

THE EXPERT'S VOICE® IN JAVA™ TECHNOLOGY

Beginning JSF™ 2 APIs and JBoss® Seam

*Begin using the new JavaServer™ Faces (JSF™) 2
APIs available in the new Java™ EE 6 platform*

Kent Ka lok Tong

apress®

Beginning JSF™ 2 APIs and JBoss® Seam



Kent Ka lok Tong

Beginning JSF™ 2 APIs and JBoss® Seam

Copyright © 2009 by Kent Ka lok Tong

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 (pbk): 978-1-4302-1922-4

ISBN-13 (electronic): 978-1-4302-1923-1

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.

Java™ and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc., in the US and other countries. Apress, Inc., is not affiliated with Sun Microsystems, Inc., and this book was written without endorsement from Sun Microsystems, Inc.

Lead Editors: Steve Anglin, Matt Moodie

Technical Reviewer: Jim Farley

Editorial Board: Clay Andres, Steve Anglin, Mark Beckner, Ewan Buckingham, Tony Campbell, Gary Cornell, Jonathan Gennick, Michelle Lowman, Matthew Moodie, Jeffrey Pepper, Frank Pohlmann, Ben Renow-Clarke, Dominic Shakeshaft, Matt Wade, Tom Welsh

Project Manager: Sofia Marchant

Copy Editors: Kim Wimpsett and Heather Lang

Associate Production Director: Kari Brooks-Copony

Production Editor: Ellie Fountain

Compositor and Artist: Kinetic Publishing Services, LLC

Proofreader: Patrick Vincent

Indexer: Toma Mulligan

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 2855 Telegraph Avenue, Suite 600, Berkeley, CA 94705. Phone 510-549-5930, fax 510-549-5939, e-mail info@apress.com, or visit <http://www.apress.com>.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales—eBook Licensing web page at <http://www.apress.com/info/bulksales>.

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.

The source code for this book is available to readers at <http://www.apress.com>. You will need to answer questions pertaining to this book in order to successfully download the code.

Contents at a Glance

About the Author.....	ix
About the Technical Reviewer	xi
CHAPTER 1 Getting Started with JSF.....	1
CHAPTER 2 Using Forms	29
CHAPTER 3 Validating Input	67
CHAPTER 4 Creating an E-shop.....	101
CHAPTER 5 Creating Custom Components	151
CHAPTER 6 Providing a Common Layout for Your Pages	173
CHAPTER 7 Building Interactive Pages with Ajax	183
CHAPTER 8 Using Conversations	215
CHAPTER 9 Supporting Other Languages.....	231
CHAPTER 10 Using JBoss Seam	253
INDEX	287

Contents

About the Author	ix
About the Technical Reviewer	xi

CHAPTER 1	Getting Started with JSF	1
	Introducing the “Hello world” Application	1
	Installing Eclipse	2
	Installing JBoss	3
	Installing a JSF Implementation	7
	Installing Web Beans	8
	Creating the “Hello world!” Application with JSF	9
	Generating Dynamic Content	17
	Retrieving Data from Java Code	20
	Exploring the Life Cycle of the Web Bean	25
	Using an Easier Way to Output Text	25
	Debugging a JSF Application	25
	Summary	27

CHAPTER 2	Using Forms	29
	Developing a Stock Quote Application	29
	Getting the Stock Quote Symbol	29
	Displaying the Result Page	36
	Displaying the Stock Value	38
	Marking Input As Required	40
	Inputting a Date	49
	Conversion Errors and Empty Input	55
	Using a Combo Box	60
	Using a Single b2 Bean	62
	Hooking Up the Web Beans	63
	Summary	66

CHAPTER 3	Validating Input	67
	Developing a Postage Calculator	67
	What If the Input Is Invalid?	73
	Null Input and Validators	78
	Validating the Patron Code	80
	Creating a Custom Validator for the Patron Code	82
	Displaying the Error Messages in Red	86
	Displaying the Error Message Along with the Field	87
	Validating a Combination of Multiple Input Values	96
	Summary	100
 CHAPTER 4	 Creating an E-shop	 101
	Listing the Products	102
	Making the Link to Show the Details	106
	Displaying Headers in the Columns	115
	Implementing a Shopping Cart	116
	Displaying the Content of the Shopping Cart	126
	The Checkout Function	127
	Getting the Credit Card Number of the Current User	131
	Forcing the User to Log In	139
	Implementing Logout	146
	Protecting the Password	148
	Summary	149
 CHAPTER 5	 Creating Custom Components	 151
	Displaying a Copyright Notice on Multiple Pages	151
	Allowing the Caller to Specify the Company Name	157
	Creating a Product Editor	159
	Passing a Method in a Parameter?	162
	Creating a Box Component	163
	Accepting Two Pieces of XHTML Code	166
	Creating a Reusable Component Library	168
	Creating a Component Library Without taglib.xml	170
	Summary	172

CHAPTER 6	Providing a Common Layout for Your Pages	173
	Using the Same Menu on Different Pages	173
	Using Global Navigation Rules	177
	Using Two Abstract Parts	178
	Creating Page-Specific Navigation Cases	180
	Summary	182
CHAPTER 7	Building Interactive Pages with Ajax	183
	Displaying a FAQ	183
	Refreshing the Answer Only	185
	Hiding and Showing the Answer	189
	Using Ajax to Hide or Show the Answer	191
	Giving a Rating to a Question	194
	Updating the Rating as the User Types	199
	Using a Dialog Box to Get the Rating	200
	Setting the Look and Feel with Skins	204
	Displaying Multiple Questions	206
	Summary	212
CHAPTER 8	Using Conversations	215
	Creating a Wizard to Submit Support Tickets	215
	Interference Between Browser Windows	219
	URL Mismatched?	225
	Summary	229
CHAPTER 9	Supporting Other Languages	231
	Displaying the Current Date and Time	231
	Supporting Chinese	232
	Easier Way to Access Map Elements	237
	Internationalizing the Date Display	238
	Letting the User Change the Language Used	238
	Localizing the Full Stop	243
	Displaying a Logo	246
	Making the Locale Change Persistent	248
	Localizing Validation Messages	250
	Summary	251

CHAPTER 10	Using JBoss Seam	253
	Installing Seam.....	253
	Re-creating the E-shop Project.....	254
	Allowing the User to Add Products.....	257
	Restricting Access to the Product-Editing Page.....	265
	Creating a Shopping Cart.....	267
	Turning the Shopping Cart into a Stateful Session Bean	273
	Creating the Checkout Page	277
	Using WebLogic, WebSphere, or GlassFish.....	284
	Summary.....	284
INDEX		287

About the Author

■ **KENT KA IOK TONG** is the manager of the IT department of the Macau Productivity and Technology Transfer Center. With a master's degree in computer science from the University of New South Wales in Sydney, Australia, and having won the Macao Programming Competition (Open Category) in 1992, Kent has been involved in professional software development, training, and project management since 1993. He is the author of several popular books on web technologies including *Essential JSF*, *Facelets and Seam*, *Enjoying Web Development with Tapestry*, *Enjoying Web Development with Wicket*, and *Developing Web Services with Apache Axis 2*.

About the Technical Reviewer

■ **JIM FARLEY** is a technology architect, strategist, writer, and manager. His career has touched a wide array of domains, from commercial to nonprofit and from finance to higher education. In addition to his day job, Jim teaches enterprise development at Harvard University. Jim is the author of several books on technology and contributes articles and commentary to various online and print publications.



Getting Started with JSF

In this chapter you'll learn how to set up a development environment and create a "Hello world!" application with JSF.

Introducing the "Hello world" Application

Suppose that you'd like to develop the application shown in Figure 1-1.

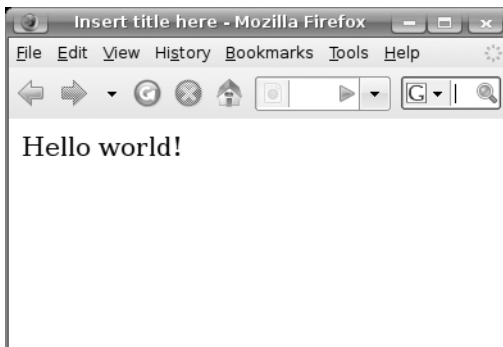


Figure 1-1. A simple "Hello world!" application with a single page

To do that, you'll need to install some software (see Figure 1-2). First, you'll need an IDE to create your application. This book will use Eclipse, but other popular IDEs will do just fine too. Next, you'll need to install JBoss, which provides a platform for running web applications (there are also fine alternatives to JBoss). In addition, your application will use JSF and Web Beans as libraries. So, you'll need to download them too.

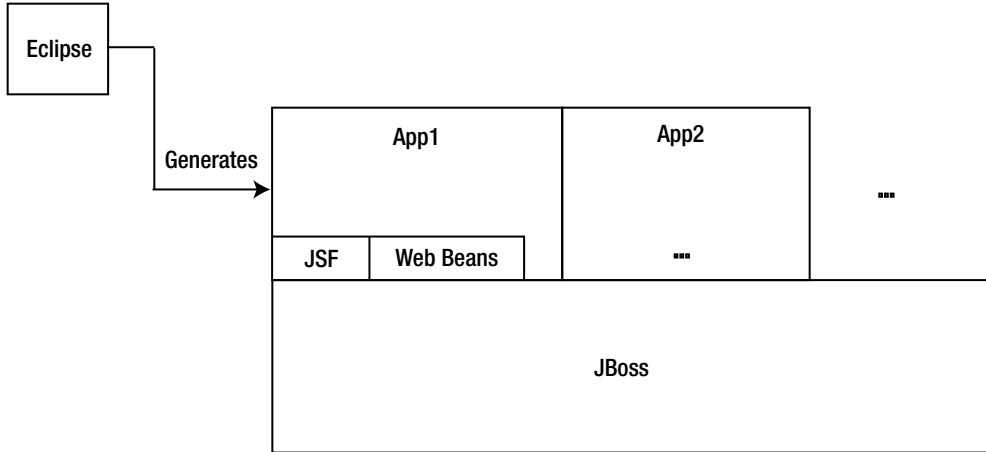


Figure 1-2. *The software that you'll need*

Installing Eclipse

You need to make sure you have the Eclipse IDE for Java EE Developers, as shown in Figure 1-3 (note that the Eclipse IDE for Java Developers is *not* enough, because it doesn't include tools for developing web applications). You can go to <http://www.eclipse.org> to download it. For example, you'll need the `eclipse-jee-ganymede-SR1-win32.zip` file if you use Windows. Unzip it into a convenient location, such as `c:\eclipse`. Then, create a shortcut to run `c:\eclipse\eclipse -data c:\workspace`. This way, it will store your projects under the `c:\workspace` folder.

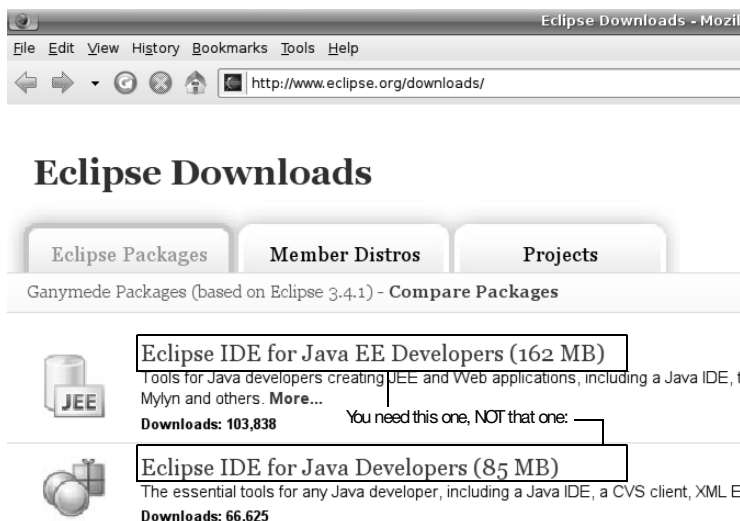


Figure 1-3. *Getting the right bundle of Eclipse*

To see whether it's working, run it, and make sure you can switch to the Java EE perspective (it should be the default; if not, choose Window ► Open Perspective ► Other), as shown in Figure 1-4.

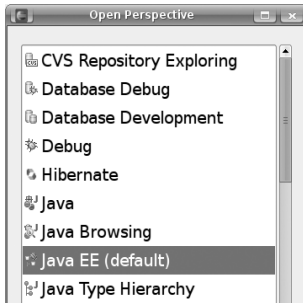


Figure 1-4. *The Java EE perspective*

Installing JBoss

To install JBoss, go to <http://www.jboss.org/jbossas/downloads> to download a binary package of JBoss Application Server 5.x (or newer), such as jboss-5.0.1.GA.zip. Unzip it into a folder such as c:\jboss. To test whether it is working, you can try to launch JBoss in Eclipse. To do that, choose Windows ► Preferences in Eclipse, and then choose Server ► Installed Runtime Environments. You'll see the window shown in Figure 1-5.

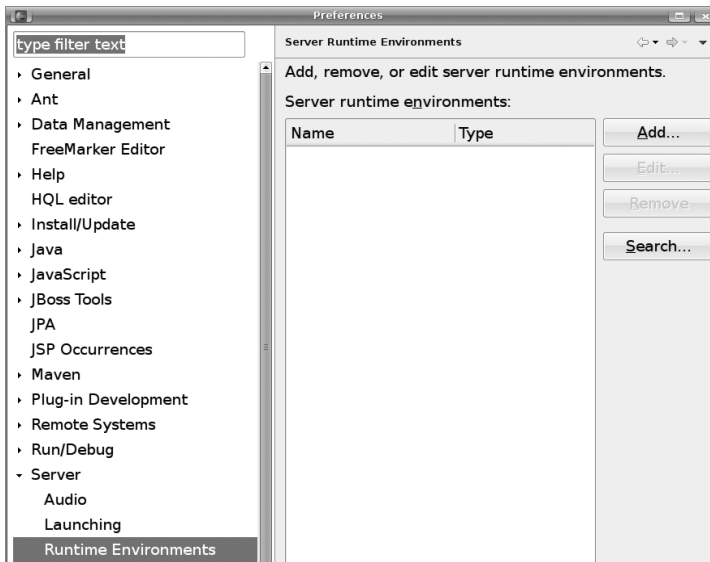


Figure 1-5. *The installed runtime environments*

Click Add, and choose JBoss ► JBoss v5.0 (Figure 1-6).

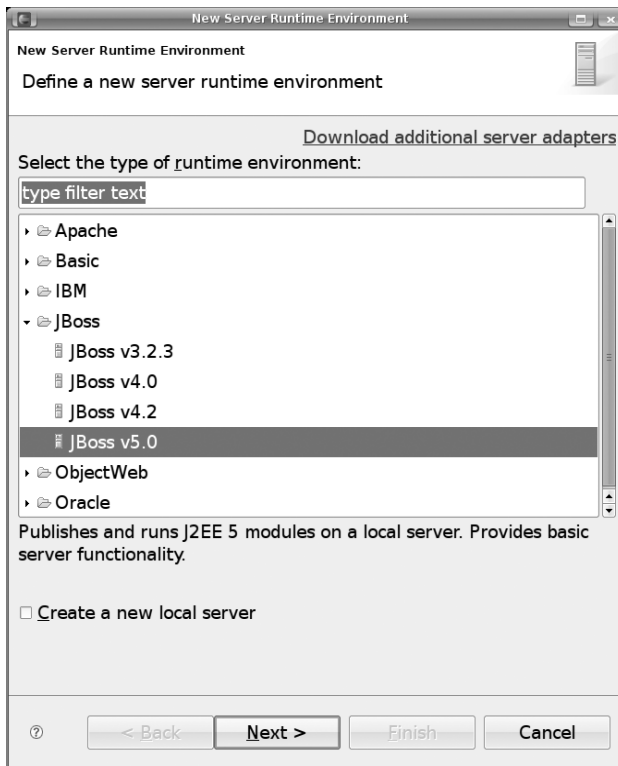


Figure 1-6. *The JBoss 5.0 runtime*

Click Next. Specify **c:\jboss** as the application server directory (Figure 1-7).

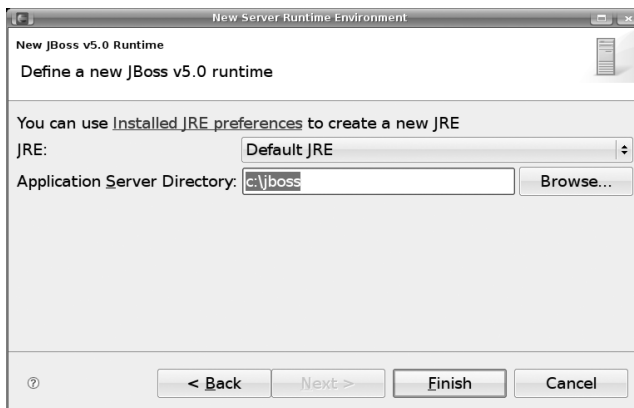


Figure 1-7. *Specifying the JBoss application server directory*

Click Finish. Next, you need to create a JBoss instance. In the bottom part of the Eclipse window, you'll see a Servers tab (you'll see this tab only when you're in the Java EE perspective); right-click anywhere on the tab, choose New ► Server, and choose the JBoss v5.0 server runtime environment (Figure 1-8).

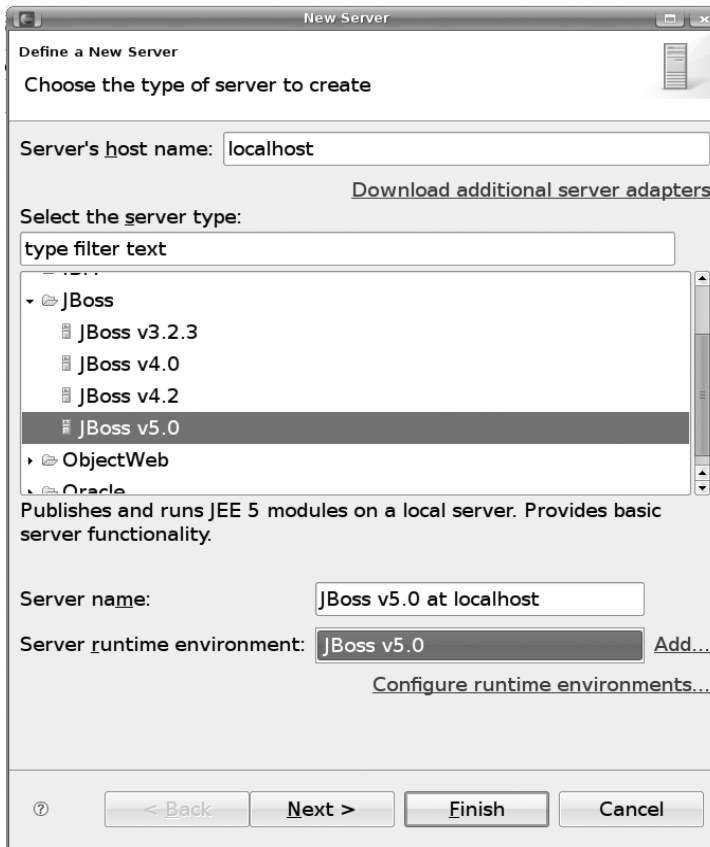


Figure 1-8. *Choosing the JBoss runtime environment*

Click Next until you see the screen in Figure 1-9, where you can add web applications to the JBoss instance.

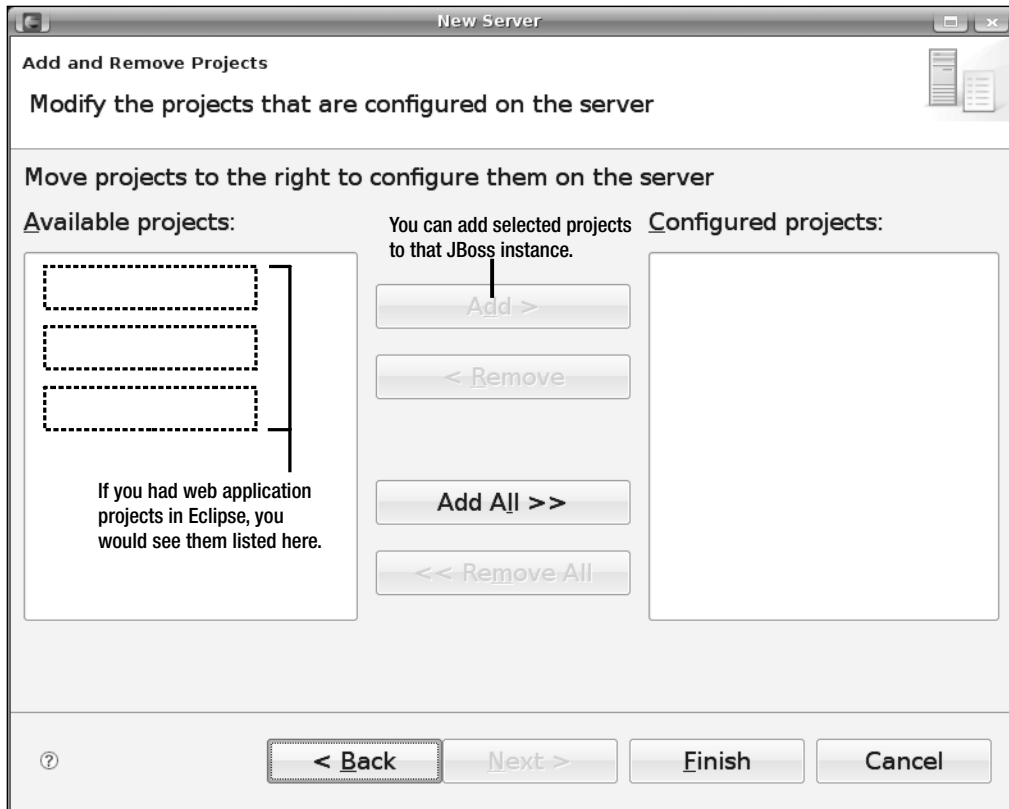


Figure 1-9. *Adding web applications*

For the moment, you'll have none. Click Finish. Then you should see your JBoss instance on the Servers tab (Figure 1-10).

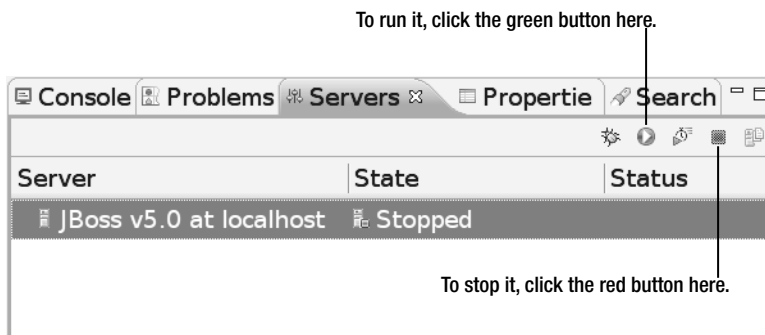


Figure 1-10. *JBoss instance*

Click the green icon as shown in Figure 1-10 to run JBoss. Then you will see some messages on the Console tab, as shown here:

```
...
14:47:06,820 INFO  [TomcatDeployment] deploy, ctxPath=/
14:47:06,902 INFO  [TomcatDeployment] deploy, ctxPath=/jmx-console
14:47:06,965 INFO  [Http11Protocol] Starting Coyote HTTP/1.1 on http-127.0.0.1-8080
14:47:06,992 INFO  [AjpProtocol] Starting Coyote AJP/1.3 on ajp-127.0.0.1-8009
14:47:07,001 INFO  [ServerImpl] JBoss (Microcontainer) [5.0.1.GA (build:
SVNTag=JBoss_5_0_1_GA date=200902231221)] Started in 26s:587ms
```

Note If your computer is not that fast, JBoss will take so long to start that Eclipse may think it has stopped responding. In that case, double-click the JBoss instance, click Timeouts, set the timeout for starting to a longer value such as 100 seconds, and then start JBoss again.

To stop JBoss, click the red icon (as shown earlier in Figure 1-10).

Installing a JSF Implementation

JSF stands for JavaServer Faces and is an API (basically, it's some Java interfaces). To use JSF, you need an implementation (which means you need Java classes that implement those interfaces). There are two main implementations: the reference implementation from Sun and MyFaces from Apache. In this book, you'll use the former, but you could use MyFaces with no practical difference.

So, go to <https://jaserverfaces.dev.java.net> to download a binary package of the JSF 2.0 implementation, which is called Mojarra. The file is probably called something like `mojarra-2.0.0-PR2-binary.zip`; unzip it into a folder, say `c:\jsf`.

Installing Web Beans

To install Web Beans, go to <http://www.seamframework.org/WebBeans> to download it. Make sure it is strictly newer than 1.0.0 ALPHA2; otherwise, get the nightly snapshot. The file is probably called something like `webbeans-ri-distribution-1.0.0-SNAPSHOT.zip`; unzip it into a folder such as `c:\webbeans`.

Next, you'll need to install Web Beans into JBoss. To do that, you'll need to run Ant 1.7.0 or newer. If you don't have this tool, you can download it from <http://ant.apache.org> and unzip it into a folder such as `c:\ant`.

Next, modify the `c:\webbeans\jboss-as\build.properties` file to tell it where JBoss is, as shown in Listing 1-1. Make sure that there is no leading # character on that line!

Listing 1-1. Tell Web Beans Where JBoss Is

```
jboss.home=c:\jboss
java.opts=...
webbeans-ri-int.version=5.2.0-SNAPSHOT
webbeans-ri.version=1.0.0-SNAPSHOT
jboss-ejb3.version=1.0.0
```

Open a command prompt, make sure you're connected to the Internet, and then issue the commands shown in Listing 1-2.

Listing 1-2. Issue These Commands at the Command Prompt

```
c:\>cd \webbeans\jboss-as
c:\>set ANT_HOME=c:\ant
c:\>ant update
```

This will output a lot of messages. If everything is fine, you should see a “BUILD SUCCESSFUL” message at the end, as shown here:

```
...
[copy] Copying 2 files to /home/kent/jboss-
5.0.1.GA/server/default/deployers/webbeans.deployer/lib-int
[copy] Copying 8 files to /home/kent/jboss-
5.0.1.GA/server/default/deployers/webbeans.deployer

update:

BUILD SUCCESSFUL
```

Creating the “Hello world!” Application with JSF

To create the “Hello world!” application, right-click in Package Explorer, and choose New ► Dynamic Web Project (Figure 1-11).

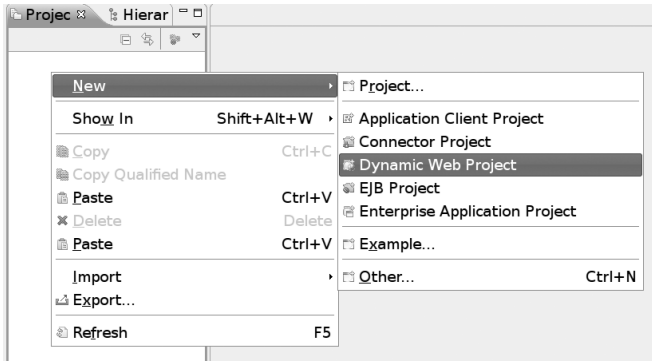


Figure 1-11. *Creating a dynamic web project*

Enter the information shown in Figure 1-12.

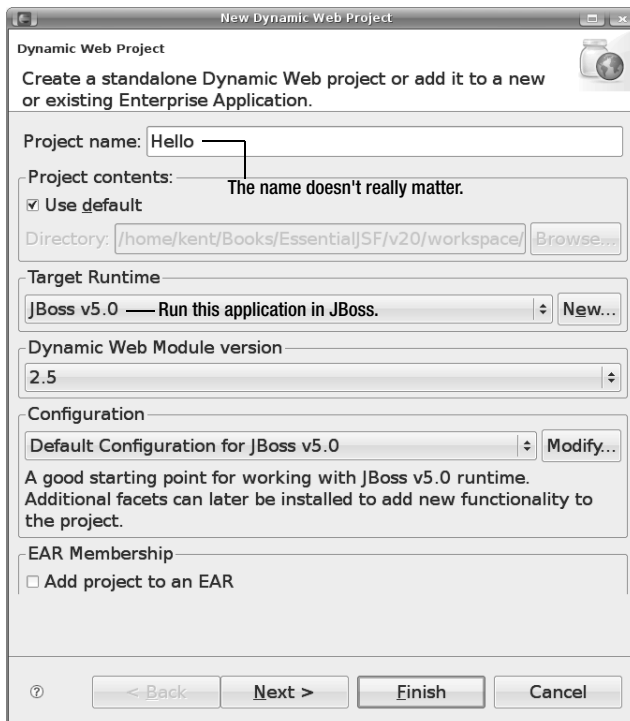


Figure 1-12. *Entering the project information*

Keep clicking Next until you finish. Finally, you should end up with the project structure shown in Figure 1-13.

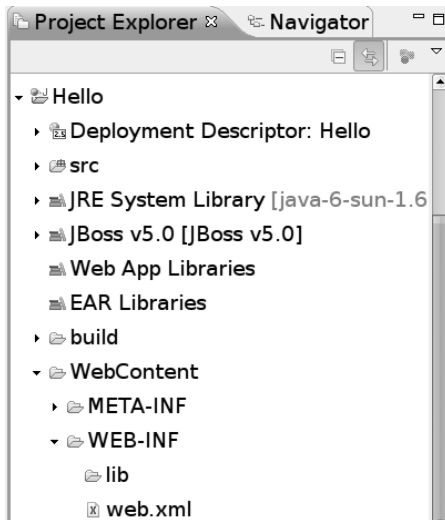


Figure 1-13. *Project structure*

To make JAR files from the JSF implementation available to your project, copy the JAR files into JBoss, as shown in Figure 1-14.

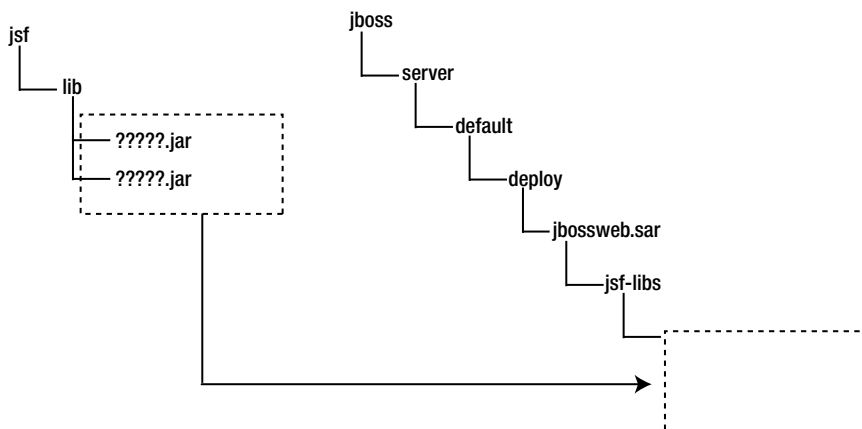


Figure 1-14. *Copying the JAR files into the JBoss*

To see the Web Beans classes available to you at compile time, right-click the project, choose Build Path ► Configure Build Path, and add `c:\jboss\server\default\deployers\webbeans.deployer\jsr299-api` to the build path.

Next, you'll create the "Hello world!" page. To do that, right-click the WebContent folder, and choose New ► HTML. Enter **hello** as the file name, as in Figure 1-15.

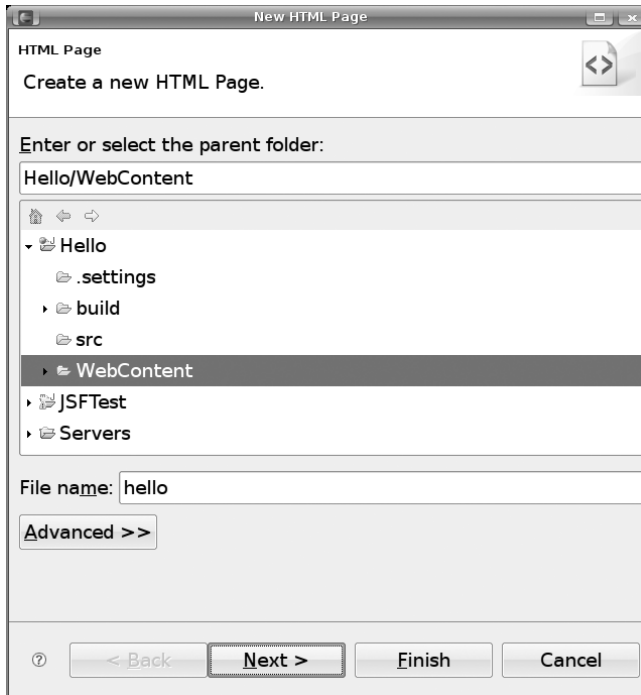


Figure 1-15. Creating the “Hello world!” page

Click Next, and choose the template named New XHTML File (1.0 Strict), as in Figure 1-16.

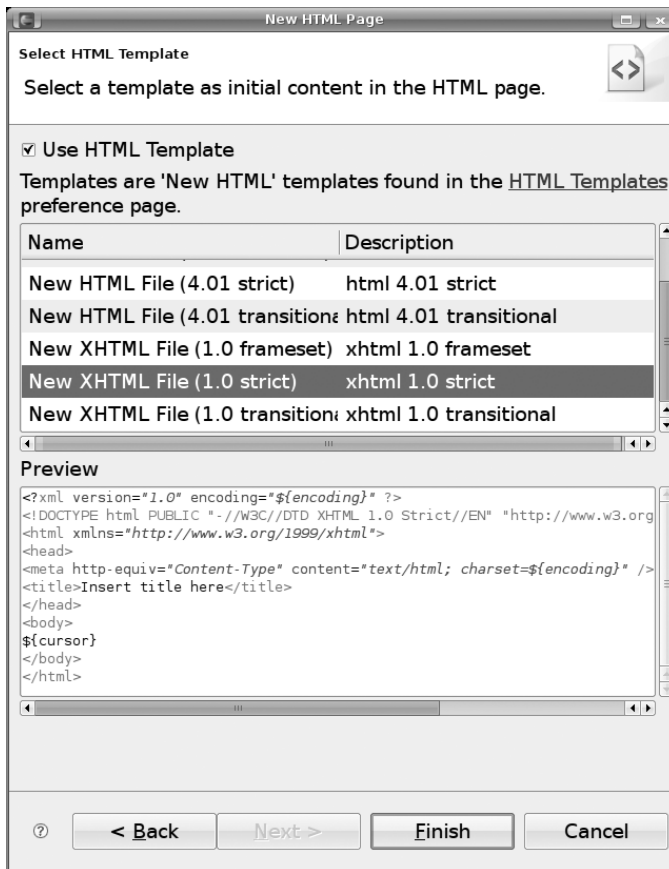


Figure 1-16. *Using the XHTML strict template*

Click Finish. This will give you a file named `hello.html`. This XHTML file will serve as the “Hello world!” page. However, JSF by default assumes that XHTML files use the `.xhtml` extension, so rename the file as `hello.xhtml` (see Figure 1-17).



Figure 1-17. *Renaming the file*

Open the file, and input the content shown in Listing 1-3.

Listing 1-3. *Content of `hello.xhtml`*

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://java.sun.com/jsf/html">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
<title>Insert title here</title>
</head>
<body>
Hello world!
</body>
</html>
```

Next, modify the `web.xml` file in the `WebContent/WEB-INF` folder as shown in Figure 1-18.

```

<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd" id="WebApp_ID"
version="2.5">
  <display-name>Hello</display-name>
  <welcome-file-list>
    <welcome-file>index.html</welcome-file>
    <welcome-file>index.htm</welcome-file>
    <welcome-file>index.jsp</welcome-file>
    <welcome-file>default.html</welcome-file>
    <welcome-file>default.htm</welcome-file>
    <welcome-file>default.jsp</welcome-file>
  </welcome-file-list>
  <servlet>
    <servlet-name>JSF</servlet-name>
    <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>JSF</servlet-name>
    <url-pattern>/faces/*</url-pattern>
  </servlet-mapping>
</web-app>

```

You can give it any name you'd like.

This "servlet" is the JSF engine.

You will access the application using a URL like this. This way, JBoss will send the request to the JSF engine for handling.

This "servlet" is the JSF engine.

You can give it any name you'd like.

`http://localhost:8080/Hello/faces/???`

The Project Name

Hello

WebContent

...

Figure 1-18. *web.xml*

Next, create a file called `faces-config.xml` in the `WebContent/WEB-INF` folder. This is the configuration file for JSF, as shown in Listing 1-4. Without it, JSF will not initialize. Because you have no particular configuration to set, it contains only an empty `<faces-config>` element.

Listing 1-4. *faces-config.xml*

```
<faces-config xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-facesconfig_2_0.xsd"
  version="2.0">

</faces-config>
```

To register your application with JBoss, right-click the JBoss instance on the Servers tab, and choose Add and Remove Projects; then you'll see Figure 1-19.

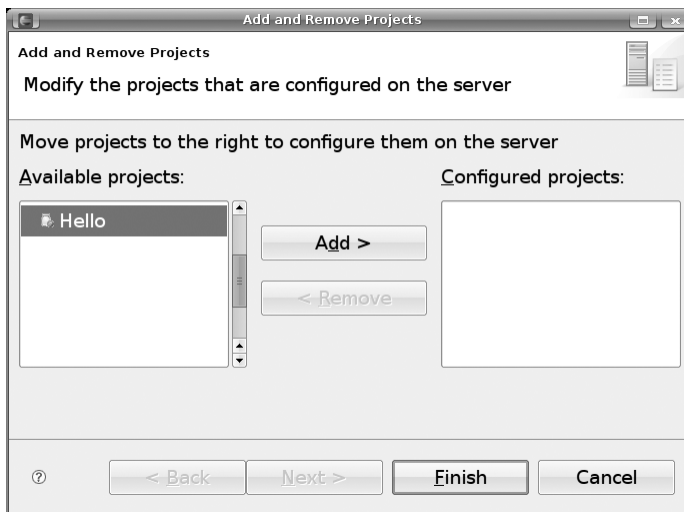


Figure 1-19. *Adding projects to the JBoss instance*

Choose your Hello project to add to the JBoss instance.

Now, start JBoss, and try to access `http://localhost:8080/Hello/hello.xhtml` in a browser. Note that this URL does *not* include the `/faces` prefix and thus will *not* be handled by the JSF engine. Instead, JBoss will directly read the `hello.xhtml` page and return its content (see Figure 1-20). We're doing this just to check whether the basic web application is working.

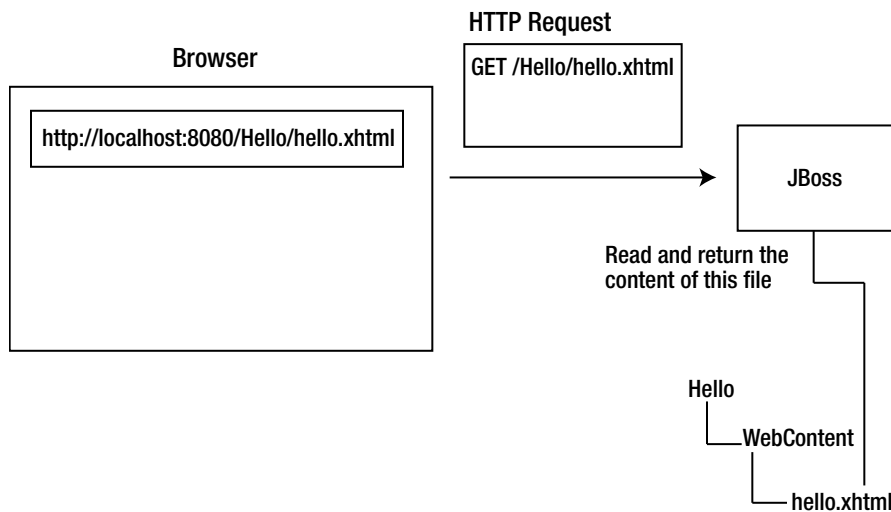


Figure 1-20. *Directly accessing the content of `hello.xhtml`*

If everything is working, the browser should either prompt you to save the file (Firefox) or display the “Hello world!” page (Internet Explorer).

To access it through the JSF engine, use `http://localhost:8080/Hello/faces/hello.xhtml` instead, as shown in Figure 1-21. Simply put, JSF will take path `/hello.xhtml` (the view ID) from the URL and use it to load the XHTML file.

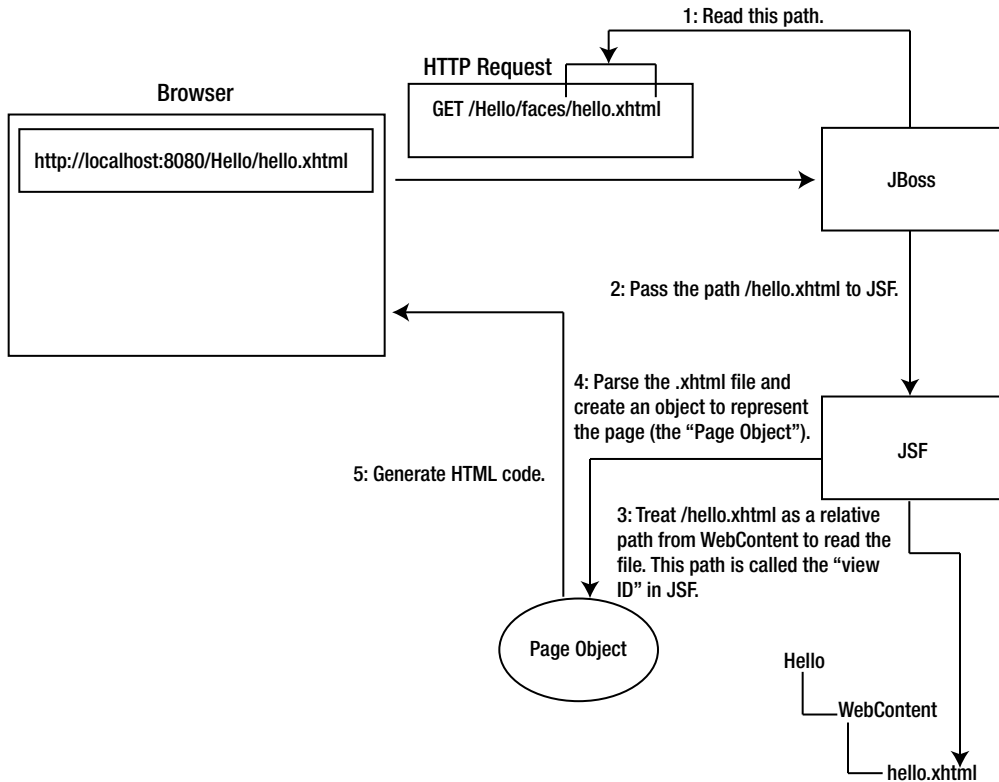


Figure 1-21. Accessing the `hello.xhtml` file through JSF

You'll see "Hello world!" displayed in the browser.

Generating Dynamic Content

Displaying static text is not particularly interesting. Next, you'll learn how to output some dynamic text. Modify `hello.xhtml` as shown in Figure 1-22. The page object created is also shown in the figure.

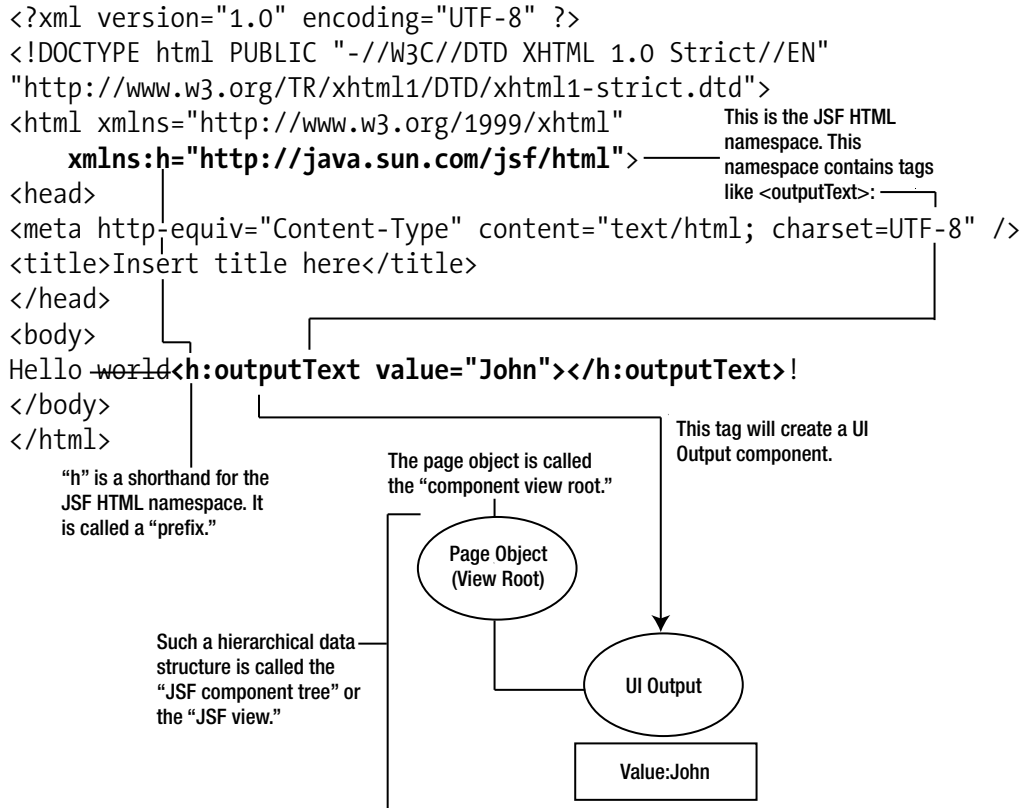


Figure 1-22. JSF component tree

The component tree generates HTML code, as shown in Figure 1-23. In JSF, the process is called *encoding*.

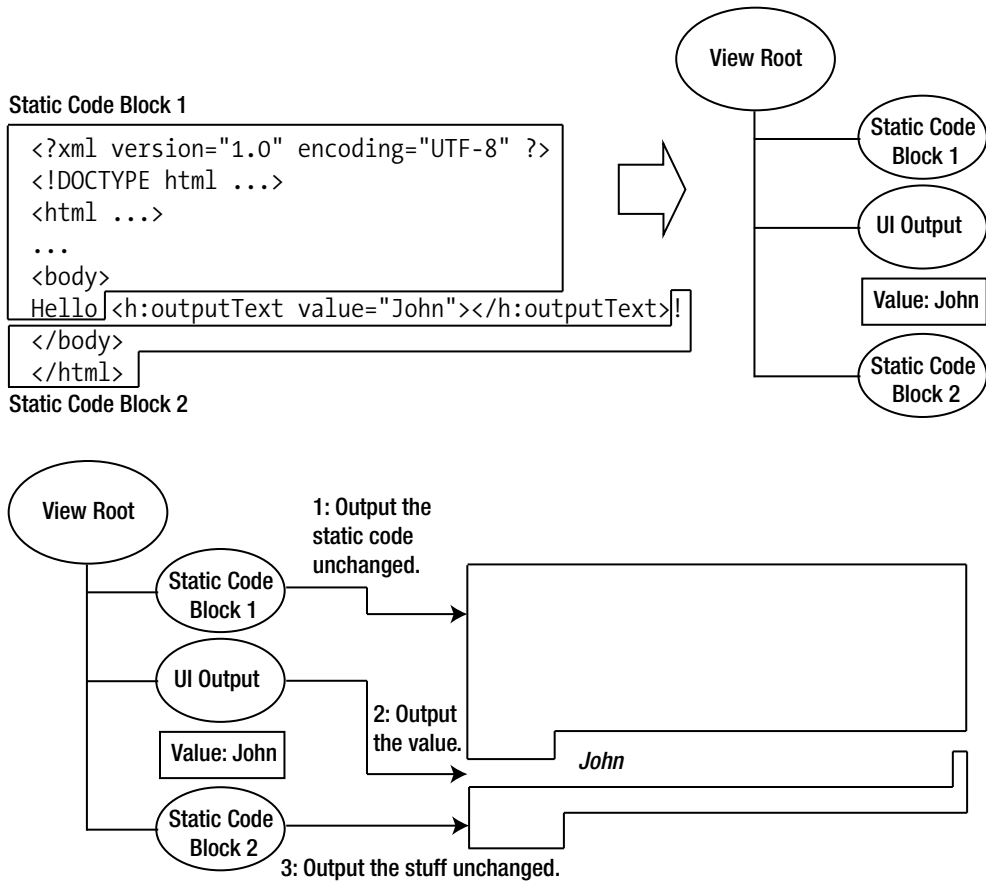


Figure 1-23. JSF component tree generating HTML code

Now access the page again in the browser. Do you need to start JBoss again? No. By default Eclipse will update the web application in JBoss every 15 seconds after you make changes to the source files. If you can't wait, you can right-click the JBoss instance and choose Publish to force it to do it immediately. Anyway, the HTML page should look like Figure 1-24.



Figure 1-24. *Generated HTML code*

Note that there is no space between “Hello” and “John.” This is because JSF ignores the spaces surrounding JSF tags. You can easily fix this problem, but let’s ignore it for now; we’ll fix it later in the chapter.

Retrieving Data from Java Code

Next, you’ll let the UI Output component retrieve the string from Java code. First, create the Java class `GreetingService` in the `hello` package. Input the content shown in Listing 1-5.

Listing 1-5. *GreetingService.java*

```
package hello;

public class GreetingService {
    public String getSubject() {
        return "Paul";
    }
}
```

So, how do you get the UI Output component to call the `getSubject()` method in the class? Figure 1-25 shows how it works. Basically, in each HTTP request, there is a table of objects, and each object has a name. (Each object is called a *web bean*.) If you set the value attribute of the UI Output component to something like `#{foo}`, which is called an *EL expression*, at runtime it will ask the JSF engine for an object named `foo`. The JSF engine will in turn ask the Web Beans manager for an object named `foo`.

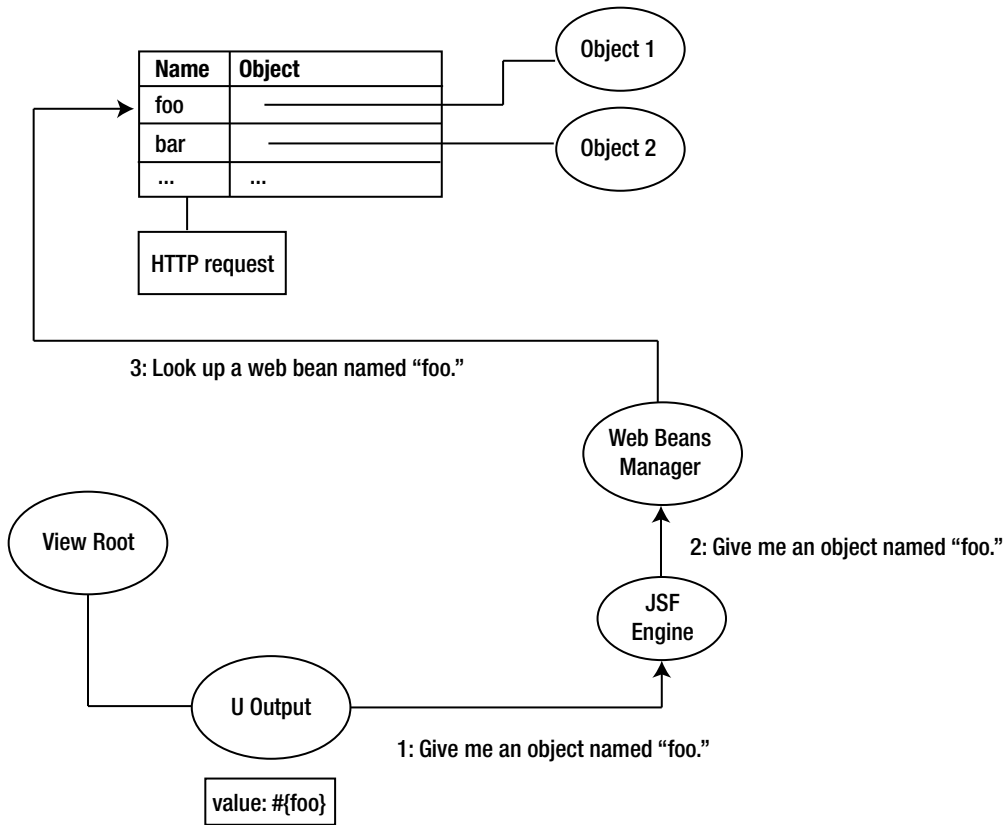


Figure 1-25. Accessing a web bean

For your current case, what if **Object1** were a `GreetingService` object (let's ignore how to create one of those for the moment)? Then the **UI Output** component can already reach the `GreetingService` object. How can the output call the `getSubject()` method on it? To do that, modify the value attribute of the `outputText` tag as shown in Listing 1-6.

Listing 1-6. *Accessing the subject Property of a GreetingService Object*

```
<html ...>
...
<body>
Hello <h:outputText value="#{foo.subject}"></h:outputText>!
</body>
</html>
```

Now, let's return to the question of how to put a `GreetingService` object into the web bean table. To do that, modify the `GreetingService` class as shown in Figure 1-26.

