–270
steps for creating a new Ajax-enabled deck component, 226–227
steps in blueprint for creating new, 306–307
JSF component libraries, custom servlet or filter solutions provided by, 214
JSF configuration file
adding a new renderer to, 26–27
code sample for renderer type as defined in, 25
navigation rules in JSF defined inside of, 8–9
JSF developer types, table of, 3–4
JSF EL syntax, using in the faces-config.xml file to set pointers to the managed bean, 405–406
JSF Expression Language (EL) expression, using, 88–89
JSF HTC components
building applications with, 398–401
designing using a blueprint, 363–398
JSF implementations
extending, 249–262
helper classes provided by for UIComponents, 22–23
overview of components provided by, 15–16
JSF implementers, 4
JSF JSP document, deck component as it would be used in, 112–113
JSF Lifecycle, event handling in, 125–127
JSF NamingContainer marker interface. *See* NamingContainer marker interface
JSF pages, enabling JSP support for, 86
JSF page source, for HTC implementation, 399–400
JSF specification
behavioral superclasses available in, 50–51
default page description defined by, 12–13
JSF tag handler, main purpose of, 27
JSF view identifier, viewID, 35
JSF view layer, function of, 12–13
JSF Web application
factory classes instantiated upon startup, 31–33
initial request, 33–39
players involved at application start-up, 32
JSF XUL components
building applications with, 357–359
using a blueprint for designing, 306–307
JSON (JavaScript Object Notation)
defined, 268
and Mabon, 280
syntax for showing the simple data source, 187
valid data types in, 280
JSP 2.0 ${} expressions and JSF 1.1 #{} expressions, 99
JSP 2.1, <deferred-value> and <deferred-method>, 102
JSP document
default processing of, 250
diagram showing processing of, 254
JSP page directive, providing a custom content type via, 250–251
JSP tag, as JSF component building block, 13
JSP tag handler
code sample for, 347–349
JSP tag handler and TLD
creating for defining the date field component, 86–102
creating for the date field component, 297–300
creating for the deck component, 155–169
creating for the ProDocument class, 347–350
<jsp:text> tag, for adding a label to each input field, 46
JSR-276: Design-Time Metadata for JavaServer Faces Components, currently under development, 153

Find it faster at http://superindex.apress.com/

L

Lifecycle approach, function of in Ajax enabling of the date field component, 275
Lifecycle phase
  effect of calling renderResponse() method during any, 71
  effect of calling responseComplete() method during any, 71
LifecycleFactory, function of, 32–33
LifecycleFactoryImpl class, for adding the MabonLifecycle, 282
LifecyclePhase class, in Mabon, 281
listArray argument, for the transferListItems() function, 186–187
Listbox renderer type, for UISelectOne component, 14
listener management interface, ShowSource interface, 128–129
Listeners, supported by JSF, 119–120
listeners and events, creating for deck components, 113–127
Login page
  code sample for building for a JSF application, 28–29
  example of, 29
login.jspx page, building, 28–29

M

Mabon (Managed Bean Object Notation)
  classes in, 281–282
  data fetch request, 283–285
  defined, 268
  initial request, 282–283
  and JSON, 280
  sequence diagram of at application start-up, 283
  sequence diagram of initial request, 283
  sequence diagram over Mabon lifecycle during postback, 284
  structure of, 281–282
  Web site address for, 268
  what it is, 279
Mabon APIs and how to register Mabon with an application, 285–291
Mabon JavaScript APIs, 285–286
Mabon lifecycle, sequence diagram over during postback, 284
Mabon protocol
  code sample of, 282
  code showing string returned after Mabon has evaluated it, 282
  used to reference the backing bean and a JavaScript convenience function, 279
  using, 286–287
Mabon send() function, passing arguments to, 286
Mabon servlet configuration, adding to the Web application configuration file, 285
Mabon/Ajax data fetch request, sequence diagram of, 284
mabon.js library, code sample, 286
MabonLifecycle class, phases in and function of, 281
MabonViewHandler class, function of, 282
managed bean, defined in the faces-config.xml file, 11
managed bean scopes, table of, 12
markup elements, leveraging some convenience methods to generate proper, 65–67
Menu renderer type, for UISelectOne component, 14
<menu> component, Mozilla XUL, 189
<menubar> component, Mozilla XUL, 189
<menuitem> component, Mozilla XUL, 189
<menupopup> component, Mozilla XUL, 189
METHOD element, in HTC, 201
method parameter, function of with <form> tag, 177
<method> XBL element, function of, 312
MethodBinding class
  use of by UICommand components, 92
  vs. ValueBinding class, 91
methods, that can be added to the binding, 195
Microsoft Dynamic HTML and HTC. *See also* DHTML; HTC
  introduction to, 199–205
Microsoft HTC. *See* HTC
Microsoft HTC Renderers. *See* HTC Renderers
Microsoft Internet Explorer, all of the Ajax pieces available in, 361
Microsoft Windows Explorer, expandable deck used in, 105
MIME type, weblets configuration file defining a custom, 217
Model 2 pattern, elements in, 10–11
Model-View-Controller (MVC) architecture, defined, 5
Moore, Gordon (Fairchild Camera and Instrument Corporation), quotation by, 4
Mozilla Amazon Browser, as example of SPIF application, 303
Mozilla Firefox browser, Mozilla XUL as development platform for, 187–199
Mozilla Thunderbird email client, Mozilla XUL as development platform for, 187–199
Mozilla XUL (XML User Interface Language). *See also* XUL applications; XUL components
  chrome system, 188
  creation of, 174
  deck implementation prototype, 317–318
  introduction to, 187–199
  what is needed to support, 323
  what JSF brings to, 304
Mozilla XUL renderers, providing, 303–359

Mozilla's Venkman JavaScript debugger, Web site address for, 179
multiple attribute, of <select> element, 14
MVC architecture, with JSF Model 2, 10–11

N

namespaced HTML elements, how to embed into base XUL controls, 190
NamingContainer, example of unique IDs within, 64
NamingContainer marker interface, function of, 65
navigation model, JSF (JavaServer Faces), 8–9
newInstance() method, for creating a new instance of a class, 164
nodeTypes, Web site address for information about, 181

O

objects, function of in JSON, 280
onblur event handler, function of, 192
oncommand event handler, function of, 192
onCommit() method
  calling on the DeferredContentTypeResponse, 259–260
  function of, 258
oncontentready HTC-specific event, function of, 369
oncontentsave HTC-specific event, function of, 369
ondetach HTC-specific event, function of, 369
ondocumentready HTC-specific event, function of, 369
one-tier applications, history of, 4–5
onfocus event handler, function of, 192
onget event handler, for getting the value attribute on your <pro:welcome> tag, 196
onload event handler, function of, 192
onreadystatechange property, XMLHttpRequest object, 176
onset event handler, for setting the value attribute on your <pro:welcome> tag, 196
open("method", "URL") method, XMLHttpRequest object, 176
OpenLaszlo's Amazon Store, as example of SPIF application, 303
Oracle ADF Faces
  getting information about the Apache MyFaces open source project, 403
  User-Agent abstraction provided by, 408

P

PackagedWeblet, function of, 216–217
page author, 3
<parameter> XBL element, function of, 312
Partial-Page Rendering (PPR), as first successful implementation of Ajax in JSF, 230–231
phaseId property, setting, 116–117
PhaseId values, table of valid, 117
PhaseListener approach, function of in Ajax enabling of the date field component, 275
POJO (plain old Java object), backing bean as, 11–12
populateBookList() function, in Ajax book filter implementation, 184–185
_popup() function, function of, 371–372
POST method
  length restriction for, 177
  using when submitting a form, 177–178
postback. *See also* Ajax postback; regular postback
  Ajax, 180–181
  decode on, 70–73
  form required for file upload functionality, 181
  with navigation in Invoke Application phase, 43–45
  process validation and conversion during, 73–75
  regular in traditional Web application development, 177–178
  render response on, 44
postback request, how JSF handles, 39–45
postback with navigation, in Invoke Application phase, 43–45
preventBackButtonFix workaround, for Dojo toolkit to be configured to work with XUL, 308
pro-bindings.xml, code sample for adding properties and methods, 196
process validation and conversion, during postback, 73–75
Process Validations phase
  conversion and validation, 40–41
  entered after the Apply Request Values phase, 73
  in request-processing lifecycle, 27–28
  use of the processValidators() and validate() methods in, 74
processApplication() method
  function of in Invoke Application phase, 126–127
  use of in Invoke Application phase, 42–43
processDecodes() method
  called on the UIViewRoot during the Apply Request Values phase, 71–72
  function of in Invoke Application phase, 126
  for processing decodes on the UIViewRoot, 133–134
  recursively calling for each UIComponent in the component hierarchy, 394
  use of during the Apply Request Values phase, 146–148
processListener() method, of the FacesEvent base class, 116