```
package hello;

import javax.annotation.Named;
import javax.context.RequestScoped;

@Named("foo")
@RequestScoped
public class GreetingService {
    public String getSubject() {
        return "Paul";
    }
}
```

The web beans-related annotations are defined in those packages.

The name of the web bean.

Put the web bean into the table in the request.

Figure 1-26. *Declaring a web bean class*

How does it work? When the Web Beans manager looks for a web bean named `foo` in the request (see Figure 1-27), there is none because initially the table is empty. Then it will check each class on the `CLASSPATH` to find a class annotated with `@Named` and with a matching name. Here, it will find the `GreetingService` class. Then it will create an instance of the `GreetingService` class, create a new row using the name `foo`, and add it to the web bean table.

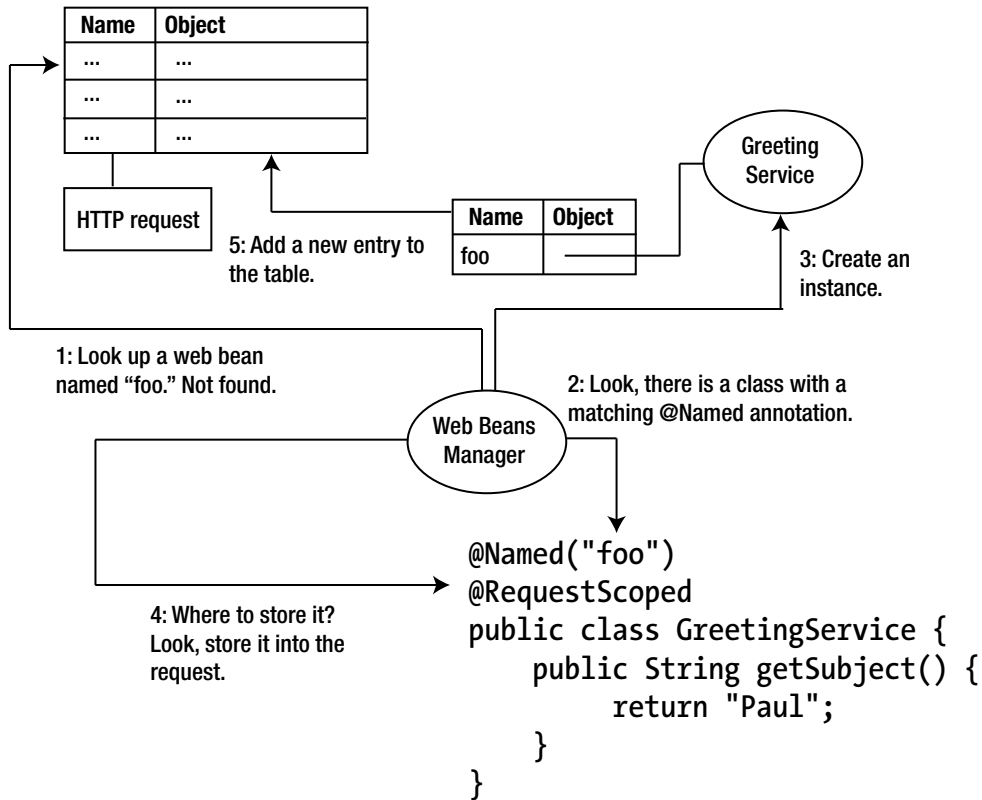


Figure 1-27. How the Web Beans manager creates the web bean

Note that in order for the Web Beans manager to create an instance of the class, it needs to have a no-argument constructor. For the JSF engine to get its subject property, it needs to have a corresponding getter, in other words, `getSubject()`. In summary, the class needs to be a Java bean.

When you need to use Web Beans, you must enable the Web Beans manager by creating a configuration file for it. So, create an empty file named `beans.xml` in the `WebContent/WEB-INF` folder.

Because you have no configuration for it, leave it empty.

Now run the application, and it will work as shown in Figure 1-28.

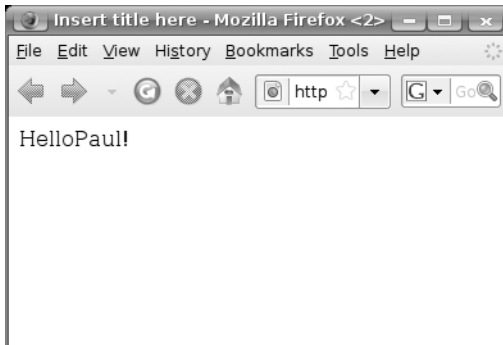


Figure 1-28. Successfully getting the value from a web bean

Now let's fix that space issue we talked about earlier; just add a space to the value attribute of the `outputText` tag, as shown in Figure 1-29.

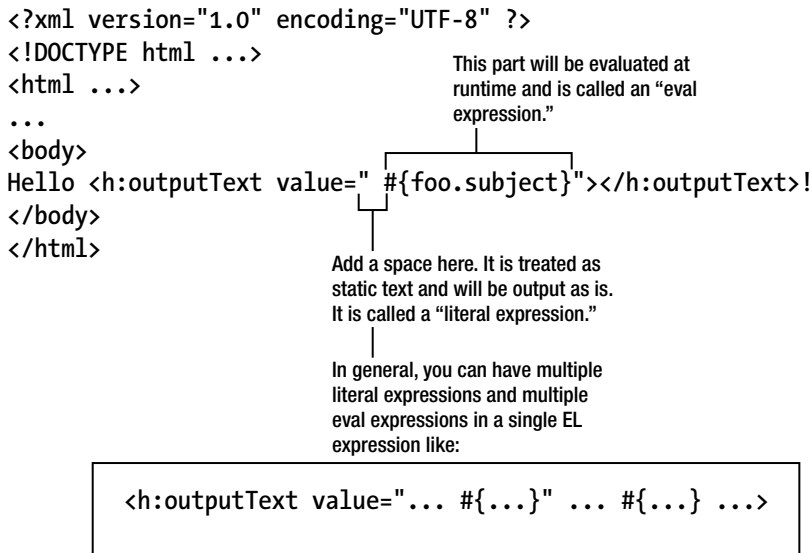


Figure 1-29. Adding a space to the value attribute

Run the application again, and it will work.

Exploring the Life Cycle of the Web Bean

Will the web bean stay there forever? No; the web bean table is stored in the HTTP request, so as HTML code is returned to the client (the browser), the HTTP request will be destroyed and so will the web bean table and the web beans in it.

Note If you have worked with servlets and JSP before, you may wonder whether it's possible to store web beans in the session instead of the request. The answer is yes; you'll see this in action in the subsequent chapters.

Using an Easier Way to Output Text

You've seen how to use the `<h:outputText>` tag to output some text. In fact, there is an easier way to do that. For example, you could modify `hello.xhtml` as shown in Listing 1-7.

Listing 1-7. *Using an EL Expression Directly in the Body Text*

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html ...>
<html ...>
...
<body>
Hello<h:outputText value="#{foo.subject}"></h:outputText>
Hello #{foo.subject}!
</body>
</html>
```

Run the application, and it will continue to work.

Debugging a JSF Application

To debug your application in Eclipse, you can set a breakpoint in your Java code, as shown in Figure 1-30, by double-clicking where the breakpoint (the little filled circle) should appear.



Figure 1-30. *Setting a breakpoint*

Then click the Debug icon in the Server window (Figure 1-31). Now go to the browser to load the page again. Eclipse will stop at the breakpoint (Figure 1-32). Then you can step through the program and check the variables and whatever else. To stop the debug session, just stop or restart JBoss in normal mode.

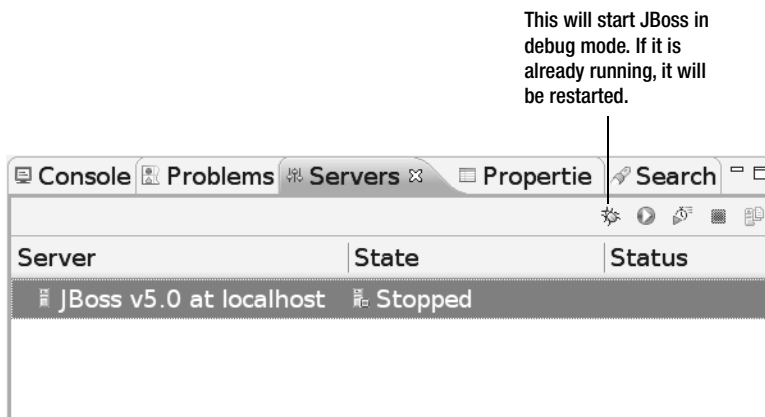


Figure 1-31. *Launching JBoss in debug mode*

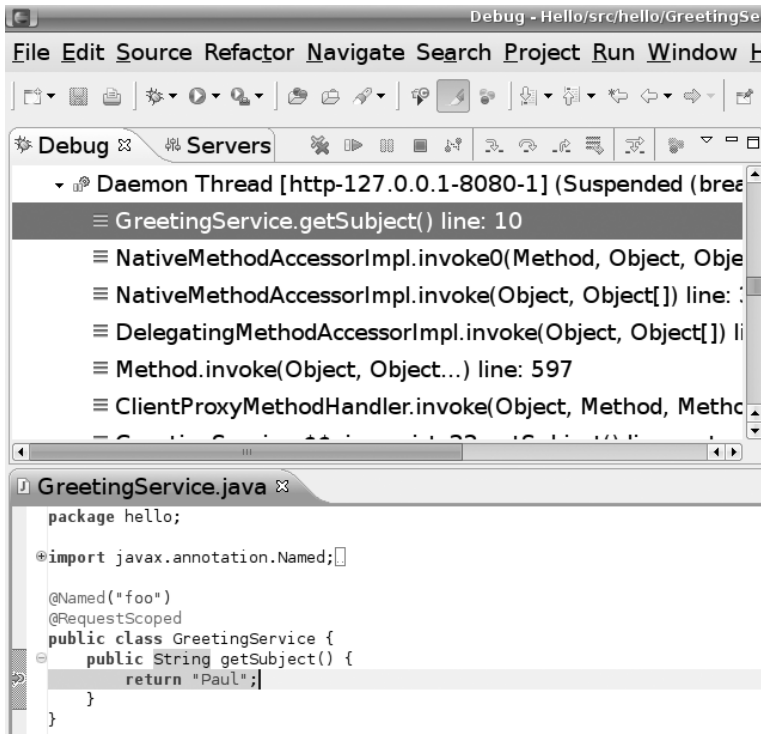


Figure 1-32. Stopping at a breakpoint

Summary

In this chapter, you learned that you can run one or more web applications on top of JBoss. If a web application uses the JSF library, it is a JSF application. In a JSF application, a page is defined by an `.xhtml` file and is identified by its view ID, which is the relative path to it from the `WebContent` folder.

You also learned that an `.xhtml` file consists of tags. Each tag belongs to a certain namespace, which is identified by a URL. To use a tag in an `.xhtml` file, you need to introduce a shorthand (prefix) for the URL and then use the prefix to qualify the tag name. The JSF tags belong to the JSF HTML namespace.

To create a JSF component in the component tree, you use a JSF tag such as `<h:outputText>` in the `.xhtml` file. The root of the component tree is the view root. The component tree will generate HTML code to return to the client. The process of generating markup in JSF is called *encoding*.

To output some text, you can use the `<h:outputText>` tag, which will create a UI Output component. That component will output the value of its `value` attribute. That value can be a static string or an EL expression.

As an alternative to the `<h:outputText>` tag, you can directly put the EL expression into the body text.

In addition, this chapter also covered EL expressions, which typically look like `#{foo.p1}`. If you use an EL expression, the JSF engine will try to find an object named `foo`. It will in turn ask the Web Beans manager to do it, and the Web Beans manager will look up the web beans in the web bean table in the HTTP request or create it appropriately. Then the JSF engine will call `getP1()` on the web bean, and the result is the final value of the EL expression.

Finally, you learned that web beans are JavaBeans created and destroyed automatically by the Web Beans manager. To enable web beans, you need to have a `META-INF/web-beans.xml` file on your CLASSPATH. To define a Java class as a web bean class, the class needs to be a JavaBean; in other words, it has a no-argument constructor and provides getters and/or setters for certain properties. Then it must be annotated with the `@Named` annotation to be given a name.



Using Forms

In this chapter, you'll learn how to use forms to get input from users and store it in a web bean for processing.

Developing a Stock Quote Application

Suppose that you'd like to develop the application shown in Figure 2-1. That is, if the user enters a stock symbol and clicks the button, then the application will display the stock value.

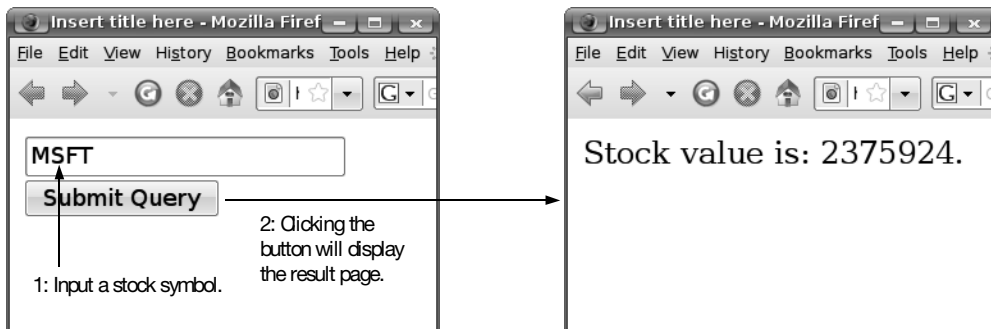


Figure 2-1. *A stock quote application*

Getting the Stock Quote Symbol

Let's create the example application now. In Eclipse, copy the Hello project, and paste it as a new project called Stock. Then choose Window ► Show View ► Navigator, and locate the `org.eclipse.wst.common.component` file shown in Figure 2-2.



Figure 2-2. *Locate this configuration file.*

Open the file, and modify it as shown in Listing 2-1. Eclipse “forgot” to update the project name there, so you need to do it yourself.

Listing 2-1. *Update the Content with the New Project Name*

```
<?xml version="1.0" encoding="UTF-8"?>
<project-modules id="moduleCoreId" project-version="1.5.0">
  <wb-module deploy-name="Stock">
    <wb-resource deploy-path="/" source-path="/WebContent"/>
    <wb-resource deploy-path="/WEB-INF/classes" source-path="/src"/>
    <property name="context-root" value="Stock"/>
    <property name="java-output-path"/>
  </wb-module>
</project-modules>
```

Save this file, and close the Navigator view. Then rename the `hello.xhtml` file as `getsymbol.xhtml`. Modify the new `getsymbol.xhtml` file as shown in Figure 2-3.

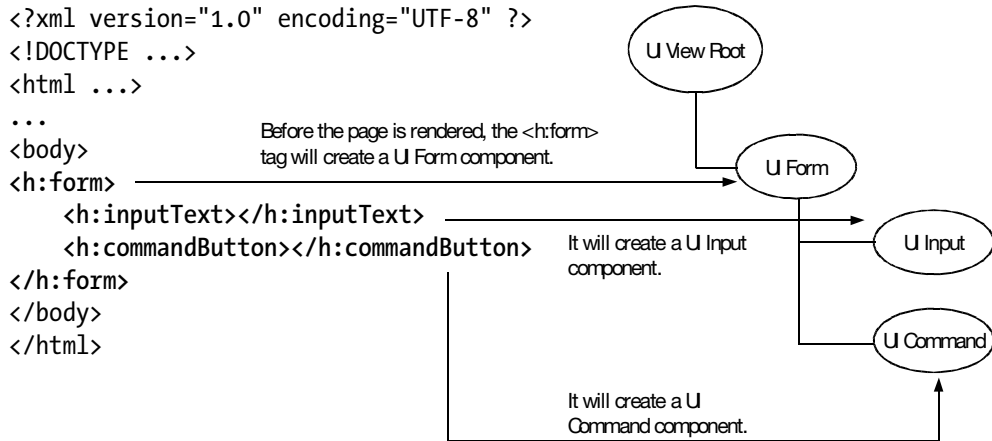


Figure 2-3. *getsymbol.xhtml*

What do the components such as UI Form do? During rendering, these components will generate HTML elements, as shown in Figure 2-4.

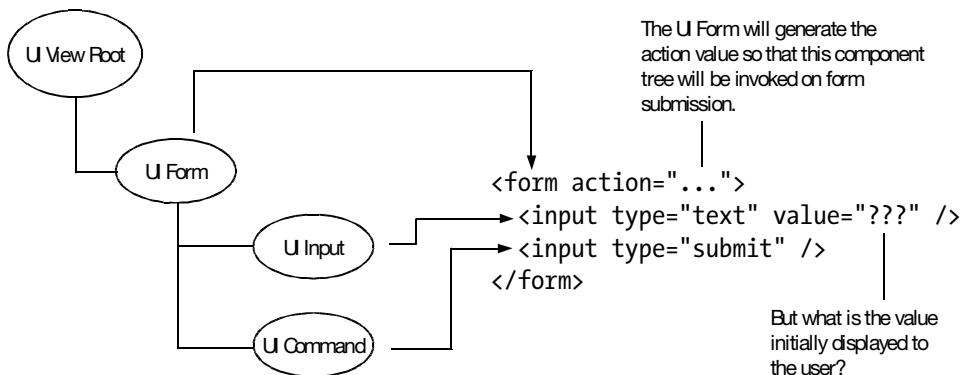


Figure 2-4. *Rendering of form-related components*

What is the initial symbol value displayed to the user? You can link a web bean to UI Input component (see Figure 2-5). That is, on rendering, the UI Input component will evaluate the EL expression and use the result as the initial value.

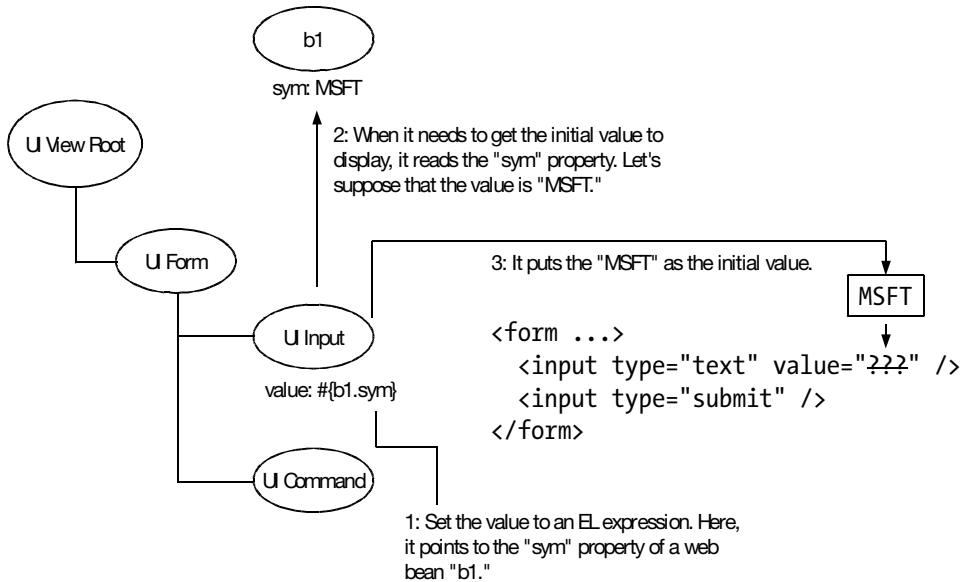


Figure 2-5. Linking a property of a web bean to a UI Input component

Note that after rendering the page, the HTTP request is gone and so is the `b1` bean (see Figure 2-6).

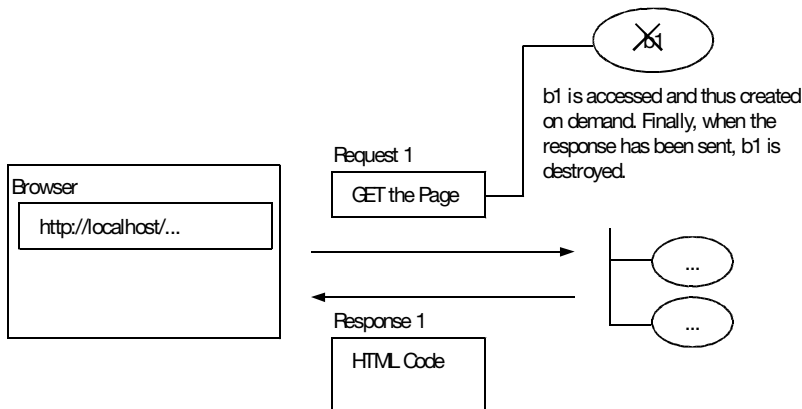


Figure 2-6. The `b1` bean will be gone after rendering.

Suppose that the user changes the value from “MSFT” to “IBM” and then submits the form. What will happen? Figure 2-7 shows the process. Note that this b1 bean is not the original; it is newly created and associated with the new request representing the form submission.

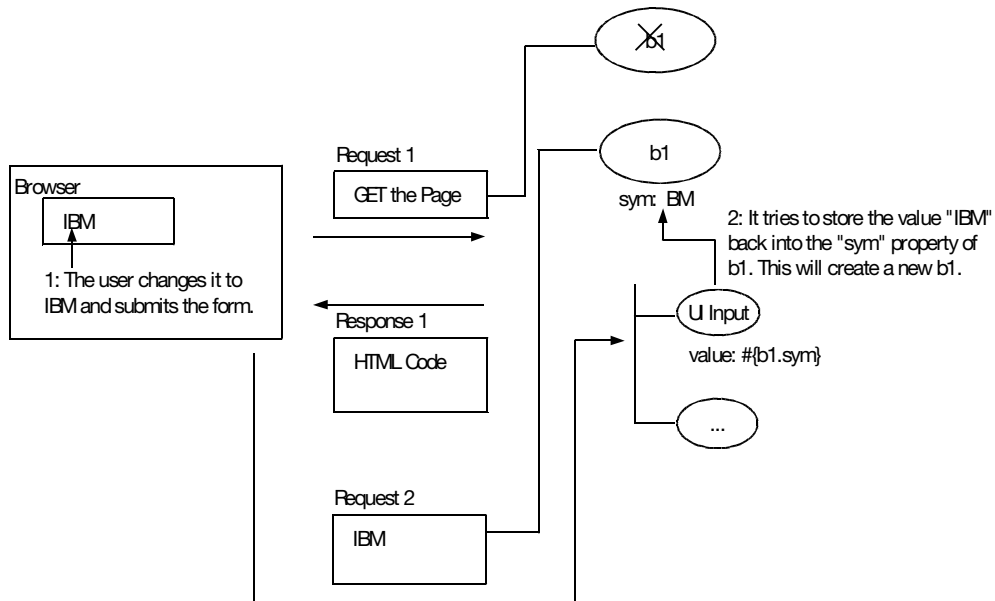


Figure 2-7. *The form submission process*

But what is the HTTP response? By default, the same page will render again. Therefore, it will display “IBM” as the value again because the b1 bean just created is still there (see Figure 2-8).

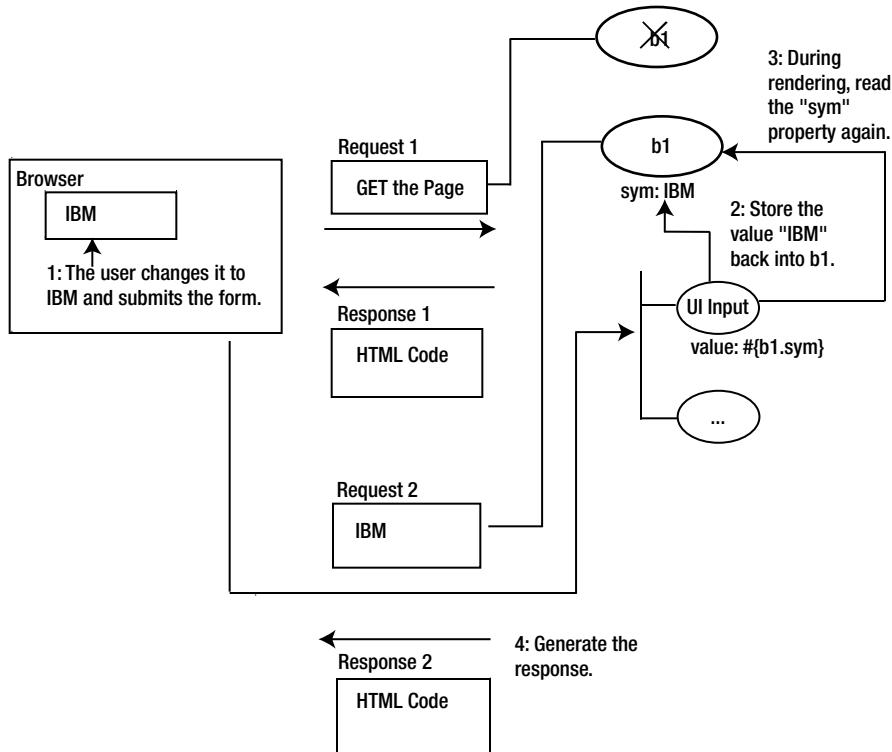


Figure 2-8. The rendering process after form submission

Now, to implement these ideas, modify `getsymbol.xhtml` as shown in Listing 2-2.

Listing 2-2. `getsymbol.xhtml`

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE ...>
<html ...>
...
<body>
<h:form>
    <h:inputText value="#{b1.sym}"></h:inputText>
    <h:commandButton></h:commandButton>
</h:form>
</body>
</html>
```

Of course, you need to define the `b1` web bean. To do that, create a class called `QuoteRequest` in a `stock` package (see Listing 2-3). Note that the `sym` property is initialized to `MSFT`, and some messages are printed in various methods to show the order of the events. You can also delete the `hello` package.

Listing 2-3. *Defining the `b1` Web Bean*

```
package stock;
...
@Named("b1")
@RequestScoped
public class QuoteRequest {
    private String sym = "MSFT";

    public QuoteRequest() {
        System.out.println("Creating b1");
    }
    public String getSym() {
        System.out.println("getting sym");
        return sym;
    }
    public void setSym(String sym) {
        System.out.println("setting sym to: " + sym);
        this.sym = sym;
    }
}
```

Now, start JBoss, and access the page at <http://localhost:8080/Stock/faces/getsymbol.xhtml>. You should see the messages in the console that are shown in Listing 2-4.

Listing 2-4. *Messages Showing the Rendering Process*

```
Creating b1
getting sym
```

Change the symbol to “IBM,” and then submit the form. You should see the messages shown in Listing 2-5. From these messages you can see that a new `b1` bean is created. Then for some reason the `sym` property is read (it is because the UI Input component is checking whether the new value is really different from the old one and, if so, notifying some interested parties). Next, the UI Input component stores `IBM` into it, and finally it is read again to generate the HTML code.

Listing 2-5. *Messages Showing the Form Submission Process*

```
Creating b1
getting sym
Creating b1
getting sym
setting sym to: IBM
getting sym
```

Displaying the Result Page

For the moment, when handling the form submission, you're simply letting JSF redisplay the current page containing the form. This is no good. You'd like to display a result page showing the stock price instead. To do that, create a `stockvalue.xhtml` file in the `WebContent` folder. For the moment, the content is hard-coded (see Listing 2-6).

Listing 2-6. *Result Page*

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://java.sun.com/jsf/html">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
<title>Insert title here</title>
</head>
<body>
Stock value is: 123.
</body>
</html>
```

The question is, how do you tell JSF to display the result page? This is done using a navigation rule (see Figure 2-9).

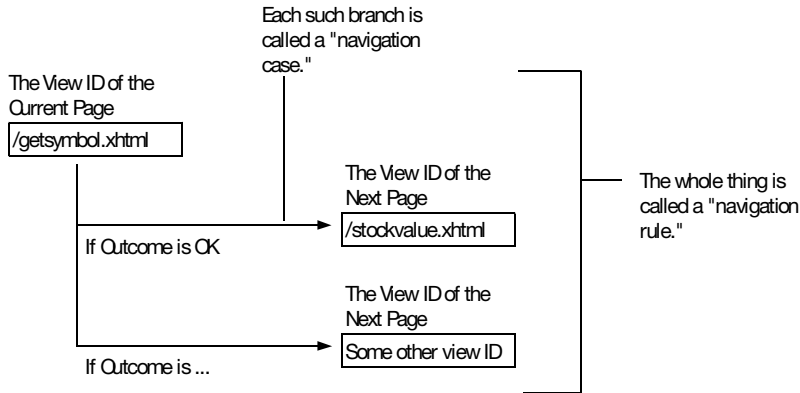


Figure 2-9. *Navigation rule*

To create the navigation rule, modify `faces-config.xml` as shown in Listing 2-7.

Listing 2-7. *Navigation Rule in `faces-config.xml`*

```
<faces-config ...>
  <navigation-rule>
    <from-view-id>/getsymbol.xhtml</from-view-id>
    <navigation-case>
      <from-outcome>ok</from-outcome>
      <to-view-id>/stockvalue.xhtml</to-view-id>
    </navigation-case>
    ...
    ...You could have more <navigation-case> elements here...
    ...
  </navigation-rule>
</faces-config>
```

Now that you've defined the navigation rule and cases (only one, actually), the next step is to set the outcome to ok. To do that, modify `getsymbol.xhtml` as shown in Listing 2-8.

Listing 2-8. *Setting the action Attribute of UI Command Component*

```

<?xml version="1.0" encoding="UTF-8" ?>
...
<h:form>
    <h:inputText value="#{b1.sym}"></h:inputText>
    <h:commandButton action="ok"></h:commandButton>
</h:form>
</body>
</html>

```

Then, the UI Command component reads its action attribute and uses the value to set the outcome, as shown in Figure 2-10.

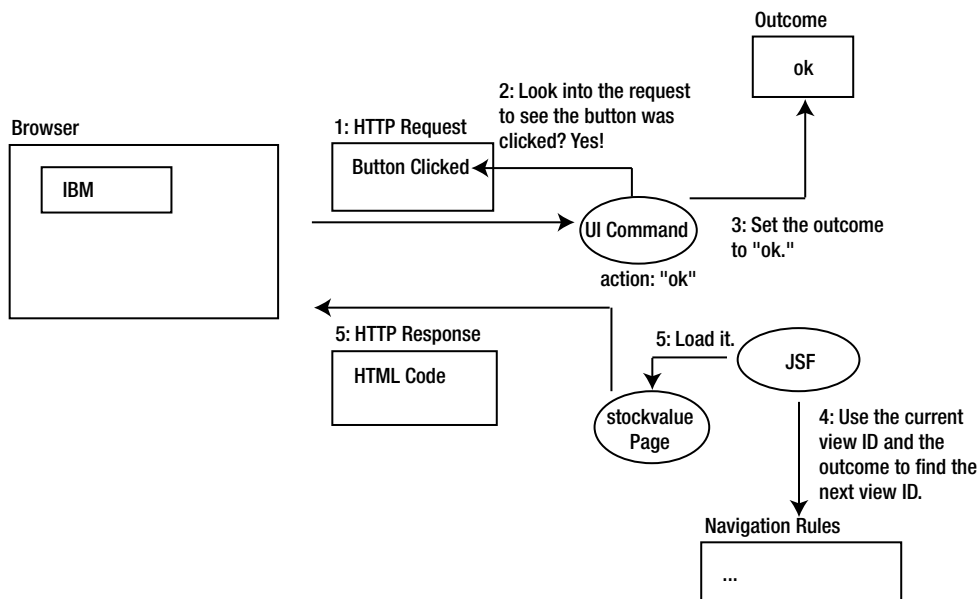


Figure 2-10. *Using the action attribute to control the outcome*

Now, run the application, and it should work.

Displaying the Stock Value

For the moment, you're hard-coding the stock value. Next, you'll calculate a dynamic value. In a real implementation, you would look up the stock price probably from an online service. For simplicity in this example, you'll just use the hash code of the symbol

as the stock value. To do that, modify `stockvalue.xhtml` so that it gets the stock value from the `b1` bean (see Listing 2-9).

Listing 2-9. *Getting the Stock Value from a Web Bean*

```
...
<body>
Stock value is: #{b1.stockValue}.
</body>
...
```

For this to work, define a getter in the `b1` bean, as shown in Listing 2-10.

Listing 2-10. *Providing the Stock Value from the `b1` Bean*

```
...
public class QuoteRequest {
    private String sym = "MSFT";

    public QuoteRequest() {
        System.out.println("Creating b1");
    }
    public String getSym() {
        System.out.println("getting sym");
        return sym;
    }
    public void setSym(String sym) {
        System.out.println("setting sym to: " + sym);
        this.sym = sym;
    }
    public int getStockValue() {
        return Math.abs(sym.hashCode());
    }
}
```

Run the application, and the stock value should change depending on the symbol. How does it do that? See Figure 2-11. Basically, when the HTTP request arrives, the UI Input and UI Command components are each given the opportunity to handle the request such as reading values from it, checking whether a value is provided, validating the value as needed, and finally storing the value into a web bean (see Figure 2-11). Let's call this phase the Input Processing phase. In this phase for the UI Command component, after finding that the button was clicked, it will *not* immediately set the outcome. Instead, it schedules a listener to be executed in the next phase.

Assuming that there is no error of any kind, JSF will enter the next phase in which all scheduled listeners will be executed. In this example, the listener scheduled by the UI Command component will execute to set the outcome. Then JSF will check the outcome and use the navigation rules to determine the next view ID of the next page. This phase is called the Invoke Application phase.

In the next phase, JSF uses the next view ID to load the page and let it render. This phase is called the Render Response phase.

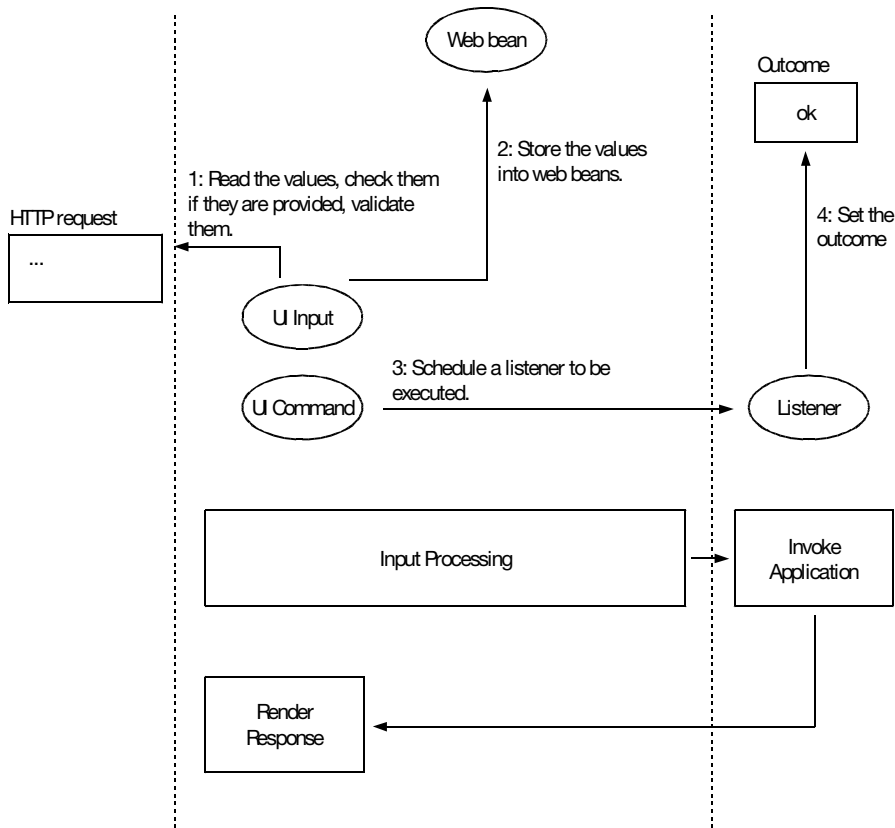


Figure 2-11. JSF handles a request in phases.

Marking Input As Required

What if the user deletes the initial symbol displayed and then submits the form? You'll get an empty string. For this stock quote application, this should be treated as an error; in other words, the user should be forced to enter something. To do that, you just need to mark the UI Input component as required (see Figure 2-12).

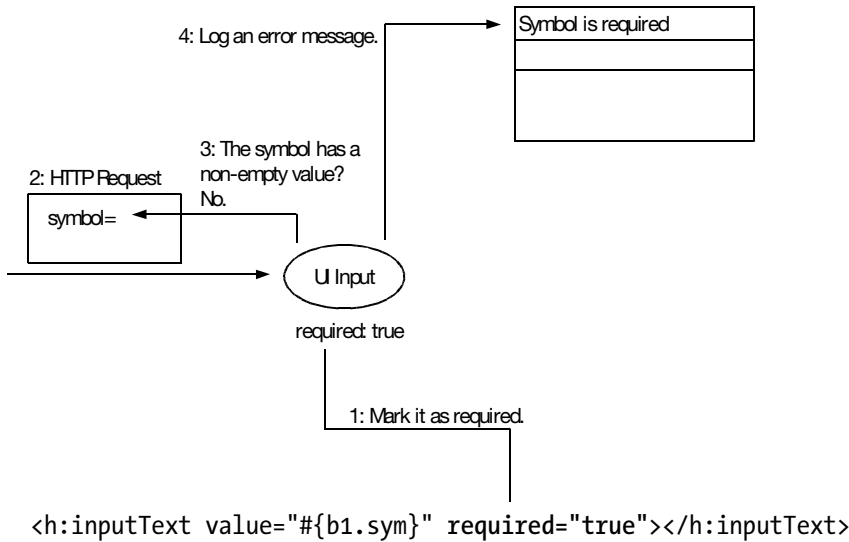


Figure 2-12. *Marking input as required*

However, if the UI Command component still sets the outcome to ok, JSF will go ahead and display the stock result. This is obviously not what you want. You'd like to do the following:

1. Redisplay getsymbol.xhtml.
2. Have it display the error message.

In order to do step 1, you need to first understand how JSF handles the request when there is any error in the Input Processing phase, such as when no value is provided but the UI Input component is marked as required (see Figure 2-13). Obviously, the UI Input component has no value to set into the web bean. Because the UI Command component doesn't know about the error, it will still schedule the listener. JSF will note the error, skip the Invoke Application phase, and go directly to the Render Response phase. Then the outcome will not have been set and will remain at its initial value of `null`. JSF will treat it as a signal to not change the current view ID and thus will redisplay the current page.

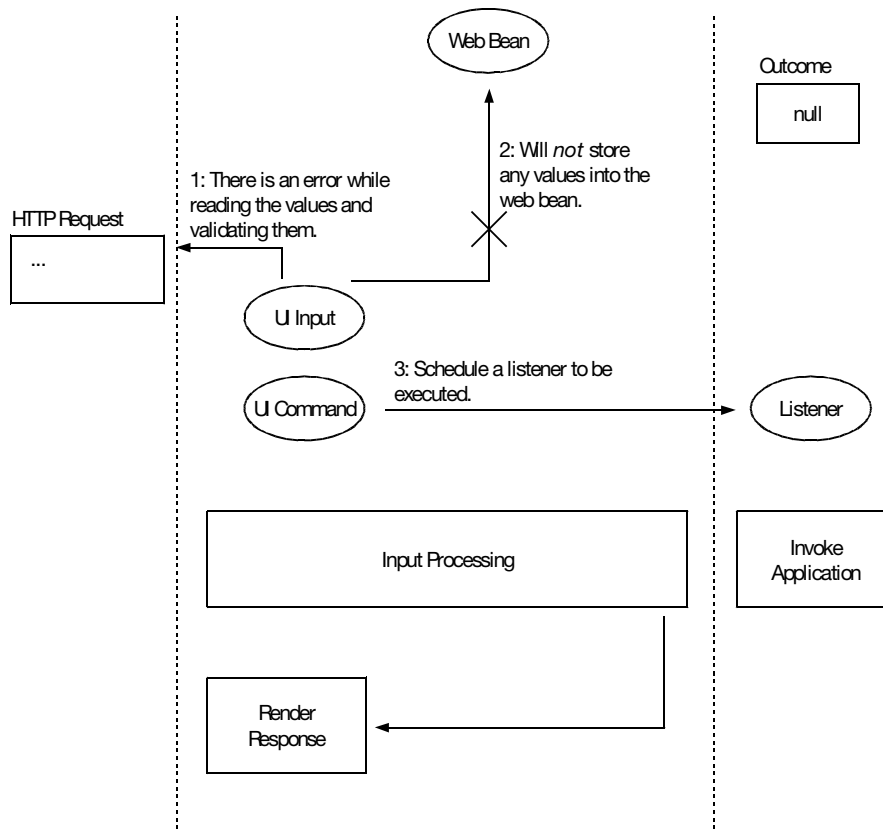


Figure 2-13. *Skipping the Invoke Application phase if there is any error when processing input*

What if you had two UI Input components and one failed? Would the other one store the value into a web bean (see Figure 2-14)?

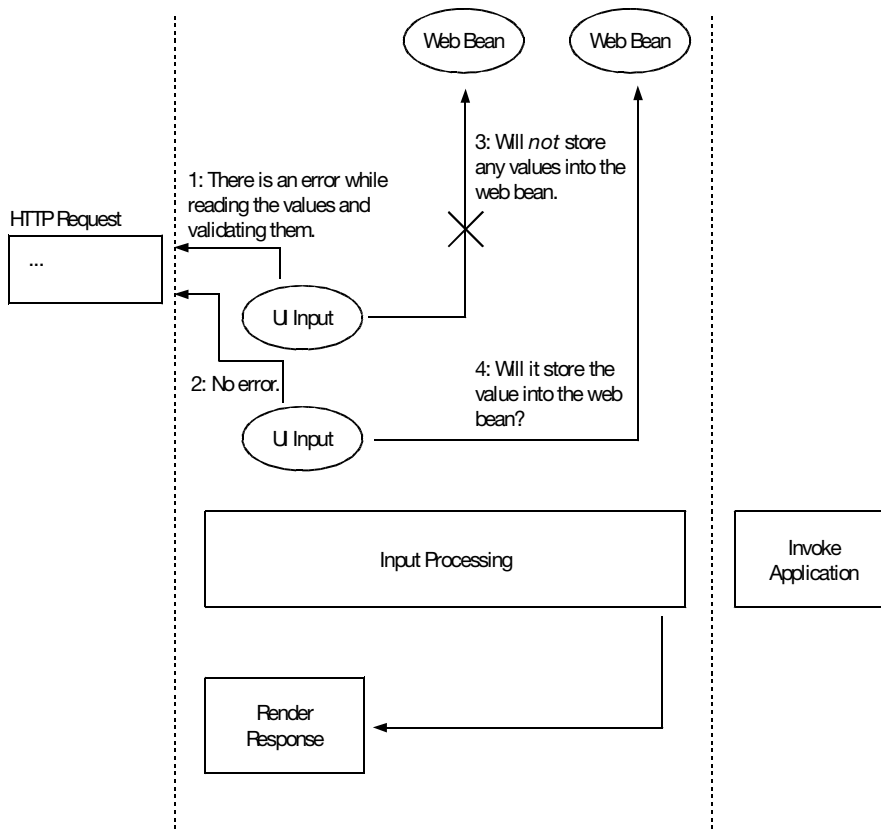


Figure 2-14. *What would happen if just one UI Input component failed?*

You certainly hope that it wouldn't. To achieve this effect, the part of updating the web beans is always split from the Input Processing phase to form a new phase called Update Domain Values (see Figure 2-15). That is, if there is any error when processing the input, the entire Update Domain Values phase will be skipped.

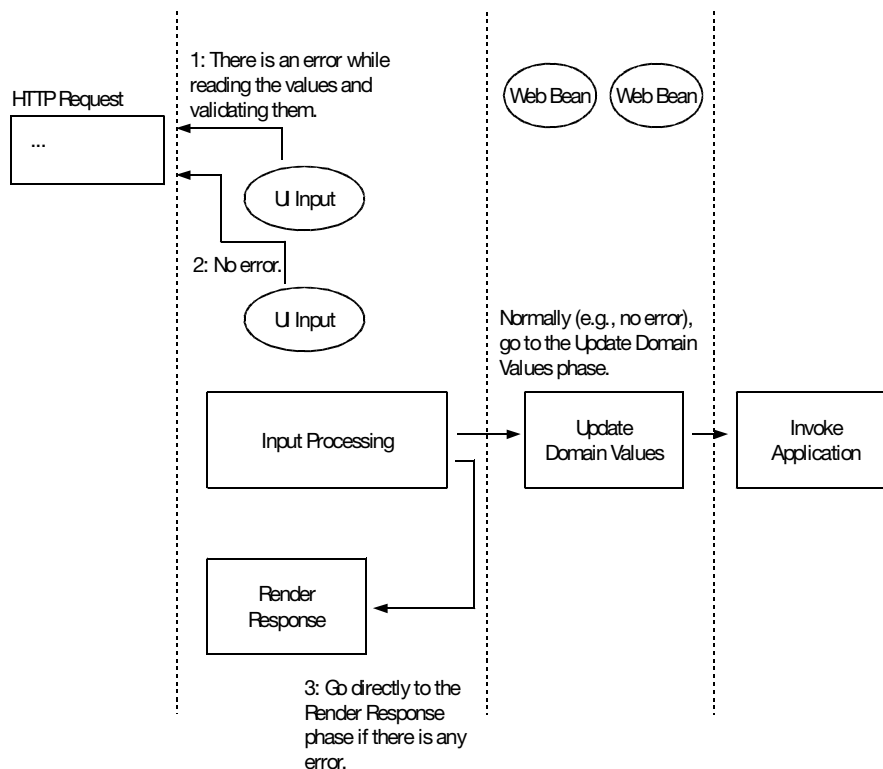


Figure 2-15. *Skipping the Update Domain Values phase if there is any error when processing input*

Finally, in order to display the error messages recorded, modify `getsymbol.xhtml` as shown in Figure 2-16. That is, the UI Messages component will render the error messages recorded.

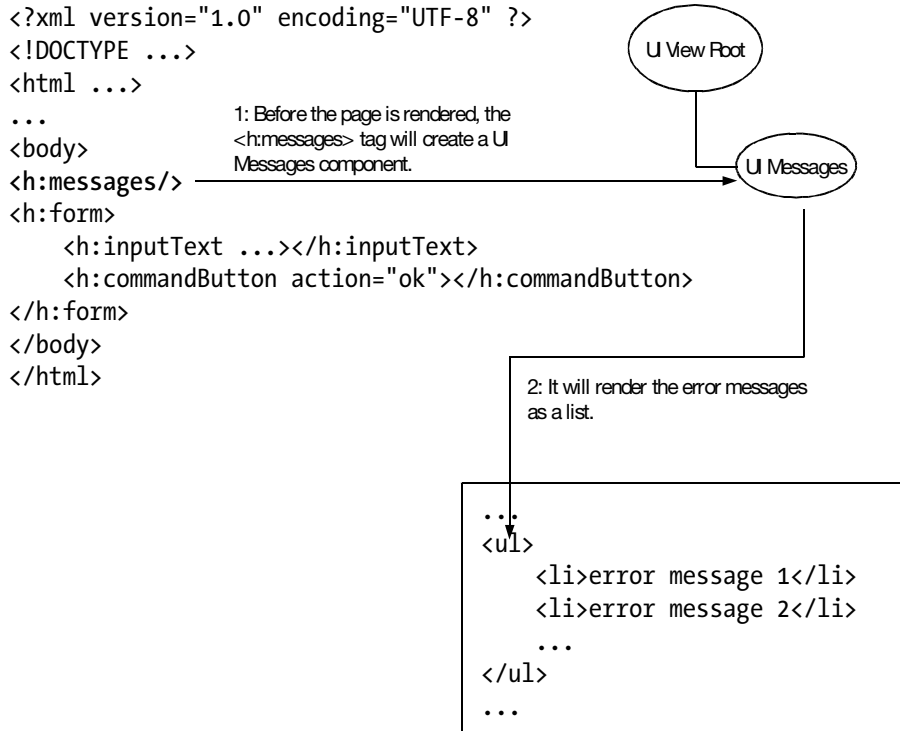


Figure 2-16. *Displaying error messages*

Now run the application, and submit the form with an empty symbol. The application will display an error message, as shown in Figure 2-17.

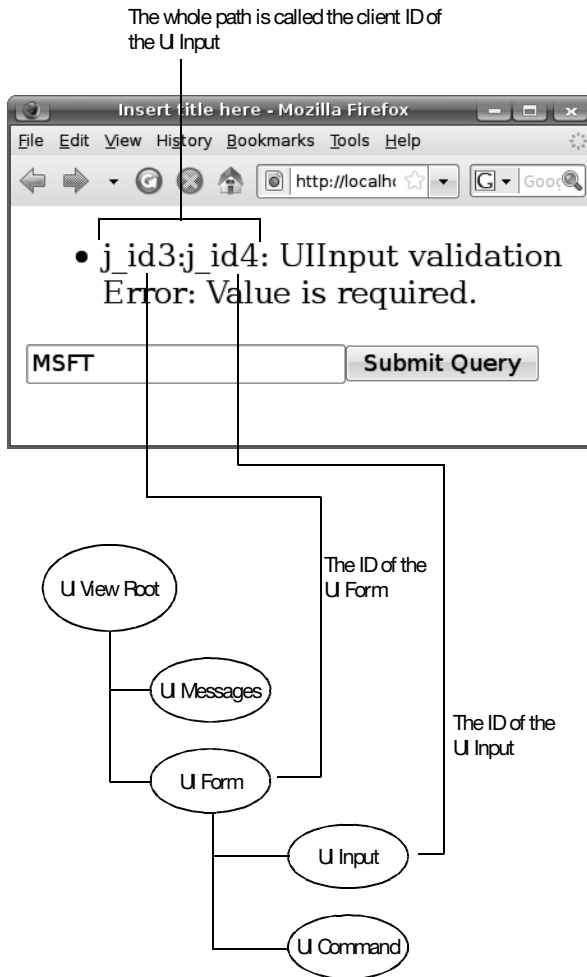


Figure 2-17. Error messages are displayed.

As you can see in Figure 2-17, the client ID of the UI Input component is the ID path from the form to the component concerned. In general, client IDs are mainly used as the values of the `id` or `name` attributes of the HTML elements generated. If you view the source of the HTML page, you'll see how various client IDs are used (see Listing 2-11).

Listing 2-11. Client IDs Used As *id* or *name* Attributes

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html ...>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8"></meta>
```

```

<title>Insert title here</title>
</head>
<body>
<form name="j_id3" id="j_id3" method="post" action="/Stock/faces/getsymbol.xhtml"
  enctype="application/x-www-form-urlencoded">
<input type="hidden" name="j_id3"></input>
<input type="text" name="j_id3:j_id4" value="MSFT"></input>
...
</form>
</body>
</html>

```

So, displaying the client ID is quite confusing to users. Instead, you should display a user-friendly description for the UI Input component. To do that, modify `getsymbol.xhtml` as shown in Listing 2-12.

Listing 2-12. *Specifying the Label*

```

...
<body>
<h:messages />
<h:form>
  <h:inputText value="#{b1.sym}" required="true"
    label="Stock symbol"></h:inputText>
  <h:commandButton action="ok"></h:commandButton>
</h:form>
</body>
</html>

```

Run the code again, and it will display the label instead of the client ID (see Figure 2-18).



Figure 2-18. *Labels displayed in error messages*

If you don't like this error message, you can provide your own. To do that, create a text file named `messages.properties` in the `stock` package (the file name is not really significant as long as it has a `.properties` extension). Figure 2-19 shows the content to input.

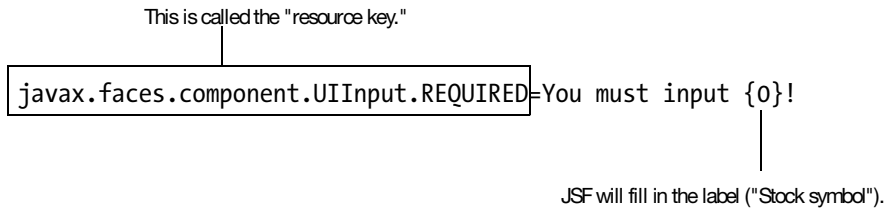


Figure 2-19. *Specifying the error message for missing input*

JSF will *not* load the file automatically; you must tell it to do so. Therefore, modify `faces-config.xml` as shown in Figure 2-20.

```
<faces-config ...>
  <application>
    <message-bundle>stock.messages</message-bundle>
  </application>
  <navigation-rule>
    <from-view-id>/getsymbol.xhtml</from-view-id>
    <navigation-case>
      <from-outcome>ok</from-outcome>
      <to-view-id>/stockvalue.xhtml</to-view-id>
    </navigation-case>
  </navigation-rule>
</faces-config>
```

The Package

The Base Filename without the .properties Extension

Figure 2-20. *Telling JSF to load a properties file*

Now JSF will load messages from this file and use them to override the default messages. Run the application, and it should work (see Figure 2-21).



Figure 2-21. Custom error messages in effect

Note that this error message will apply to all UI Input components in your application. If you'd like this error message to apply only to a single UI Input component, you can do that by using the code in Listing 2-13. This will override the message provided by the UI Input component (either the default text or the text loaded from a `.properties` file). Note that you can't use placeholders like `{0}` in this string.

Listing 2-13. Specifying the Error Message for a Single UI Input Component

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE ...>
<html ...>
...
<body>
<h:messages/>
<h:form>
    <h:inputText value="#{b1.sym}" required="true" label="Stock symbol"
        requiredMessage="Input is missing!"></h:inputText>
    <h:commandButton action="ok"></h:commandButton>
</h:form>
</body>
</html>
```

Run the application, and it will display the error message you specified.

Inputting a Date

You've learned how to let the user input a string (the symbol). What if you need to input a `Date` object? For example, say you'd like to allow the user to query the stock value on a particular date, as shown in Figure 2-22.

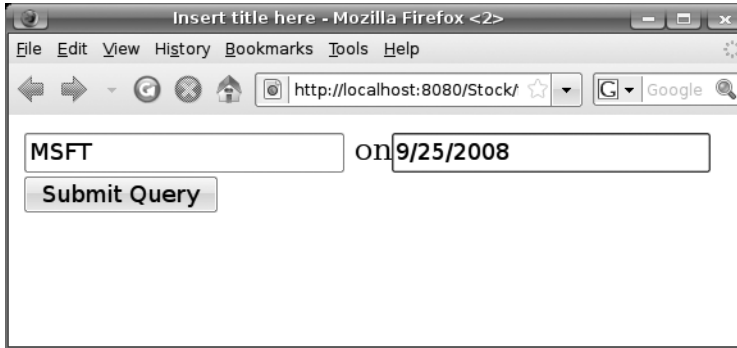


Figure 2-22. *Inputting a date*

How does that differ from inputting a string? During rendering (the Render Response phase), the UI Input component now needs to convert a Date object into a string (see Figure 2-23).

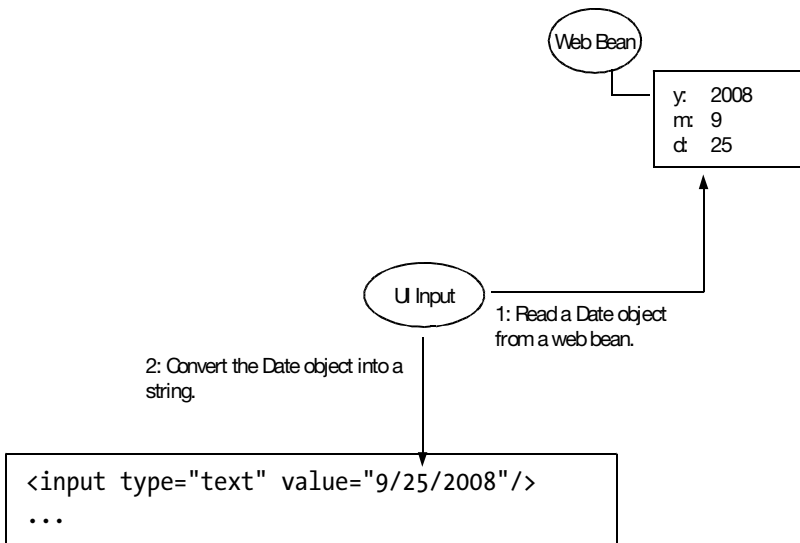


Figure 2-23. *Converting a Date object into a string in the Render Response phase*

When the user submits the form, in the Input Processing phase the UI Input component needs to convert the string back to a Date object (see Figure 2-24). Then in the Update Domain Values phase, it will store the Date object into a web bean.

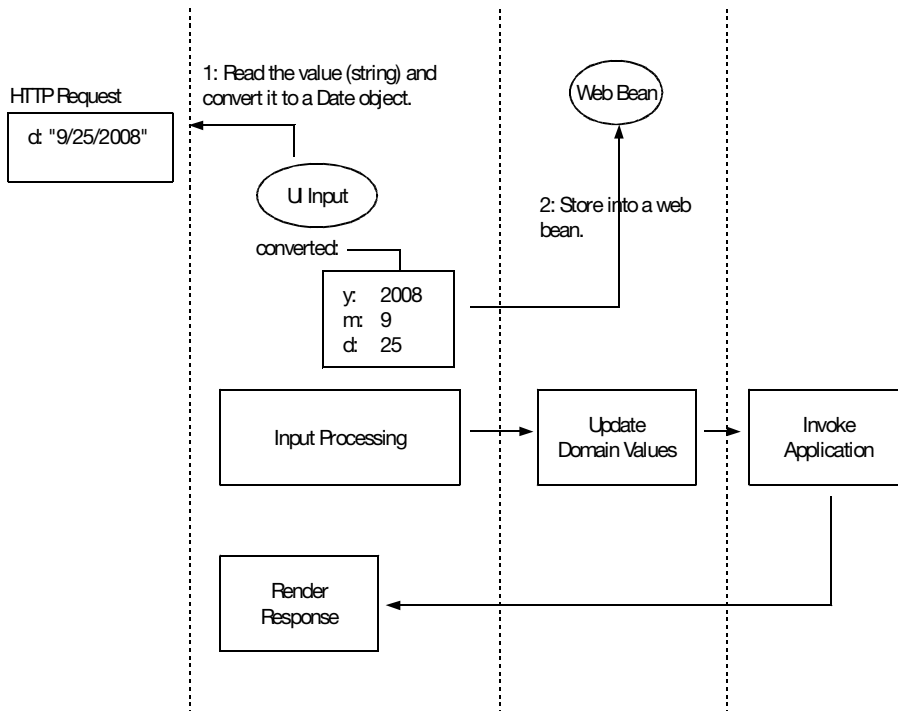


Figure 2-24. Converting a string into the Date object in the Input Processing phase

The UI Input component knows about a few common types such as `java.lang.Integer` and `java.lang.Double` and can convert between a value of such types and a string. Unfortunately, it doesn't know how to convert between a `java.util.Date` and a string. To solve this problem, you need to tell it to use a Date converter, as shown in Figure 2-25.

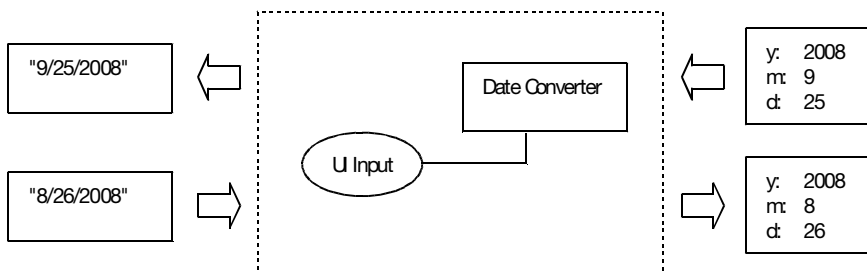


Figure 2-25. Using a Date converter

To implement this idea, modify `getsymbol.xhtml` as shown in Figure 2-26.

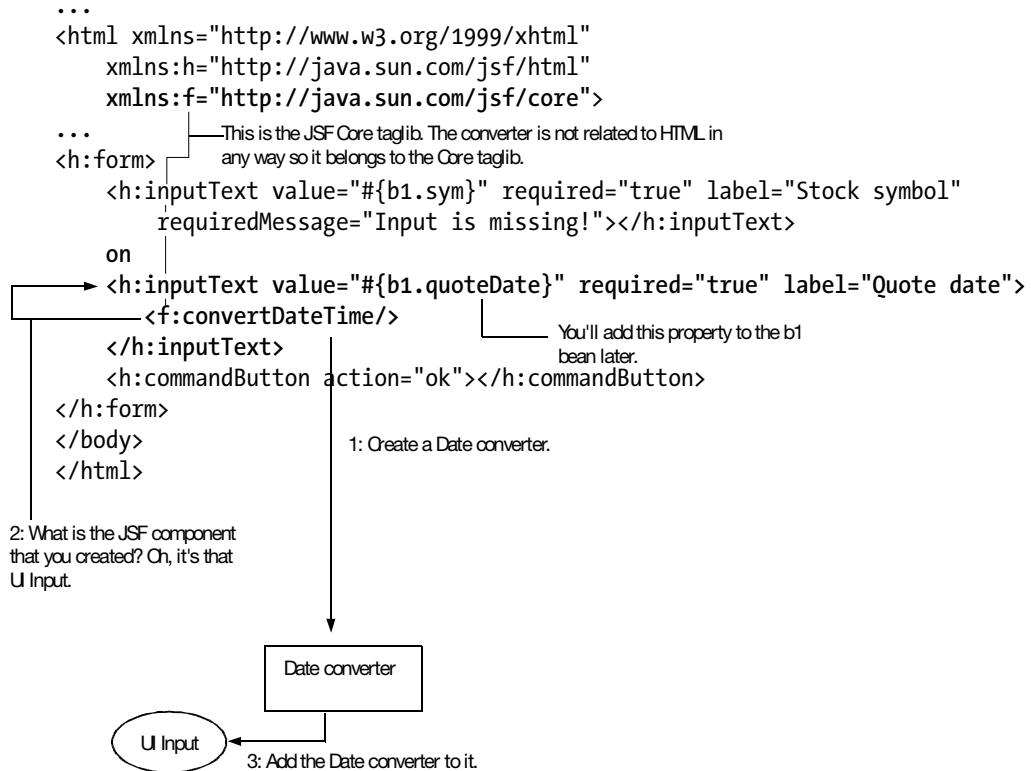


Figure 2-26. Specifying a Date converter for a UI Input component

Add the `quoteDate` property to the `b1` bean, as shown in Listing 2-14. You'll simply append the `Date` to the symbol before getting the hash code so that the calculated stock price will depend on both the `Date` and the symbol.

Listing 2-14. Providing a `quoteDate` Property

```

@Named("b1")
@RequestScoped
public class QuoteRequest {
    private String sym = "MSFT";
    private Date quoteDate = new Date();
}

```

```

public QuoteRequest() {
    System.out.println("Creating b1");
}
public String getSym() {
    System.out.println("getting sym");
    return sym;
}
public void setSym(String sym) {
    System.out.println("setting sym to: " + sym);
    this.sym = sym;
}
public Date getQuoteDate() {
    return quoteDate;
}
public void setQuoteDate(Date quoteDate) {
    this.quoteDate = quoteDate;
}
public int getStockValue() {
    return Math.abs((sym+quoteDate.toString()).hashCode());
}
}

```

Now run the application, and it should work (see Figure 2-27). Note that you’re creating a new `Date` object and assigning it to the `quoteDate` property for each request, so the UI Input component will display the current date on render.



Figure 2-27. Quote date working

Why does it show “Oct 26, 2008” instead of say “10/26/2008” or “26/10/2008”? This is controlled by two factors: the preferred language set in the browser and the style used by the converter. Table 2-1 shows some examples.

Table 2-1. *How the Date Format Is Determined*

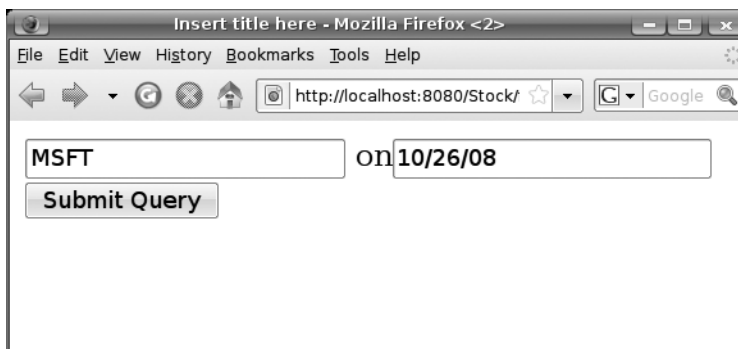
Preferred Language	Short Style	Medium Style	Long Style	Full Style
U.S. English	10/26/2008	Oct 26, 2008	October 26, 2008	Sunday, October 26, 2008
U.K. English	26/10/2008	26 Oct, 2008	...	...

If you don't set the style, it will use the medium style. To tell the converter to use, say, the short style, you can use the code shown in Listing 2-15.

Listing 2-15. *Specifying the Date Style*

```
...
<h:form>
  <h:inputText value="#{b1.sym}" required="true" label="Stock symbol"
    requiredMessage="Input is missing!"></h:inputText>
  on
  <h:inputText value="#{b1.quoteDate}" required="true" label="Quote date">
    <f:convertDateTime dateStyle="short"/>
  </h:inputText>
  <h:commandButton action="ok"></h:commandButton>
</h:form>
```

Now, run the application, and it should look like Figure 2-28. Obviously, the user now has to input the date using this short style too.

**Figure 2-28.** *Quote date displayed in short style*

You can also change the preferred language in the browser. For example, in Firefox, you can set the preferred language by selecting Tools ► Options ► Content ► Languages ► Choose. For this to work, you still need to tell JSF that you support that language in `faces-config.xml`. For example, in Listing 2-16, you're telling JSF that English (en), French

(fr), German (de), and Chinese (zh) are supported and that English is the default. What does *default* mean here? If an unsupported language such as Italian is requested, English will be used instead.

Listing 2-16. *Configuring the Supported Languages in faces-config.xml*

```
<faces-config ...>
  <application>
    <message-bundle>stock.messages</message-bundle>
    <locale-config>
      <default-locale>en</default-locale>
      <supported-locale>fr</supported-locale>
      <supported-locale>de</supported-locale>
      <supported-locale>zh</supported-locale>
    </locale-config>
  </application>
  <navigation-rule>
    <from-view-id>getsymbol.xhtml</from-view-id>
    <navigation-case>
      <from-outcome>ok</from-outcome>
      <to-view-id>stockvalue.xhtml</to-view-id>
    </navigation-case>
  </navigation-rule>
</faces-config>
```

Now, change your preferred language to, say, French, and run the application again. It will display the date in French.

You may wonder what will happen if you don't configure the supported languages in faces-config.xml at all. In that case, if your OS account (which runs JBoss) is set to use, say, Japanese, then JSF will assume that you support Japanese only.

Conversion Errors and Empty Input

Having to convert a string into a Date object introduces some new issues:

- What if the string is, say, "abc" and thus can't be converted?
- What if the user doesn't input anything (empty string) and submits the form?

For the first issue, the Date converter will log an error, as shown in Figure 2-29. This is exactly like the case when input is required but nothing was provided.

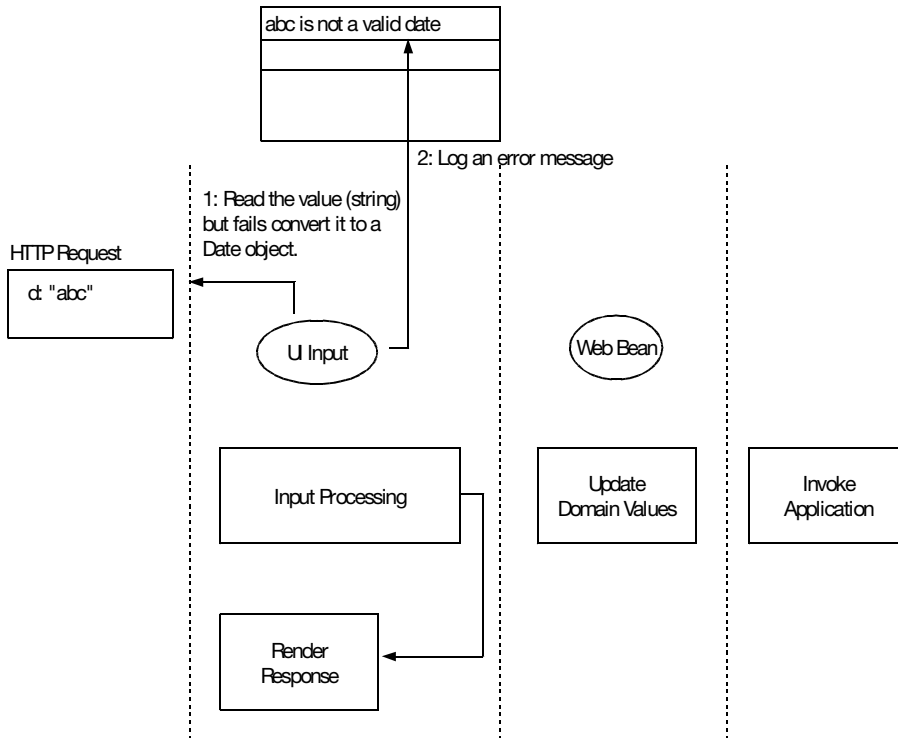


Figure 2-29. *Conversion failure*

However, there is a slight difference here from the previous scenario. When the UI Input component renders again in the Render Response phase, it would like to redisplay the raw input entered by the user (“abc”) instead of retrieving it from the web bean again. This is so the user can correct it. To do that, at the beginning of the Input Processing phase all UI Input components will always store the raw input string into themselves first (see Figure 2-30). This processing is split from the Input Processing phase to form a new phase called Apply Request Values. The rest of the Input Processing phase deals with data conversion and validation and is called the Process Validations phase.

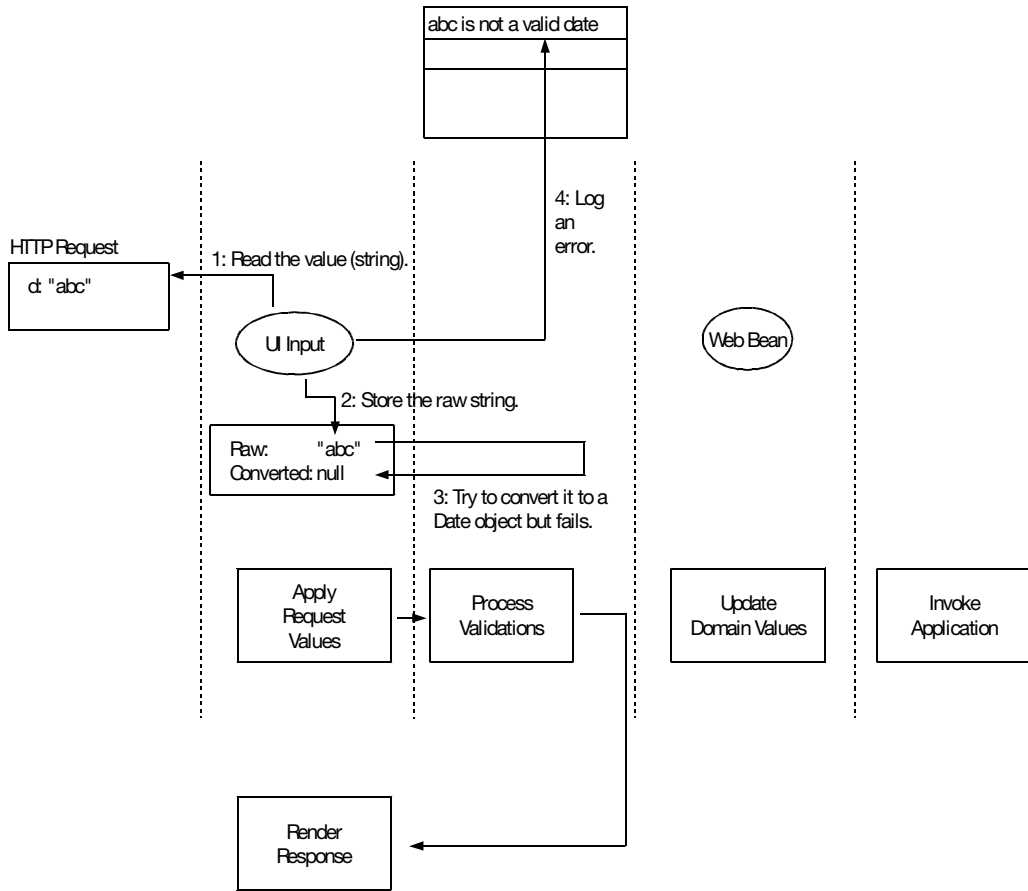


Figure 2-30. The *Apply Request Values* phase and the *Process Validations* phase

Note I've made up the term *Input Processing phase*; in the official JSF specification, there is no such term. You'll find only the phases shown in Figure 2-30.

Now run the application, and enter **abc** as the date. You'll see something like Figure 2-31.

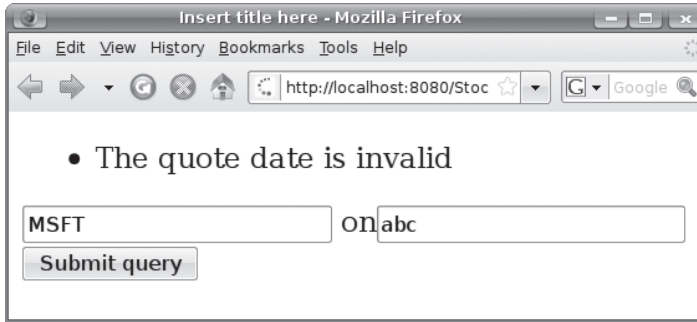


Figure 2-31. “abc” can’t be converted, and it is redisplayed.

Again, if you don’t like the error message, you can override it in the `messages.properties` file (see Figure 2-32).

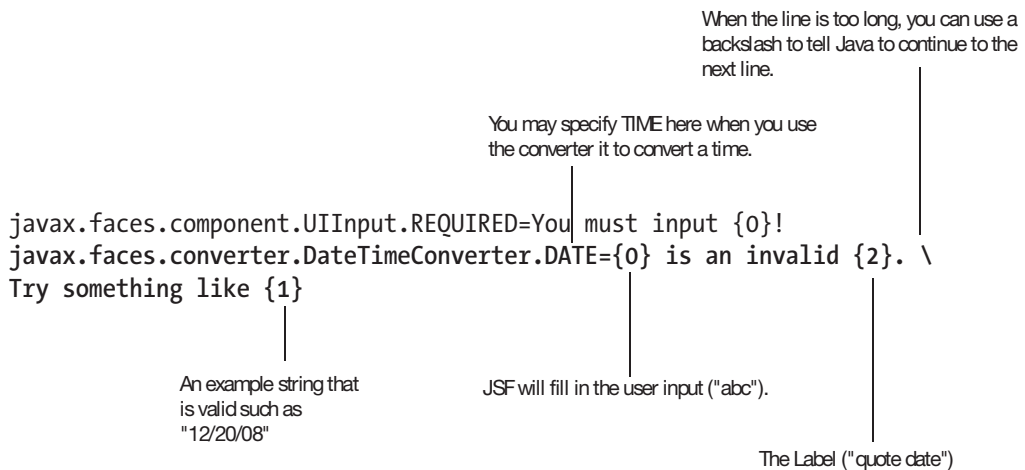


Figure 2-32. Customizing error messages using a `.properties` file

You may wonder how I found out what placeholders are supported by the Date converter. This is documented in the Javadoc of the `DateTimeConverter` class (see Figure 2-33).

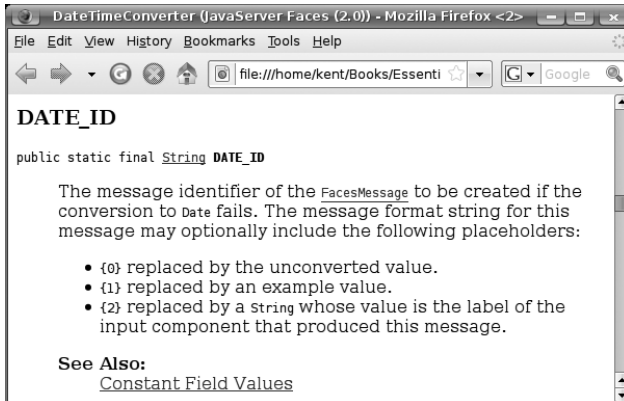


Figure 2-33. Finding out what placeholders are supported

Using a `.properties` file will affect all UI Input components. If you'd like to set a `.properties` file just for this UI Input component, you can do so using the code shown in Listing 2-17.

Listing 2-17. Specifying the Conversion Error Message for a Single UI Input Component

```
...
<h:form>
    <h:inputText value="#{b1.sym}" required="true" label="Stock symbol"
        requiredMessage="Input is missing!"></h:inputText>
    on
    <h:inputText value="#{b1.quoteDate}" required="true" label="Quote date"
        converterMessage="The quote date is invalid">
        <f:convertDateTime dateStyle="short"/>
    </h:inputText>
    <h:commandButton action="ok"></h:commandButton>
</h:form>
</body>
</html>
```

We've covered conversion errors, but what about empty input? Because an empty string can't be converted to a `Date`, will it be treated as a conversion error? No. The UI Input component will assume all input is optional, and an empty string is treated as "no input." In that case, it converts the empty string into `null` (if the property type is not a string) and stores it in the property of the web bean. As mentioned earlier, if the property type is a string, no conversion is needed, and it will store just an empty string in the property.

Again, if the input is not optional, you can simply set the `required` attribute to `true` (as you did in Figure 2-12).

For it to work, create a new class to serve as the b2 bean. Let's call it `StockService` (see Figure 2-36).

```
package stock;

import java.util.ArrayList;
import java.util.List;
import javax.faces.model.SelectItem;
import javax.annotation.Named;
import javax.context.ApplicationScoped;

@Named("b2")
@RequestScoped
public class StockService {
    private List<SelectItem> symbols;

    public StockService() {
        symbols = new ArrayList<SelectItem>();
        symbols.add(new SelectItem("MSFT"));
        symbols.add(new SelectItem("IBM"));
        symbols.add(new SelectItem("RHAT"));
    }

    public List<SelectItem> getSymbols() {
        return symbols;
    }
}
```

This class is provided by JSF. It represents an item for the user's selection.

This string will be displayed to the user.

It can return a List or an array.

Figure 2-36. *StockService* class

Run the application, and it should work. However, you may wonder why you need to provide it with a `List<SelectItem>` instead of just a `List<String>`. Say, for example, that instead of displaying short codes such as “MSFT” to the user, you’d like to display a longer description such as “Microsoft.” Internally all your processing will still use “MSFT,” though. To do that, modify the code as shown in Figure 2-37.

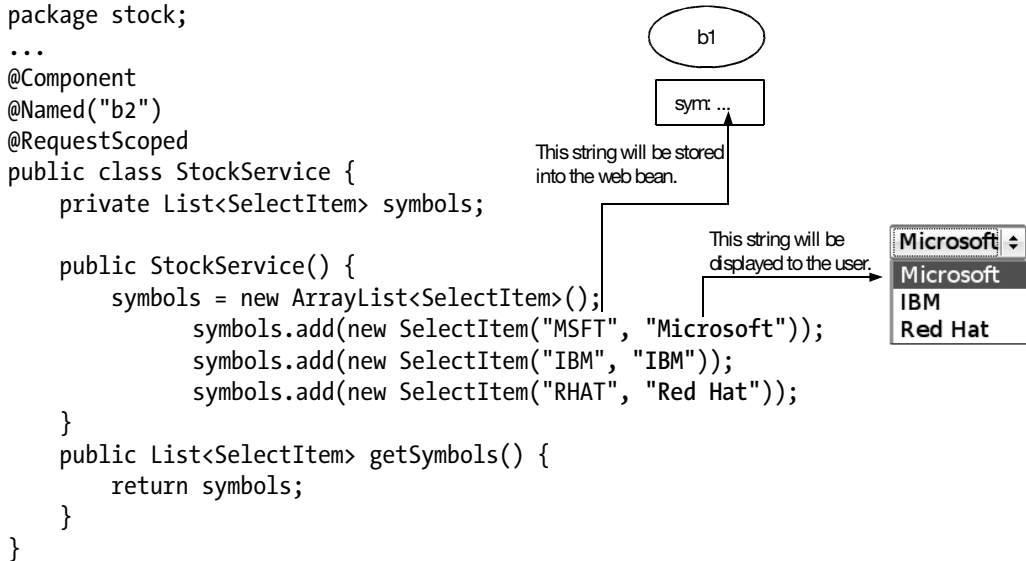


Figure 2-37. Using a short ID and a long description

Using a Single b2 Bean

At the moment, you're creating a new b2 bean for each request. However, because the list of symbols should be global, a single instance should be enough for all requests from all users. To do that, you need to know that in addition to the web bean table in each request, there is a web bean table for the whole application (see Figure 2-38).

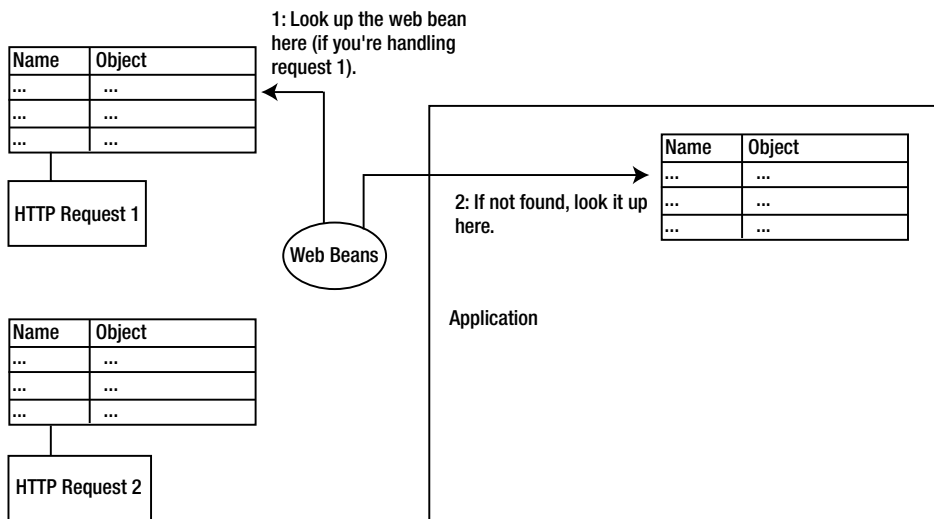


Figure 2-38. A web bean table for the whole application

To put the b2 bean into the application table, modify the StockService class as shown in Listing 2-18.

Listing 2-18. *Using the Application Scope*

```
package stock;

import javax.faces.model.SelectItem;
import javax.context.ApplicationScoped;
import javax.annotation.Named;

@Named("b2")
@ApplicationScoped
public class StockService {
    private List<SelectItem> symbols;

    public StockService() {
        symbols = new ArrayList<SelectItem>();
        symbols.add(new SelectItem("MSFT", "Microsoft"));
        symbols.add(new SelectItem("IBM", "IBM"));
        symbols.add(new SelectItem("RHAT", "Red Hat"));
    }
    public List<SelectItem> getSymbols() {
        return symbols;
    }
}
```

Run the application, and it will continue to work.

Hooking Up the Web Beans

For the moment, the stock value calculation is done in the QuoteRequest class (see Listing 2-19).

Listing 2-19. *Stock Value Calculation in QuoteRequest Class*

```
@Named("b1")
@RequestScoped
public class QuoteRequest {
    private String sym = "MSFT";
    private Date quoteDate = new Date();
    ...
}
```

```

    public int getStockValue() {
        return Math.abs((sym+quoteDate.toString()).hashCode());
    }
}

```

In a real implementation, you will need to hook up to a database or connect to a network service provider to get the stock value. This kind of work is best done in the `StockService` class. So, to make the code more realistic, let's move the calculation logic into the `StockService` class (see Listing 2-20).

Listing 2-20. *Moving the Stock Value Calculation into the StockService Class*

```

@Named("b2")
@ApplicationScoped
public class StockService {
    private List<SelectItem> symbols;

    public StockService() {
        symbols = new ArrayList<SelectItem>();
        symbols.add(new SelectItem("MSFT", "Microsoft"));
        symbols.add(new SelectItem("IBM", "IBM"));
        symbols.add(new SelectItem("RHAT", "Red Hat"));
    }
    public List<SelectItem> getSymbols() {
        return symbols;
    }
    public int getStockValue(QuoteRequest r) {
        return Math.abs((r.getSym() + r.getQuoteDate().toString()).hashCode());
    }
}

```

Then the code in the `QuoteRequest` class should call the `StockService` to get the stock value. But how do we get access to it (see Listing 2-21)?

Listing 2-21. *How Can b1 Get Access to b2?*

```

@Named("b1")
@RequestScoped
public class QuoteRequest {
    private String sym = "MSFT";
    private Date quoteDate = new Date();

    public QuoteRequest() {
        System.out.println("Creating b1");
    }
    public int getStockValue() {
        StockService stkSrv = ???;
        return stkSrv.getStockValue(this);
    }
}

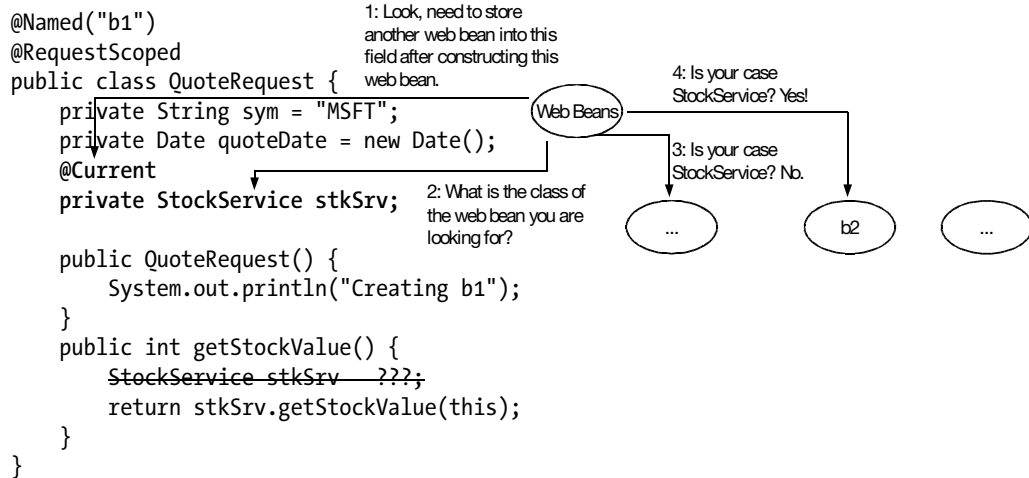
```

To solve this problem, you can tell web beans to inject *b2* into *b1* (see Figure 2-39). You may wonder why the annotation is called `@Current` instead of something like `@Inject`. For the moment, you don't need to worry about it.

```

import javax.inject.Current;
import javax.annotation.Named;
import javax.context.RequestScoped;

```

**Figure 2-39.** *Injecting b2 into b1*

Run the application, and it should continue to work.

Summary

In this chapter, you learned about how to use forms to get input from users. To handle a form submission, JSF will go through the following phases: Apply Request Values (store the raw input strings), Process Validations (convert the strings into objects and validate them), Update Domain Values (store the converted values into web beans), Invoke Application (set the outcome and determine the view ID of the next page), and Render Response (render the next page).

If there is any error in the Process Validations phase, it will jump right to the Render Response phase so that the web beans are not updated, the outcome is not set, and the current page is redisplayed.

To let the user edit a string in a text field, use the UI Input component, and set its value attribute to link it to the property of a web bean. To let the user choose an entry from a combo box, use the UI Select One component. You can also set its value attribute to link it to the property of a web bean. In addition, you need to provide a list of `SelectItem` items to it. Each `SelectItem` contains an object (the value) and its string presentation (the label).

For a UI Input component, if the type of the property is not a string or a built-in type such as `Integer` or `Double`, you need provide a converter to the UI Input component. If there is a conversion error, it will log an error message.

The UI Input component assumes that the input is optional and will convert an empty string into `null` (or leave it unchanged if the type of the property is string). If the input is mandatory, you need set the `required` attribute to `true`. Then it will log an error if no input is provided.