Find it faster at http://superindex.apress.com/

processUpdates() method
  calling on each UIComponent in the component hierarchy, 75–76
  function of in Update Model phase, 41–42
processValidators() method
  calling on the UIViewRoot, 40–41
  conversion and validation performed by calling on the UIViewRoot, 73
ProDocument class
  class diagram for, 341
  code sample for building, 342–344
ProDocument component, ProDocumentTag class representing, 305
ProDocument renderer-specific subclass, for the UIDocument class, 305
ProDocumentTag class
  class diagram showing, 347
  for creating the component that will create a ProDocument instance, 347–350
  representing the ProDocument component, 305
  Tag Library Descriptor (TLD) for, 349–350
ProDocumentTag handler, code for registering and setting rules for, 349–350
ProInputDate attributes, code sample, 86
ProInputDate class, as renderer-specific subclass, 52
ProInputDate component
  with attached dateValidator, 277
  implemented in DHTML/Ajax, 270
  providing support for the valueChangeListener attribute, 91
  requirements for Ajax implementation of, 267
  requirements for HTC implementation, 362
  requirements for XUL implementation, 304
  restoring and saving state in, 81–82
ProInputDate renderer-specific attributes, code for defining, 102
ProInputDate renderer-specific subclass, class diagram over, 78
<pro:inputDate> component
  code using the HTC <body> element, 366
  HTML markup needed for the HTC implementation of, 364–365
  implemented in HTC, 398–401
  implemented in HTML and HTC, 364
  JSF page rendered using the XUL RenderKit and, 357
<pro:inputDate> tag, code showing JSF document using, 103
ProInputDateTag class
  code sample, 93–94
  function of, 93–98
  tag handler for the date field component, 52
projsf-bindings.xml, code showing an XBL file containing one binding, 193
projsf.css, a sample CSS file that has the -moz-binding property set, 194
PROPERTY element, in HTC, 201
property item, that can be added to the binding, 195
<property> XBL element, function of, 312
<pro:showDeckOne> component, JSF page rendered using the XUL RenderKit and, 358
<pro:showItem> component, as part of the <pro:showOneDeck> component, 319
<pro:showItemActive> component, function of, 320–321
<pro:showItemHeader> component, function of, 321
ProShowOneDeck class
  class diagram showing, 148
  client-specific subclass, 128
  renderer-specific subclass, 107
ProShowOneDeck client-specific subclass, code sample for, 148–150
ProShowOneDeck component
  examining how to Ajax enable, 224–264
  ProShowOneDeckTag class that represents, 107
  requirements for HTC implementation, 362
  requirements for XUL implementation, 304
  running in multiple clients using client-specific markup, 412
ProShowOneDeck renderer-specific class, registering, 154–155
<pro:showOneDeck> component
  implemented in HTC, 400
  implemented in XUL, 317
  prototype implemented in HTML and HTC, 375
ProShowOneDeckTag class
  code sample for, 156–159
  that represents ProShowOneDeck component, 107
prototype-based programming, defined, 292
prototype-ch4.xul
  code sample of a XUL file with XBL components, 197
  a sample HTML file with XUL components, 194
prototype-oriented programming. *See* prototype-based programming
<public:attach> element, coupling an event raised on the client with an underlying function with, 200–201
<public:component>, used to define element and attached behavior types, 200
<public:event> HTC-specific element, for listing events that define the HTC component, 200
<public:method> HTC-specific element, for listing methods that define the HTC component, 200
<public:property> HTC-specific element, for listing properties that define the HTC component, 200

Q–R

queue() method, of the FacesEvent base class, 116

Radio renderer type, for UISelectOne component, 14
<radio> component, Mozilla XUL, 189
<radiogroup> component, Mozilla XUL, 189
readyState property, XMLHttpRequest object, 176
registering
  HtmlInputDateRenderer class as a renderer for JSF, 82–83
  the HtmlShowOneDeckRenderer class in faces-config.xml, 153
  ProShowOneDeck renderer-specific class, 154–155
  render-specific subclass in the faces-config.xml file, 83–86
  a UIComponent and Renderer, 82–86
  UIShowOne and UIShowItem, 150–155
regular postback
  obvious undesired side effects of, 178–179
  sequence diagram over, 178
  in traditional Web application development, 177–178
relative variables
  issues with, 275–276
  possible solutions to, 276
release() method
  for releasing the internal state used by a tag, 164
  for resetting all the internal storage, 98
render() method, function of in Restore View phase in JSF lifecycle, 34
Render Response phase
  during initial request in the JSF lifecycle, 34
  during postback, 76–77
  processing of the response object during, 253
  in request-processing lifecycle, 27–28
Renderer
  adding functionality to for detecting the Ajax request, 274
  client-side attributes provided by, 55
  registering UIComponent and, 82–86
Renderer and RenderKit, code sample to register the Ajax-enabled, 262–263
renderer types, function of, 25–26
RENDERER_TYPE, passing to the setRendererType() method, 78–79
Renderers
  function of, 24
  as JSF component building blocks, 13
  vs. UIComponents, 24
renderer-specific attributes
  accessing, 20
  providing convenience getters and setters for each, 79–80
renderer-specific component subclass
  creating, 77–82
  creating a new, 78–79
  creating for the document component, 341–344
  function of, 16–17
  as JSF component building block, 13
  using, 17–20
renderer-specific subclass, creating for the deck component, 148–150
RenderKit, registering to wrap, 239–240
RenderKit and JSF extension
  registering, 262–263
  registering for the HTC solution, 396–398
RenderKit and Renderer, code sample to register the Ajax-enabled, 262–263
RenderKit and ResponseWriter
  creating to provide Ajax functionality, 238–249
  creating to provide support for XML documents, 350–354
RenderKit ID
  code for setting the default, 405–406
  using the managed bean to set the default, 406
RenderKit identifier, using default as a base to locate agent-specific RenderKits for incoming requests, 409–410
RenderKitFactory class
  extending and wrapping the standard HTML RenderKit, 240
  function of, 32
RenderKitFactoryWrapper class, function of, 226
RenderKits
  function of, 404
  functionality of, 26–27
  as JSF component building block, 13
  registering, 354–355
  registering the dynamic RenderKit solution, 411–412
  requirements for dynamically switching, 404
  responsibility of in JSF, 9–10
  structure of dynamic implementation of, 405
  switching dynamically, 403–412
renderResponse() method
  calling to skip directly to the Render Response phase, 41–42
  effect of calling during any Lifecycle phase, 71
  function of in Restore View phase in JSF lifecycle, 34
RenderResponsePhase class, in Mabon, 281
rendersChildren property
  calling on a component, 45
  controlling rendering of child components with, 70
  function of in client-specific renderer, 62
  rendering of non-JSF content with it set to true, 46
  setting flag to true, 144

Find it faster at http://superindex.apress.com/

render-specific subclass, registering in the faces-config.xml file, 83–86
renderView() method, function of in Restore View phase in JSF lifecycle, 34
request managed bean scope, 12
request-processing lifecycle
  of JSF, 27–47
  navigation and completion of, 29–30
  phases of, 27
resource loading, introduction to, 213–215
resources, code for the unique keys used to identify, 60
responseComplete() method, effect of calling during any Lifecycle phase, 71
ResponseStateManager class, management of client-side state saving by, 132–133
responseText property, XMLHttpRequest object, 176–177
responseText type, function of, 280
ResponseWriter
  creating the right one to provide Ajax functionality, 239
  sequence diagram of creating the right one for the response, 351
ResponseWriter (JSF major), determining when the contentType should be set by, 254
ResponseWriter class
  creating and storing an instance of on the FacesContext, 35
  table of useful methods, 65–66
  using to leverage some convenience method to generate proper markup, 65–67
  using to write output to the client, 65–67
ResponseWriterWrapper class, function of, 225
responseXML property, XMLHttpRequest object, 176–177
responseXML type, function of, 280
Restore View phase
  during initial request in the JSF lifecycle, 33–34
  in request-processing lifecycle, 27–28
  restoring the saved state of the component hierarchy, 39–40
restoreView() method, using in the Restore View phase, 39–40
result.jspx page, 28–29
Rich Internet Applications (RIAs), 173–174
  importance of cross-platform support in developing, 208–209
Rich Internet Technologies (RITs), using, 173–211
RITs. *See* Rich Internet Technologies (RITs)
Russel, Alex, Dojo toolkit written in JavaScript by, 224

S

<script> elements, encapsulation of scripts in, 201
scripting languages, supported by HTC, 201
_scroll() function, allowing user navigation plus or minus one month in the calendar, 372–373
security, setting up for weblets, 219
<select> element, function of multiple attribute of, 14
selectId argument, for the transferListItems() function, 186–187
send(content) method, XMLHttpRequest object, 176
sequence diagram
  over regular postback, 178
  over the book filter XMLHttpRequest, 184
  over XMLHttpRequest postback, 180–181
Servlet specification, Web site address, 250
session managed bean scope, 12
setAvailability() method
  code sample for with getAvailability() method, 278
  for setting the method binding, 299
setBooleanProperty() method
  code sample method handling boolean attributes and properties, 90
  implementing for UIComponents, 88
setMethodBindingProperty() method, implementing for UIComponents, 88
setPhaseId method, of the FacesEvent base class, 116
setProperties() method, code sample, 97
setRendererType() method, passing the RENDERER_TYPE to, 78–79
setRequestHeader("label", "value") method, XMLHttpRequest object, 176
setStringProperty() method, implementing for UIComponents, 88–89
setSubmittedValue() method, calling only from the decode() method of components Renderer, 73
setValueBindingProperty() method
  code sample method handling ValueBinding attributes and properties, 90–91
  implementing for UIComponents, 88
shopping cart application, saving and restoring state, 20–21
ShowAdapter class
  for adapting a JSF 1.1 MethodBinding into a ShowListener instance, 121–123
  supports adding a MethodBinding as a ShowListener, 107
ShowEvent class
  custom event class, 107
  needed for the new UIComponents for the deck component, 113
showItemId behavioral attributes, code for accessor and mutator for, 131
ShowItemTag class
  code sample for, 160–161
  that represents leaf nodes of the deck component, 107