To display error messages, use the UI Messages component.

To let the user click a button, use a UI Command component. Specify the outcome in its action attribute. JSF will use the current view ID to look up the right navigation rule and use the outcome to look up the right navigation case to find the next view ID. The UI Command component will schedule a listener to set the outcome in the Invoke Application phase so that if there is any conversion or validation error, it will not set the outcome and the original page will be redisplayed.

You can customize the error messages using a message bundle (that is, one or more `.properties` files). This will affect the whole application. To customize it for a particular component, simply set the `right` attribute of the component.

Finally, you learned that you can inject one web bean into a field of another using `@Current`. Web beans will use the type of the field to locate the web bean to be injected.



Validating Input

In the previous chapter, you learned some basic ways of input validation: forcing the user to input something for a mandatory field and enforcing the format of the input (it can be converted into, say, a date properly). That is, you learned how to make sure that there is a converted value. In this chapter, you'll learn how to further validate that converted value.

Developing a Postage Calculator

Suppose that you'd like to develop an application to calculate the postage for sending a package from one place to another. The user will enter the weight of the package in kilograms (see Figure 3-1). Optionally, he can enter a "patron code" identifying himself as a patron to get a certain discount. After clicking OK, the calculator will display the postage (Figure 3-1).

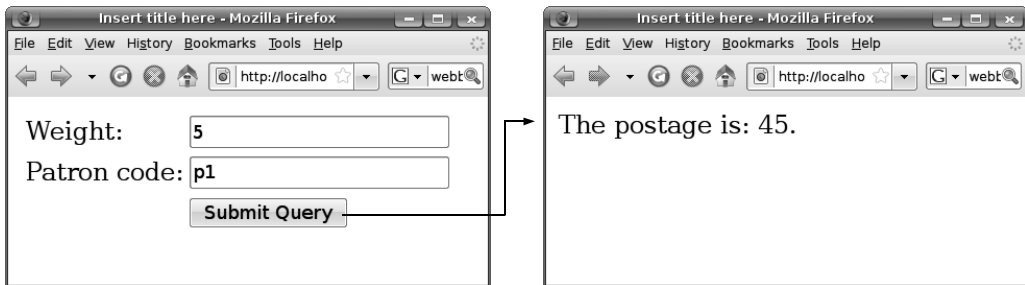


Figure 3-1. *A postage calculator*

To do that, create a new JSF project named Postage as usual (for example, copy an existing project and then do some manual updates). Then create a `getRequest.xhtml` file. To get the required tabular layout shown in Figure 3-1, you could use an HTML `<table>` element, as shown in Listing 3-1.

Listing 3-1. *Using the HTML <table> to Get the Desired Tabular Layout*

```

<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://java.sun.com/jsf/html">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
<title>Insert title here</title>
</head>
<body>
<h:form>
    <table>
        <tr>
            <td>Weight:</td>
            <td><h:inputText .../></td>
        </tr>
        <tr>
            <td>Patron code:</td>
            <td><h:inputText .../></td>
        </tr>
        <tr>
            <td></td>
            <td><h:commandButton .../></td>
        </tr>
    </table>
</h:form>
</body>

```

However, a design objective of JSF is to make it easier to support markups other than HTML (for example, simplified markup for low-powered mobile devices). Therefore, you can use an `<h:panelGrid>` tag instead of the HTML `<table>`, as shown in Figure 3-2. At runtime, this tag will create a UI Panel component, and more important, it will create another object (the HTML renderer). When the UI Panel needs to render itself, it will ask the renderer to do it. In this case, the HTML renderer will read the properties of the UI Panel and generate the corresponding HTML code such as a `<table>` element. The idea is that if you needed to generate, say, WML output, you could reuse the UI Panel but give it a WML renderer.

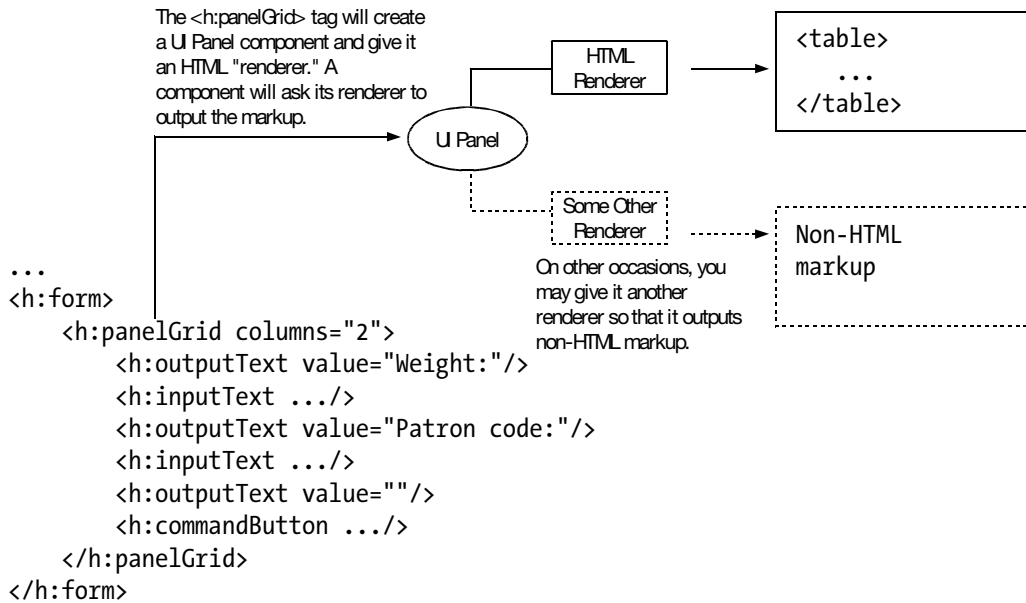


Figure 3-2. A component using a renderer

The HTML renderer will output the child components of the UI Panel sequentially, as shown in Figure 3-3.

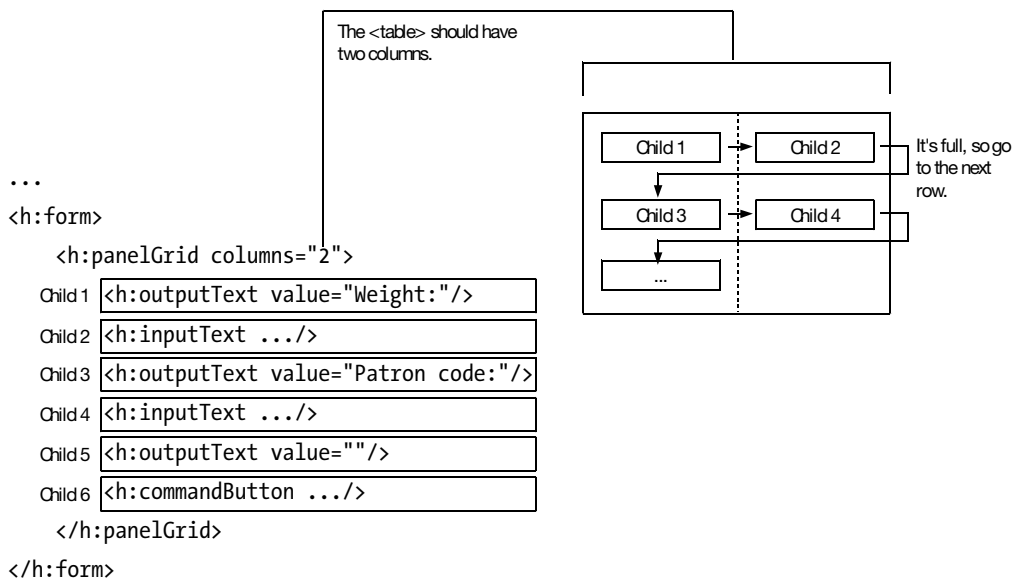


Figure 3-3. The HTML renderer lays out the child components in a `<table>` sequentially.

Next, you need to link a web bean to the two `<h:inputText>` tags. To do that, create a class called `Request` in the `postage` package as shown in Listing 3-2. You're making it a request-scoped web bean named `r`. To allow Web Beans to create it, you need a constructor that takes no argument. To calculate the postage, you're injecting a `PostageService` web bean into it. Finally, you need getters and setters for the properties to be edited.

Listing 3-2. *The Request Class*

```
package postage;
...
@Named("r")
@RequestScoped
public class Request {
    private int weight;
    private String patronCode;
    @Current
    private PostageService postageService;

    public Request() {
    }
    public Request(int weight, String patronCode) {
        this.weight = weight;
        this.patronCode = patronCode;
    }
    public int getWeight() {
        return weight;
    }
    public void setWeight(int weight) {
        this.weight = weight;
    }
    public String getPatronCode() {
        return patronCode;
    }
    public void setPatronCode(String patronCode) {
        this.patronCode = patronCode;
    }
    public int getPostage() {
        return postageService.getPostage(this);
    }
}
```

Listing 3-3 shows the `PostageService` class. In the constructor you hard-code some patrons and their respective discounts. For example, `p1` has 10 percent off. When calculating the postage, you assume that the postage is \$10 per kilogram.

Listing 3-3. *The PostageService Class*

```
package postage;

import java.util.HashMap;
import java.util.Map;
import javax.context.ApplicationScoped;

@ApplicationScoped
public class PostageService {
    private Map<String, Integer> patronCodeToDiscount;

    public PostageService() {
        patronCodeToDiscount = new HashMap<String, Integer>();
        patronCodeToDiscount.put("p1", 90);
        patronCodeToDiscount.put("p2", 95);
    }

    public int getPostage(Request r) {
        Integer discount = (Integer) patronCodeToDiscount
            .get(r.getPatronCode());

        int postagePerKg = 10;
        int postage = r.getWeight() * postagePerKg;
        if (discount != null) {
            postage = postage * discount.intValue() / 100;
        }
        return postage;
    }
}
```

A very important point in Listing 3-3 is that you are *not* giving it a name using `@Name`. Because the `Request` bean will inject it using its type (the `PostageService` class) and there is no EL expression referring to it using a name (yet), you don't need to name it.

Next, link the request properties and the UI Input components as shown in Listing 3-4.

Listing 3-4. *Linking the Request Properties and the UI Input Components*

```
...
<h:form>
    <h:panelGrid columns="2">
        <h:outputText value="Weight:"/>
        <h:inputText value="#{r.weight}"/>
        <h:outputText value="Patron code:"/>
        <h:inputText value="#{r.patronCode}"/>
        <h:outputText value=""/>
        <h:commandButton/>
    </h:panelGrid>
</h:form>
```

Next, create the result page. Let's call it `showpostage.xhtml`. Listing 3-5 shows the content.

Listing 3-5. *showpostage.xml*

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
<title>Insert title here</title>
</head>
<body>
The postage is: #{r.postage}.
</body>
</html>
```

Set the outcome in the `<h:commandButton>` tag as shown in Listing 3-6.

Listing 3-6. *Setting the Outcome*

```
<h:form>
    <h:panelGrid columns="2">
        <h:outputText value="Weight:"/>
        <h:inputText value="#{r.weight}"/>
        <h:outputText value="Patron code:"/>
        <h:inputText value="#{r.patronCode}"/>
```

```

        <h:outputText value=""/>
        <h:commandButton action="ok" value="OK"/>
    </h:panelGrid>
</h:form>

```

Define the navigation rule as shown in Listing 3-7.

Listing 3-7. *Navigation Rule*

```

<faces-config ...>
    <navigation-rule>
        <from-view-id>/getrequest.xhtml</from-view-id>
        <navigation-case>
            <from-outcome>ok</from-outcome>
            <to-view-id>/showpostage.xhtml</to-view-id>
        </navigation-case>
    </navigation-rule>
</faces-config>

```

Right-click the JBoss instance, and choose Add and Remove Projects to add the project. Then, run the application by going to <http://localhost:8080/Postage/faces/getrequest.xhtml>. It should work.

What If the Input Is Invalid?

At the moment, if the user enters a negative number as the weight (such as -5), the calculator will return a negative postage. This is no good. Instead, you'd like the application to tell the user that the weight is invalid.

Similarly, for the moment, if the user enters a nonexistent patron code such as p3, the calculator will simply treat it as “no discount” because the patron code is not found in the discount map. Ideally, it should tell him that this patron code is not found (see Figure 3-4).

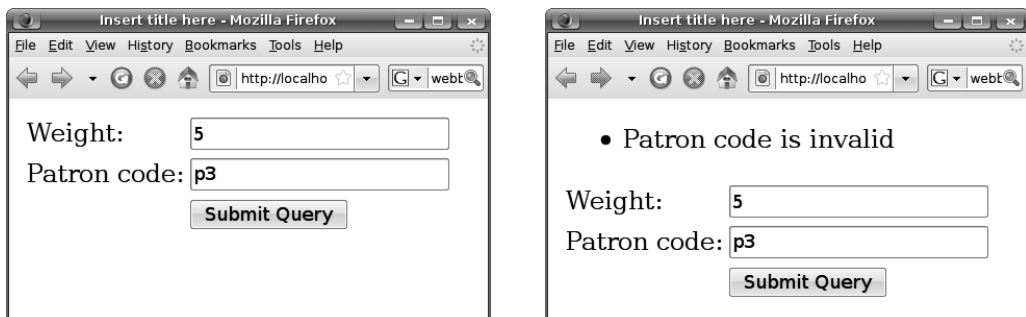


Figure 3-4. *Catching unknown patron codes*

Note that because the patron code is optional, if the user doesn't enter anything, this will *not* be treated as an error. This is a very important rule in validation: if some input is optional and it is indeed not provided, you must *not* perform any validation because there is simply no value to validate.

To validate the user input, you can add one or more validator objects to a UI Input component (see Figure 3-5). When the form is submitted, as mentioned earlier, in the Apply Request Values phase the UI Input component will store the raw input string (–5) locally. In the Process Validations phase, it will convert it into an object (an integer, –5, here). Then it will ask each of its validators (if any) in turn to validate the converted value (–5 here). If a validator fails, it will log an error message and tell the JSF engine to jump to the Render Response phase directly, without updating the web beans or setting the outcome.

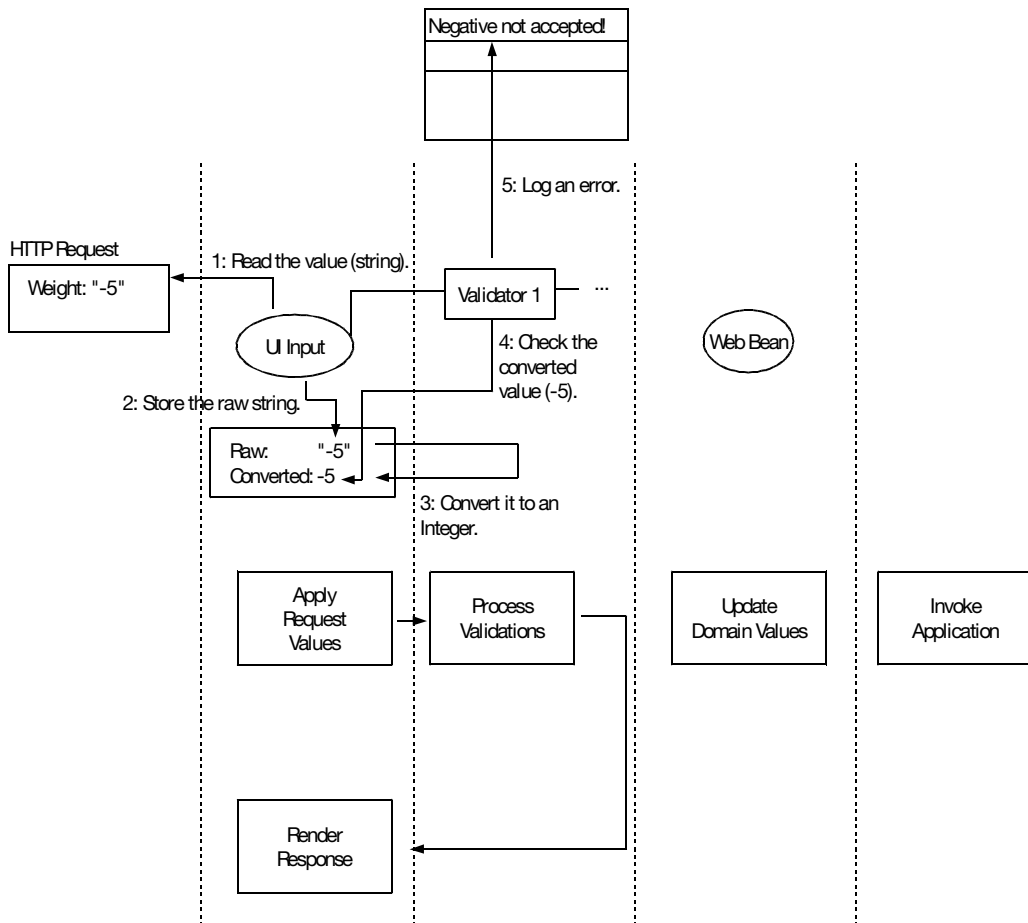


Figure 3-5. *How validators work*

In order to create such a validator, modify `getrequest.xhtml` as shown in Listing 3-8. This `<f:validateLongRange>` tag will create a “long range validator,” which will assume the converted value is a long and will check whether it is in a specified range. Here, you’re setting the minimum value to 0 so that anything less than 0 is an error. You could set the maximum value too, but there is no need for the current case. Note that the long range validator has nothing to do with the markup, so it is in the Core tag lib.

Listing 3-8. *Creating a Long Range Validator*

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE ...>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core">
...
<h:form>
  <h:panelGrid columns="2">
    <h:outputText value="Weight:"/>
    <h:inputText value="#{r.weight}">
      <f:validateLongRange minimum="0"/>
    </h:inputText>
    <h:outputText value="Patron code:"/>
    <h:inputText value="#{r.patronCode}"/>
    <h:outputText value=""/>
    <h:commandButton action="ok" value="OK"/>
  </h:panelGrid>
</h:form>
```

To display the error message, you need an `<h:messages>` tag, and you need to set the label as shown in Listing 3-9. Why set the label? As explained in the previous chapter, if you don’t set the label, the error message will display the client ID, which is daunting to the user.

Listing 3-9. *Displaying the Error Message*

```
<h:messages/>
<h:form>
  <h:panelGrid columns="2">
    <h:outputText value="Weight:"/>
    <h:inputText value="#{r.weight}" label="weight">
      <f:validateLongRange minimum="0"/>
    </h:inputText>
```

```
<h:outputText value="Patron code:"/>
<h:inputText value="#{r.patronCode}"/>
<h:outputText value=""/>
<h:commandButton action="ok" value="OK"/>
</h:panelGrid>
```

Now run the application again, and it should work (see Figure 3-6).

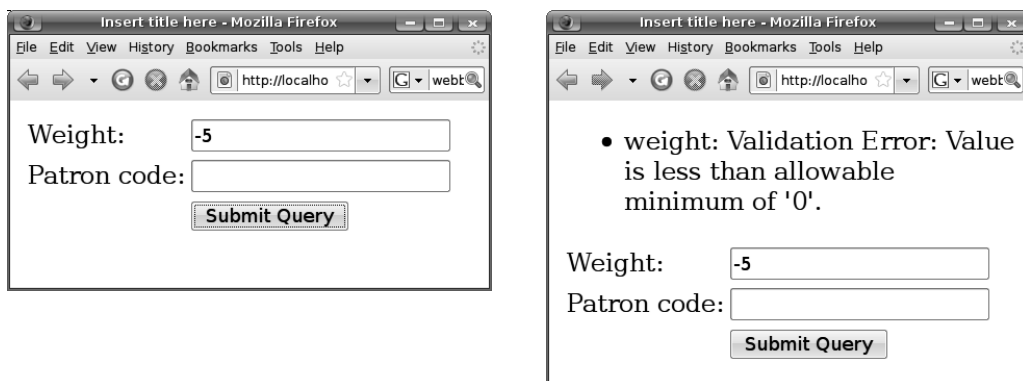


Figure 3-6. *Negative weight caught as an error*

Just like the error messages for missing required input or conversion errors, you can customize the error message using a message bundle. For example, create `Postage.properties` in the `postage` package, as shown in Figure 3-7.

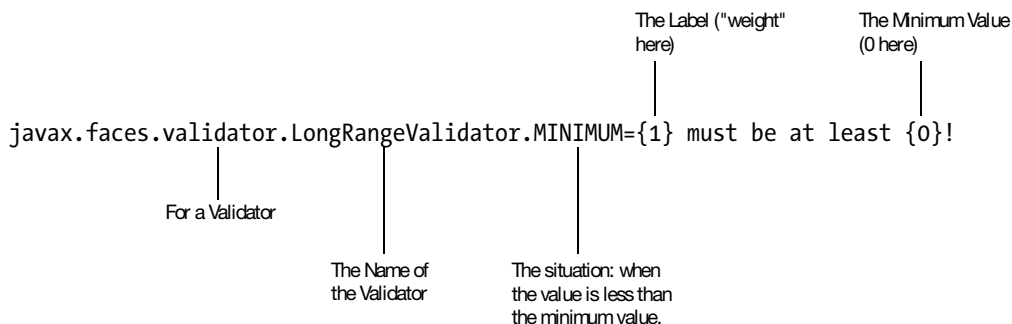


Figure 3-7. Customizing validator error messages

You may wonder how I found out what placeholders are supported by the long range validator. This is documented in the Javadoc of the `LongRangeValidator` class.

Specify the message bundle in `faces-config.xml` as shown in Listing 3-10.

Listing 3-10. *Specifying the Message Bundle*

```
<faces-config ...>
  <application>
    <message-bundle>postage.Postage</message-bundle>
  </application>
  <navigation-rule>
    <from-view-id>/getrequest.xhtml</from-view-id>
    <navigation-case>
      <from-outcome>ok</from-outcome>
      <to-view-id>/showpostage.xhtml</to-view-id>
    </navigation-case>
  </navigation-rule>
</faces-config>
```

Make sure the application is reloaded. Then run the app, and it should work (Figure 3-8).

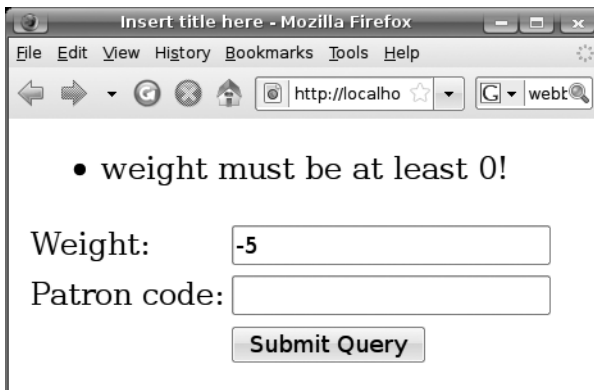


Figure 3-8. *Customized error message displayed*

This will affect all uses of the long range validator, though. If you'd like to customize it for a single UI Input only, you can do it as shown in Listing 3-11.

Listing 3-11. *Specifying the Validation Error Message for a Single Component*

```
...
<h:form>
  <h:panelGrid columns="2">
    <h:outputText value="Weight:"/>
```

```

<h:inputText value="#{r.weight}" label="weight"
    validatorMessage="weight cannot be negative! ">
    <f:validateLongRange minimum="0"/>
</h:inputText>
<h:outputText value="Patron code:"/>
<h:inputText value="#{r.patronCode}"/>
<h:outputText value=""/>
<h:commandButton action="ok" value="OK"/>
</h:panelGrid>
</h:form>

```

In addition to this validator, there are similar ones for checking doubles and another for checking the lengths of strings. They are shown in Listing 3-12, and their resource keys are shown in Listing 3-13. The first tag creates a `LengthValidator` enforcing the length of a string to be between 3 and 20. The second tag creates a `DoubleRangeValidator` enforcing a double to be between 0 and 999999. What if you'd like to validate an integer? You'll simply use the long range validator. What if you'd like to validate a float? You'll simply use the double range validator.

Listing 3-12. Double Validator and String Validator

```

<f:validateLength minimum="3" maximum="20"/>
<f:validateDouble minimum="0" maximum="999999"/>

```

Listing 3-13 shows their resource keys.

Listing 3-13. Resource Keys for the Other Validators

```

javax.faces.validator.LengthValidator.MINIMUM=...
javax.faces.validator.LengthValidator.MAXIMUM=...
javax.faces.validator.DoubleRangeValidator.MINIMUM=...
javax.faces.validator.DoubleRangeValidator.MAXIMUM=...

```

Null Input and Validators

If the user doesn't input anything as the weight, what will the long range validator do? Recall the rule regarding validation: if there is no input, you must skip the validation because there is simply no value to validate. JSF does that automatically. In this case, if the user doesn't input the weight, it will be converted to null, and all validation will be skipped. This is no good, because the weight should be mandatory. To fix it, simply mark it as required (see Listing 3-14).

Listing 3-14. *Marking the Weight As Required*

```
...
<h:form>
  <h:panelGrid columns="2">
    <h:outputText value="Weight:"/>
    <h:inputText value="#{r.weight}" label="weight" required="true"
      validatorMessage="weight cannot be negative!">
      <f:validateLongRange minimum="0"/>
    </h:inputText>
    <h:outputText value="Patron code:"/>
    <h:inputText value="#{r.patronCode}"/>
    <h:outputText value=""/>
    <h:commandButton action="ok" value="OK"/>
  </h:panelGrid>
</h:form>
```

Then not inputting the weight will result in Figure 3-9.



Figure 3-9. *The weight is mandatory.*

Validating the Patron Code

Now the weight field is working fine. How do you validate the patron code? No built-in validator is suitable. In that case, you can specify a validator method as shown in Figure 3-10.

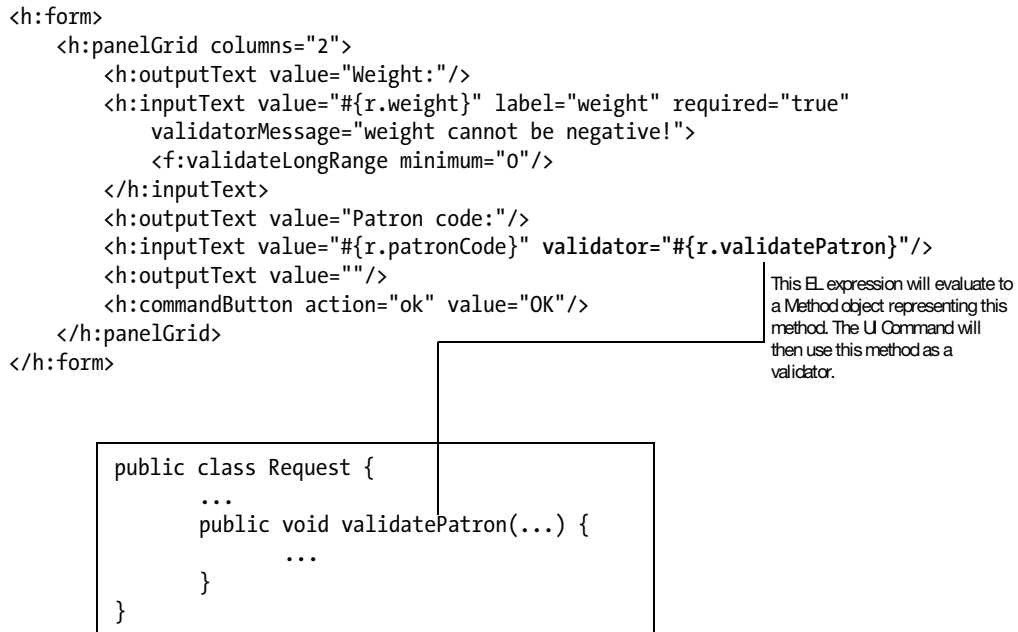


Figure 3-10. *Using a validator method*

So, create that `validatePatron()` method as shown in Figure 3-11. In short, the method will be passed the component and the converted value, and it is free to perform whatever check it desires. If the converted value is considered invalid, it should throw a `ValidatorException` and provide a `FacesMessage`.

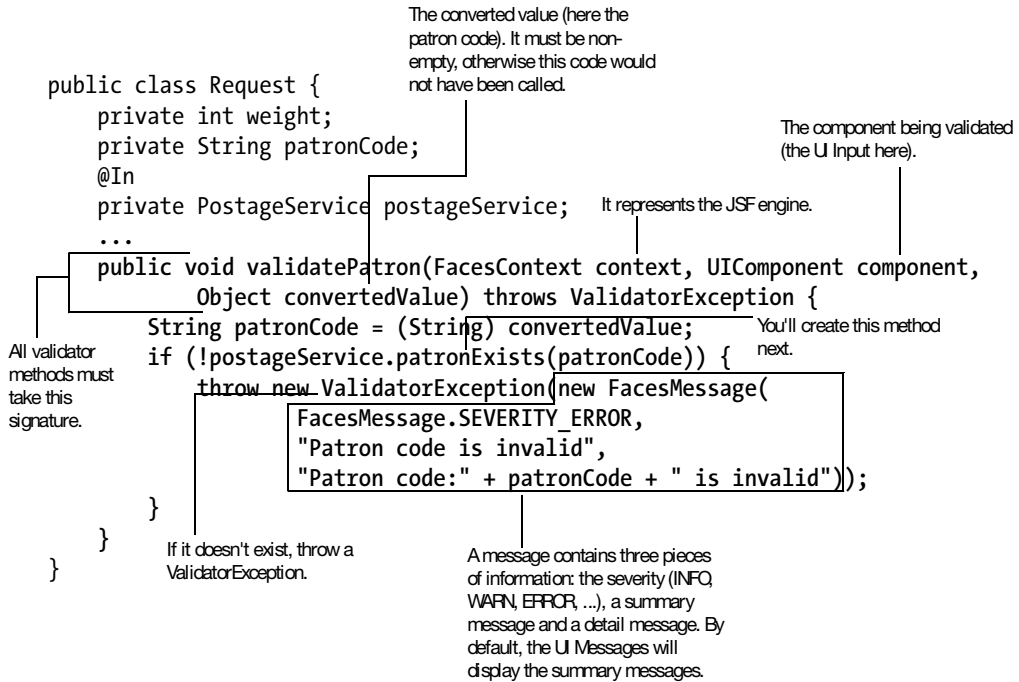


Figure 3-11. *Implementing a validator method*

Create the `patronExists()` method in the `PostageService` class as shown in Listing 3-15.

Listing 3-15. *Implementing the `patronExists()` Method*

```

...
public class PostageService {
    private Map<String, Integer> patronCodeToDiscount;

    public PostageService() {
        patronCodeToDiscount = new HashMap<String, Integer>();
        patronCodeToDiscount.put("p1", 90);
        patronCodeToDiscount.put("p2", 95);
    }
    ...
    public boolean patronExists(String patronCode) {
        return patronCodeToDiscount.containsKey(patronCode);
    }
}

```

Now run the application, and it should work (see Figure 3-12).

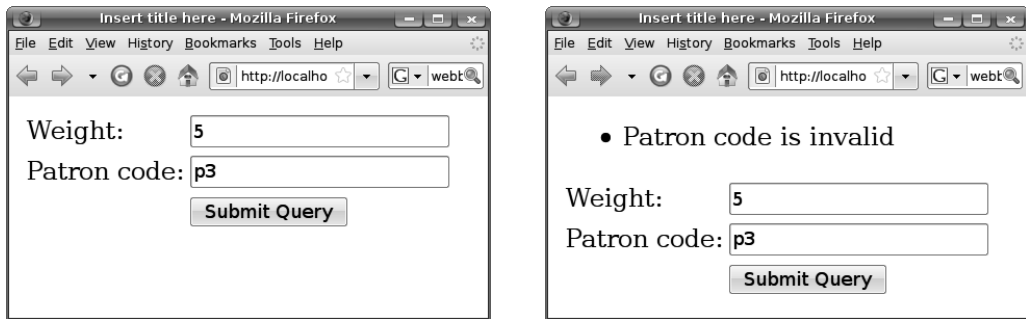


Figure 3-12. *Unknown patron code caught*

Creating a Custom Validator for the Patron Code

Suppose that you have multiple pages on which the user can input the patron code. Some such pages may not store information to the Request bean at all. In that case, you can no longer use the `patronExists()` method in the Request class as the validator method. To solve this problem, it would be great if you had a validator for the patron code as shown in Figure 3-13.

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE ...>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:x="http://foo.com">
...
<h:messages/><!-- This namespace represents your own tag lib. -->
<h:form>
  <h:panelGrid columns="2">
    <h:outputText value="Weight:"/>
    <h:inputText value="#{r.weight}" ...>
      <f:validateLongRange minimum="0"/>
    </h:inputText>
    <h:outputText value="Patron code:"/>
    <h:inputText value="#{r.patronCode}" validator="#{r.validatePatron}"
      <x:validatePatron/>
    </h:inputText>
    <h:outputText value=""/>
    <h:commandButton action="ok" value="OK"/>
  </h:panelGrid>
</h:form>
```

Don't need it anymore.

This tag will create your own patron code validator.

Figure 3-13. *Using a custom validator for the patron code*

To create such a tag (and the tag lib), create a META-INF folder in your Java source folder, and then create a file `foo.taglib.xml` in it. (The file name is not important as long as it ends with `.taglib.xml`.) Figure 3-14 shows the content. In short, it defines a namespace to identify your tag lib, define a `<validatePatron>` tag, and link it to the validator whose ID is `foo.v1`.

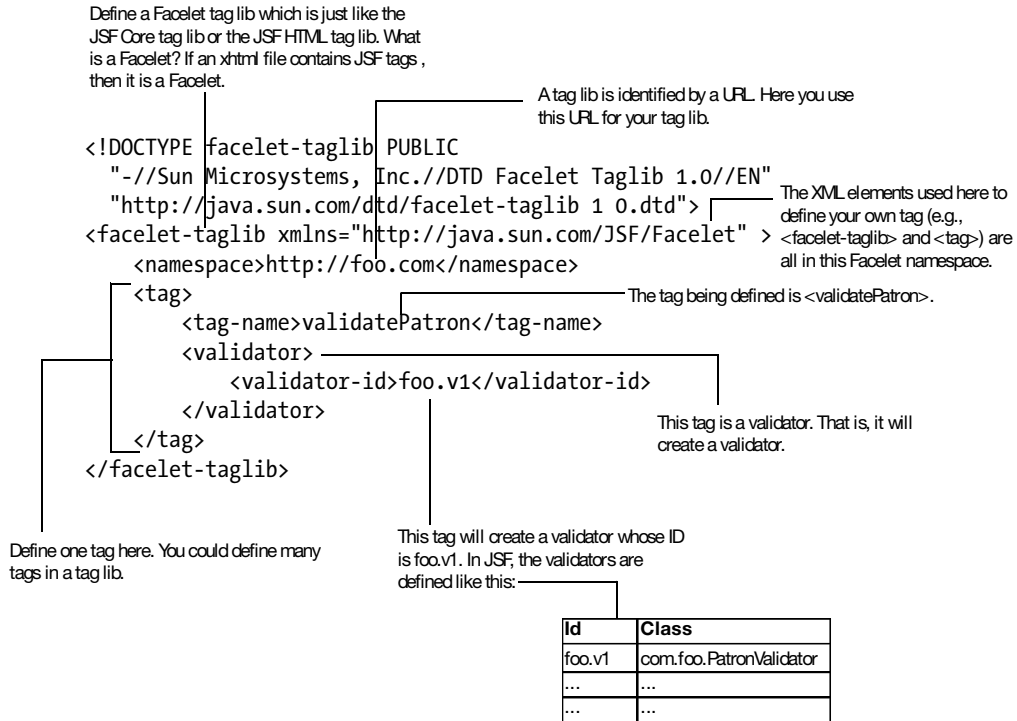


Figure 3-14. *Defining your own validator tag*

Note In Mojarra 2.0.0.PR2, there is a bug preventing `*.taglib.xml` files in the META-INF folder on the classpath to be discovered. To work around it, put the whole META-INF folder into `WebContent` and then explicitly specify the tag lib in `web.xml`, as shown in Listing 3-16.

Listing 3-16. Explicitly Specifying a Tag Lib

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app ...>
  ...
  <servlet>
    <servlet-name>JSF</servlet-name>
    <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
```

```

    </servlet>
    <servlet-mapping>
        <servlet-name>JSF</servlet-name>
        <url-pattern>/faces/*</url-pattern>
    </servlet-mapping>
    <context-param>
        <param-name>javax.faces.FACELETS_LIBRARIES</param-name>
        <param-value>/META-INF/foo.taglib.xml</param-value>
    </context-param>
</web-app>

```

To define the `foo.v1` validator, modify `faces-config.xml` as shown in Listing 3-17.

Listing 3-17. *Defining a JSF Validator*

```

<faces-config ...>
    <application>
        <message-bundle>postage.Postage</message-bundle>
    </application>
    <navigation-rule>
        <from-view-id>/getrequest.xhtml</from-view-id>
        <navigation-case>
            <from-outcome>ok</from-outcome>
            <to-view-id>/showpostage.xhtml</to-view-id>
        </navigation-case>
    </navigation-rule>
    <validator>
        <validator-id>foo.v1</validator-id>
        <validator-class>postage.PatronValidator</validator-class>
    </validator>
</faces-config>

```

Create the `PatronValidator` class in the `postage` package. Listing 3-18 shows the content. Note that you must implement the `Validator` interface provided by JSF, and the `validate()` method must carry exactly the same signature as that of a validator method.

Listing 3-18. *Creating the Validator Class*

```

package postage;

import javax.faces.component.UIComponent;
import javax.faces.context.FacesContext;

```

```
import javax.faces.validator.Validator;
import javax.faces.validator.ValidatorException;

public class PatronValidator implements Validator {
    @Override
    public void validate(FacesContext context, UIComponent component,
        Object convertedValue)
        throws ValidatorException {
        ...
    }
}
```

Fill in the code in the method as shown in Figure 3-15. You may wonder why you use an EL expression to get access to the `PostageService` object instead of injection. This is because the `PatronValidator` object will be created by JSF, not by Web Beans. Because JSF knows nothing about injection, no injection will be performed.

```
public class PatronValidator implements Validator {
    @Override
    public void validate(FacesContext context, UIComponent component,
        Object convertedValue)
        throws ValidatorException {
        String patronCode = (String) convertedValue;
        Application app = context.getApplication();
        PostageService ps = (PostageService) app.evaluateExpressionGet(context,
            "#{ps}", PostageService.class);
        if (!ps.patronExists(patronCode)) {
            throw new ValidatorException(new FacesMessage(
                FacesMessage.SEVERITY_ERROR,
                "Patron code is invalid",
                "Patron code:" + patronCode + " is invalid"));
        }
    }
}

@ApplicationScoped
@Named("ps")
public class PostageService {
    private Map<String, Integer> patronCodeToDiscount;
    ...
    public boolean patronExists(String patronCode) {
        return patronCodeToDiscount.containsKey(patronCode);
    }
}
```

The application contains various helpers that can be customized. Here, you'll use it to evaluate an EL expression.

Evaluate the EL expression "#{ps}" and then call `get()` on it. As an EL expression can be set too, you need to be explicit about you'd like to get or set it.

Need to specify a name for the `PostageService` bean so that it can be looked up in an EL expression.

The result should belong to this class. JSF will try to convert the result to this class if required.

Figure 3-15. *Performing validation in validator class*

Finally, delete the `validatePatron()` method in the `Request` class because it is no longer used. Run the application, and it should continue to work.

For the moment, you're embedding the error messages directly in the Java code. If you need to support multiple languages, you may want to put them into the `Postage.properties` file and then load the error messages in Java code, as shown in Figure 3-16.

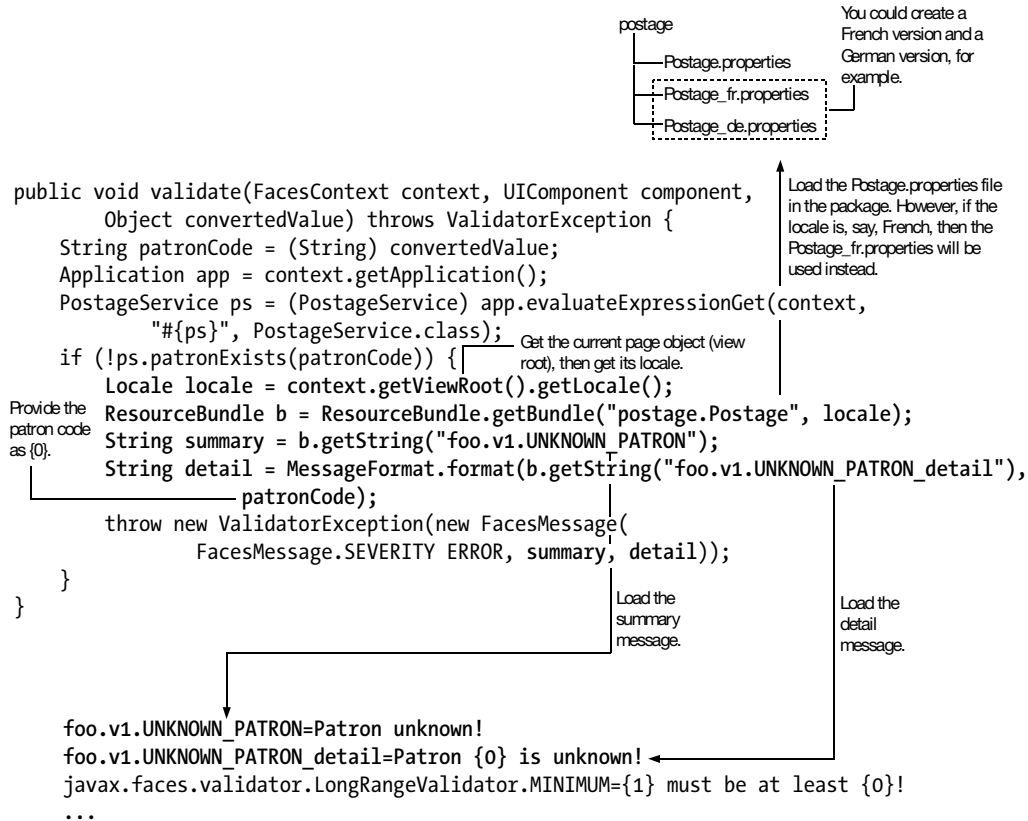


Figure 3-16. Providing error messages in a *.properties* file

Run the application, and it should work.

Displaying the Error Messages in Red

Suppose that you'd like the error messages to be in red. To do that, modify `getrequest.xhtml` as shown in Figure 3-17.

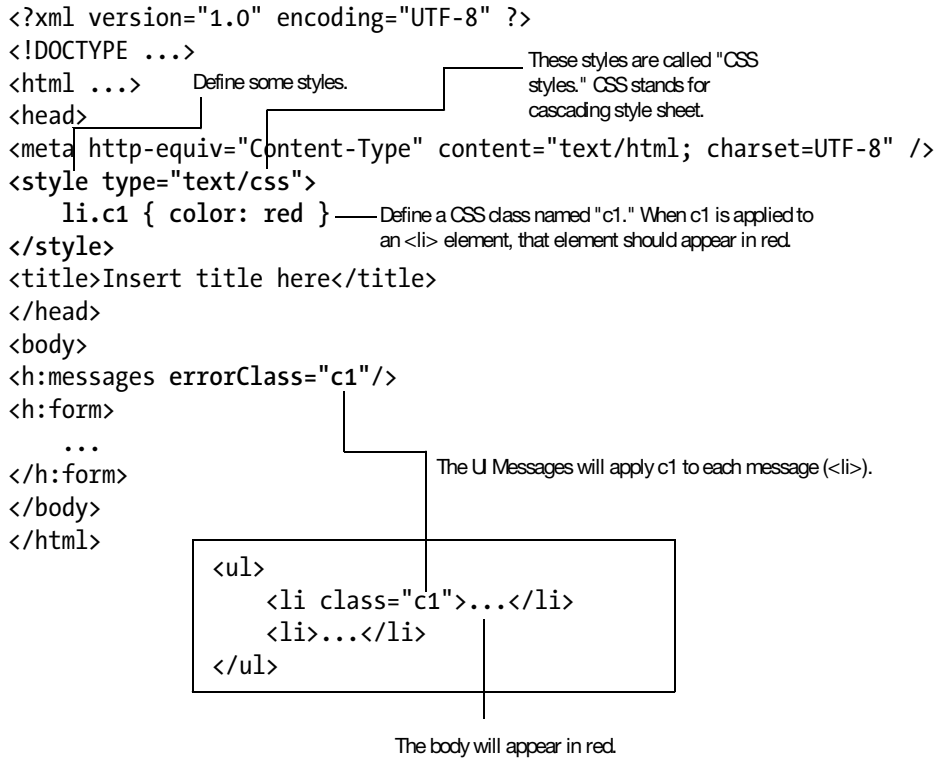


Figure 3-17. Assigning CSS class to error messages

Now run the application, and it will work.

This example shows how <h:messages> accepts CSS classes. In fact, many other JSF tags do the same. For example, the <h:dataTable> accepts CSS classes for its headers and columns. Look up the tag documentation to find out the details.

Displaying the Error Message Along with the Field

You may wonder what the purpose of the detail message in a FacesMessage is. It is intended to be displayed along with the field (see Figure 3-18).



Figure 3-18. Detail message displayed along with the field

To do that, you need to understand how JSF stores the error message. For example, if the weight is negative, the error message is associated with the client ID of the weight component, as shown in Figure 3-19.

Client id	Severity	Summary	Detail
f:w	ERROR	weight is negative	...
...	...	...	...
...	...	...	...

Let's assume this is whatever client ID generated by JSF for the weight component.

```
<form id="f" ...>  
  <input id="f:w" ...>  
  ...  
</form>
```

Figure 3-19. Client ID and error message stored together

To display the detail error message associated with client ID f:w (if any), you can use the `<h:message>` tag (*not* the plural `<h:messages>`), as shown in Listing 3-19. It will create a UI Message component that will output the detail error message.

Listing 3-19. Using the `<h:message>` Tag

```
...  
<h:messages errorClass="c1"/>  
<h:form>  
  <h:panelGrid columns="2">  
    <h:outputText value="Weight:"/>
```

```

    <h:inputText value="#{r.weight}" label="weight" required="true"
        validatorMessage="weight cannot be negative!">
        <f:validateLongRange minimum="0"/>
    </h:inputText>
    <h:message for="f:w"/>
    <h:outputText value="Patron code:"/>
    <h:inputText value="#{r.patronCode}">
        <x:validatePatron/>
    </h:inputText>
    <h:outputText value=""/>
    <h:commandButton action="ok" value="OK"/>
</h:panelGrid>
</h:form>

```

However, by default JSF will generate client IDs any way it likes. To guarantee that it will be `f:w`, you need to modify the code as shown in Listing 3-20.

Listing 3-20. Specifying Client IDs

```

...
<h:messages errorClass="c1"/>
<h:form id="f">
    <h:panelGrid columns="2">
        <h:outputText value="Weight:"/>
        <h:inputText id="w" value="#{r.weight}" label="weight" required="true"
            validatorMessage="weight cannot be negative!">
            <f:validateLongRange minimum="0"/>
        </h:inputText>
        <h:message for="f:w"/>
        <h:outputText value="Patron code:"/>
        <h:inputText value="#{r.patronCode}">
            <x:validatePatron/>
        </h:inputText>
        <h:outputText value=""/>
        <h:commandButton action="ok" value="OK"/>
    </h:panelGrid>
</h:form>

```

Run the application, and it should basically work, but the layout will be incorrect (see Figure 3-20).

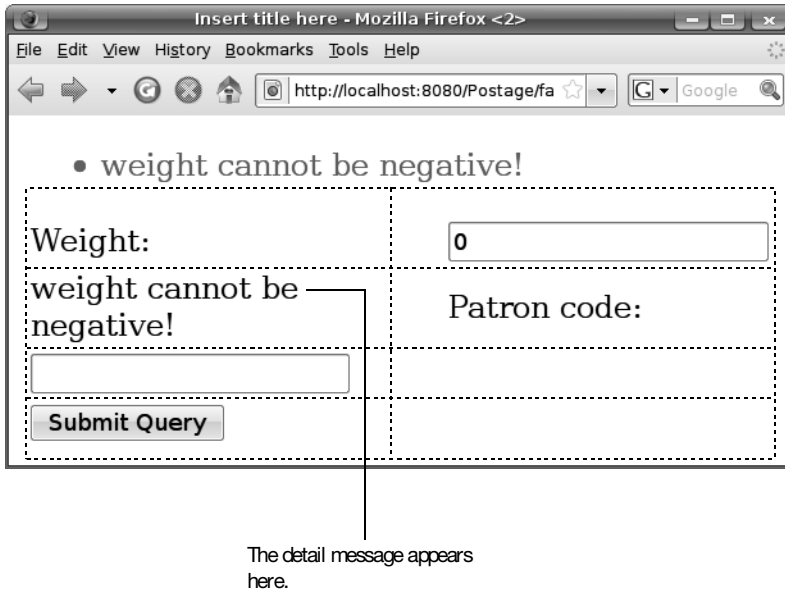


Figure 3-20. Detail message displayed but layout is incorrect

This is because `<h:panelGrid>` lays out the child components sequentially (in two columns, as specified) and the UI Message is also a child component (see Figure 3-21).

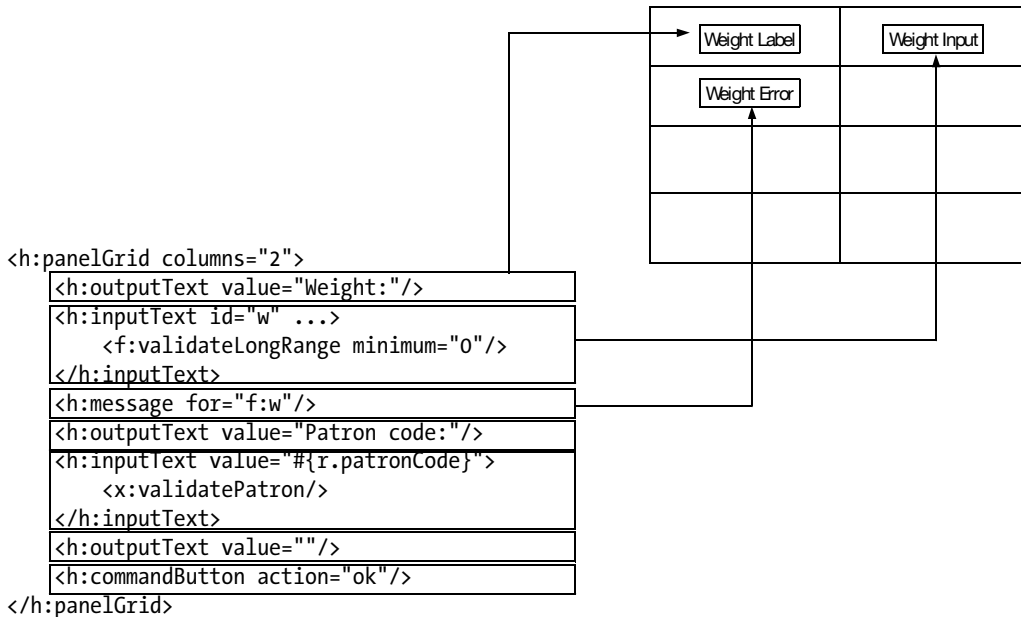


Figure 3-21. The UI Message component is also a child component.

To solve this problem, you can use another component to group the UI Input component and the UI Message component together (see Figure 3-22).

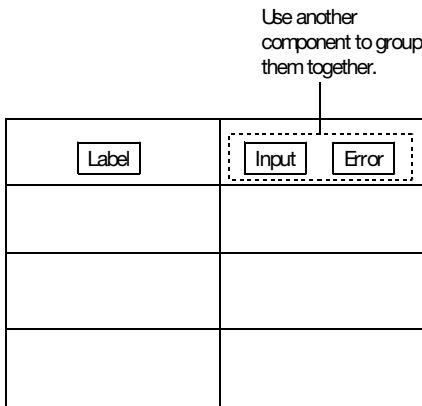


Figure 3-22. Grouping multiple components into one

How do you do that? Could you use an `<h:panelGrid>` tag to group them together? In that case, it will create a UI Panel with a renderer that renders the children in an HTML table (see Figure 3-23). Let's call this renderer the HTML grid renderer. However, for the current case, you don't really need to arrange them in an HTML table; all you need is to arrange them one by one sequentially without adding any extra markup for the UI Panel (see Figure 3-23 again). To do that, you can give the UI Panel a so-called group renderer.

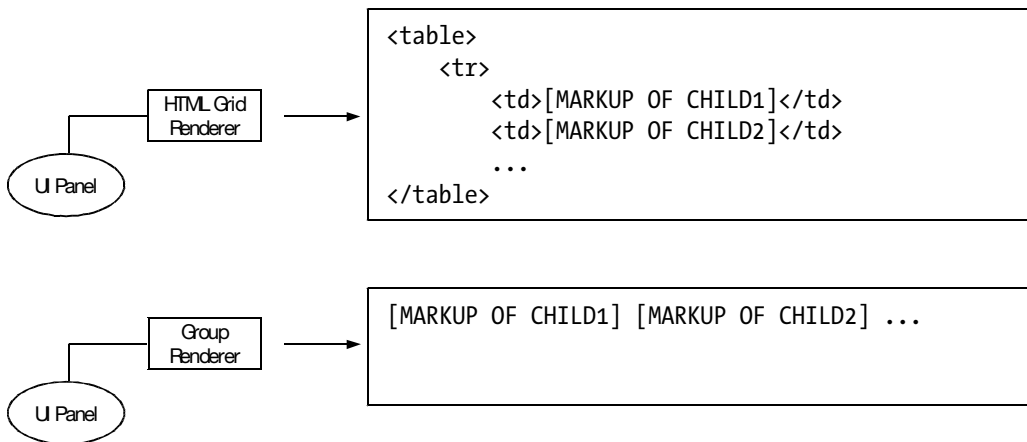


Figure 3-23. Group renderer vs. grid renderer

To create a UI Panel and give it a group renderer, you can use the `<h:panelGroup>` tag as shown in Listing 3-21.

Listing 3-21. *Using the <h:panelGroup> Tag*

```

...
<h:messages errorClass="c1"/>
<h:form id="f">
    <h:panelGrid columns="2">
        <h:outputText value="Weight:"/>
        <h:panelGroup>
            <h:inputText id="w" value="#{r.weight}" label="weight" required="true"
                validatorMessage="weight cannot be negative!"/>
            <f:validateLongRange minimum="0"/>
        </h:panelGroup>
        </h:inputText>
        <h:message for="f:w"/>
    </h:panelGrid>
    <h:outputText value="Patron code:"/>
    <h:inputText value="#{r.patronCode}">
        <x:validatePatron/>
    </h:inputText>
    <h:outputText value=""/>
    <h:commandButton action="ok" value="OK"/>
</h:form>

```

Then the page will look fine (see Figure 3-24).



Figure 3-24. *Input field and error message sticking together*

You may have noticed that the detail message is the same as the summary message. This is because you're setting the error message using the `validatorMessage` attribute (see Listing 3-22). This will set both the summary and detail messages.

Listing 3-22. *Using `validatorMessage` Will Set Both the Summary and Detail Messages*

```

...
<h:form id="f">
    <h:panelGrid columns="2">
        <h:outputText value="Weight:"/>
        <h:panelGroup>
            <h:inputText id="w" value="#{r.weight}" label="weight" required="true"
                validatorMessage="weight cannot be negative! ">
            <f:validateLongRange minimum="0"/>
        </h:panelGroup>
    </h:panelGrid>
</h:form>
...

```

If you were still using the message bundle, you could provide the detail message (see Listing 3-23). That is, just add the string `_detail` to the resource key. In addition, if the value is too long, you can type a backslash and continue on the next line.

Listing 3-23. *Providing a Detail Message in a Message Bundle*

```

javax.faces.validator.LongRangeValidator.MINIMUM={1} must be at least {0}!
javax.faces.validator.LongRangeValidator.MINIMUM_detail={1} is invalid. It must \
be at least {0}!

```

Run the application again, and it should work (see Figure 3-25).



Figure 3-25. *Detail message displayed*

For the `<h:message>` tag, you are specifying the client ID of `f:w`. In fact, you can specify a so-called relative client ID instead (see Figure 3-26).

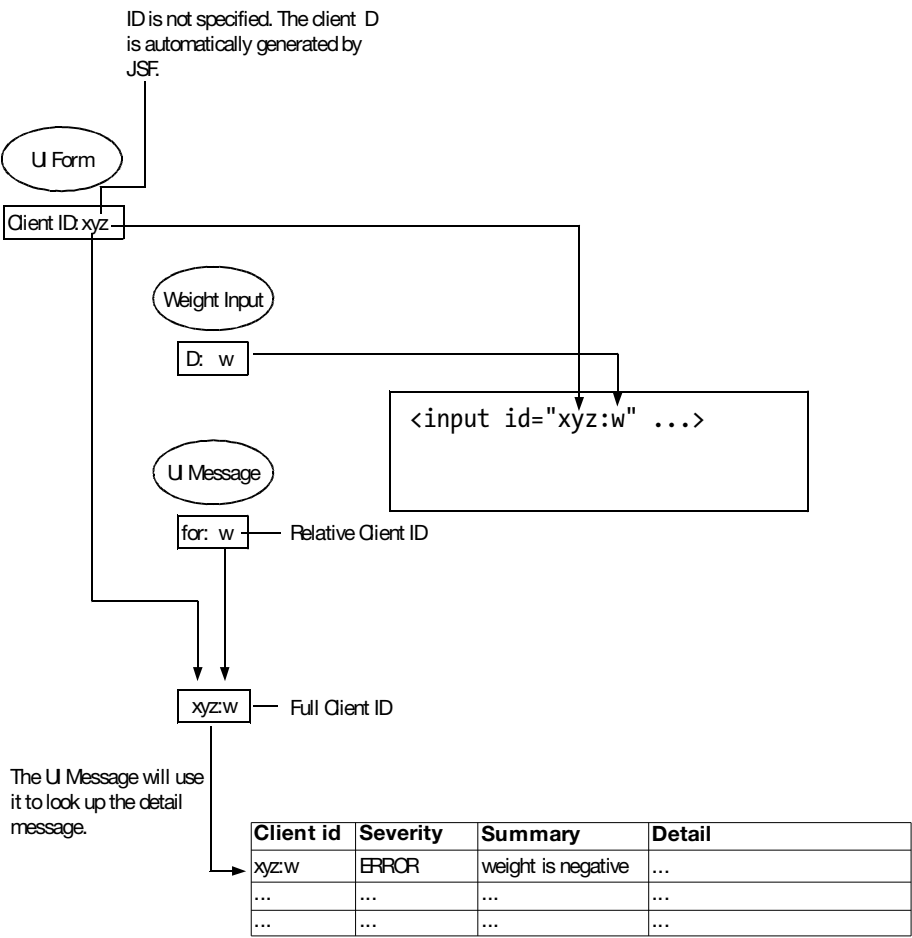


Figure 3-26. Using a relative client ID

Therefore, you can simplify the code a little bit by deleting the ID for the `<h:form>` tag and using a relative ID for the `<h:message>` tag (see Listing 3-24).

Listing 3-24. Using Relative Client ID in `getrequest.xhtml`

```
...
<h:messages errorClass="c1"/>
<h:form>
  <h:panelGrid columns="2">
```

```

    <h:outputText value="Weight:"/>
    <h:panelGroup>
        <h:inputText id="w" value="#{r.weight}" label="weight" required="true"
            validatorMessage="weight cannot be negative!">
            <f:validateLongRange minimum="0"/>
        </h:inputText>
        <h:message for="w"/>
    </h:panelGroup>
    ...
</h:panelGrid>
</h:form>

```

Run the application, and it will continue to work. Finally, you can make the detail message appearing in red as shown in Listing 3-25.

Listing 3-25. *Specifying CSS Class for UI Message*

```

...
<style type="text/css">
    li.c1 { color: red }
    span.c1 { color: red }
</style>
...
<h:messages errorClass="c1"/>
<h:form>
    <h:panelGrid columns="2">
        <h:outputText value="Weight:"/>
        <h:panelGroup>
            <h:inputText id="w" value="#{r.weight}" label="weight" required="true"
                validatorMessage="weight cannot be negative!">
                <f:validateLongRange minimum="0"/>
            </h:inputText>
            <h:message for="w" errorClass="c1"/>
        </h:panelGroup>
        ...
    </h:panelGrid>
</h:form>

```

Validating a Combination of Multiple Input Values

Suppose that for a particular patron *p1*, you will never ship a package that is weighted more than 50 kilograms. Because this involves both the weight and the patron code (two components), you can't make a validator and assign it to a single component. One way to do it is to perform the checking in an action method. To do that, modify `getRequest.xhtml` as shown in Listing 3-26.

Listing 3-26. *Invoking an Action Method for Validation*

```
...
<h:form>
    <h:panelGrid columns="2">
        <h:outputText value="Weight:" />
        <h:panelGroup>
            ...
        </h:panelGroup>
        <h:outputText value="Patron code:" />
        <h:inputText value="#{r.patronCode}">
            <x:validatePatron />
        </h:inputText>
        <h:outputText value="" />
        <h:commandButton action="#{r.onOK}" value="OK" />
    </h:panelGrid>
</h:form>
</body>
</html>
```

Implement the `onOK()` method as shown in Figure 3-27. In short, it performs the checking, and if it fails, it will log an error message and return null as the outcome so that the current page is redisplayed.

```

public class Request {
    private int weight;
    private String patronCode;
    ...
    public String onOK() {
        if (!isValid()) {
            FacesContext context = FacesContext.getCurrentInstance();
            context.addMessage("f:w", new FacesMessage(
                FacesMessage.SEVERITY_ERROR,
                "weight too heavy for the patron", null));
            return null;
        }
        return "ok";
    }
    public boolean isValid() {
        if (patronCode.equals("p1") && weight > 50) {
            return false;
        }
        return true;
    }
}

```

Validate the request. This method is defined below.

Record the error for this component. You'll have to set the client ID of the UI Form explicitly.

Re-display the current page.

If you specify null as the detail message, it will be treated as equal to the summary.

Figure 3-27. The `onOK()` method

Explicitly set the client ID of the UI Form component as shown in Listing 3-27.

Listing 3-27. *Setting the Client ID of UI Form*

```

...
<h:messages errorClass="c1"/>
<h:form id="f">
    <h:panelGrid columns="2">
        <h:outputText value="Weight:"/>
        <h:panelGroup>
            <h:inputText id="w" value="#{r.weight}" label="weight" required="true"
                validatorMessage="weight cannot be negative!">
                <f:validateLongRange minimum="0"/>
            </h:inputText>
            <h:message for="w"/>
        </h:panelGroup>
    </h:panelGrid>
</h:form>

```

Run the app, and it should work. However, if the `Request` class is intended to be independent of the UI technology so that it can be reused in different types of UIs, then it is now a problem because it is referring to JSF-specific classes such as `FacesContext` and `FacesMessage`. If this bothers you, you can move this UI-specific code into a so-called action listener to perform the validation (see Listing 3-28). Here you're specifying that the Java class of the action listener is `postage.RequestValidatingListener`. You'll create this class next. JSF will call all action listeners before calling the action method (if any) in the Invoke Application phase. Now you don't need the `onOK()` action method anymore, so go ahead and delete it.

Listing 3-28. *Using an Action Listener*

```
...
<h:form>
    <h:panelGrid columns="2">
        <h:outputText value="Weight:" />
        <h:panelGroup>
            ...
        </h:panelGroup>
        <h:outputText value="Patron code:" />
        <h:inputText value="#{r.patronCode}">
            <x:validatePatron />
        </h:inputText>
        <h:outputText value="" />
        <h:commandButton action="ok" value="OK">
            <f:actionListener type="postage.RequestValidatingListener"/>
        </h:commandButton>
    </h:panelGrid>
</h:form>
</body>
</html>
```

Create this `RequestValidatingListener` class in the `postage` package. Listing 3-29 shows the content. It has to implement the `ActionListener` interface provided by JSF and provide a `processAction()` method. In that method it acquires the current request by evaluating an EL expression. If the request is invalid, it tells JSF to stop any further processing on this event by throwing an `AbortProcessingException`. In that case, all further action listeners (if any) and the action method will be skipped, and thus the outcome will not be set.

Listing 3-29. *The RequestValidatingListener Class*

```
public class RequestValidatingListener implements ActionListener {
    public void processAction(ActionEvent event)
        throws AbortProcessingException {
        FacesContext context = FacesContext.getCurrentInstance();
        Application app = context.getApplication();
        Request req = (Request) app.evaluateExpressionGet(context, "#{r}",
            Request.class);
        if (!req.isValid()) {
            context.addMessage("f:w", new FacesMessage(
                FacesMessage.SEVERITY_ERROR,
                "weight too heavy for the patron",
                null));
            throw new AbortProcessingException();
        }
    }
}
```

Note that the properties of the Request object will have been updated in the Update Domain Values phase while the action listener is executed in the Invoke Application phase. Now, run the application, and it should continue to work (see Figure 3-28).

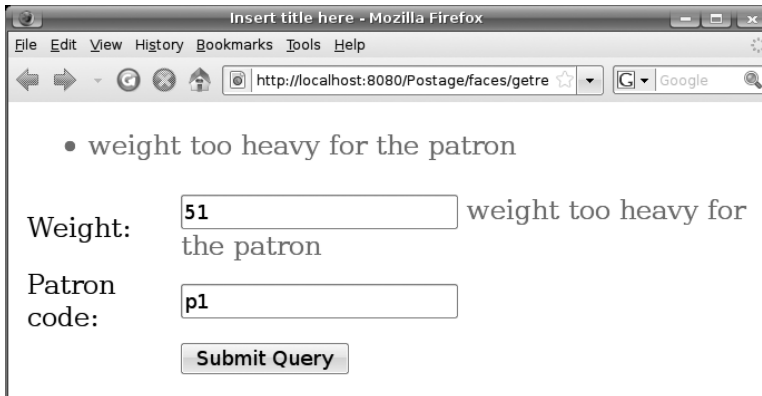


Figure 3-28. *The whole Request object is validated.*

Summary

In this chapter, you learned how to validate user input. To validate the user input in a single UI Input component, you can add one or more validators to it. They will be invoked one by one in the Process Validations phase to check the converted value. If any one fails, it will log an error for that component (or rather, for its client ID) and tell the JSF engine to jump to the Render Response phase directly.

As you learned, JSF provides a few predefined validators for checking the range of a long, the range of a double, or the length of a string. To customize their error messages, use a message bundle or provide the message directly in the validator tag.

To perform custom validation, you learned that you can create a custom validator by providing a Java class and a validator ID. In addition, you need to define a Facelet tag that will create the validator. If the validation involves two or more components, you can add an action listener to a UI Command. It will be executed in the Invoke Application phase. As such, the beans will have been updated so you can check their properties.

I also covered how to specify a JSF message. Specifically, a JSF message contains a severity level, a summary, and a detail. Usually you will display the summary using a UI Message component and display the detail using a UI Message component along with each UI Input component.

To customize the appearance of the HTML output, you can define CSS style classes and let the components refer to them.

A UI Panel component lays out its child components according to its renderer. It can lay them out in a table (<panelGrid>) or just arrange them one by one (<panelGroup>).

Finally, you learned that if a web bean is never looked up by name, you don't need to use @Named on it.



Creating an E-shop

In this chapter, you'll learn how to create an e-shop. This involves displaying a list of products (using a loop), implementing a shopping cart for each user, supporting user login and logout, and requiring authenticated access for the checkout page.

Suppose that you'd like to create the e-shop as shown in Figure 4-1. Initially, the page lists all the products. Clicking a product link will display the detail page for the product.

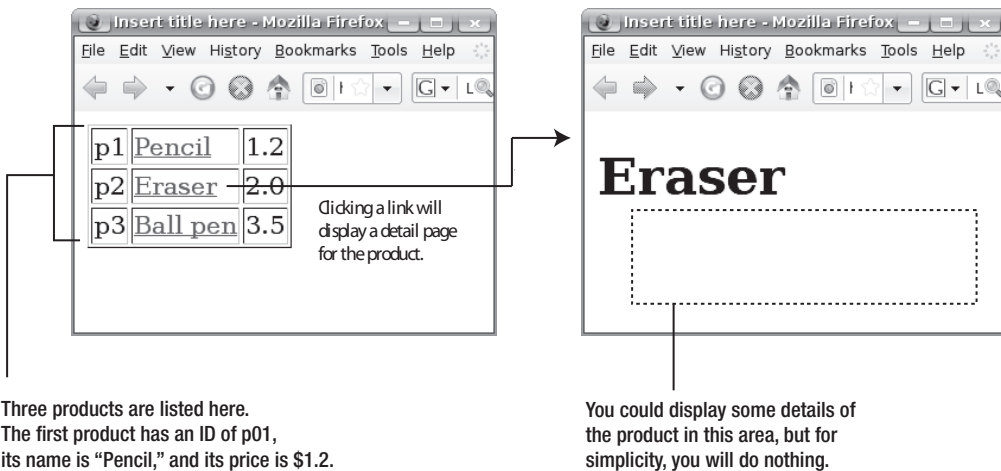


Figure 4-1. *Your e-shop*

Listing the Products

OK, let's do it. Create a new dynamic web project named Shop. Then create a catalog.xhtml page. How will you list the products? The problem here is that supposedly the products are loaded from a database; thus, you don't know the number of products in advance, so you can't lay them out in advance using `<h:panelGrid>`, as shown in Listing 4-1.

Listing 4-1. *You Don't Know How Many Products to Lay Out*

```
<h:panelGrid...>
  PRODUCT 1
  PRODUCT 2
  PRODUCT 3 -> But how do you know there are only three products?
</h:panelGrid>
```

To perform a loop at runtime, you can use `<h:dataTable>`, as shown in Figure 4-2. In short, it will loop over each element of the list. For each element, it will render all the columns specified inside. Figure 4-3 shows the component tree that will be created.

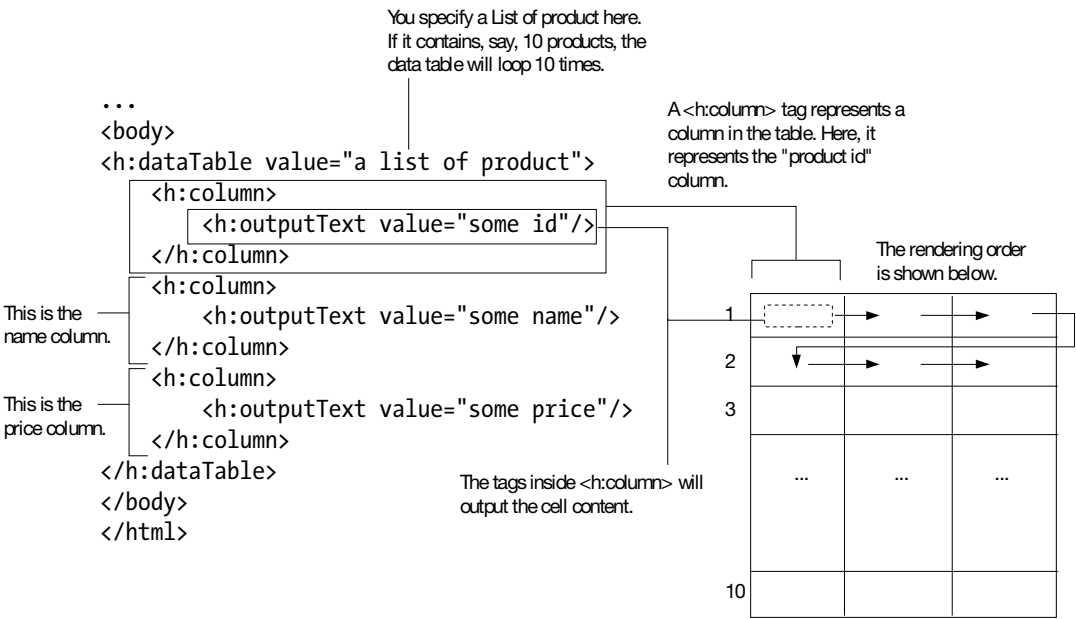


Figure 4-2. *Looping with `<h:dataTable>`*

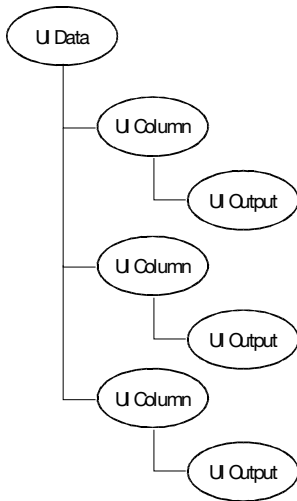


Figure 4-3. Component tree created by `<h:dataTable>` and `<h:column>`

To implement this idea, you need to provide a `List` of products using a web bean. So, create a `Catalog` class in the `shop` package, as shown in Listing 4-2. Because the catalog is a global thing, use the application scope. In addition, instead of loading the products from a database, for simplicity you'll simply hard-code them into a `List`.

Listing 4-2. *The Catalog Class*

```

package shop;
...

@Named("catalog")
@ApplicationScoped
public class Catalog {
    private List<Product> products;

    public Catalog() {
        products = new ArrayList<Product>();
        products.add(new Product("p1", "Pencil", 1.20));
        products.add(new Product("p2", "Eraser", 2.00));
        products.add(new Product("p3", "Ball pen", 3.50));
    }
    public List<Product> getProducts() {
        return products;
    }
}

```

Define the `Product` class in the same package (Listing 4-3).

Listing 4-3. *The Product Class*

```
package shop;

public class Product {
    private String id;
    private String name;
    private double price;

    public Product(String id, String name, double price) {
        this.id = id;
        this.name = name;
        this.price = price;
    }
    public String getId() {
        return id;
    }
    public String getName() {
        return name;
    }
    public double getPrice() {
        return price;
    }
}
```

Provide the `List` to the `dataTable` (Listing 4-4).

Listing 4-4. *Providing the List to the dataTable*

```
...
<h:dataTable value="#{catalog.products}">
    <h:column>
        <h:outputText value="some id"/>
    </h:column>
    <h:column>
        <h:outputText value="some name"/>
    </h:column>
    <h:column>
        <h:outputText value="some price"/>
    </h:column>
</h:dataTable>
```

Now, the UI Data component will render the `Product` objects one by one (one row for each `Product`). However, how can the UI Output component in the columns access the information in the current `Product` object? You can do that with the `var` attribute, as shown in Listing 4-5. In short, the UI Data will use `p` as a looping variable to point to each element in turn. This variable is implemented as an attribute (a name-value pair) in the request.

So, you can finally access the attribute just like a web bean (Listing 4-5).

Listing 4-5. *Accessing an Attribute Like a Web Bean*

```
...
<h:dataTable value="#{catalog.products}" var="p">
  <h:column>
    <h:outputText value="#{p.id}"/>
  </h:column>
  <h:column>
    <h:outputText value="#{p.name}"/>
  </h:column>
  <h:column>
    <h:outputText value="#{p.price}"/>
  </h:column>
</h:dataTable>
```

Note that an attribute is *not* a web bean. When evaluating an EL expression, JSF will try to look up an attribute first, before looking up a web bean. Now when you run the application, the products will be displayed (see Figure 4-4).

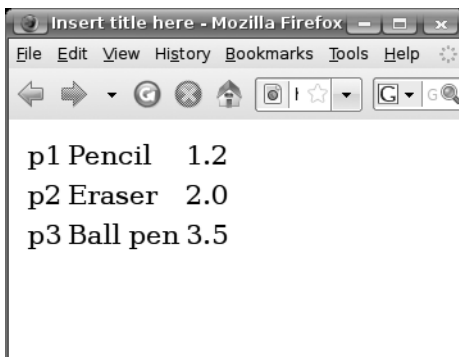


Figure 4-4. *Products are displayed.*

To make the grid visible, modify the page as shown in Listing 4-6.

Listing 4-6. *Setting the Width of the Border*

```
...
<h:dataTable value="#{catalog.products}" var="p" border="1">
  <h:column>
    <h:outputText value="#{p.id}"/>
  </h:column>
  <h:column>
    <h:outputText value="#{p.name}"/>
  </h:column>
  <h:column>
    <h:outputText value="#{p.price}"/>
  </h:column>
</h:dataTable>
```

When you run the application now, the table border will now display.

Making the Link to Show the Details

Now, let's create the link to show the product details. For this, you'll use the

`<h:commandLink>` tag, as shown in Figure 4-5.

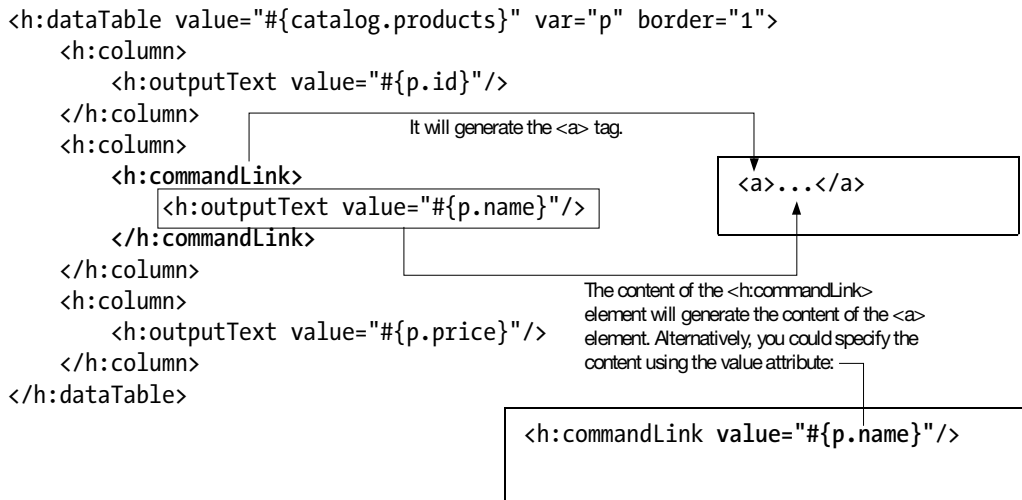


Figure 4-5. *Using `<h:commandLink>`*

Note that the behavior of the `<h:commandLink>` tag is the same as the `<h:commandButton>` tag because they both will create a UI Command component. The only difference is that the former will render the UI Command as a link, while the latter will render a button (see Figure 4-6). An important consequence of this is that the `<h:commandLink>` tag will submit a form just like `<h:commandButton>`, and thus it must appear inside a form.

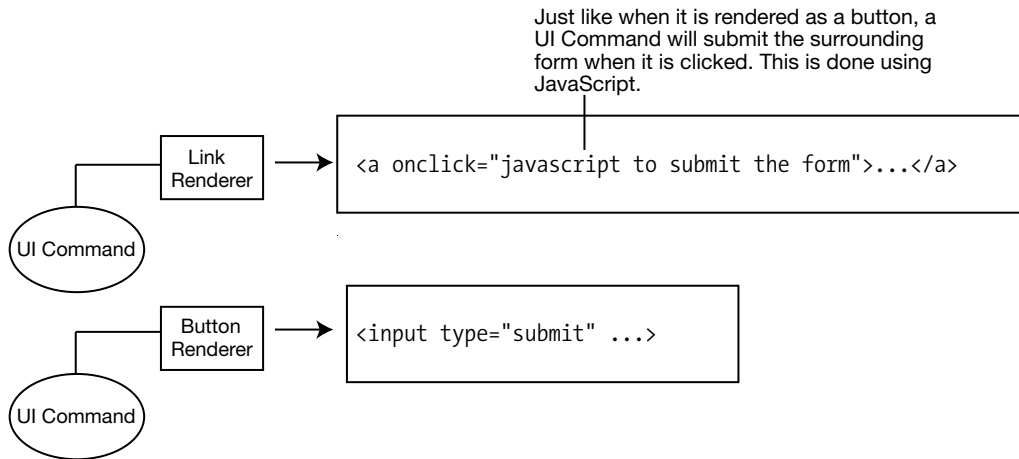


Figure 4-6. `<h:commandLink>` vs. `<h:commandButton>`

Therefore, you need to modify the code as shown in Listing 4-7.

If you were using `<h:commandButton>`, you would set the outcome using the `action` attribute. Because `<h:commandLink>` has exactly the same behavior, you do the same thing.

Listing 4-7. *Setting the Outcome of `<h:commandLink>`*

```

...
<h:dataTable value="#{catalog.products}" var="p" border="1">
  <h:column>
    <h:outputText value="#{p.id}"/>
  </h:column>
  <h:column>
    <h:form>
      <h:commandLink action="detail">
        <h:outputText value="#{p.name}"/>
      </h:commandLink>
    </h:form>
  </h:column>
</h:dataTable>

```

```

        <h:outputText value="#{p.price}"/>
    </h:column>
</h:dataTable>

```

For it to work, create a `detail.xhtml` page as shown in Listing 4-8. Note that for simplicity it contains static content only at the moment.

Listing 4-8. *Detail Page*

```

<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://java.sun.com/jsf/html">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
<title>Insert title here</title>
</head>
<body>
This is the detail
</body>
</html>

```

Create a navigation case in `faces-config.xml`, as shown in Listing 4-9.

Listing 4-9. *Navigation Case to Show the Detail Page*

```

<faces-config ...>
    <application>
        <view-handler>com.sun.facelets.FaceletViewHandler
        </view-handler>
    </application>
    <navigation-rule>
        <from-view-id>/catalog.xhtml</from-view-id>
        <navigation-case>
            <from-outcome>detail</from-outcome>
            <to-view-id>/detail.xhtml</to-view-id>
        </navigation-case>
    </navigation-rule>
</faces-config>

```

When you run the application now, the detail link should be working (see Figure 4-7).

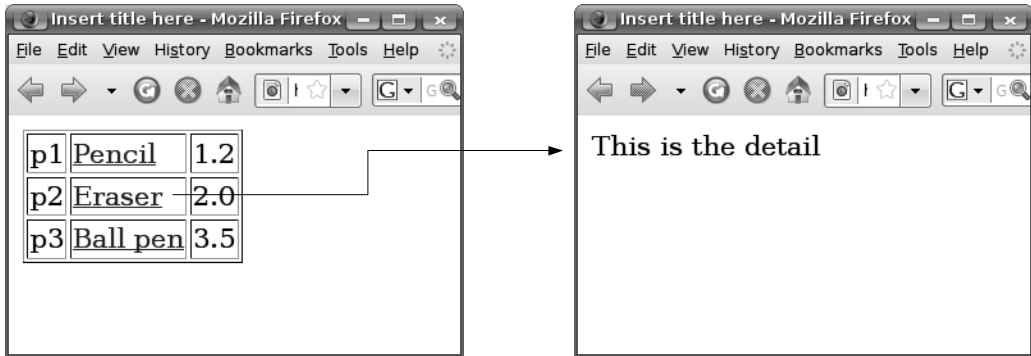


Figure 4-7. *Detail link working*

The next question is, how do you get access to the selected product in the detail page? As a first step, let's create an action listener to print the ID of the selected product to the console (we covered action listeners in Chapter 3). To do that, modify `catalog.xhtml` as shown in Listing 4-10.

Listing 4-10. *Using an Action Listener to Handle the Click*

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core">
...
<h:dataTable value="#{catalog.products}" var="p" border="1">
  <h:column>
    <h:outputText value="#{p.id}"/>
  </h:column>
  <h:column>
    <h:form>
      <h:commandLink action="detail">
        <f:actionListener type="shop.OnDetailActionListener"/>
        <h:outputText value="#{p.name}"/>
      </h:commandLink>
    </h:form>
  </h:column>
  <h:column>
    <h:outputText value="#{p.price}"/>
  </h:column>
</h:dataTable>
```

How does the action listener get access to the selected product? When the UI Data loops through the rows (see Figure 4-8), conceptually it will create an environment object surrounding the UI Command created by `<h:commandLink>`. The environment object contains the current row index. When the UI Command needs to output its client ID, it will send its client ID through that environment, which will append the row index to the client ID for display on the page.

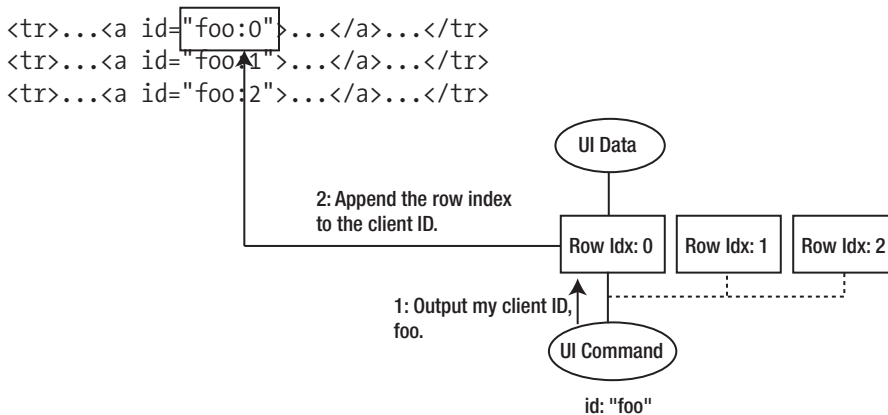


Figure 4-8. Row-specific environment appending row index to client ID

When the user clicks a link (see Figure 4-9), in the Apply Request Values phase the UI Data will loop through each row again. For each row, it will again create an environment object surrounding the UI Command. When the UI Command checks whether it was clicked, it sends its client ID to the environment object. That environment object again will append the row index to get the final client ID before checking it against the client ID in the request. If it is matched, the UI Command will schedule the execution of the action listener through the environment object.

The environment object will attach to the action listener waiting for execution. When it is executed, the environment object will first set the `p` attribute in request scope according to its row ID, before calling the action listener. Finally, to clean up, it will remove the `p` attribute.

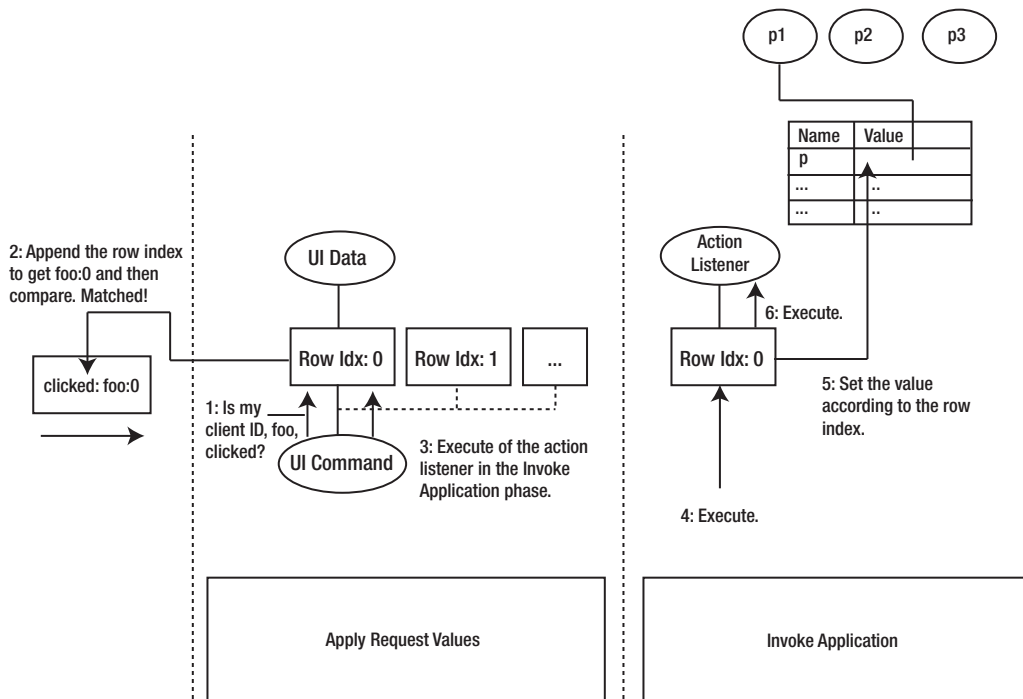


Figure 4-9. How UI Data handles form submissions

Therefore, in your action listener, you can readily access the selected Product object from the `p` attribute. So, create the `OnDetailActionListener` class in the `shop` package (see Listing 4-11).

Listing 4-11. Accessing the Current Product in Your Action Listener

```
package shop;
...
public class OnDetailActionListener implements ActionListener {
    @Override
    public void processAction(ActionEvent ev) throws AbortProcessingException {
        FacesContext context = FacesContext.getCurrentInstance();
        Application app = context.getApplication();
        Product p = (Product) app.evaluateExpressionGet(context, "#{p}",
            Product.class);
        System.out.println(p.getId());
    }
}
```

When you run the application now, it should print the product ID to the console. Now, the next step is to display the detail of the `Product` in the detail page. Can the detail page find the `Product` in the `p` attribute? No, it can't, because the attribute will have been removed. To solve this problem, you can use your action listener to store the `Product` object into a web bean. To do that, create a `ProductHolder` class in the `shop` package, as shown in Listing 4-12.

Listing 4-12. *Using a Web Bean to Hold the Current Product Object*

```
package shop;
...
@Named("ph")
@RequestScoped
public class ProductHolder {
    private Product currentProduct;

    public Product getCurrentProduct() {
        return currentProduct;
    }
    public void setCurrentProduct(Product currentProduct) {
        this.currentProduct = currentProduct;
    }
}
```

Then modify your action listener as shown in Listing 4-13.

Listing 4-13. *Using a Web Bean to Hold the Current Product Object*

```
public class OnDetailActionListener implements ActionListener {
    @Override
    public void processAction(ActionEvent ev) throws AbortProcessingException {
        FacesContext context = FacesContext.getCurrentInstance();
        Application app = context.getApplication();
        Product p = (Product) app.evaluateExpressionGet(context, "#{p}",
            Product.class);
    }
}
```

```

        ProductHolder ph = (ProductHolder) app.evaluateExpressionGet(
            context,
            "#{ph}",
            ProductHolder.class);
        ph.setCurrentProduct(p);
    }
}

```

Alternatively, recall that an EL expression not only can be queried to get the value but also can be used to set the value. Therefore, you could modify the code as shown in Figure 4-10.