ShowListener class
a Listener interface, 107
needed for the new UIComponents for the deck component, 113
ShowListener interface
code sample for, 120
implementation of, 124–125
showListener tag, page source with, 123
ShowListenerTag class
code sample for, 161–162
representing a custom action for registering a ShowListener instance, 107
showOneClientId argument, to showOneDeck(), 139
showOneDeck(), arguments taken by, 139
showOneDeck Ajax implementation, 237–238
showOneDeck binding component, code sample of, 319
showOneDeck renderer, code for parameterized HTML for, 110–111
showOneDeck.js file, the source of, 140
showOneDeck.js library
the Ajax version of, 237
the HTML version of, 237
ShowSource class
isolates the event listener management methods, 128–129
for isolating the event listener management methods, 107
ShowSource interface, code for implementing, 132
single-page interface (SPIF) applications. *See* SPIF applications
SmallTalk, introduction of Model-View-Controller (MVC) architecture in, 5
SPIF applications
examples of, 303
RIAs that behave like desktop applications, 303
<splitter> component, Mozilla XUL, 189
startDocument() method, provided by the JSF ResponseWriter class, 65
startElement() method
arguments taken by, 328, 382
provided by the JSF ResponseWriter class, 65
using ResponseWriter's to improve performance, 139
state, saving and restoring, 20–21, 80–82
state management, as benefit of using JSF to build applications, 20–21, 80–82
state saving
drawbacks of on the server, 21
managing, 132–133
and restoring, 80–82
StateManager class
automatic state handling through, 80–82
function of, 20–21
management of server-side state saving by, 132–133
status property, XMLHttpRequest object, 176
statusText property, XMLHttpRequest object, 176
Struts, as JSF component, 7
style sheet, using to define the binding element, 321–322
Swing framework vs. JSF, 114–115

T

Tag Library Descriptor (TLD)
code for registering and setting rules for the ProDocumentTag handler, 349–350
code sample for, 165–168, 300
for grouping custom actions to make up a JSF tag library, 98–99
Tapestry, Struts,Tiles, TopLink, Hibernate, ADF UIX, as JSF components, 7
target parameter, function of with <form> tag, 177
three-tier or multitier (Web) applications, development of, 5
Tiles, as JSF components, 7
tools provider, 4
TopLink, as JSF component, 7
transferListItems() function, that returns the data requested and populates the <select> element, 186
two-tier or client-server applications, history and limitations of, 5

U

UI prototype
creating, 53–55, 108–113, 270–276
creating using HTML, DHTML behaviors, and HTC file types, 363–380
UIColumn component
available in JSF specification, 50
provided by the JSF implementation, 15
UICommand component
available in JSF specification, 50
provided by the JSF implementation, 15
setting immediate attribute on, 97
use of method-binding expressions to reference, 92
UIComponent
as JSF component building block, 13
separation of from behavior and data model, 14
UIComponent and Renderer, registering, 150–155, 238
UIComponent and renderer
registering, 297
registering the JSF XUL implementation, 345–346
UIComponent attributes, code sample, 99–100
UIComponent component argument, for encodeEnd() method, 62
UIComponent inheritance, example of, 17

Find it faster at http://superindex.apress.com/

UIComponentBase class
code sample for, 129–130
subclass that implements almost all methods of UIComponent class, 15
UIComponent.encodeAll(FacesContext) method, added to the JSF 1.2 release, 146, 340
UIComponent-inherited attributes, code sample, 84
UIComponents
as cornerstones of a JSF application, 12–13
identifying, 63–64
registering, 82–86
vs. Renderers, 24
that differentiate JSF from other technologies, 13–15
utility methods for handling attributes for, 88
UIComponentTagSupport class, code sample for, 88
UIComponentTagSupport setStringProperty() method, changes in for JSF 1.2, 89
UIComponentTagSupport tag handler class
creating, 87–98
for providing functionality that is common among all components, 52
UIData component
available in JSF specification, 50
as example of a MethodBinding expression using relative variables, 275–276
provided by the JSF implementation, 15
UIDocument attributes, code sample of the getters for, 328–329
UIDocument class
function of, 305
introducing, 323–324
UIForm component
available in JSF specification, 50
getting the ID of, 141
provided by the JSF implementation, 15
UIGraphic component
available in JSF specification, 50
provided by the JSF implementation, 15
UIInput component
available in JSF specification, 50
provided by the JSF implementation, 15
setting immediate attribute on, 97
using, 51
UIInput-inherited attributes, code sample, 85–86
UIMessage component
available in JSF specification, 50
provided by the JSF implementation, 15
UIMessages component
available in JSF specification, 50
provided by the JSF implementation, 15
UIOutput component
available in JSF specification, 50
provided by the JSF implementation, 15–16
UIPanel component
available in JSF specification, 50
provided by the JSF implementation, 16
UIParameter component
available in JSF specification, 50
provided by the JSF implementation, 16
UISelectBoolean component, available in JSF specification, 50
UISelectItem component
available in JSF specification, 51
provided by the JSF implementation, 16
UISelectItems component
available in JSF specification, 51
provided by the JSF implementation, 16
UISelectMany component
available in JSF specification, 51
behavior of, 14
provided by the JSF implementation, 16
UISelectOne component
available in JSF specification, 51
behavior of, 14
and its renderers, 14
provided by the JSF implementation, 16
renderer types, 14
UISelectOneBoolean component, provided by the JSF implementation, 16
UIShowItem behavioral superclass
acts as a clickable parent container that shows or hides its children, 113
for adding labeled items to the deck component, 134–136
class diagram of the implementation, 135
representing each child component to the UIShowOne component, 107
UIShowItem component
code for processing facet and children of, 142–143
processing facet and children of, 142–143
UIShowOne behavioral superclass
acts as a top-level container controlling which child component to display, 107
class diagram showing implementation of, 127
function of, 129–134
handling of associated listeners, 131–132
for keeping track of which node the user has selected, 113
UIShowOne component
encoding the children of, 141–144
getting the ID of, 141
JavaScript implementation of, 139–141
UIViewRoot component
method for processing decodes on, 133–134
provided by the JSF implementation, 16
responsible for calling processDecodes() on each UIComponent, 71–72
UIViewRoot view identifier, 35
Update Model phase, updating the underlying model in, 41–42