```

public class OnDetailActionListener implements ActionListener {

    @Override
    public void processAction(ActionEvent ev) throws AbortProcessingException {
        FacesContext context = FacesContext.getCurrentInstance();
        Application app = context.getApplication();
        Product p = (Product) app.evaluateExpressionGet(context, "#{p}",
            Product.class);
        ProductHolder ph = (ProductHolder) app.evaluateExpressionGet(context,
            "#{ph}",
            ProductHolder.class);
        ph.setCurrentProduct(p);
        ELContext elContext = context.getELContext();
        ValueExpression ve = app.getExpressionFactory().createValueExpression(
            elContext, "#{ph.currentProduct}", Product.class);
        ve.setValue(elContext, p);
    }
}

```

The expression factory can create EL expression objects.

The EL context provides the variable bindings and etc.

Store the value of p into the EL expression (ph.currentProduct).

Given this string-form EL expression, create an EL expression object.

The value of the EL expression should be Product. This is used for possible type conversion.

Figure 4-10. Taking advantage of the modifiability of EL expressions

Modify the detail.xhtml page to display the product name as shown in Listing 4-14.

Listing 4-14. *Displaying the Product Name*

```

<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
<title>Insert title here</title>
</head>
<body>
<h1>#{ph.currentProduct.name}</h1>
</body>
</html>

```

When you run the application and click the link for a certain product, its details should be displayed (see Figure 4-11).

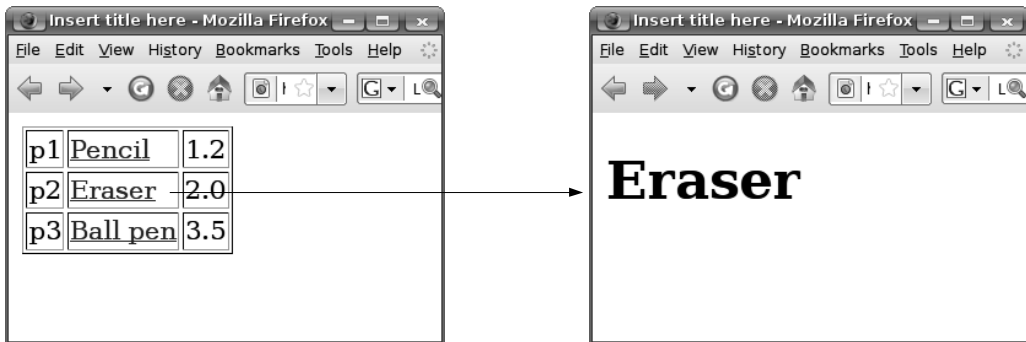


Figure 4-11. *Detail page displaying the correct product name*

Because it is very common to store the value of one EL expression into another in an action listener, JSF provides a built-in action listener to do that. To use it, modify `catalog.xhtml`, as shown in Listing 4-15.

Listing 4-15. *Displaying the Product Name*

```

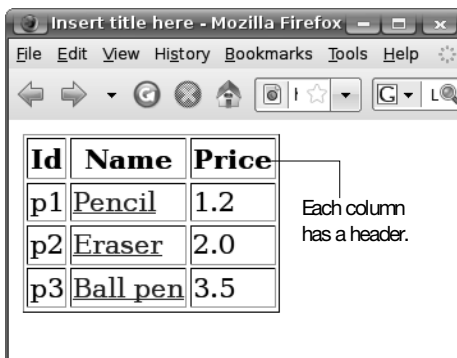
<h:dataTable value="#{catalog.products}" var="p" border="1">
  <h:column>
    <h:outputText value="#{p.id}"/>
  </h:column>
  <h:column>
    <h:form>
      <h:commandLink action="detail">
        <f:setPropertyActionListener
          value="#{p}" target="#{ph.currentProduct}"/>
        <h:outputText value="#{p.name}"/>
      </h:commandLink>
    </h:form>
  </h:column>
  <h:column>
    <h:outputText value="#{p.price}"/>
  </h:column>
</h:dataTable>

```

Delete the `OnDetailActionListener` class. Run the application now, and it will continue to work.

Displaying Headers in the Columns

Next, you'd like to display headers as shown in Figure 4-12.



Id	Name	Price
p1	Pencil	1.2
p2	Eraser	2.0
p3	Ball pen	3.5

Each column has a header.

Figure 4-12. *Column headers*

To do that, modify `catalog.xhtml` as shown in Figure 4-13.

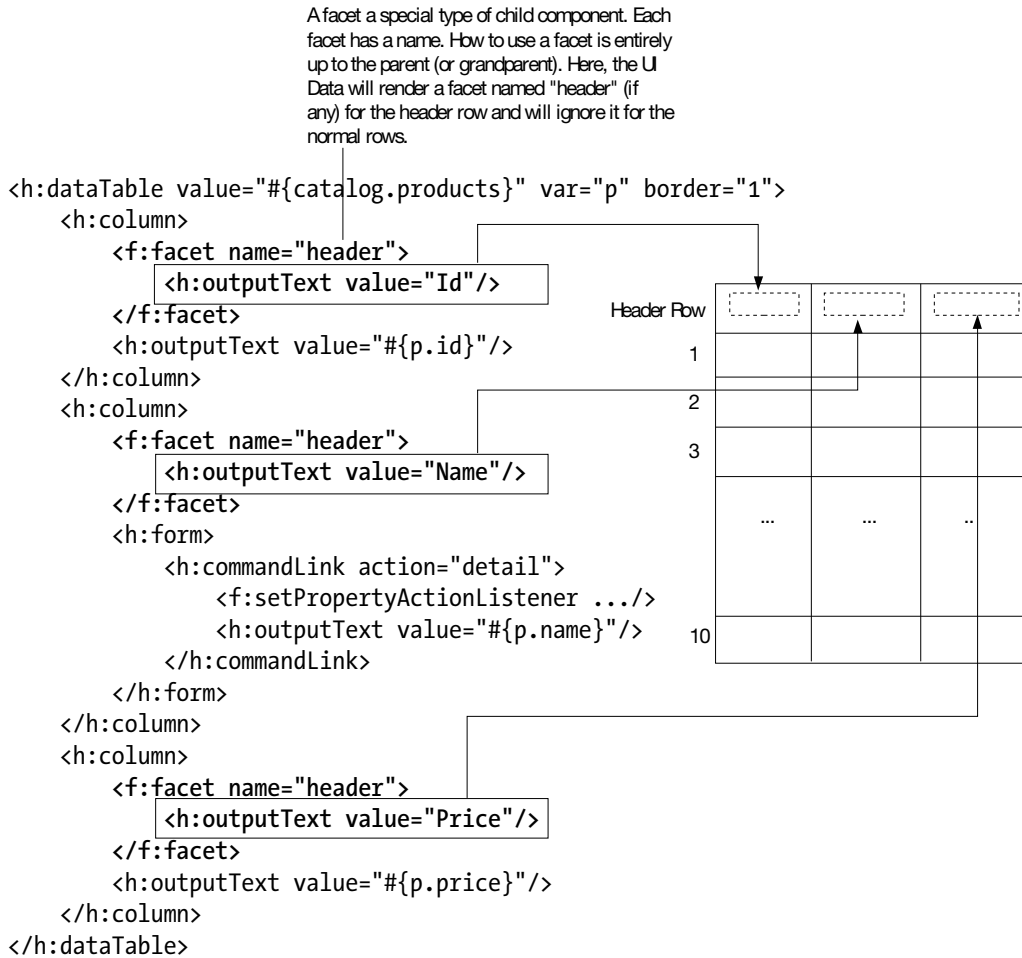


Figure 4-13. *Using header facets*

When you run the application now, it will display the headers.

Implementing a Shopping Cart

Now, let's allow the user to add products to the shopping cart (see Figure 4-14).

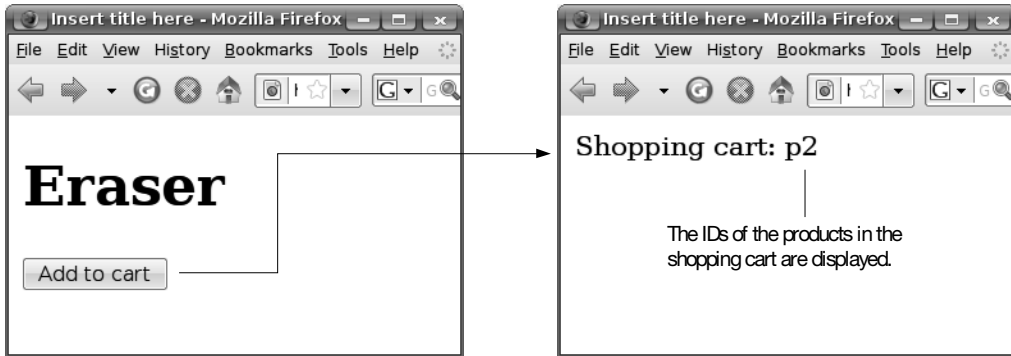


Figure 4-14. Adding a product to the shopping cart

To do it, modify `detail.xhtml` as shown in Figure 4-15.

```
...
<body>
<h1>#{ph.currentProduct.name}</h1>
<h:form>
    <h:commandButton
        value="Add to cart"
        action="#{ph.addToCart}"/>
</h:form>
</body>
</html>
```

Usually it is just the outcome, but here you specify a method. It should take no argument and return a string which is the outcome.

```
public class ProductHolder {
    ...
    public String addToCart() {
        ...
    }
}
```

Figure 4-15. Using a business method as the action

Define this `addToCart()` method in the `ProductHolder` class as shown in Listing 4-16. In the code, you first inject the shopping cart by marking it with the `@Current` annotation (see the “Dependency Injection” sidebar for details). The `@Current` annotation is the default binding type for web beans (you must specify a binding type for field injection, even if it is the default binding type). A combination of the binding type and the object’s type is used to select the web bean to inject.

In the `addToCart()` method, you simply print out the product ID to verify that it is working. Then you add the product ID to the shopping cart. Finally, you return the string “added” as the outcome.

Listing 4-16. *The `addToCart()` Method*

```
@Named("ph")
@RequestScoped
public class ProductHolder {
    private Product currentProduct;
    @Current
    private Cart cart;
    ...
    public String addToCart() {
        System.out.println("Adding "+currentProduct.getId());
        cart.add(currentProduct.getId());
        return "added";
    }
}
```

You’ll implement the `Cart` class next.

DEPENDENCY INJECTION

Java applications consist of Java interfaces and classes, which make up the application components that interact with each other to accomplish the application’s job. These component objects depend on each other, so an object is *dependent* if it uses other objects to do its job. It then follows that the other objects used by the dependent object are called the object’s *dependencies*.

An object pulls its dependencies if it is responsible for providing its dependencies from its environment. The object may do this by instantiating dependencies or by looking up an outside object for them. Pulling dependencies is the traditional way of using objects in Java.

In contrast, another object could be responsible for providing dependencies and pushing them into the object. This approach is called *dependency injection*, and the dependencies are *injected* by a third party to the dependent object. In the case of web beans, the web bean implementation pushes their dependencies into them, and this is how one web bean gains a reference to another web bean.

You need to have the shopping cart as a web bean. What scope should it be in? If you put it into the request scope, it will be gone when the request is finished. If you put it into the application scope, all users will share the same shopping cart. To solve this problem, you need to understand that whenever a new user starts using your application, the web container will allocate memory for that session. When will a session be deleted? If the user doesn't send any request in a certain period, such as 30 minutes, the web container will delete the session. This timeout can be configured. The session can also be destroyed programmatically.

Because of the per-user nature of sessions, it is the best place to store per-user temporary data, such as the shopping cart. How to do that? Initially there is an empty web bean table in each session. To put a shopping cart into the session, create the `Cart` class as in Listing 4-17. The `@SessionScoped` says that, once created, the `Cart` object should be put into the session. But why does it need to implement `Serializable`? This is because the web container may need to save the content of the session to disk or send it to another computer over the network if you have a cluster. In that case, it needs to convert all the objects in the session into bytes. This requires that all their classes implement `Serializable`.

Listing 4-17. *Putting the Shopping Cart into Session*

```
package shop;
...
@SessionScoped
public class Cart implements Serializable {

}
```

Implement the rest of the `Cart` class as shown in Listing 4-18.

Listing 4-18. *Implementing the Rest of the Cart Class*

```
...
@SessionScoped
public class Cart implements Serializable {
    private List<String> productIds;

    public Cart() {
        productIds = new ArrayList<String>();
    }
    public void add(String pid) {
        productIds.add(pid);
    }
}
```

The `addToCart()` method of the `ProductHolder` class returns the outcome “added.” You need to define a navigation case for it to display the next page in `faces-config.xml` (see Listing 4-19).

Listing 4-19. *Displaying the Next Page After Adding a Product to the Cart*

```
<faces-config ...>
    <application>
        <view-handler>com.sun.facelets.FaceletViewHandler
        </view-handler>
    </application>
    <navigation-rule>
        <from-view-id>/catalog.xhtml</from-view-id>
        <navigation-case>
            <from-outcome>detail</from-outcome>
            <to-view-id>/detail.xhtml</to-view-id>
        </navigation-case>
    </navigation-rule>
    <navigation-rule>
        <from-view-id>/detail.xhtml</from-view-id>
        <navigation-case>
            <from-outcome>added</from-outcome>
            <to-view-id>/cart.xhtml</to-view-id>
        </navigation-case>
    </navigation-rule>
</faces-config>
```

Create the `cart.xhtml` page as shown in Listing 4-20. For simplicity, it contains static content only for the moment.

Listing 4-20. *A Page Displaying the Content of the Shopping Cart*

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
<title>Insert title here</title>
</head>
<body>
Shopping cart
</body>
</html>
```

Run the application. Unfortunately, it will throw a `NullPointerException` at the line highlighted in Listing 4-21.

Listing 4-21. *NullPointerException When Accessing the Current Product*

```
@Named("ph")
@RequestScoped
public class ProductHolder {
    private Product currentProduct;
    @Current
    private Cart cart;
    ...
    public String addToCart() {
        System.out.println("Adding "+currentProduct.getId());
        cart.add(currentProduct.getId());
        return "added";
    }
}
```

Why? The short answer is that it's because the `ph` bean is in the request scope. On form submission it will be gone, and a new one will be created. Thus, the current product will be `null`. For a more detailed explanation, see Figure 4-16.

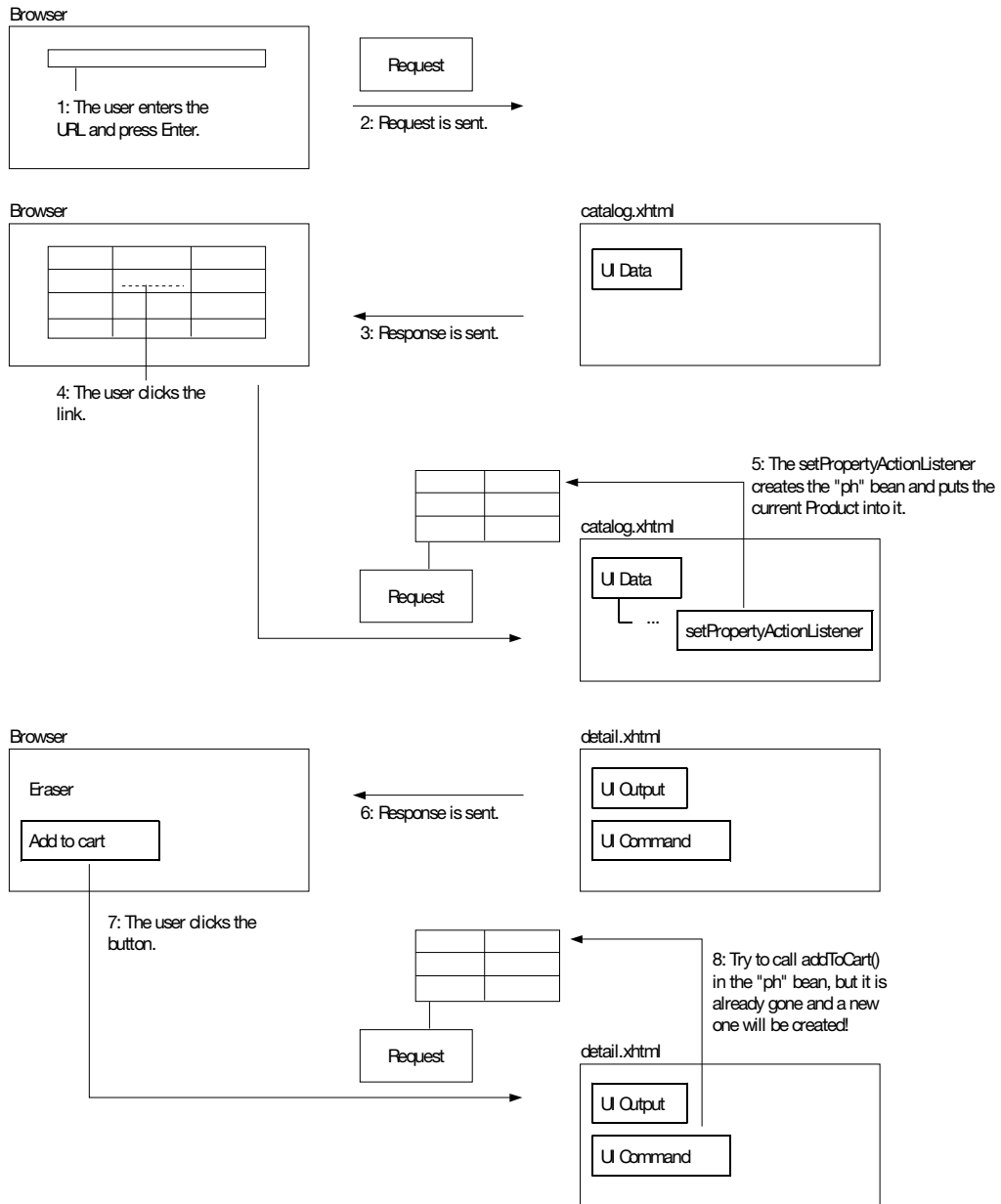


Figure 4-16. *How the ph bean is lost*

To solve this problem, you can store the product ID into the web page (see Figure 4-17).

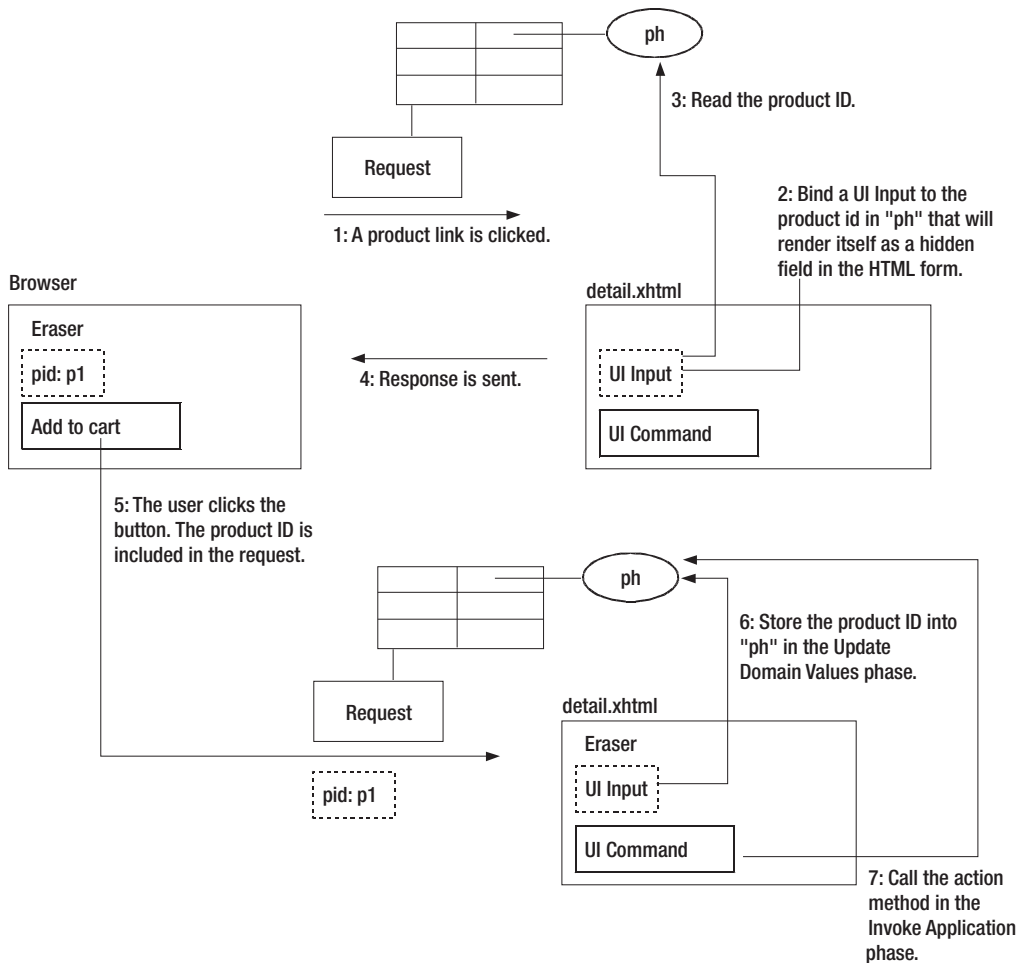


Figure 4-17. Using a hidden UI Input to restore the product ID

To create such a UI Input, use the `<h:inputHidden>` tag. It behaves exactly the same as `<h:inputText>` except that the former will render as an HTML hidden input, while the latter will render as an HTML text input (see Figure 4-18).

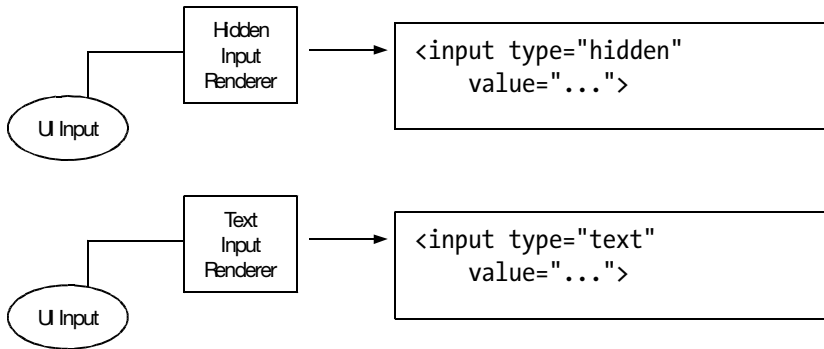


Figure 4-18. `<h:inputHidden>` vs. `<h:inputText>`

To use it, modify `detail.xhtml` as shown in Listing 4-22. It is used just like the `<h:inputText>` tag.

Listing 4-22. *Using the `<h:inputHidden>` Tag*

```

...
<body>
<h1>#{ph.currentProduct.name}</h1>
<h:form>
    <h:inputHidden value="#{ph.productId}"/>
    <h:commandButton
        value="Add to cart"
        action="#{ph.addToCart}"/>
</h:form>
</body>
</html>

```

Modify the `ProductHolder` class to provide this property as shown in Listing 4-23. To find the `Product` object given a product ID, you need to inject the catalog. In addition, in `getProductId()`, why could the current product be `null`? The issue is, before calling the setter on form submission, `UI Input` will try to get the existing value first to see whether the new value is indeed different. So when the getter is called, the current product could indeed be `null`.

Listing 4-23. *Using the `<h:inputHidden>` Tag*

```

...
@Named("ph")
@RequestScoped

```

```

public class ProductHolder {
    private Product currentProduct;
    @Current
    private Cart cart;
    @Current
    private Catalog catalog;
    ...
    public String getProductId() {
        return currentProduct != null ? currentProduct.getId() : null;
    }
    public void setProductId(String pid) {
        currentProduct = catalog.getProduct(pid);
    }
    public String addToCart() {
        System.out.println("Adding "+currentProduct.getId());
        cart.add(currentProduct.getId());
        return "added";
    }
}

```

Define the `getProduct()` method in the `Catalog` class as shown in Listing 4-24.

Listing 4-24. *Providing the `getProduct()` Method in the `Catalog` Class*

```

@Named("catalog")
@ApplicationScoped
public class Catalog {
    private List<Product> products;

    public Catalog() {
        products = new ArrayList<Product>();
        products.add(new Product("p1", "Pencil", 1.20));
        products.add(new Product("p2", "Eraser", 2.00));
        products.add(new Product("p3", "Ball pen", 3.50));
    }
    public List<Product> getProducts() {
        return products;
    }
    public Product getProduct(String pid) {
        for (Product p : products) {
            if (p.getId().equals(pid)) {
                return p;
            }
        }
    }
}

```

```

        }
    }
    return null;
}
}

```

Now run the application, and try to add a product to the shopping cart. It should print the product ID to the console and then display the cart page.

Displaying the Content of the Shopping Cart

How do you display the product IDs on the cart page? Obviously, you need to loop through the product IDs stored in the shopping cart. Can you use `<h:dataTable>`? Unfortunately, it will always output an HTML `<table>`, which is not what's desired here. To loop but without adding its own markup, you can use the `<ui:repeat>` tag. It works almost exactly like the `<h:dataTable>` tag. For example, modify `cart.xhtml` as shown in Listing 4-25. `<ui:repeat>` will simply loop through the list and render its children in each iteration. Because it outputs no markup by itself, it has nothing to do with HTML, so it is not in the JSF HTML tag lib. Note that `<h:outputText>` is outputting a space after the product ID.

Listing 4-25. *Using `<ui:repeat>` to Display the Product IDs*

```

<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:ui="http://java.sun.com/jsf/facelets">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
<title>Insert title here</title>
</head>
<body>
Shopping cart:
<ui:repeat value="#{cart.productIds}" var="pid">
    <h:outputText value="#{pid} " />
</ui:repeat>
</body>
</html>

```

Provide the `getProductIds()` method in the `Cart` class as shown in Listing 4-26. Because you need to refer to it by name, you need to give it a name too.

Listing 4-26. *Providing the `getProductIds()` Method and a Name in the `Cart` Class*

```
...
@Named("cart")
@SessionScoped
public class Cart implements Serializable {
    private List<String> productIds;

    public Cart() {
        productIds = new ArrayList<String>();
    }
    public void add(String pid) {
        productIds.add(pid);
    }

    public List<String> getProductIds() {
        return productIds;
    }
}
```

When you run the application and add some products to the cart, their IDs will be displayed (see Figure 4-19).

**Figure 4-19.** *Content of shopping cart displayed*

The Checkout Function

So far, you have implemented the catalog page, the detail page, and the shopping cart page.

Next, you'd like to allow the user to check out (see Figure 4-20). That is, this new page, the confirm page, will display the total charge and the credit card number of the user.

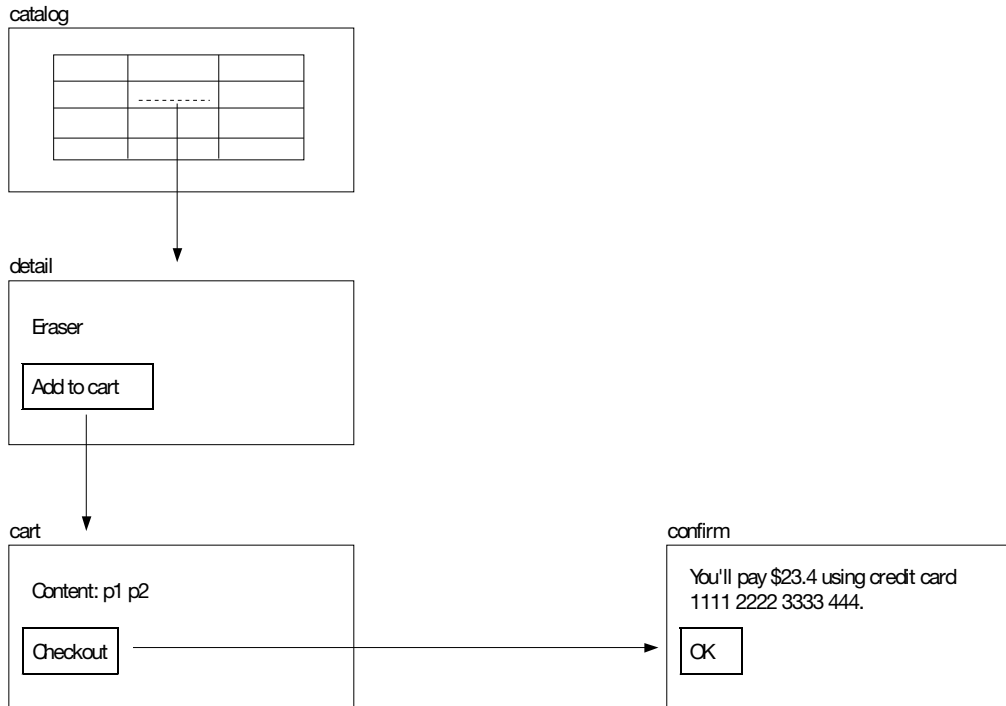


Figure 4-20. *The checkout function*

For the total charge, you can get the product IDs from the shopping cart, so it is easy. But how to get the credit card number of the user? Let's hard-code it for the moment. Now, create the `confirm.xhtml` page as shown in Listing 4-27.

Listing 4-27. *The `confirm.xhtml` Page*

```

<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
<title>Insert title here</title>
</head>
<body>
You'll pay #{confirmService.total} with credit card #{confirmService.creditCardNo}.
</body>
</html>

```

For it to work, create the `ConfirmService` class in the `shop` package as shown in Listing 4-28.

Listing 4-28. *The `ConfirmService` Class*

```
package shop;
...
@Named("confirmService")
@RequestScoped
public class ConfirmService {
    @Current
    private Cart cart;
    @Current
    private Catalog catalog;

    public double getTotal() {
        double total = 0;
        for (String pid : cart.getProductIds()) {
            total += catalog.getProduct(pid).getPrice();
        }
        return total;
    }
    public String getCreditCardNo() {
        return "1111 2222 3333 4444";
    }
}
```

Create the Checkout button in `cart.xhtml`, as shown in Listing 4-29.

Listing 4-29. *Checkout Button in the `cart.xhtml` Page*

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:ui="http://java.sun.com/jsf/facelets"
    xmlns:h="http://java.sun.com/jsf/html">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
<title>Insert title here</title>
</head>
<body>
```

Shopping cart:

```
<ui:repeat value="#{cart.productIds}" var="pid">
    <h:outputText value="#{pid}" />
</ui:repeat>
<h:form>
    <h:commandButton value="Confirm" action="confirm"/>
</h:form>
</body>
</html>
```

Define the navigation case in `faces-config.xml` as shown in Listing 4-30.

Listing 4-30. *Navigation Case for the `confirm.xhtml` Page*

```
<faces-config ...>
    <application>
        <view-handler>com.sun.facelets.FaceletViewHandler
        </view-handler>
    </application>
    <navigation-rule>
        <from-view-id>/catalog.xhtml</from-view-id>
        <navigation-case>
            <from-outcome>detail</from-outcome>
            <to-view-id>/detail.xhtml</to-view-id>
        </navigation-case>
    </navigation-rule>
    <navigation-rule>
        <from-view-id>/detail.xhtml</from-view-id>
        <navigation-case>
            <from-outcome>added</from-outcome>
            <to-view-id>/cart.xhtml</to-view-id>
        </navigation-case>
    </navigation-rule>
    <navigation-rule>
        <from-view-id>/cart.xhtml</from-view-id>
        <navigation-case>
            <from-outcome>confirm</from-outcome>
            <to-view-id>/confirm.xhtml</to-view-id>
        </navigation-case>
    </navigation-rule>
</faces-config>
```

When you run the application and try to check out, it should display the total amount and the hard-coded credit card number (Figure 4-21).

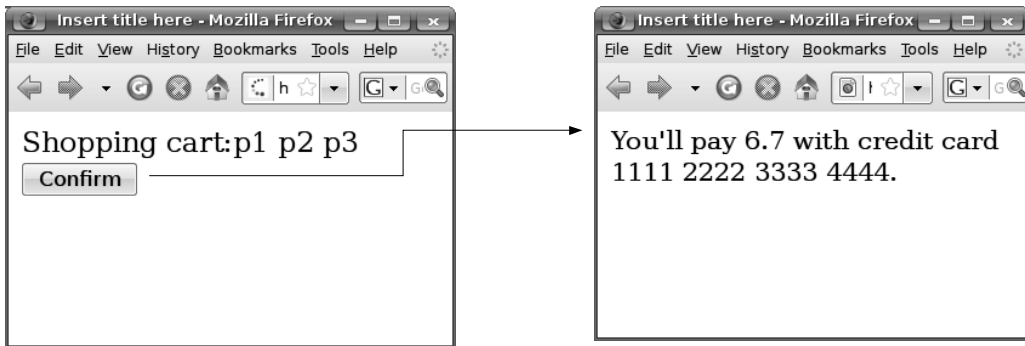


Figure 4-21. *Confirming page working (with a hard-coded credit card number)*

Getting the Credit Card Number of the Current User

So, how do you get the credit card number of the current user? Suppose that the user accounts are stored in a secure database such as the one Amazon.com uses to remember your card details (see Figure 4-22). If the user logs in, then you can use the user ID to load the data into a User object and put it into the session for later use.

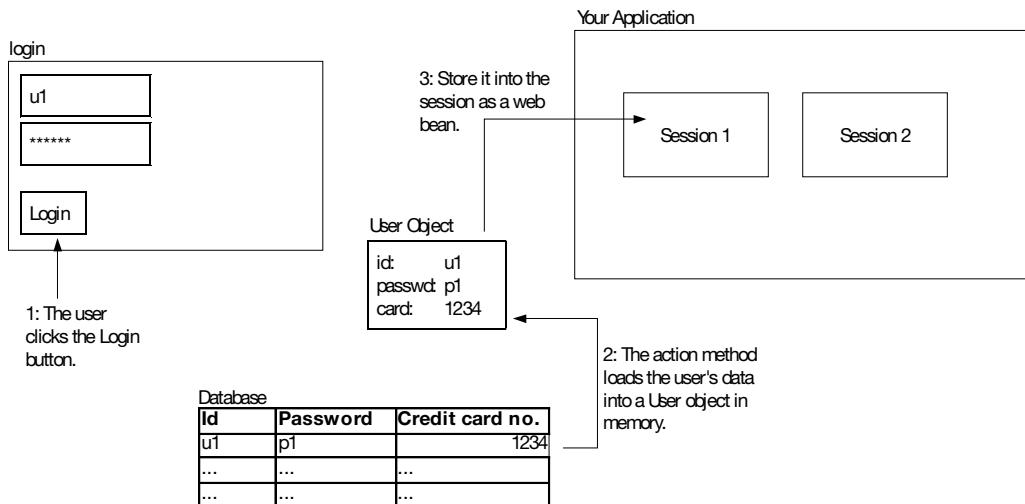


Figure 4-22. *Loading the user's data on login*

To implement this idea, you'll make a Login link on the catalog page to show the login page (see Figure 4-23). On a successful login, the user will be returned to the catalog page.

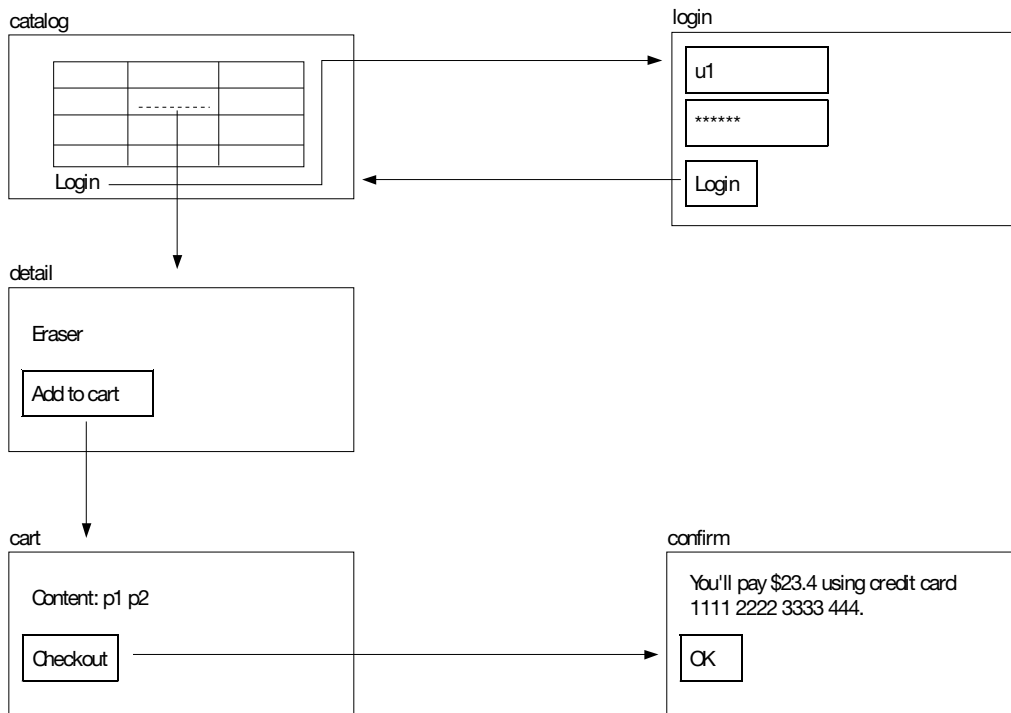


Figure 4-23. Page flow involving the login page

So, modify `catalog.xhtml` as shown in Listing 4-31.

Listing 4-31. Login Link

```

...
<body>
<h:dataTable value="#{catalog.products}" var="p" border="1">
  <h:column>
    <f:facet name="header">
      <h:outputText value="Id"/>
    </f:facet>
    <h:outputText value="#{p.id}"/>
  </h:column>
  <h:column>
    <f:facet name="header">

```

```

        <h:outputText value="Name"/>
    </f:facet>
    <h:form>
        <h:commandLink action="detail">
            <f:setPropertyActionListener .../>
            <h:outputText value="#{p.name}"/>
        </h:commandLink>
    </h:form>
</h:column>
<h:column>
    <f:facet name="header">
        <h:outputText value="Price"/>
    </f:facet>
    <h:outputText value="#{p.price}"/>
</h:column>
</h:dataTable>
<h:form>
    <h:commandLink action="login" value="Login"/>
</h:form>
</body>
</html>

```

Define the navigation case in `faces-config.xml` as shown in Listing 4-32.

Listing 4-32. *Navigation Case for the Login Page*

```

<faces-config ...>
    <application>
        <view-handler>com.sun.facelets.FaceletViewHandler
        </view-handler>
    </application>
    <navigation-rule>
        <from-view-id>/catalog.xhtml</from-view-id>
        <navigation-case>
            <from-outcome>detail</from-outcome>
            <to-view-id>/detail.xhtml</to-view-id>
        </navigation-case>
    </navigation-rule>
    <navigation-rule>
        <from-view-id>/detail.xhtml</from-view-id>
        <navigation-case>
            <from-outcome>added</from-outcome>

```

```

        <to-view-id>/cart.xhtml</to-view-id>
    </navigation-case>
</navigation-rule>
<navigation-rule>
    <from-view-id>/cart.xhtml</from-view-id>
    <navigation-case>
        <from-outcome>confirm</from-outcome>
        <to-view-id>/confirm.xhtml</to-view-id>
    </navigation-case>
</navigation-rule>
<navigation-rule>
    <from-view-id>/catalog.xhtml</from-view-id>
    <navigation-case>
        <from-outcome>login</from-outcome>
        <to-view-id>/login.xhtml</to-view-id>
    </navigation-case>
</navigation-rule>
</faces-config>

```

Create login.xhtml as shown in Listing 4-33.

Listing 4-33. *Login Page*

```

<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://java.sun.com/jsf/html">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
<title>Insert title here</title>
</head>
<body>
<h:messages/>
<h:form>
    <h:inputText value="#{loginRequest.username}" />
    <h:inputText value="#{loginRequest.password}" />
    <h:commandButton value="Login" action="#{loginRequest.login}" />
</h:form>
</body>
</html>

```

Create the `LoginRequest` class in the `shop` package as shown in Listing 4-34. You'll create the (session-scoped) `UserHolder` web bean later in this chapter. It is like the `ProductHolder` bean except that it contains the current `User` object. For simplicity, instead of looking up the database, you'll just hard-code a known user here. If the username and password are correct, you'll put the `User` object into the `UserHolder` web bean (and thus into the session). If the username or password is incorrect, you'll log an error message. In that case, you'll return `null` as the outcome, which tells JSF to *not* change the view ID, that is, redisplay the current page (the login page).

Listing 4-34. *The `LoginRequest` Class*

```
package shop;

...
@Named("loginRequest")
@RequestScoped
public class LoginRequest {
    private String username;
    private String password;
    @Current
    private UserHolder userHolder;

    public String getUsername() {
        return username;
    }
    public String getPassword() {
        return password;
    }
    public void setUsername(String username) {
        this.username = username;
    }
    public void setPassword(String password) {
        this.password = password;
    }
    public String login() {
        if (username.equals("u1") && password.equals("p1")) {
            userHolder.setCurrentUser(new User("u1", "p1", "1234"));
            return "loggedIn";
        } else {
            FacesContext context = FacesContext.getCurrentInstance();
            context.addMessage(null, new FacesMessage(
                FacesMessage.SEVERITY_ERROR, "Login failed", null));
            return null;
        }
    }
}
```

```

    }
}
}

```

For convenience, you are providing the user's details rather than pulling them from a database.

Create the `UserHolder` class in the `shop` package as shown in Listing 4-35. Note that it is a session-scoped web bean and thus needs to implement `Serializable`.

Listing 4-35. *The `UserHolder` Class*

```

package shop;
...
@SessionScoped
public class UserHolder implements Serializable {
    private User currentUser;

    public User getCurrentUser() {
        return currentUser;
    }
    public void setCurrentUser(User currentUser) {
        this.currentUser = currentUser;
    }
}

```

Create the `User` class in the `shop` package as shown in Listing 4-36. Note that because it will be dragged into the session by the `UserHolder` web bean, it needs to implement `Serializable` too.

Listing 4-36. *The `User` Class*

```

package shop;
...
public class User implements Serializable {
    private String username;
    private String password;
    private String creditCardNo;

    public User(String username, String password, String creditCardNo) {
        this.username = username;
        this.password = password;
        this.creditCardNo = creditCardNo;
    }
}

```

```

    }
    public String getCreditCardNo() {
        return creditCardNo;
    }
}

```

Define the navigation case for a successful login as shown in Listing 4-37. Note that on a successful login it will always return to the catalog page.

Listing 4-37. *Navigation Case for Successful Login*

```

<faces-config ...>
    <application>
        <view-handler>com.sun.facelets.FaceletViewHandler
        </view-handler>
    </application>
    ...
    <navigation-rule>
        <from-view-id>/login.xhtml</from-view-id>
        <navigation-case>
            <from-outcome>loggedIn</from-outcome>
            <to-view-id>/catalog.xhtml</to-view-id>
        </navigation-case>
    </navigation-rule>
</faces-config>

```

When you run the application now, the login page should be working (see Figure 4-24).

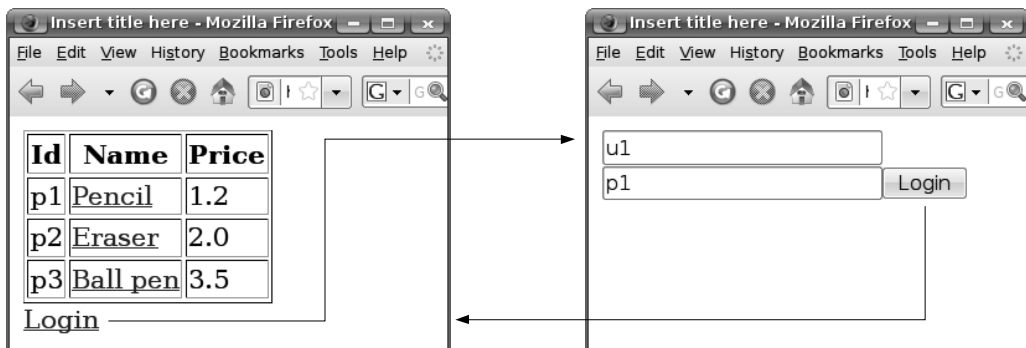


Figure 4-24. *Login page working*

Then, modify `ConfirmService` so that it retrieves the credit card number from the current `User` object (see Listing 4-38).

Listing 4-38. *Retrieving the Credit Card Number of the Current User*

```
...
@Named("confirmService")
@RequestScoped
public class ConfirmService {
    @Current
    private Cart cart;
    @Current
    private Catalog catalog;
    @Current
    private UserHolder uh;

    public double getTotal() {
        double total = 0;
        for (String pid : cart.getProductIds()) {
            total += catalog.getProduct(pid).getPrice();
        }
        return total;
    }
    public String getCreditCardNo() {
        return uh.getCurrentUser().getCreditCardNo();
    }
}
```

Run the application, log in, and then go to the confirm page. It should display “1234” as the credit card number (Figure 4-25) because that was hard-coded in Listing 4-34, which means it is working.

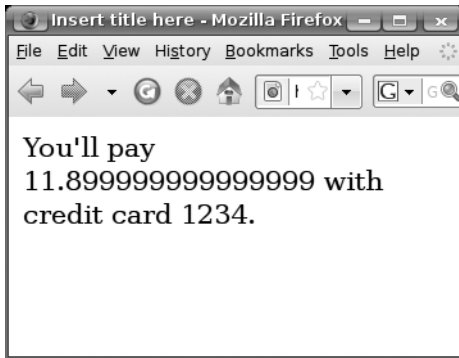


Figure 4-25. Checkout page displaying the user's credit card number

Forcing the User to Log In

The example is working fine if the user logs in and then tries to check out. But what if the user tries to check out without logging in first? Then the current `User` object will be `null` and the code highlighted in bold in Listing 4-39 will be `null`.

Listing 4-39. Problems If the User Hasn't Logged In

```
...
@Named("confirmService")
@RequestScoped
public class ConfirmService {
    @Current
    private Cart cart;
    @Current
    private Catalog catalog;
    @Current
    private UserHolder uh;

    public double getTotal() {
        double total = 0;
        for (String pid : cart.getProductIds()) {
            total += catalog.getProduct(pid).getPrice();
        }
        return total;
    }
    public String getCreditCardNo() {
        return uh.getCurrentUser().getCreditCardNo();
    }
}
```

```

    }
}

```

To handle this scenario, the ideal behavior is to send the user to the login page and then, after a successful login, return the user to the confirm page (see Figure 4-26).

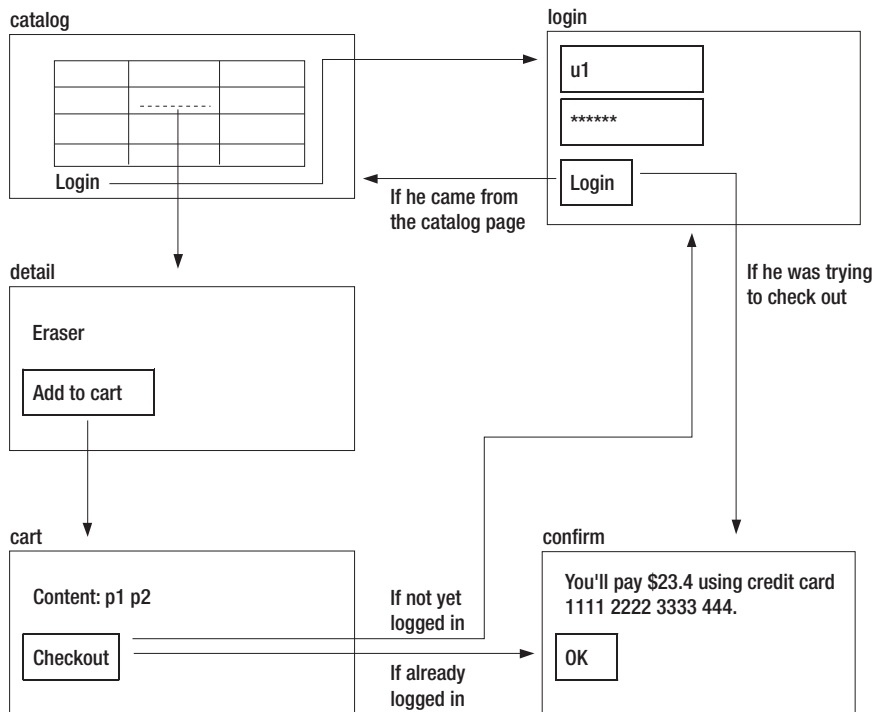


Figure 4-26. Page flow forcing the user to log in

To achieve this effect, you will provide a firewall protecting the Render Response phase, as shown in Figure 4-27. The firewall will check whether JSF is trying to render the `confirm.xhtml` page but there is no current `User` object. If JSF is trying this, the firewall will change the view ID to `/login.xhtml`; if it's not, the firewall lets JSF continue the processing as usual.

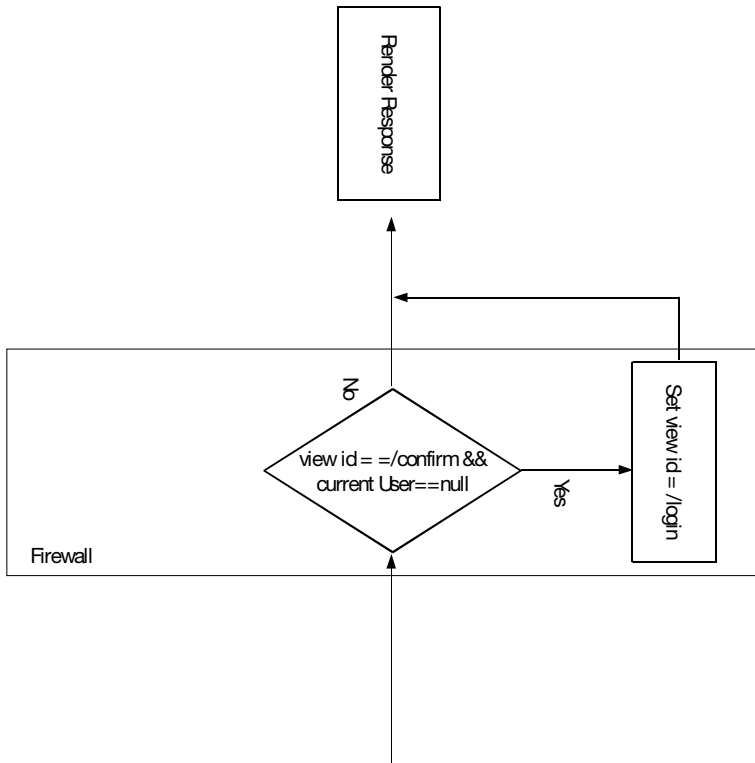


Figure 4-27. Firewall protecting the Render Response phase

Such a firewall can be implemented as a phase listener in JSF. It will get notified whenever JSF is entering a certain phase. So, create a `ForceLoginPhaseListener` class in the shop package, as shown in Figure 4-28.