Update Model Values phase
entering after the Process Validations phase, 75–76
the processUpdates() and updateModel() methods in, 76
in request-processing lifecycle, 27–28
UxlAjaxShowOneDeckRenderer class, function of, 305

V

validate() method, code sample for, 277–278
ValidateDateTag class
actual code behind, 298–299
class diagram for, 298
function of, 269
Validator
creating to perform validation on a strongly typed Date object, 276–279
use of for Ajax enabling the date field component, 273
validators, function of, 22
validators and converters, creating, 276–279
ValueBinding class
function of, 77
vs. MethodBinding class, 91
and renderer-specific attributes, 79–80
valueChangeListener attribute, providing support for, 91
values
converting, 69–70
in JSON, 280
Venkman JavaScript debugger, Mozilla's, 179
video terminals (VTs), 6
view layer. *See* JSF view layer
viewHandler, function of, 38
ViewHandler.renderView() method, function of in Restore View phase in JSF lifecycle, 34
ViewHandlerWrapper class, function of, 405
viewlink property, manually setting on the defaults declaration, 204–205
visual calendar, requirement for Ajax implementation of ProInputDate component, 267
VTs (video terminals), 6

W

W3C HTTP specification, Web site address for, 177
Web application. *See* JSF Web application
Web applications, traditional development of, 177–179
Web application startup, upon receiving a JSF request, 31–33
Web Hypertext Applications Technology (WHAT), working to create a standard tag library for extensions to HTML, 207–208
Web site address
for "A Modular Way of Defining Behavior for XML and HTML" (Netscape), 193
for Apache MyFaces, 128
for "Behavioral Extensions to CSS", 200
for "Componentizing Web Applications" proposal sent to W3C, 200
for Dojo toolkit written in JavaScript by Alex Russel, 224
for HttpServletResponse object Servlet specification, 250
for information about DHTML behaviors and HTC, 364
for information about HTML elements and their supported attributes, 54
for information about JavaScript and the DOM, 141
for information about JSON, 280
for information about Mabon open source project, 279
for information about nodeTypes, 181
for information about visual formatting using block boxes, 377
for JSF Central, 128
for Mabon information, 268
for Mozilla Amazon Browser, 303
for Mozilla's Venkman JavaScript debugger, 179
for OpenLaszlo's Amazon Store, 303
for subset of available XUL components, 189
for W3C HTTP specification, 177
for weblets open source project, 213
for the Weblets project, 263
for WHAT, 208
for wikipedia, 292
for XHTML 1.0 specification, 65
for XULPlanet, 312
Web technologies, emergence of new, 173–174
weblet architecture, exploring, 215–222
Weblet configuration
for the D^2 library, 264
for the Dojo toolkit, 264
weblet filter, using to optimize weblets, 221–222
weblet protocol
syntax for, 219
using to serve up resources, 220
WebletContainer, function of, 216
weblets
configuration file using 1.0 versioning for production, 218
configuration file using SNAPSHOT versioning for development, 219
exploring the architecture of, 215–222
loading resources with, 213–222
registering HTC resources with, 398
registering XUL resources with, 355–356
registering your Ajax resources with, 301
setting up security for, 219
specifying MIME types, 217

Find it faster at http://superindex.apress.com/

weblets *(continued)*
specifying versioning of the component library, 217–219
using as a mediator that intercepts requests from the client, 215
using in the HtmlInputDateRenderer, 220
using in the HtmlShowOneDeckRenderer, 220–221
using in your component library, 216–221
using the weblet protocol, 219
Web site address for information about, 213, 355
weblets mapping
overriding, 222
Weblets project, Web site address for information about, 263
WebletsPhaseListener, function of, 216
WebletsViewHandler, function of, 215
web.xml file, weblet container configuration in, 221–222
welcome HTC component, example of a page using, 204
welcome XBL component, example of a page using, 196
WHAT. *See* Web Hypertext Applications Technology (WHAT)
wikipedia, Web site address, 292
<window> component, Mozilla XUL, 189
World Wide Web Consortium (W3C), keeping up-to-date with emerging technologies and standards through, 5–7
writeAttribute() method, provided by the JSF ResponseWriter class, 65, 66
writeComment() method, provided by the JSF ResponseWriter class, 66
writeScriptInline() method, function of, 386
writeScriptResource() method
function of, 386
for guaranteeing script resourse is written only once during rendering, 139
for writing script resource to the client, 58–59
writeState() method, called by the </f:view> end tag, 38
writeStyleResource() method
guarantees a style resource is written only once during rendering, 68
for writing style resources to the client, 58
writeText() method, provided by the JSF ResponseWriter class, 66
writeURIAttribute() method, provided by the JSF ResponseWriter class, 66

X

XAML vs. HTC, 199
XBL (Extensible Binding Language)
adding content in vs. HTC, 206
attaching components in vs. HTC, 207
comparing HTC and, 206–207
deck component prototype, 318–321
defined, 192
defining a component in vs. HTC, 206
elements used in the <pro:inputDate> component, 312
event handling in vs. HTC, 207
function of, 174
sequence diagram of the inputDate binding popup method, 314
types of items that can be added to the binding, 195
using to add new properties and methods, 195–197
using to create custom XUL components, 192–198
XBL bindings
creating, 192–193
event handling and, 197–198
extending, 195–197
a simple XUL page using with attached event handler, 198
using, 193–195
XBL component, a page's DOM tree with, 195
XBL Date component prototype, function of, 308–318
XHTML 1.0 specification, Website address for, 65
XML data islands, in Internet Explorer, 367–368
XMLHTTP. *See* Ajax (Asynchronous JavaScript and XML)
XMLHttpRequest object
in Ajax, 174–187
creating, 185–186
creating an instance of, 175
introduced by Microsoft, 361
methods provided by, 175–176
properties common to all implementations, 176–177
sequence diagram of using the HTTP GET method, 274
XMLHttpRequest object methods, parameters required by, 176
XMLHttpRequest postback, sequence diagram over, 180–181
XMLResponseWriter, code sample for, 353–354
XMLResponseWriter class
function of, 305
for writing the required XML markup to the requesting client, 353–354
XUL. *See* Mozilla XUL (XML User Interface Language)
XUL applications. *See also* Mozilla XUL (XML User Interface Language)
building, 188–192
deploying and running on a remote server, 190
XUL button. *See also* Mozilla XUL (XML User Interface Language)
adding one that triggers the oncommand event handler, 196–197
XUL components. *See also* Mozilla XUL (XML User Interface Language)

base set of available through the Mozilla GRE, 188–189
creating custom using XBL, 192–198
XUL date implementation prototype. *See also* Mozilla XUL (XML User Interface Language)
the <pro:inputDate> component implemented in XUL, 307
XUL elements. *See also* Mozilla XUL (XML User Interface Language)
used in the "Providing Mozilla XUL Renderers" chapter, 309–310
XUL event handling. *See also* Mozilla XUL (XML User Interface Language)
events, state, and data, 190–192
XUL file. *See also* Mozilla XUL (XML User Interface Language)
a simple rendered in the Firefox browser, 191
a simple with embedded HTML elements, 190
XUL for Web development. *See also* Mozilla XUL (XML User Interface Language)
events, state, and data, 190–192
XUL implementations. *See also* Mozilla XUL (XML User Interface Language)
requirements for the Deck and Date components', 304
XUL Renderers, class diagram showing, 325
XUL resources, registering with Weblets, 355–356
XulAjaxInputDateRenderer class
class diagram showing, 329
code sample extending the XulRenderer, 329–330
function of, 305
XulAjaxRenderKit class
class diagram showing, 351
function of, 305, 351–352
setting the CONTENT_TYPE variable to the accepted XUL contentType, 352
XulAjaxShowOneDeckRenderer class
arguments taken by the encodeBegin() method, 336
class diagram for, 334
encodeResources() method, 336
function of, 334–341
getters from the different style classes supported by, 340–341
<xul:button> element, function of, 310
<xul:column> element, function of, 310
<xul:columns> element
function of, 310
XulDocumentRenderer class
arguments taken by the startElement() method, 328
class diagram showing, 325
encodeBegin() method, 326–327
function of, 305, 325–329
<xul:grid> element, function of, 310
<xul:hbox> element, function of, 309
<xul:label> element, function of, 309
<xul:popup> element, function of, 310
<xul:popupset> element, function of, 310
XulRenderer class, function of, 305
<xul:row> element, function of, 310
<xul:rows> element, function of, 310
<xul:textnode> element, function of, 309
<xul:vbox> element, function of, 309

Find it faster at http://superindex.apress.com/

FIND IT FAST with the Apress *SuperIndex*™

Quickly Find Out What the Experts Know

Leading by innovation, Apress now offers you its ***SuperIndex***™, a turbocharged companion to the fine index in this book. The Apress ***SuperIndex***™ is a keyword and phrase-enabled search tool that lets you search through the entire Apress library. Powered by dtSearch™, it delivers results instantly.