```

package shop;

import javax.faces.application.Application;
import javax.faces.application.ViewHandler;
import javax.faces.component.UIViewRoot;
import javax.faces.context.FacesContext;
import javax.faces.event.PhaseEvent;
import javax.faces.event.PhaseId;

public class ForceLoginPhaseListener implements PhaseListener {
    public PhaseId getPhaseId() {
        return PhaseId.RENDER_RESPONSE;
    }

    public void beforePhase(PhaseEvent event) {
        FacesContext context = FacesContext.getCurrentInstance();
        String viewId = context.getViewRoot().getViewId();
        if (viewId.equals("/confirm.xhtml")) {
            Application app = context.getApplication();
            UserHolder uh = (UserHolder) app.evaluateExpressionGet(context,
                "#{uh}", UserHolder.class);
            if (uh.getCurrentUser() == null) {
                ViewHandler viewHandler = app.getViewHandler();
                UIViewRoot viewRoot = viewHandler.createView(context,
                    "/login.xhtml");
                context.setViewRoot(viewRoot);
            }
        }
    }

    public void afterPhase(PhaseEvent event) {
    }
}

```

It will be called before and after a certain phase. Which phase? It tells JSF it is only interested in the Render Response phase:

About to render the confirm page?

No User object (not logged in)?

This is the view ID.

View handler is the XHTML parser that creates the component trees from XHTML files.

Ask the view handler to create the component tree from the login.XHTML file.

Tell JSF to render this view.

Figure 4-28. *ForceLoginPhaseListener*

You need to register this phase listener with JSF in `faces-config.xml` (see Listing 4-40).

Listing 4-40. *Registering the ForceLoginPhaseListener*

```

<faces-config ...>
    <lifecycle>
        <phase-listener>shop.ForceLoginPhaseListener</phase-listener>
    </lifecycle>

```

```

<navigation-rule>
  <from-view-id>/catalog.xhtml</from-view-id>
  <navigation-case>
    <from-outcome>detail</from-outcome>
    <to-view-id>/detail.xhtml</to-view-id>
  </navigation-case>
</navigation-rule>
...
</faces-config>

```

Because the phase listener needs to access the `UserHolder` web bean by name, you need to give it a name (see Listing 4-41).

Listing 4-41. *Naming the `UserHolder` Web Bean*

```

...
@Named("uh")
@SessionScoped
public class UserHolder implements Serializable {
    private User currentUser;

    public User getCurrentUser() {
        return currentUser;
    }
    public void setCurrentUser(User currentUser) {
        this.currentUser = currentUser;
    }
}

```

Now, you'd like to go to the confirm page without logging in to test your progress. But you just logged in not long ago, so how do you make the application forget that you logged in? You could wait, say, 30 minutes so that the session is timed out, but a faster way is to close the browser and start a new one. Then JBoss will treat it as a new browser (and thus a new user and a new session).

Try it, and it should display the login page. But how do you return the user to the right page once he has logged in? You can store the original view ID in the `UserHolder` web bean before redirecting to the login page and let the login page return to there on a successful login (see Listing 4-42, Listing 4-43, and Listing 4-44).

Listing 4-42. *Keeping the Original View ID in the UserHolder Web Bean*

```

...
@Named("uh")
@SessionScoped
public class UserHolder implements Serializable {
    private User currentUser;
    private String originalViewId;

    public String getOriginalViewId() {
        return originalViewId;
    }
    public void setOriginalViewId(String originalViewId) {
        this.originalViewId = originalViewId;
    }
    public User getCurrentUser() {
        return currentUser;
    }
    public void setCurrentUser(User currentUser) {
        this.currentUser = currentUser;
    }
}

```

Listing 4-43. *Storing the Original View ID in the UserHolder Web Bean*

```

public class ForceLoginPhaseListener implements PhaseListener {
    public PhaseId getPhaseId() {
        return PhaseId.RENDER_RESPONSE;
    }
    public void beforePhase(PhaseEvent event) {
        FacesContext context = FacesContext.getCurrentInstance();
        String viewId = context.getViewRoot().getViewId();
        if (viewId.equals("/confirm.xhtml")) {
            Application app = context.getApplication();
            UserHolder uh = (UserHolder) app.evaluateExpressionGet(context,
                "#{uh}", UserHolder.class);
            if (uh.getCurrentUser() == null) {
                uh.setOriginalViewId(viewId);
                ViewHandler viewHandler = app.getViewHandler();
                UIViewRoot viewRoot = viewHandler.createView(context,
                    "/login.xhtml");
                context.setViewRoot(viewRoot);
            }
        }
    }
}

```

```

        }
    }
}
public void afterPhase(PhaseEvent event) {
}
}

```

Listing 4-44. *Returning to the Original View (If Any) on Successful Login*

```

...
@Named("loginRequest")
@RequestScoped
public class LoginRequest {
    private String username;
    private String password;
    @Current
    private UserHolder userHolder;
    ...
    public String login() {
        if (username.equals("u1") && password.equals("p1")) {
            userHolder.setCurrentUser(new User("u1", "p1", "1234"));
            String viewId = userHolder.getOriginalViewId();
            if (viewId != null) {
                FacesContext context = FacesContext.getCurrentInstance();
                Application app = context.getApplication();
                ViewHandler viewHandler = app.getViewHandler();
                UIViewRoot root = viewHandler.createView(context, viewId);
                context.setViewRoot(root);
                userHolder.setOriginalViewId(null);
                return null;
            } else {
                return "loggedIn";
            }
        } else {
            FacesContext context = FacesContext.getCurrentInstance();
            context.addMessage(null, new FacesMessage(
                FacesMessage.SEVERITY_ERROR, "Login failed", null));
            return null;
        }
    }
}
}

```

Note that it is possible that the user explicitly clicked the login link to go to the login page instead of being redirected to here. In that case, there is no original view ID (`null`), so you'll just return `loggedIn` as the outcome as usual. Then the navigation system will send the user to the catalog page. If there is indeed an original view ID, you'll use it to load the view root and set it as the current view root. In that case, because you have set the view root yourself, you must tell the navigation system to *not* change the view root again. This is done by returning `null` as the outcome.

Now, start a new session, and run it again. Try to check out without logging in. The application should display the login page. Then it will return you to the confirm page once you have logged in. Then start a new session, but this time log in from the catalog page. It should then return you to the catalog page.

Implementing Logout

Suppose that you'd like to allow the user to log out by clicking a Logout link, as shown in Figure 4-29.

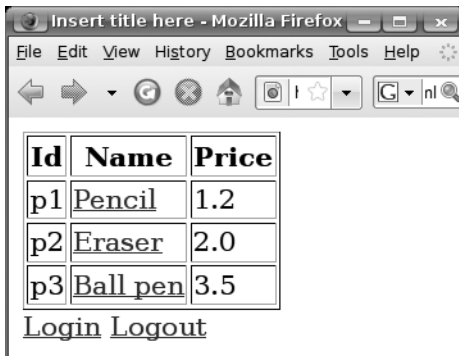


Figure 4-29. *The Logout link*

The minimum that you need to do is to remove the `User` object from the `UserHolder` web bean. However, a better way is to delete the session altogether (including the shopping cart, for example) because it will free up the memory. To do that, modify `catalog.xhtml` as shown in Listing 4-45. Note that you're not setting the outcome (action) so that it remains on the catalog page after logging out. In addition, you will create the action listener to remove the session. Why not specify a method in the action attribute? You could do that, but removing the session is a UI-specific task, not a business task. So, an action listener is better.

Listing 4-45. *Logout Link on the Catalog Page*

```

...
<h:dataTable value="#{catalog.products}" var="p" border="1">
    ...
</h:dataTable>
<h:form>
    <h:commandLink action="login" value="Login"/>
    <h:commandLink value="Logout">
        <f:actionListener type="shop.LogoutActionListener"/>
    </h:commandLink>
</h:form>
</body>

```

Create the `LogoutActionListener` class in the `shop` package, as shown in Figure 4-30.

```

package shop;

import javax.faces.context.ExternalContext;
import javax.faces.context.FacesContext;
import javax.faces.event.AbortProcessingException;
import javax.faces.event.ActionEvent;
import javax.faces.event.ActionListener;
import javax.servlet.http.HttpSession;

public class LogoutActionListener implements ActionListener {
    public void processAction(ActionEvent event)
        throws AbortProcessingException {
        FacesContext context = FacesContext.getCurrentInstance();
        ExternalContext externalContext = context.getExternalContext();
        Object session = externalContext.getSession(false);
        HttpSession httpSession = (HttpSession) session;
        httpSession.invalidate();
    }
}

```

The external context means the platform JSF is running on. In this case, it's JBoss.

Get access to the session. The session is maintained by the platform (JBoss).

Remove the session.

The session is an `Object`, not an `HttpSession`. This is because JSF could potentially run on a platform that doesn't use HTTP. Here you're sure it is using HTTP, so you can typecast it.

Figure 4-30. *The `LogoutActionListener` class*

Run the application, and log in and then log out. Then try to check out, and it should ask you to log in again. You may notice that there is no space between the Login link and the Logout link. To fix this, modify `catalog.xhtml` as shown in Listing 4-46.

Listing 4-46. *Inserting a Space*

```
...
<h:form>
    <h:commandLink action="login" value="Login"/>
    <h:outputText value=" "/>
    <h:commandLink value="Logout">
        <f:actionListener type="shop.LogoutActionListener"/>
    </h:commandLink>
</h:form>
</body>
</html>
```

When you run the application, there should be a space between the two links.

Protecting the Password

Currently the login page shows the password as the user types it in. This is no good because someone watching over the user's shoulders could steal the password. It's better to display it as asterisks. To do that, modify `login.xhtml` as shown in Listing 4-47. `<h:inputSecret>` is just like `<h:inputText>` in that it will create a UI Input. The only difference is that the user input will appear as asterisks. Again, this is done using a different renderer.

Listing 4-47. *Using `<h:inputSecret>`*

```
...
<body>
<h:messages />
<h:form>
    <h:inputText value="#{loginRequest.username}" />
    <h:inputSecret value="#{loginRequest.password}" />
    <h:commandButton value="Login" action="#{loginRequest.login}" />
</h:form>
</body>
</html>
```

When you run the application now, the password should be displayed as asterisks.

Summary

You learned the following in this chapter:

- A facet is a child component that is subjected to special processing by its parent.
- A request has a table of attributes. Each attribute has a name and a value (Object). It allows you to give a name to an object. It is like request-scoped web beans except that it doesn't create the object; it only associates it with a name.
- To loop through some items in a table, if the number of items can only be determined at runtime, use the `<h:dataTable>` tag, which will create a UI Data component. You provide a List of items to it, and it will loop through it for each item. Its children must be UI Column components. Each UI Column represents a column in the table. For each row, the UI Data will create an environment containing the row index and store the current item into an attribute before asking each UI Column to render itself. Each UI Column will render its own child components (excluding the facet named header). A UI Column can optionally have a facet named header. In that case, the UI Data will render a header row and ask the UI Columns to render the header facet as the header for the column before it starts to process the data rows.
- On form submission, the UI Data will loop through the items again, re-creating the environment (to set up the attribute) and giving each component inside an opportunity to apply request values, process validations, update domain values and invoke application in each respective phase.
- Because the current item is stored into an attribute and that attribute will be cleared by the environment, if you need to pass it onto the next page, most likely you'll want to use the Set Property action listener.
- If you need to loop through some items but they won't be presented in an HTML `<table>`, you can use the `<ui:repeat>` tag. It works very much like the `<h:dataTable>` tag except that it won't output markup of its own.
- To create a link, use the UI Command component with a link renderer. In terms of behavior, it is just like a UI Command component with a button renderer.
- For a UI Command, in addition to setting an outcome in its action attribute, you can also specify a method. This is useful when you need to perform a business action. That method should return a string indicating the outcome. If you need to perform a UI-specific action, it's better to add an action listener to the UI Command.
- A UI Input can be rendered such that user input is echoed as stars. This is good for password input.

- If you display the properties of a request-scoped web bean using a page, you must be careful when the form is submitted because a new request-scoped web bean will be created. Usually you will use a UI Input component along with an HTML Hidden Input renderer to store the ID in the browser as a hidden field. You will then load the object when the ID is set.
- JSF uses a view handler to create the JSF component tree from a specified view ID. You need to use a view handler when you want to bypass the JSF navigation system. You need to load a view and set it as the one to be rendered.
- A session is a memory area for each user (or rather, each browser instance). To start a new session, either restart the browser or wait until the timeout. To remove the session on the server, call `invalidate()` on the session. This is commonly done when logging out. Everything put into the session must implement `Serializable`.
- There is a web bean table in each session. You can put per-user temporary data into there.
- A phase listener will be notified before entering a phase or after exiting from a phase. You can use it to make sure the user has logged in before rendering a certain pages.
- The external context means the platform on which JSF is running. In your case, it is JBoss. JSF assumes this platform is responsible for maintaining the session.



Creating Custom Components

In this chapter, you'll learn how to create your own components that can be reused on multiple pages.

Displaying a Copyright Notice on Multiple Pages

Suppose that you'd like to display a copyright notice on multiple pages like that shown in Figure 5-1.

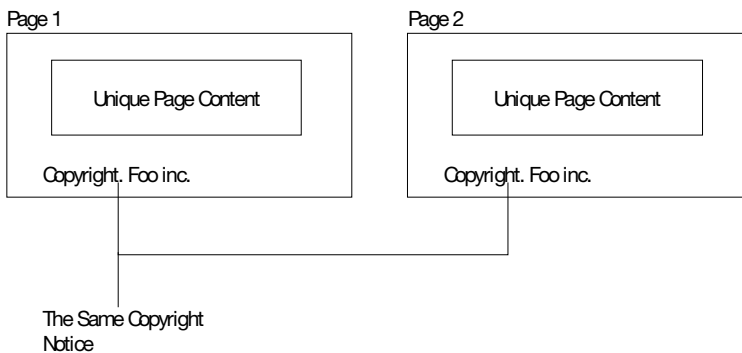


Figure 5-1. *Copyright notice on multiple pages*

This is no good, because if later you need to modify the copyright notice, you'll have to do it multiple times (once for each page). To solve this problem, you can extract the common HTML code into a separate XHTML file, as shown in Figure 5-2.

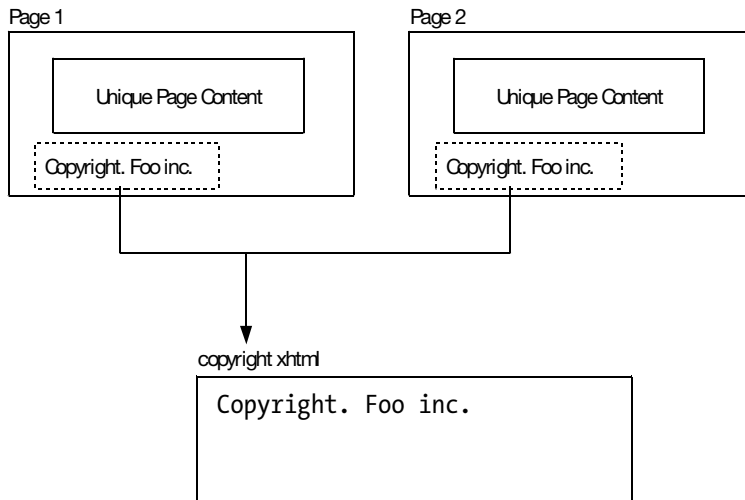


Figure 5-2. *Extracting common code into a separate XHTML file*

Then to include that XHTML file into a particular page, let's assume that someone has developed a custom tag that can be used, as shown in Figure 5-3.

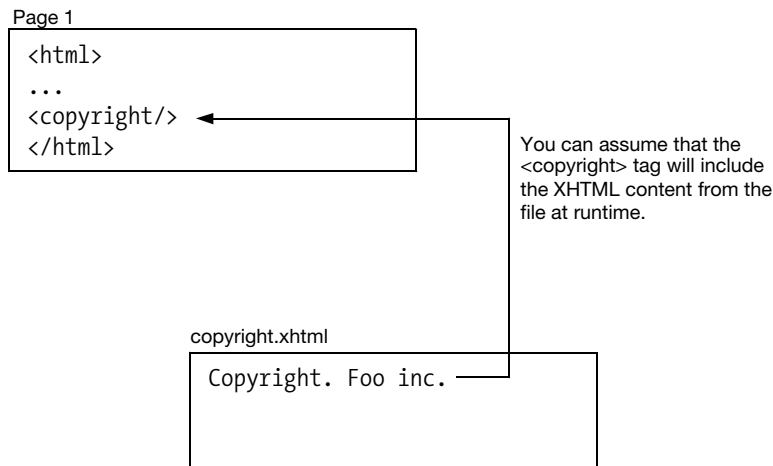


Figure 5-3. *Including the XHTML file using a custom tag*

However, this is not a simple text inclusion. When JSF is creating the component tree, it will use the content of `copyright.xhtml` to create a subtree and then graft that subtree into the page, as shown in Figure 5-4.

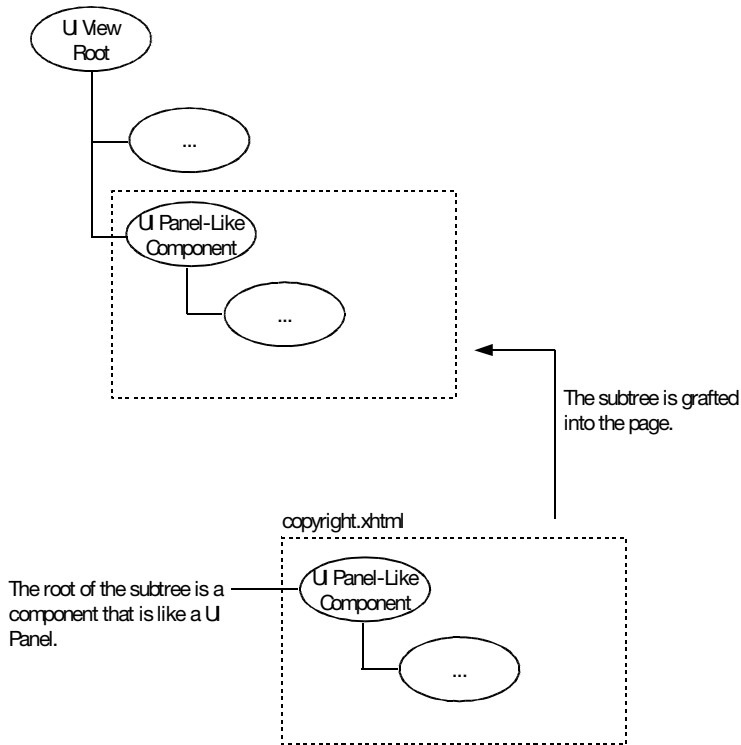


Figure 5-4. *Grafting the subtree into the page*

In addition, in XHTML each tag must belong to a namespace, so your `<copyright>` tag must as well. Let's choose `http://foo.com` as the namespace (you could choose any unique URL that you'd like, though). Then you will use the tag as shown in Listing 5-1.

Listing 5-1. *The `<copyright>` Tag in the `http://foo.com` Namespace*

```
<html xmlns:foo="http://foo.com">
...
<foo:copyright/>
</html>
```

In addition, just because the tag is named `copyright`, JSF will not simply assume that the XHTML is named `copyright.xhtml`. Instead, you must explicitly tell JSF the file name when defining the tag. Conceptually, it looks like Figure 5-5.



Figure 5-5. *Explicitly specifying the file name when defining a custom tag*

Because a namespace could contain more than one tag, conceptually you could define multiple tags, as shown in Listing 5-2.

Listing 5-2. *Defining Multiple Tags Conceptually*

```

namespace: http://foo.com
tag1: copyright
source1: copyright.xhtml
tag2: ...
source2: ...

```

This means that you’re defining a tag library instead of just a single tag. You would put such a tag lib definition into a file in a folder named META-INF in the classpath. The file name must end with .taglib.xml, such as foo.taglib.xml. On startup, JSF will look for such file names and load the definitions.

Now, let’s do it. Create a new dynamic web project named CustomComp. Create p1.xhtml as shown in Listing 5-3. Here p1 stands for “page 1” and serves as a simple page to use your <copyright> tag.

Listing 5-3. *Sample Page Using the Custom Tag*

```

<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:foo="http://foo.com">
<p>This is page1.</p>
<foo:copyright/>
</html>

```

Create a META-INF folder in your Java source folder, and then create a file foo.taglib.xml in it. Listing 5-4 shows the content. Here you’re defining a Facelet tag lib, which is the same as a namespace. The tag lib (namespace) is identified by the URL http://foo.com.

You're defining a single tag called `<copyright>`, while you could define many tags in a tag lib. The XML tags used in Listing 5-4 to define a tag (for example, `<facelet-taglib>` and `<tag>`) are all in the `http://java.sun.com/JSF/Facelet` namespace.

Listing 5-4. *Defining a Tag Lib*

```
<!DOCTYPE facelet-taglib PUBLIC
"-//Sun Microsystems, Inc.//DTD Facelet Taglib 1.0//EN"
"http://java.sun.com/dtd/facelet-taglib_1_0.dtd">
<facelet-taglib xmlns="http://java.sun.com/JSF/Facelet">
  <namespace>http://foo.com</namespace>
  <tag>
    <tag-name>copyright</tag-name>
    <source>copyright.xhtml</source>
  </tag>
</facelet-taglib>
```

Note In Mojarra 2.0.0.PR2, there is a bug preventing *.taglib.xml files in the META-INF folder on the classpath to be discovered. To work around it, put the whole META-INF folder into WebContent and then explicitly specify the tag lib in web.xml, as shown in Listing 5-5.

Listing 5-5. *Explicitly Specifying a Tag Lib*

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app ...>
  ...
  <servlet>
    <servlet-name>JSF</servlet-name>
    <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>JSF</servlet-name>
    <url-pattern>/faces/*</url-pattern>
  </servlet-mapping>
  <context-param>
    <param-name>javax.faces.FACELETS_LIBRARIES</param-name>
    <param-value>META-INF/foo.taglib.xml</param-value>
  </context-param>
</web-app>
```

Create the `copyright.xhtml` file in the same `META-INF` folder. The content contains a single line only (see Listing 5-6).

Listing 5-6. *XHTML Code for the `<copyright>` Tag*

```
Copyright. Foo inc.
```

However, JSF expects that the file is a complete XHTML page such as Listing 5-7, probably so that you can use a visual editor to edit the XHTML code.

Listing 5-7. *Complete XHTML Page for the `<copyright>` Tag*

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<body>
Copyright. Foo inc.
</body>
</html>
```

However, this is a problem, because you definitely don't want to include the `<html>` and `<body>` tags in the real page. To solve this problem, JSF will require that you indicate the real content, as shown in Listing 5-8, by surrounding it with the `<component>` tag. This tag will create the UI Panel-like component as the root of the subtree. Everything outside will not go into the JSF component tree and thus will have no effect on the output. The `<component>` tag is defined in the JSF Facelets tag lib, which is the third tag lib in addition to the JSF Core tag lib and JSF HTML tag lib. The tags in the JSF Facelets tag lib are mainly used to define components.

Listing 5-8. *Indicating the Real Content with `<ui:component>`*

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets">
<body>
<ui:component>
Copyright. Foo inc.
</ui:component>
</body>
</html>
```

Now when you run the application, you should see the copyright notice on the page.

Allowing the Caller to Specify the Company Name

Suppose that in your application you'd like to display "Foo" as the company name on some pages, but on other pages you'd like to display "Bar" instead. How do you do that? You can let your `<copyright>` tag accept a parameter, as shown in Listing 5-9.

Listing 5-9. Providing Parameters to a Custom Tag

```
...
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:foo="http://foo.com">
  <p>This is page1.</p>
  <foo:copyright company="Foo"/>
</html>
```

To output the company parameter in `copyright.xhtml`, access it just like a web bean or an attribute (see Listing 5-10).

Listing 5-10. Accessing a Parameter in an EL Expression

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets">
  <body>
    <ui:component>
```

Copyright. #{company}

```
</ui:component>
</body>
</html>
```

How does it work? When JSF is building the component tree, the `<copyright>` tag will copy all its attributes into a table (see Figure 5-6). The attribute values are treated as EL expressions, and they will be copied as is, without being evaluated. Then JSF will go into `copyright.xhtml`. When it sees the EL expression `#{company}`, it will link it to the surrounding variable table so that it can find the variables when it is evaluated in the future. This kind of variable is called an *EL variable*. If you know C/C++, an EL variable is very much like a macro in C/C++ in that you can assign a name to an expression.

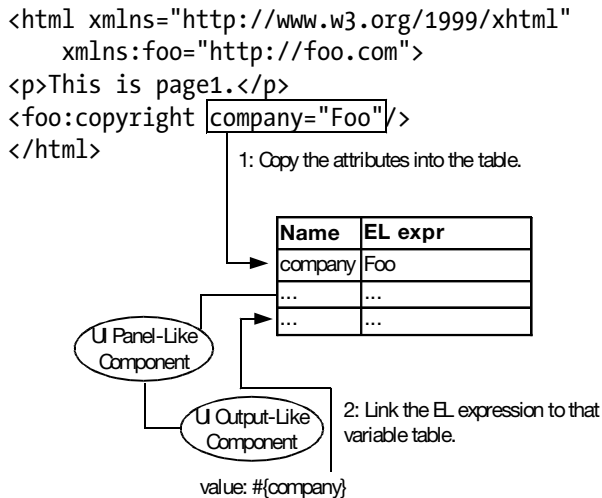


Figure 5-6. Custom tag parameters gathered to form a variable table

After the tree is completed, the connection between the variable table and the UI Panel-like component will be removed, and the tree will look like what's shown in Figure 5-7. Now, there is no longer any concept of parameters.

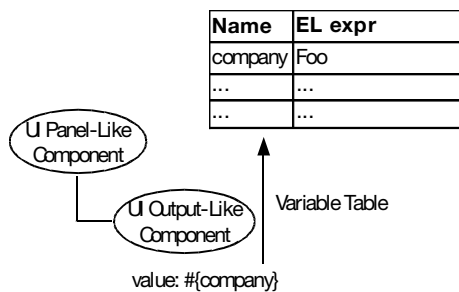


Figure 5-7. EL expression linked to a variable table

When the UI Output-like component needs to render itself, it will evaluate the EL expression. The EL expression will look up the variable table (see Figure 5-8) and reach Foo. Then it will evaluate Foo as an EL expression again. Because it is a literal, the result is still Foo, so that's the output you'll see on the screen.

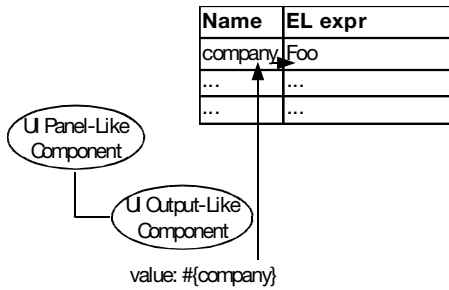


Figure 5-8. Looking up the variable to find the EL expression to evaluate

Now run the application again, and it will display the copyright notice with “Foo” as the company name.

Creating a Product Editor

You aren’t limited to passing strings as parameters; you can pass objects. For example, suppose that you’d like to have a form to edit the details of a `Product` object (containing, say, a product ID and a product name) and that the form is used on multiple pages. So, you’d like to create a custom tag to represent such a form and pass a `Product` object to it for editing (see Listing 5-11). Here, `<pe>` stands for “product editor.”

Listing 5-11. Passing an Object to a Custom Tag

```
...
<foo:pe product="...EL EXPR TO RETURN A PRODUCT..." />
<foo:copyright company="Foo" />
```

To do that, modify `foo.taglib.xml` as shown in Listing 5-12.

Listing 5-12. Defining the `<pe>` Tag

```
<!DOCTYPE facelet-taglib PUBLIC
"-//Sun Microsystems, Inc.//DTD Facelet Taglib 1.0//EN"
"http://java.sun.com/dtd/facelet-taglib_1_0.dtd">
<facelet-taglib xmlns="http://java.sun.com/JSP/Facelet">
  <namespace>http://foo.com</namespace>
  <tag>
    <tag-name>copyright</tag-name>
    <source>copyright.xhtml</source>
  </tag>
```

```

    <tag>
        <tag-name>pe</tag-name>
        <source>pe.xhtml</source>
    </tag>
</facelet-taglib>

```

Create the `pe.xhtml` file as shown in Listing 5-13.

Listing 5-13. *XHTML Code for the `<pe>` Tag*

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:ui="http://java.sun.com/jsf/facelets"
    xmlns:h="http://java.sun.com/jsf/html">
<body>
<ui:component>
    <h:form>
        <h:inputHidden value="#{product.id}"/>
        <h:inputText value="#{product.name}"/>
        <h:commandButton action="#{product.onUpdated}" value="OK"/>
    </h:form>
</ui:component>
</body>
</html>

```

Note how the EL expressions refer to the product parameter in the variable table. Finally, the caller has to provide a `Product` object. Let's do it in `p1.xhtml` (see Listing 5-14).

Listing 5-14. *Providing a Product Object to the `<pe>` Tag*

```

...
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:foo="http://foo.com">
<p>This is page1.</p>
<foo:pe product="#{currentProduct}"/>
<foo:copyright company="Foo"/>
</html>

```

Create the `Product` class in the custom package, and create the `currentProduct` web beans from it (see Listing 5-15).

Listing 5-15. *The Product Class*

```
package custom;
...
@Named("currentProduct")
@RequestScoped
public class Product {
    private String id;
    private String name;

    public Product() {
        this("p1", "pen");
    }
    public Product(String id, String name) {
        this.id = id;
        this.name = name;
    }
    public String onUpdated() {
        System.out.println(id + ": " + name);
        return "updated";
    }
    public String getId() {
        return id;
    }
    public void setId(String id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
}
```

Note that its `onUpdate()` method will simply print its data to the console. In practice, it could update the database, and so on. Now, restart JBoss so that JSF loads the tag lib definition again. Then run the application, modify the product name, and submit the form. It will now print the data to the console.

Passing a Method in a Parameter?

Note that some built-in JSF tags accept methods as parameters, as shown in Listing 5-16.

Listing 5-16. JSF Tags Accepting Method Parameters

```
<h:commandButton action="#{ph.addToCart}"/>
<h:commandLink action="..." />
```

Can this be done with your custom tags? For example, could you modify the <pe> tag so that it can be used as shown in Listing 5-17?

Listing 5-17. Custom Tags Accepting Method Parameters?

```
<foo:pe product="..." action="currentProduct.onUpdated"/>
...
```

Then you could call it as shown in Listing 5-18.

Listing 5-18. Calling a Method Parameter?

```
...
<ui:component>
  <h:form>
    <h:inputHidden value="#{product.id}"/>
    <h:inputText value="#{product.name}"/>
    <h:commandButton action="#{action}" value="OK"/>
  </h:form>
</ui:component>
</body>
</html>
```

Unfortunately, by default it won't work. That is, by default all attributes of a custom tag are expected to evaluate to values (primitive values or objects). They can't evaluate to methods. The workaround is to pass an object that has a method that you can invoke (see Listing 5-19 and Listing 5-20).

Listing 5-19. Passing an Object As an Action Provider

```
<foo:pe product="..." actionProvider="some object having an onUpdated() method"/>
...
```

Listing 5-20. *Invoking the Method of an Action Provider*

```

...
<ui:component>
    <h:form>
        <h:inputHidden value="#{product.id}"/>
        <h:inputText value="#{product.name}"/>
        <h:commandButton action="#{actionProvider.onUpdated}" value="OK"/>
    </h:form>
</ui:component>
</body>
</html>

```

This example is actually not much different from the original solution: invoking a method on the `Product` object itself.

Creating a Box Component

Suppose that you'd like to create a component that will accept any XHTML code (including JSF tags) and will render a box around the code. See Figure 5-9 for an example.

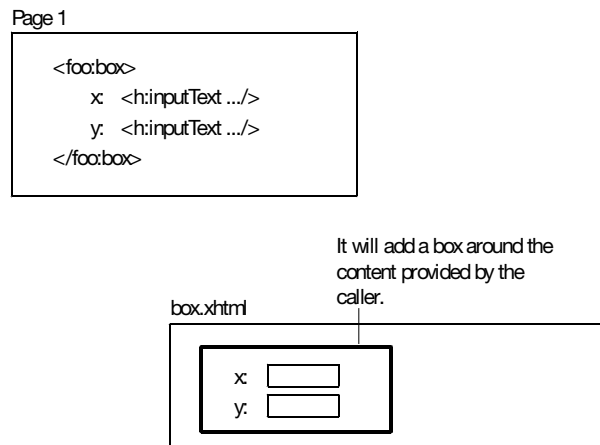


Figure 5-9. *The Box component*

A first attempt is to try to pass the XHTML code through a parameter, as shown in Listing 5-21, and then to output it as shown in Listing 5-22.

Listing 5-21. *Passing XHTML Code in Parameters?*

```
<foo:box content="x: <h:inputText .../> ...">
...

```

Listing 5-22. *Outputting XHTML Code?*

```
...
<table border="1">
  <tr>
    <td><h:outputText value="#{content}"/></td>
  </tr>
</table>

```

However, this won't work because the XHTML code will be treated as a string and JSF won't parse it to create components accordingly. Then what you'll get will look like Figure 5-10.

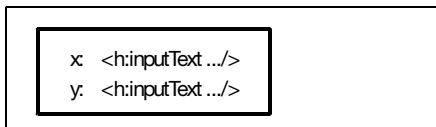


Figure 5-10. JSF tags will be output as is.

To really pass XHTML code (including JSF tags) to a custom tag, you can put the XHTML code into the tag body. Let's do it in `p1.xhtml` (see Listing 5-23).

Listing 5-23. *Passing XHTML Code As the Tag Body*

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
  xmlns:foo="http://foo.com">
  <p>This is page1.</p>
  <foo:box>
    <foo:pe product="#{currentProduct}"/>
    <foo:copyright company="Foo"/>
  </foo:box>
</html>

```

So, how do you parse the XHTML code in the custom component? Create `box.xhtml` (in the `META-INF` folder) with the content shown in Figure 5-11. Simply put, the `<ui:insert>` tag will trace back into the body of the custom tag in the calling page to build the component tree.

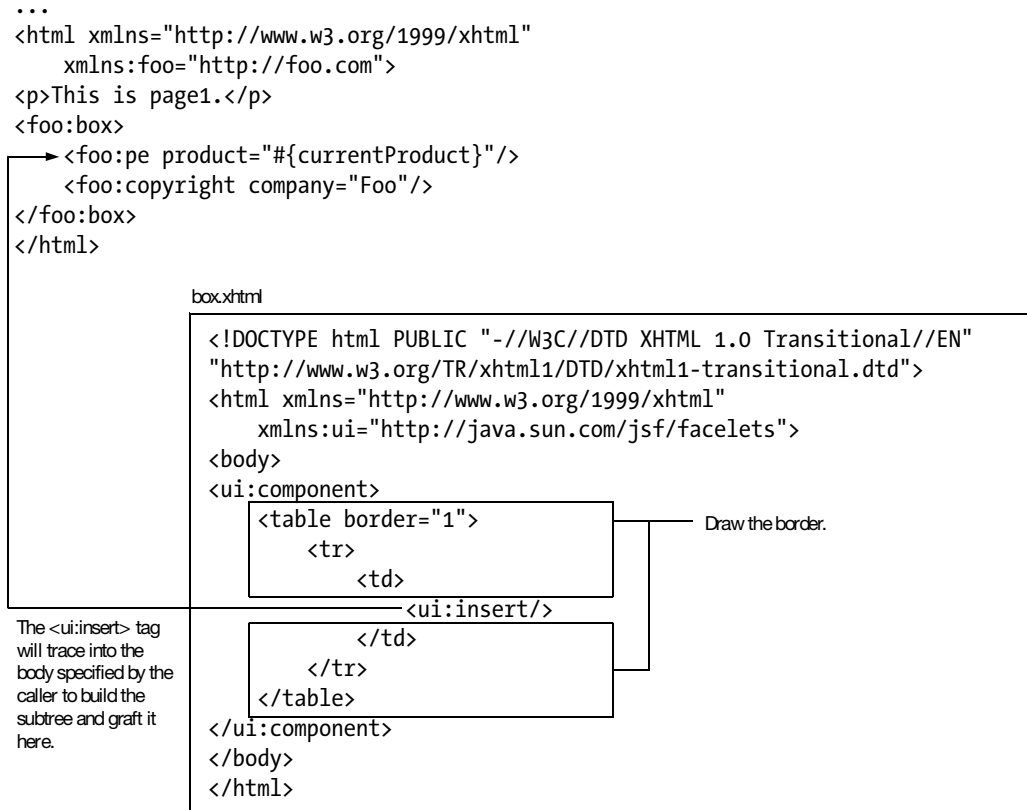


Figure 5-11. Using `<ui:insert>` to trace into the body

Finally, define the `<box>` tag in `foo.taglib.xml`, as shown in Listing 5-24.

Listing 5-24. Defining the `<box>` Tag

```

<!DOCTYPE facelet-taglib PUBLIC
"-//Sun Microsystems, Inc.//DTD Facelet Taglib 1.0//EN"
"http://java.sun.com/dtd/facelet-taglib_1_0.dtd">
<facelet-taglib xmlns="http://java.sun.com/JSP/Facelet">
  <namespace>http://foo.com</namespace>
  <tag>

```

```

    <tag-name>copyright</tag-name>
    <source>copyright.xhtml</source>
  </tag>
<tag>
  <tag-name>pe</tag-name>
  <source>pe.xhtml</source>
</tag>
<tag>
  <tag-name>box</tag-name>
  <source>box.xhtml</source>
</tag>
</facelet-taglib>

```

Restart JBoss, and run the application. When you do, you should see a border surrounding the product editor and the copyright notice.

Accepting Two Pieces of XHTML Code

Can a custom tag accept two pieces of XHTML code? For example, can you create a tag that will display the two pieces of XHTML in two cells in a row, as shown in Figure 5-12?

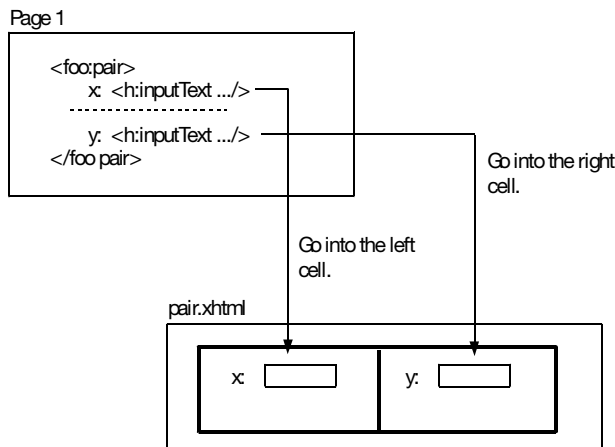


Figure 5-12. *Accepting two pieces of XHTML code*

To do that, create `pair.xhtml` (in the META-INF folder), and modify `p1.xhtml` as shown in Figure 5-13. That is, you assign a unique name to each `<ui:define>` tag. When using a `<ui:insert>` tag to trace back into the body of the custom tag in the calling page, you also specify a name so that it can look up the corresponding `<ui:define>` tag by name.

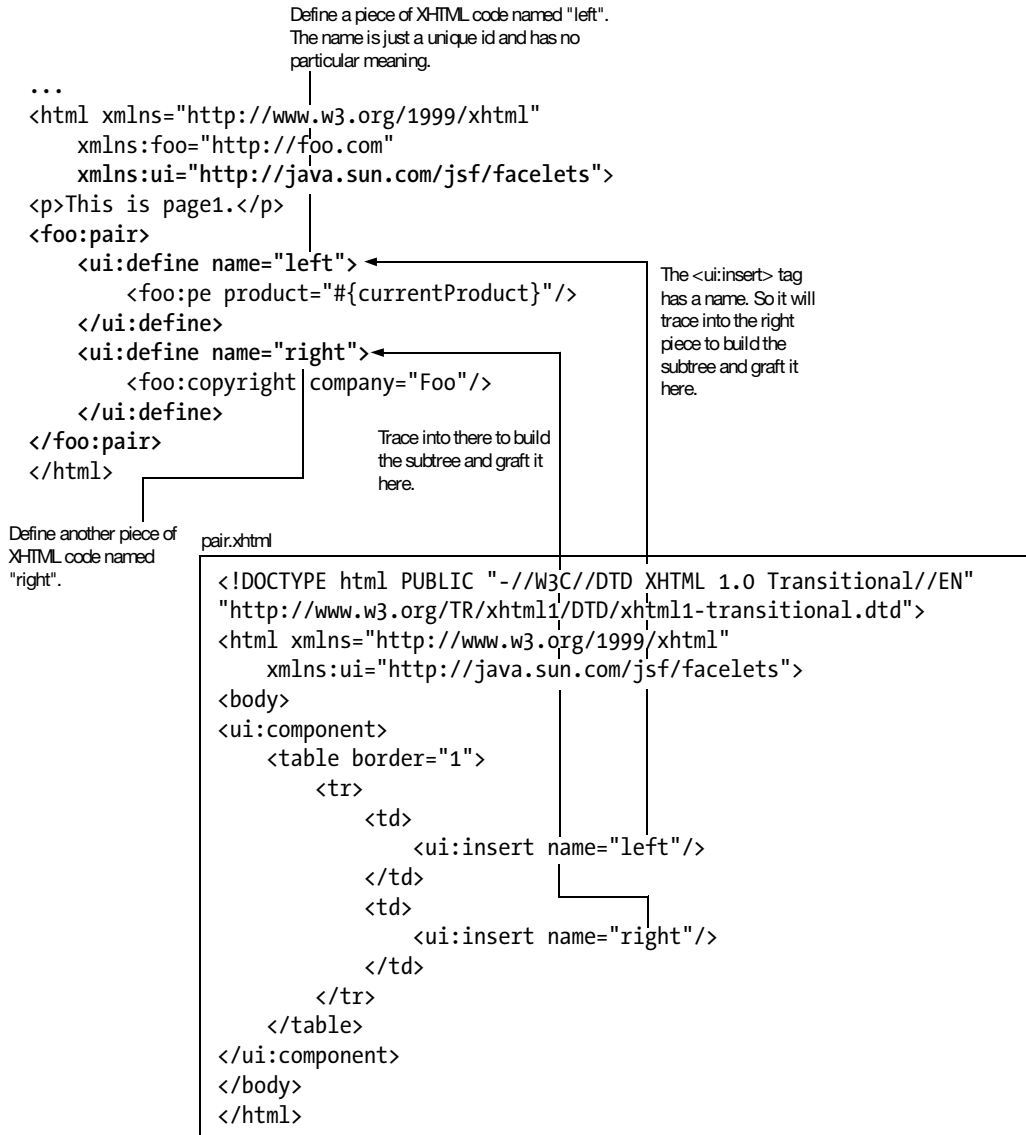


Figure 5-13. Using `<ui:insert>` with a name along with `<ui:define>`

Define the `<pair>` tag in `foo.taglib.xml` as shown in Listing 5-25.

Listing 5-25. *Defining the `<pair>` Tag*

```
<!DOCTYPE facelet-taglib PUBLIC
  "-//Sun Microsystems, Inc.//DTD Facelet Taglib 1.0//EN"
  "http://java.sun.com/dtd/facelet-taglib_1_0.dtd">
```

```

<facelet-taglib xmlns="http://java.sun.com/JSF/Facelet">
  <namespace>http://foo.com</namespace>
  <tag>
    <tag-name>copyright</tag-name>
    <source>copyright.xhtml</source>
  </tag>
  <tag>
    <tag-name>pe</tag-name>
    <source>pe.xhtml</source>
  </tag>
  <tag>
    <tag-name>box</tag-name>
    <source>box.xhtml</source>
  </tag>
  <tag>
    <tag-name>pair</tag-name>
    <source>pair.xhtml</source>
  </tag>
</facelet-taglib>

```

Restart JBoss, and run the application. You should see the product editor in the left cell and the copyright notice in the right one.

Creating a Reusable Component Library

Suppose that you'd like to use the custom tags created so far in multiple projects. Obviously, copying `foo.taglib.xml` and the source XHTML files into multiple projects is a bad idea, because if later you need to fix a bug in one of those files, you will have to do it once for each project. To solve the problem, you can pack the `META-INF` folder into a JAR file and reuse it in multiple projects (see Figure 5-14).

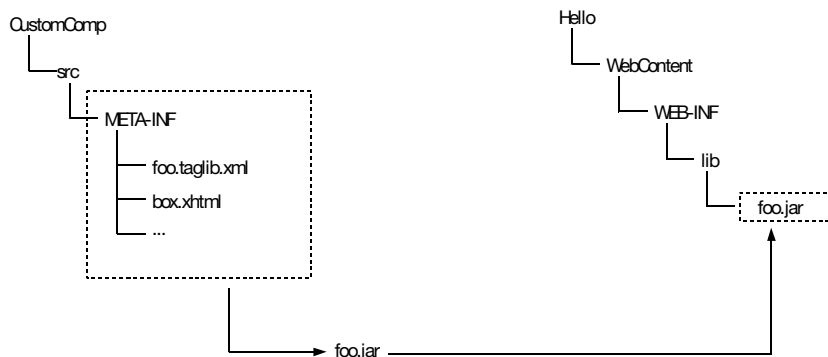


Figure 5-14. *Creating a component library*

To do that, create a new project named `CompUser` as usual. Then right-click the `META-INF` folder in the `CustomComp` project, and choose `Export`. Then choose `Java` ► `JAR File`. Click `Browse` to save the JAR file as `foo.jar` in the `WEB-INF/lib` folder of the `CompUser` project (see Figure 5-15).

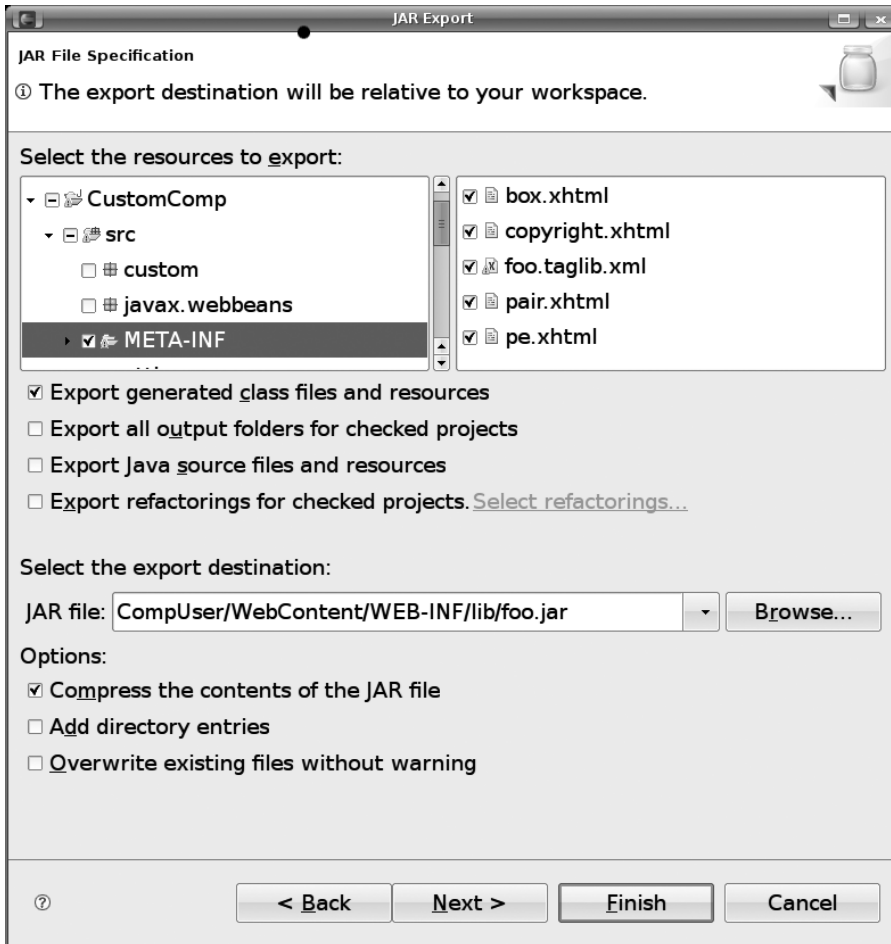


Figure 5-15. *Exporting a JAR file*

In the `CompUser` project, create a `p2.xhtml` file (which stands for “page 2”) in the `WebContent` folder, and try to use the custom tags shown in Listing 5-26.

Listing 5-26. *Using the Custom Tags in Another Project*

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
```

```

"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:foo="http://foo.com">
<foo:box>Testing</foo:box>
</html>

```

When you run the application now, you'll see the "Testing" message appear in a box. Now it should be clear why you need to put the `foo.taglib.xml` file and the source XHTML files into the classpath instead of `WebContent` or `WEB-INF`: they were designed to be packed into a JAR file for reuse.