Instead of paging through a book or a PDF, you can electronically access the topic of your choice from a vast array of Apress titles. The Apress ***SuperIndex***™ is the perfect tool to find critical snippets of code or an obscure reference. The Apress ***SuperIndex***™ enables all users to harness essential information and data from the best minds in technology.

No registration is required, and the Apress ***SuperIndex***™ is free to use.

❶ Thorough and comprehensive searches of over 300 titles

❷ No registration required

❸ Instantaneous results

❹ A single destination to find what you need

❺ Engineered for speed and accuracy

❻ Will spare your time, application, and anxiety level

Search now: ***http://superindex.apress.com***

SuperIndex

Apress®
THE EXPERT'S VOICE™

You Need the Companion eBook

Your purchase of this book entitles you to its companion eBook for only $10.

We believe this Apress title will prove so indispensable that you'll want to carry it with you everywhere, which is why we are offering the companion eBook for $10 to customers who purchase this book now. Convenient and fully searchable, the eBook version of any content-rich, page-heavy Apress book makes a valuable addition to your programming library. You can easily find, copy, and apply code—and then perform examples by quickly toggling between instructions and the application. Even simultaneously tackling a donut, diet soda, and complex code becomes simplified with hands-free eBooks!

Once you purchase this book, getting the $10 companion eBook is simple:

1. Visit **www.apress.com/promo/tendollars/**.
2. Complete a basic registration form to receive a randomly generated question about this title.
3. Answer the question correctly in 60 seconds and you will receive a promotional code to redeem for the $10 eBook.

2560 Ninth Street • Suite 219 • Berkeley, CA 94710

Apress eBookshop

ASPToday

Apress® THE EXPERT'S VOICE™

All Apress eBooks subject to copyright protection. No part may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher. The purchaser may print the work in full or in part for their own non-commercial use. The purchaser may place the eBook title on any of their personal computers for their own personal reading and reference.

Offer valid through 8/20/06.

forums.apress.com

FOR PROFESSIONALS BY PROFESSIONALS™

JOIN THE APRESS FORUMS AND BE PART OF OUR COMMUNITY. You'll find discussions that cover topics of interest to IT professionals, programmers, and enthusiasts just like you. If you post a query to one of our forums, you can expect that some of the best minds in the business—especially Apress authors, who all write with *The Expert's Voice*™—will chime in to help you. Why not aim to become one of our most valuable participants (MVPs) and win cool stuff? Here's a sampling of what you'll find:

DATABASES

Data drives everything.

Share information, exchange ideas, and discuss any database programming or administration issues.

PROGRAMMING/BUSINESS

Unfortunately, it is.

Talk about the Apress line of books that cover software methodology, best practices, and how programmers interact with the "suits."

INTERNET TECHNOLOGIES AND NETWORKING

Try living without plumbing (and eventually IPv6).

Talk about networking topics including protocols, design, administration, wireless, wired, storage, backup, certifications, trends, and new technologies.

WEB DEVELOPMENT/DESIGN

Ugly doesn't cut it anymore, and CGI is absurd.

Help is in sight for your site. Find design solutions for your projects and get ideas for building an interactive Web site.

JAVA

We've come a long way from the old Oak tree.

Hang out and discuss Java in whatever flavor you choose: J2SE, J2EE, J2ME, Jakarta, and so on.

SECURITY

Lots of bad guys out there—the good guys need help.

Discuss computer and network security issues here. Just don't let anyone else know the answers!

MAC OS X

All about the Zen of OS X.

OS X is both the present and the future for Mac apps. Make suggestions, offer up ideas, or boast about your new hardware.

TECHNOLOGY IN ACTION

Cool things. Fun things.

It's after hours. It's time to play. Whether you're into LEGO® MINDSTORMS™ or turning an old PC into a DVR, this is where technology turns into fun.

OPEN SOURCE

Source code is good; understanding (open) source is better.

Discuss open source technologies and related topics such as PHP, MySQL, Linux, Perl, Apache, Python, and more.

WINDOWS

No defenestration here.

Ask questions about all aspects of Windows programming, get help on Microsoft technologies covered in Apress books, or provide feedback on any Apress Windows book.

HOW TO PARTICIPATE:

Go to the Apress Forums site at **http://forums.apress.com/**.

Click the New User link.