Creating a Component Library Without `taglib.xml`

For the moment, you're creating a `taglib.xml` file to define a tag lib. However, there is an easier way to do that. For example, let's create another tag lib that has the same `<pe>` tag. Instead of a URL, such an "easy" tag lib is identified by a short name. Let's name it `bar`. To create it, simply create the folder `META-INF/resources/bar` in the classpath.

To define the `<pe>` tag, copy the `pe.xhtml` file into that folder, and modify it as shown in Listing 5-27. The `<composite:interface>` tag assigns a name and a display name to the `<pe>` tag, which are mainly used for visual tools. Then the `<composite:attribute>` tag states that the `<pe>` tag accepts a parameter named `product` whose type is `custom.Product` and that it is a required parameter. Finally, the `<composite:implementation>` tag plays a role similar to `<ui:component>`. That is, it indicates the real content of the component.

Listing 5-27. Defining the `<pe>` Tag

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:composite="http://java.sun.com/jsf/composite">
<body>
<composite:interface name="pe" displayName="Product Editor">
  <composite:attribute name="product" type="custom.Product" required="true"/>
</composite:interface>
<composite:implementation>
  <h:form>
    <h:inputHidden value="#{product.id}"/>
    <h:inputText value="#{product.name}"/>
    <h:commandButton action="#{product.onUpdated}" value="OK"/>

```

```

    </h:form>
</composite:implementation>
</body>
</html>

```

However, in such a component, the parameters are available only in a Map that can be accessed as `compositeComponent.attrs` (see Listing 5-28). Note how you can access a particular element in a Map using its key in an EL expression.

Listing 5-28. *Accessing the Parameters in a Map in an “Easy” Component*

```

...
<composite:implementation>
    <h:form>
        <h:inputHidden value="#{compositeComponent.attrs['product'].id}"/>
        <h:inputText value="#{compositeComponent.attrs['product'].name}"/>
        <h:commandButton
            action="#{compositeComponent.attrs['product'].onUpdated}" .../>
    </h:form>
</composite:implementation>
</body>
</html>

```

To use the bar tag lib in `p1.xhtml`, you need to use its namespace URL. But what's its URL? Because it has only a short name, its URL is always derived from the short name using this pattern: `http://java.sun.com/jsf/composite/<short name>`. In this case, it is `http://java.sun.com/jsf/composite/bar`. So, modify `p1.xhtml` as shown in Listing 5-29.

Listing 5-29. *Using the <pe> Tag in the Bar Tag Lib*

```

<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:foo="http://foo.com"
    xmlns:ui="http://java.sun.com/jsf/facelets"
    xmlns:bar="http://java.sun.com/jsf/composite/bar">
<p>This is page1.</p>
<bar:pe product="#{currentProduct}"/>
<foo:pair>
    <ui:define name="left">
        <foo:pe product="#{currentProduct}"/>
    </ui:define>
</foo:pair>

```

```
</ui:define>
<ui:define name="right">
    <foo:copyright company="Foo"/>
</ui:define>
</foo:pair>
</html>
```

Now, run the application, and it should display a new product editor outside the pair.

Summary

Whenever you see duplicated code in XHTML files, it's high time that you considered extracting the duplicated code into a custom component. To do that, as you learned in this chapter, you can define a tag lib by putting a `*.taglib.xml` file into the `META-INF` folder. For each tag, specify the source XHTML file, and put the duplicated code in that XHTML file.

You also learned that you can pass parameters to a custom tag. At tree construction time, parameters will be formed into a variable table, and the EL expressions in the source file will be linked to it. At render time, they will trace into the variable table to get the value as an EL expression for evaluation.

The EL expression can evaluate to a value/object only. It must not evaluate to a method. To pass a method to the custom tag, pass an object that hosts the method.

You can't pass XHTML code to a custom tag through a parameter and expect JSF to parse the code. To do that, put the code into the body of the tag, or use `<ui:define>` if you have multiple pieces of code. The custom component can trace into the code to build the subtree using `<ui:insert>`.

To reuse custom tags in multiple projects, you can export the `META-INF` folder into a JAR file and reuse it.

Finally, you learned you can create a tag lib without using a `taglib.xml` file by creating a folder using the short name of the tag lib in a special location on the classpath (`META-INF/resources`). The short name is also used to derive the namespace URL. Components in such a tag lib provide more information regarding their parameters for visual tools to use. To access their parameters, the components need to access them from a `Map` using a special name in EL expressions.



Providing a Common Layout for Your Pages

It is commonly required for all the pages in a given application to have a common layout. In this chapter, you'll learn how to apply a common layout to all your pages in your application easily.

Using the Same Menu on Different Pages

Suppose that you'd like to develop an application like the one shown in Figure 6-1. It is not important what the application shown does. What is important is that on each page there is the same menu on the left.

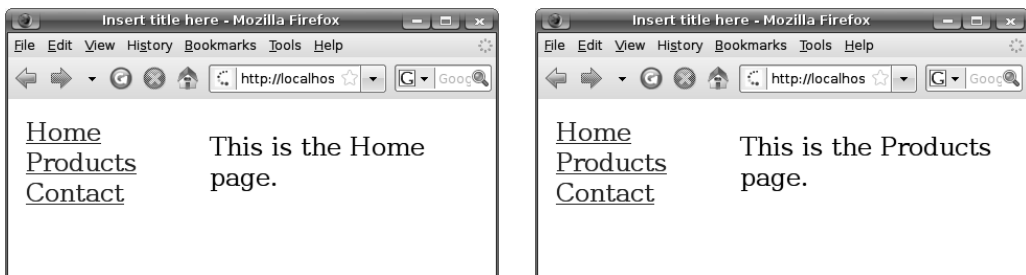


Figure 6-1. *All the pages have a menu on the left.*

To do that, create a new dynamic web project named Layout. You're about to create two pages: `home.xhtml` and `products.xhtml`. Figure 6-2 shows the structure of these pages. You can see that the XHTML page structures are the same, and the only difference is the cell content on the right.

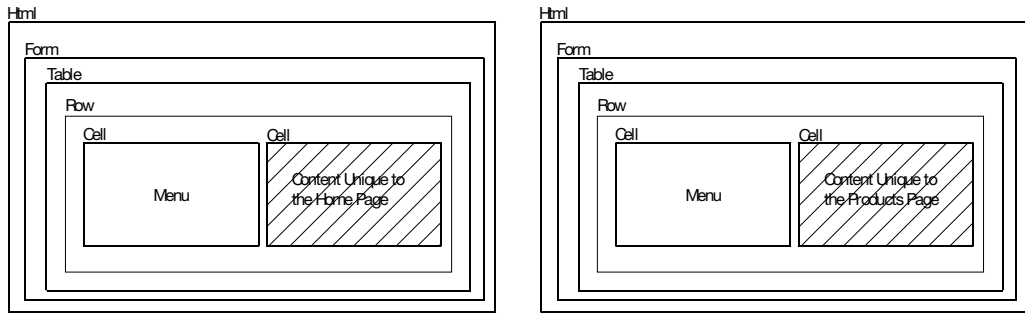


Figure 6-2. Pages sharing a common structure

Whenever you see duplicate XHTML code, you should extract it into a common place. For example, if you extracted it into a component, that would allow you to reuse it in both pages (see Listing 6-1 for `home.xhtml` using the imagined component), but you would still have to duplicate the `<html>` tag, the `<head>` tag, the `<body>` tag, any CSS styles, and so on, as highlighted in Listing 6-1.

Listing 6-1. Using a Custom Component Would Not Remove All Duplicate Code

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:foo="http://foo.com">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
<link rel="stylesheet" type="text/css" href="style1.css"/>
<link rel="stylesheet" type="text/css" href="style2.css"/>
</head>
<body>
<foo:mycomponent>
    This is the Home page.
</foo:mycomponent>
</body>
</html>
```

Instead of having to repeat all that code, you should extract the duplicate code into a base page that contains some abstract parts (see Figure 6-3). Then you can let each page extend the base page and provide its unique content, just like Java class inheritance.

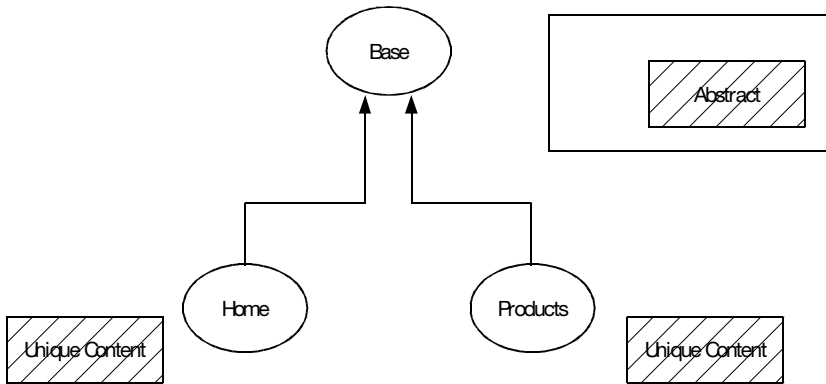


Figure 6-3. *Page inheritance*

To do that, create `base.xhtml` (also in the `WebContent` folder), as shown in Listing 6-2. Note how it uses the `<ui:insert>` tag to insert the abstract part, exactly like what you would do in a custom component in the previous chapter.

Listing 6-2. *The Base Page*

```

<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://java.sun.com/jsf/html"
    xmlns:f="http://java.sun.com/jsf/core"
    xmlns:ui="http://java.sun.com/jsf/facelets">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
<title>Insert title here</title>
</head>
<body>
<h:form>
    <table>
        <tr>
            <td width="40%"><h:commandLink action="home">Home</h:commandLink> <br />
            <h:commandLink action="products">Products</h:commandLink> <br />
            <h:commandLink action="contact">Contact</h:commandLink></td>

```

```

        <td>
            <ui:insert>unique content</ui:insert>
        </td>
    </tr>
</table>
</h:form>
</body>
</html>

```

Create `home.xhtml` to “inherit” `base.xhtml`, as shown in Figure 6-4.

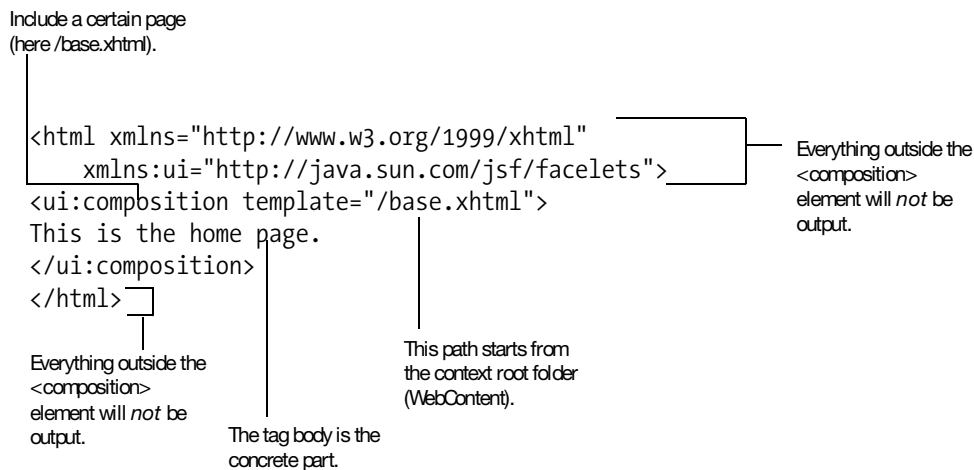


Figure 6-4. “Inheriting” the base page

Run the application by visiting `http://localhost:8080/Layout/faces/home.xhtml`. If it is working as expected, the HTML code should be that of `base.xhtml` except for the abstract part. This is very much like using a custom component except that you don’t need to define a custom tag.

Now create `products.xhtml` in a similar manner. It works in the same way.

In this setup, you can think of it like the base page providing a template with some holes to fill in and the Home page and the Products page using that template and filling those holes with their unique contents.

Using Global Navigation Rules

What is interesting is how to create the navigation rules for the links. As a first attempt, you may try what's shown in Figure 6-5.

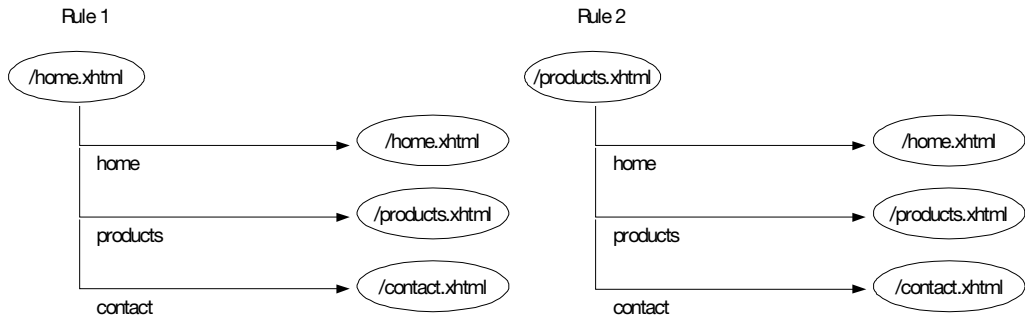


Figure 6-5. *Duplicate navigation rules*

However, that is a lot of duplication. Figure 6-6 shows a better way. Because the star will match any source view ID, the navigation rule could be considered a global fallback rule.

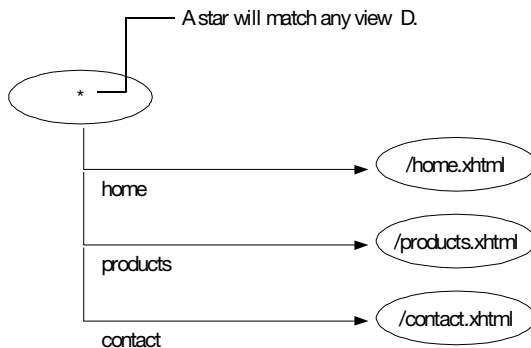


Figure 6-6. *Using a wildcard to match any view ID*

To implement this idea, modify `faces-config.xml` as shown in Listing 6-3.

Listing 6-3. *faces-config.xml Using a Wildcard to Match Any View ID*

```

<?xml version="1.0" encoding="UTF-8"?>
<faces-config ...>

    <navigation-rule>
        <from-view-id>*</from-view-id>
        <navigation-case>
            <from-outcome>products</from-outcome>
            <to-view-id>/products.xhtml</to-view-id>
        </navigation-case>
        <navigation-case>
            <from-outcome>home</from-outcome>
            <to-view-id>/home.xhtml</to-view-id>
        </navigation-case>
    </navigation-rule>
</faces-config>

```

Restart the application, and you'll see that the links work.

Using Two Abstract Parts

Suppose that each page may need to have a particular header that contains HTML elements or even JSF components, as shown in Figure 6-7.

**Figure 6-7.** *Creating a unique header for each page*

Now you'll have a structure that looks like Figure 6-8.

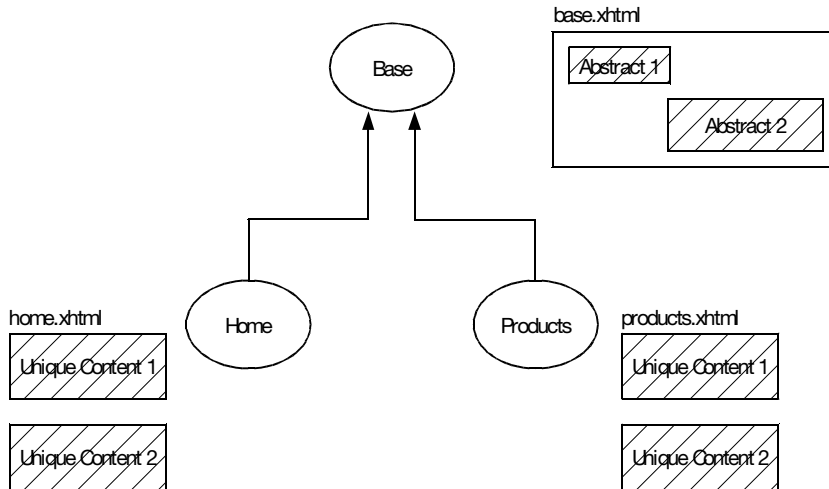


Figure 6-8. *Having two abstract parts*

This is like a base class having two abstract methods. For this to work, you need to give a unique name to each abstract part, as shown in Listing 6-4.

Listing 6-4. *Giving a Unique Name to Each Abstract Part*

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:ui="http://java.sun.com/jsf/facelets">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
<title>Insert title here</title>
</head>
<body>
<ui:insert name="p1">header</ui:insert>
<h:form>
  <table>
    <tr>
      <td width="40%"><h:commandLink action="home">Home</h:commandLink> <br />
      <h:commandLink action="products">Products</h:commandLink> <br />
      <h:commandLink action="contact">Contact</h:commandLink></td>
    <td>
      <ui:insert name="p2">unique content</ui:insert>
    </td>
  </tr>
</table>
</h:form>
</body>
</html>
```

```
        </td>
    </tr>
</table>
</h:form>
</body>
</html>
```

Modify `home.xhtml` to provide the two concrete parts as shown in Listing 6-5. Again, this is exactly like when passing multiple fragments of XHTML tags to a custom component.

Listing 6-5. *Providing Concrete Parts*

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets">
<ui:composition template="/base.xhtml">
    <ui:define name="p1">
        <h1>Home</h1>
    </ui:define>
    <ui:define name="p2">
        This is the home page.
    </ui:define>
</ui:composition>
</html>
```

Run the application, and you'll see it works. Then, modify `products.xhtml` similarly.

Creating Page-Specific Navigation Cases

Suppose that you'd like to have a link on the Products page to display hot deals, as shown in Figure 6-9.



Figure 6-9. *A link to hot deals*

Why is this requirement interesting? Consider where you should put the navigation case. If you put it into the global navigation rule, it would affect all the pages in the application, which is not what you want. To see how to do it, read on. First, modify `products.xhtml` as shown in Listing 6-6.

Listing 6-6. *Putting a JSF Tag in a Concrete Part*

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:h="http://java.sun.com/jsf/html">
  <ui:composition template="/base.xhtml">
    <ui:define name="p1">
      <h1>Products</h1>
    </ui:define>
    <ui:define name="p2">
      This is the Products page.
      Here are some <h:commandLink action="hotDeals">hot
deals</h:commandLink>.
    </ui:define>
  </ui:composition>
</html>
```

There is nothing special here. Create a simple `hotdeals.xhtml` page as shown in Listing 6-7.

Listing 6-7. *The Hot Deals Page*

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  Hot deals here!
</html>
```

Then, create a normal navigation rule for the products page as shown in Listing 6-8.

Listing 6-8. *Navigation Rule for the Products Page*

```
<?xml version="1.0" encoding="UTF-8"?>
<faces-config ...>

  <navigation-rule>
    <from-view-id>*</from-view-id>
```

```

    <navigation-case>
      <from-outcome>products</from-outcome>
      <to-view-id>/products.xhtml</to-view-id>
    </navigation-case>
    <navigation-case>
      <from-outcome>home</from-outcome>
      <to-view-id>/home.xhtml</to-view-id>
    </navigation-case>
  </navigation-rule>
  <navigation-rule>
    <from-view-id>/products.xhtml</from-view-id>
    <navigation-case>
      <from-outcome>hotDeals</from-outcome>
      <to-view-id>/hotdeals.xhtml</to-view-id>
    </navigation-case>
  </navigation-rule>
</faces-config>

```

Now, that is the interesting part: both this rule and the global rule will match the current view ID (/products.xhtml), so which one will take effect? This rule will be considered before the global rule because it is more specific (that is, it doesn't contain any wildcard). Once a navigation case is found, the search will stop. In this case, it means the specific rule will take effect because it is considered first and because it contains a matching navigation case. Now, run the application again, and try to go to the hot deals page. It should work as expected.

Summary

In this chapter, you learned that if you have pages with a common layout, you can extract the common stuff into a base page and mark the abstract parts using `<ui:define>`. Then in each child page, suck in the base page using `<ui:composition>`, and provide each concrete part using `<ui:define>`. Each part should have a unique name. If there is only one abstract part, you can omit the name and provide the concrete part as the body of the `<ui:composition>` element.

You learned that you can use a star as the view ID in a navigation rule. In that case, it will match any view ID. This is useful if multiple pages share the same navigation cases. If a page needs some additional navigation cases, it can have its own normal navigation rule, which will be checked first.

Building Interactive Pages with Ajax

In this chapter, you'll learn how to build pages that are more interactive than normal HTML pages using a technique called Ajax.

Displaying a FAQ

Suppose that you'd like to develop an application that displays a FAQ, as shown in Figure 7-1.

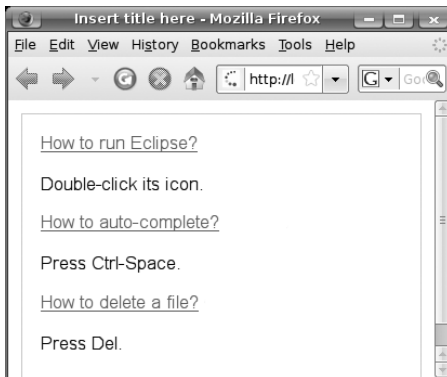


Figure 7-1. *Displaying a FAQ*

Suppose that every question has both a short answer and a long answer. Initially, the short answer is displayed. If the user clicks the question, the answer will change to the long answer. If the user clicks it again, it will change back to the short answer again.

To do that, create a new dynamic web project named FAQ in the same way as you have created projects in the rest of the book. As a first step, you'll show a single question only. So, create a `listfaq.xhtml` page as shown in Listing 7-1.

Listing 7-1. *listfaq.xhtml*

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://java.sun.com/jsf/html"
    xmlns:f="http://java.sun.com/jsf/core">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
<title>Insert title here</title>
</head>
<body>
<h:form>
    <h:commandLink action="#{faqService.trigger}"
        value="#{faqService.questionText}" />
</h:form>
<br />
<h:outputText value="#{faqService.answerText}" />
</body>
</html>

```

Create the `FAQService` class in the `faq` package as shown in Listing 7-2.

Listing 7-2. *FAQService Class*

```

package faq;
...
@Named("faqService")
@SessionScoped
public class FAQService implements Serializable {
    private String questionText = "How to run Eclipse?";
    private String answerTextShort = "Double-click its icon.";
    private String answerTextLong = "Double-click its icoooooooooooooooooooooon.";
    private boolean isShortForm = true;

    public String getQuestionText() {
        return questionText;
    }
    public String getAnswerText() {
        return isShortForm ? answerTextShort : answerTextLong;
    }
    public String trigger() {

```

```
        isShortForm = !isShortForm;  
        return null;  
    }  
}
```

Note that this class is in the session scope (and thus needs to implement `Serializable`). Why? If it were in the request scope, then a new bean would be created for each request and the `isShortForm` flag would always be true. In addition, the `trigger()` method returns `null` so that the current page is redisplayed after the link is clicked.

Now run it. Clicking the question will change the form of the answer between the short and long forms.

Refreshing the Answer Only

Note that for the moment, whenever the user clicks the link, the progress bar in the browser will go from 0 to 100 percent, as shown in Figure 7-2. This indicates that the whole HTML page is refreshed.



Figure 7-2. Progress bar indicating a full-page refresh

This can be made better. For example, you could refresh the answer only, not the whole page. This way the response will feel much faster and thus provide a better user experience. To do that (see Figure 7-3), you need to generate some JavaScript for the `onclick` event of the question link (the `<a>` element). Don't worry about how to actually do it now; it will become clear later in the chapter. When the link is clicked, that JavaScript will send a request to your application. This will cause the action method of the UI Command component to execute, which will change the form of the answer. Then only the UI Output component is asked to render again (not the whole page). The UI Output component will generate some HTML code, which will be returned to the JavaScript in the browser. The JavaScript will use that HTML code to update the answer. This entire process is called Ajax.

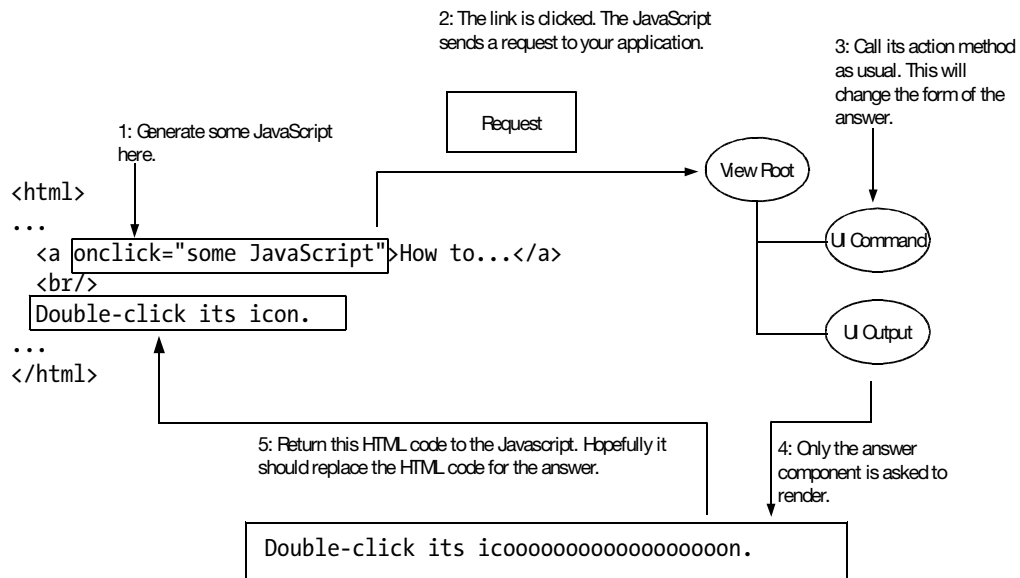


Figure 7-3. *How Ajax works*

However, there is still a problem: how does the JavaScript know which part of the page to update? To solve this problem, you need to explicitly assign an ID to the component being refreshed (the UI Output component in this case), as shown in Figure 7-4. This will cause the component to generate an HTML element with a client ID. Then the JavaScript can use the client ID to look up the existing HTML element and replace it.

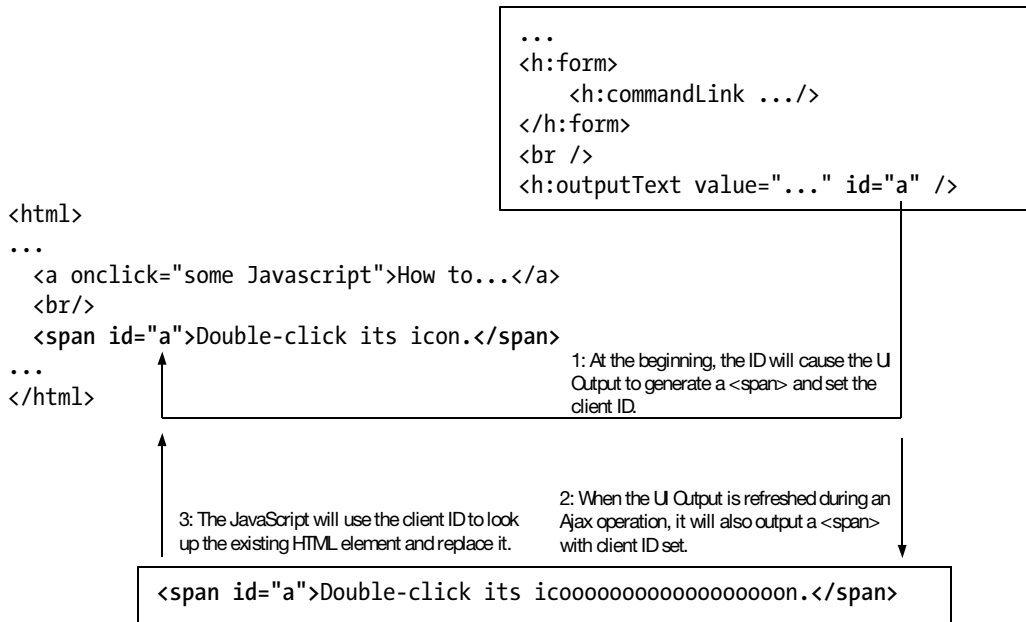


Figure 7-4. Assigning an ID to identify the HTML element to be updated

To implement these ideas, you need to use a JSF component library called RichFaces from JBoss. Go to <http://labs.jboss.com/jbossrichfaces> to download it. It may be something like `richfaces-ui-3.3.0-GA-bin.zip`. Unzip the download into, say, `c:\richfaces-ui`. To use it, copy all the JAR files in `c:\richfaces-ui\lib` into your `WEB-INF/lib`. RichFaces in turn needs a few third-party JAR files. You can download them from the Source Code section of the Apress web site (<http://www.apress.com>) and unzip the files into `WEB/lib`. Finally, refresh the project in Eclipse.

The 3.3 version of RichFaces doesn't support JSF 2.0 yet. Because you installed JSF 2.0 into the JBoss application server in Chapter 1, you can't run the application being developed on top of it. To solve this problem, you may want to download a clean copy of the JBoss application server again and unzip it into, say, `c:\jboss`. Then go to <https://facelets.dev.java.net/servlets/ProjectDocumentList> to download the latest version of Facelets. It has been included in JSF 2.0, but because you aren't using JSF 2.0, you need to include it yourself. The file may be something like `facelets-1.1.15.b1`. Unzip it into, say, `c:\facelets`. Finally, copy `c:\facelets\jsf-facelets.jar` into your `WEB-INF/lib`, and refresh the project in Eclipse. Before JSF 2.0, Facelets is an add-on that needs to be enabled. To do that, modify `faces-config.xml` as shown in Listing 7-3.

Listing 7-3. *Enabling Facelets in faces-config.xml (Before JSF 2.0)*

```

<faces-config ...>
  <application>
    <view-handler>com.sun.facelets.FaceletViewHandler
    </view-handler>
  </application>
</faces-config>

```

Next, modify WEB-INF/web.xml as shown in Listing 7-4.

Listing 7-4. *Changes to web.xml in Order to Use RichFaces*

```

<?xml version="1.0" encoding="UTF-8"?>
<web-app ...>
  ...
  <servlet>
    <servlet-name>JSF</servlet-name>
    <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>JSF</servlet-name>
    <url-pattern>/faces/*</url-pattern>
  </servlet-mapping>
  <context-param>
    <param-name>org.richfaces.SKIN</param-name>
    <param-value>blueSky</param-value>
  </context-param>
  <context-param>
    <param-name>org.richfaces.CONTROL_SKINNING</param-name>
    <param-value>enable</param-value>
  </context-param>
  <filter>
    <display-name>RichFaces Filter</display-name>
    <filter-name>richfaces</filter-name>
    <filter-class>org.ajax4jsf.Filter</filter-class>
  </filter>
  <filter-mapping>
    <filter-name>richfaces</filter-name>
    <servlet-name>JSF</servlet-name>
    <dispatcher>REQUEST</dispatcher>
    <dispatcher>FORWARD</dispatcher>

```

```

    <dispatcher>INCLUDE</dispatcher>
  </filter-mapping>
</web-app>

```

You don't need to completely understand the meaning of this code, but basically, it is used to enable RichFaces to intercept requests so that it can deliver JavaScript to the browser. Next, modify `listfaq.xhtml` as shown in Figure 7-5. In summary, the `<a4j:commandLink>` is used like a normal `<h:commandLink>`, but it will generate some JavaScript into the `onclick` attribute of the HTML link. That JavaScript will call the action method on the server and then rerender the component specified in its `reRender` attribute.

```

<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:a4j="http://richfaces.org/a4j">
  <head>
    Use the <commandLink> in the Ajax4Jsf tag lib.
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
    <title>Insert title here</title>
  </head>
  <body>
    <h:form>
      <a4j:commandLink action="#{faqService.trigger}"
                     value="#{faqService.questionText}"
                     reRender="a" />
      1: It will set the onclick
      event handler to some
      JavaScript.
      2: If clicked, it will call
      this trigger() method on
      the server side.
    </h:form>
    <br />
    3: Then, ask this "a" component only to render again.
    <h:outputText value="#{faqService.answerText}" id="a" />
  </body>
</html>

```

Figure 7-5. Using `<a4j:commandLink>`

When you run the application, clicking the question will change the form of the answer, while this time no progress bar will be displayed in the browser, indicating that the page as a whole is not refreshed.

Hiding and Showing the Answer

Suppose that instead of changing the form of the answer, now you'd like to hide or show the answer. For simplicity, let's do it without Ajax first. Modify `listfaq.xhtml` as shown in Listing 7-5.

Listing 7-5. *Using the rendered Attribute*

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:a4j="http://richfaces.org/a4j">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
<title>Insert title here</title>
</head>
<body>
<h:form>
    <h:commandLink
        action="#{faqService.trigger}"
        value="#{faqService.questionText}" />
</h:form>
<br />
<h:outputText value="#{faqService.answerText}" id="a"
    rendered="#{faqService.showingAnswer}" />
</body>
</html>

```

What does that `rendered` attribute do? When the UI Output component is about to render itself in the Render Response phase, it will check this attribute. In this case, it will call `isShowingAnswer()` on the web bean. If `isShowingAnswer()` returns `true`, it will go ahead to render itself. Otherwise, it will do nothing. For it to work, modify the web bean as shown in Listing 7-6.

Listing 7-6. *Showing or Hiding the Answer in the Web Bean*

```

package faq;
...
@Named("faqService")
@SessionScoped
public class FAQService implements Serializable {
    private String questionText = "How to run Eclipse?";
    private String answerText = "Double-click its icon.";
    private boolean isShowingAnswer = false;

    public String getQuestionText() {

```

```

        return questionText;
    }
    public String getAnswerText() {
        return answerText;
    }
    public String trigger() {
        isShowingAnswer = !isShowingAnswer;
        return null;
    }
    public boolean isShowingAnswer() {
        return isShowingAnswer;
    }
}

```

When you run the application, clicking the question should show and hide the answer. Of course, now the whole page is refreshed because Ajax is not used.

Using Ajax to Hide or Show the Answer

Now, you'd like to use Ajax to hide or show the answer, without refreshing the whole page. A first attempt is shown in Listing 7-7.

Listing 7-7. *Trying to Use Ajax to Change the Visibility of a Component*

```

<!DOCTYPE ...>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:a4j="http://richfaces.org/a4j">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
<title>Insert title here</title>
</head>
<body>
<h:form>
    <a4j:commandLink
        action="#{faqService.trigger}"
        value="#{faqService.questionText}"
        reRender="a" />
</h:form>
<br />

```

```

<h:outputText value="#{faqService.answerText}" id="a"
    rendered="#{faqService.showingAnswer}"/>
</body>
</html>

```

If you run it, initially no answer will be shown, which is the correct behavior. However, if you click the link, nothing seems to happen. This is because when the UI Output component is about to render itself initially, the `rendered` attribute is false so it will not output any HTML element (see Figure 7-6). As a result, the `onclick` JavaScript will be unable to find the HTML element to update in the subsequent Ajax operations.

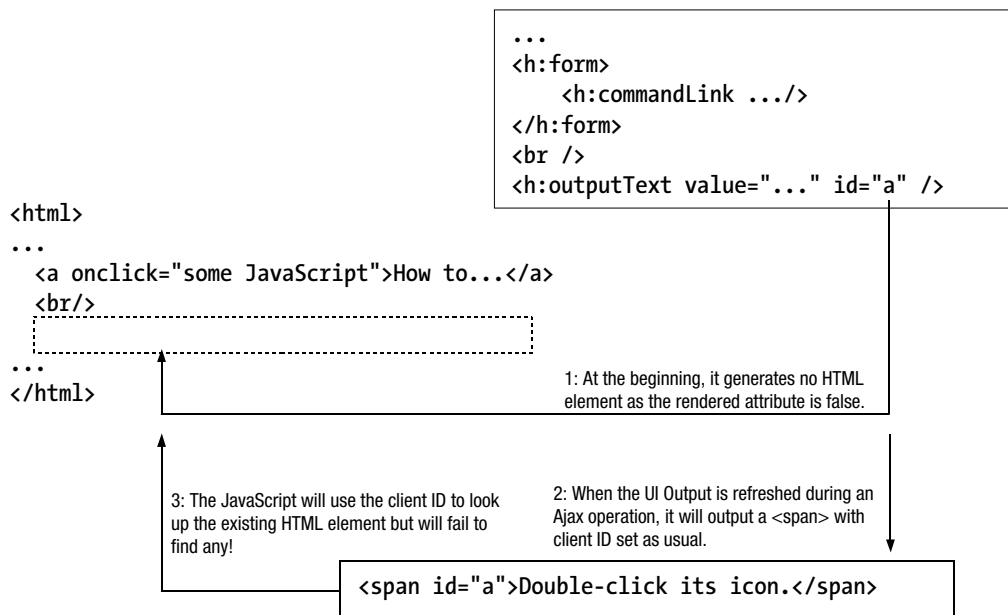


Figure 7-6. *Why Ajax will fail to show a component*

To solve this problem, you'll place a panel surrounding the answer and use Ajax to update the panel (see Figure 7-7). Simply put, that HTML `<span>` element generated by the panel will always be there. It may contain nothing in its body (if the answer is hidden) or contain the answer (if the answer is shown). As it is always there, you can use Ajax to update it.

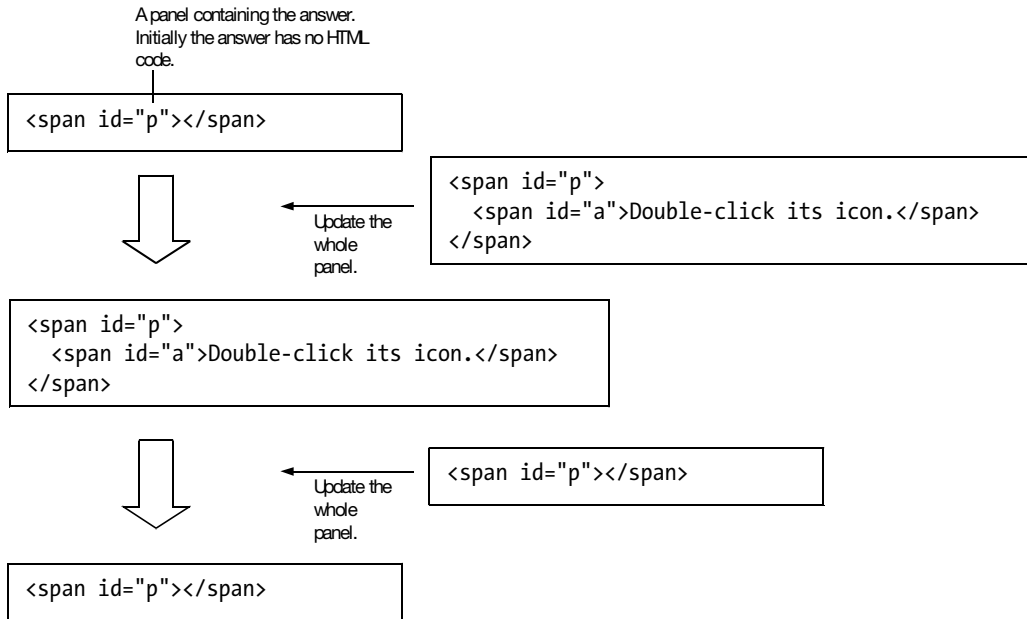


Figure 7-7. Using a panel to show or hide a component

To implement this idea, modify `listfaq.xhtml` as shown in Listing 7-8.

Listing 7-8. Refreshing the Whole Panel Group in `listfaq.xhtml`

```
<!DOCTYPE ...>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:a4j="http://richfaces.org/a4j">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
<title>Insert title here</title>
</head>
```

```

<body>
<h:form>
    <a4j:commandLink action="#{faqService.trigger}"
        value="#{faqService.questionText}"
        reRender="p" />
</h:form>
<br />
<h:panelGroup id="p">
    <h:outputText value="#{faqService.answerText}" id="a"
        rendered="#{faqService.showingAnswer}" />
</h:panelGroup>
</body>
</html>

```

Now when you run the application, clicking the question will show or hide the answer using Ajax.

Giving a Rating to a Question

Suppose that you'd like to allow the user to rate the helpfulness of the question (and its answer). The latest rating is displayed at the end of the question, as shown in Figure 7-8. Again, you don't want to refresh the whole page, but just the question. This is just like clicking the `<a4j:commandLink>` to update the answer, except that now you need to process some user input (the rating).



Figure 7-8. Getting and displaying a rating from the user

To do that, you can use an `<a4j:commandButton>` as shown in Figure 7-9. Simply put, an `<a4j:commandButton>` is just like an `<a4j:commandLink>` except that it will render itself as an HTML button instead of an HTML link.

```

...
<h:form>
  <a4j:commandLink action="#{faqService.trigger}"
    value="#{faqService.questionText}({#{faqService.rating}})" reRender="p"
    id="q" />
</h:form>
<h:form style="float:right">
  <h:inputText size="2" value="#{faqService.rating}"></h:inputText>
  <a4j:commandButton value="Rate" action="#{faqService.rate}"
    reRender="q">
  </a4j:commandButton>
</h:form>
<br />
<h:panelGroup id="p">
  <h:outputText value="#{faqService.answerText}" id="a"
    rendered="#{faqService.showingAnswer}" />
</h:panelGroup>
</body>
</html>

```

Refresh the question.

This is just a normal HTML thing to put the form to the right.

This part is just a literal expression that will be output as is.

Literal expression again.

`<commandButton>` from the Ajax4Jsf tag lib. It will put JavaScript into the onclick handler of the HTML submit button.

Figure 7-9. Using `<a4j:commandButton>`

Modify the `FAQService` class as shown in Listing 7-9.

Listing 7-9. Maintaining a Rating in `FAQService`

```

@Named("faqService")
@SessionScoped
public class FAQService implements Serializable {
    private String questionText = "How to run Eclipse?";
    private String answerText = "Double-click its icon.";
    private boolean isShowingAnswer = false;
    private int rating = 0;

```

```
public String getQuestionText() {
    return questionText;
}
public String getAnswerText() {
    return answerText;
}
public String trigger() {
    isShowingAnswer = !isShowingAnswer;
    return null;
}
public boolean isShowingAnswer() {
    return isShowingAnswer;
}
public String rate() {
    System.out.println("Setting rating to: " + rating);
    return null;
}
public int getRating() {
    return rating;
}
public void setRating(int rating) {
    this.rating = rating;
}
}
```

Now run the application, and you should be able to give a rating to the question.

What if the user enters something invalid such as “abc” as the rating? Then you should display an error. To do that, as a first attempt, modify `listfaq.xhtml` as shown in Listing 7-10. Note how to refresh two (or more) components in the `reRender` attribute: just list their IDs using a comma as the separator.

Listing 7-10. *Refreshing Both the Question and the <h:messages>*

```

<!DOCTYPE ...>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:a4j="http://richfaces.org/a4j">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
<title>Insert title here</title>
</head>
<body>
<b:h:messages id="m"/>
<h:form>
    <a4j:commandLink action="#{faqService.trigger}"
        value="#{faqService.questionText} ({#{faqService.rating})" reRender="p"
        id="q" />
</h:form>
<h:form style="float:right">
    <h:inputText size="2" value="#{faqService.rating}"></h:inputText>
    <a4j:commandButton value="Rate" action="#{faqService.rate}"
        reRender="q,m">
    </a4j:commandButton>
</h:form>
<br />
<h:panelGroup id="p">
    <h:outputText value="#{faqService.answerText}" id="a"
        rendered="#{faqService.showingAnswer}" />
</h:panelGroup>
</body>
</html>

```

However, this will not work, because if there is no error message, the UI Messages component will not generate any HTML code at all. It means subsequent Ajax operations will be unable to find the HTML element to update. To solve this problem, put that UI Messages component into a panel and update the panel instead (see Listing 7-11).

Listing 7-11. *Putting the <h:messages> Inside a Panel*

```

<!DOCTYPE ...>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:a4j="http://richfaces.org/a4j">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
<title>Insert title here</title>
</head>
<body>
<h:panelGroup id="mp">
    <h:messages id="m"/>
</h:panelGroup>
<h:form>
    <a4j:commandLink action="#{faqService.trigger}"
        value="#{faqService.questionText} ({#{faqService.rating})" reRender="p"
        id="q" />
</h:form>
<h:form style="float:right">
    <h:inputText size="2" value="#{faqService.rating}"></h:inputText>
    <a4j:commandButton value="Rate" action="#{faqService.rate}"
        reRender="q,mp">
    </a4j:commandButton>
</h:form>
<br />
<h:panelGroup id="p">
    <h:outputText value="#{faqService.answerText}" id="a"
        rendered="#{faqService.showingAnswer}" />
</h:panelGroup>
</body>
</html>

```

Now when you run the application and enter some garbage as the rating, it should be caught as an error.

Updating the Rating as the User Types

What if you'd like to update the rating as the user types (even using the Rate button)? You can do it as shown in Figure 7-10. In summary, the `<a4j:support>` tag is like the `<a4j:commandLink>` tag except that you can specify the name of the JavaScript event to control when to trigger the Ajax operation.

```

...
<h:panelGroup id="mp">
    <h:messages id="m" />
</h:panelGroup>
<h:form>
    <a4j:commandLink action="#{faqService.trigger}"
        value="#{faqService.questionText} (#{faqService.rating})"
        reRender="p" id="q" />
</h:form>
<h:form style="float:right">
    <h:inputText size="2" value="#{faqService.rating}">
        ^
        |
        |_ <a4j:support
            |_ event="onkeyup"           Whenever a key is up..
            |_ action="#{faqService.rate}" Call this action method on the
            |_ reRender="q,mp"           server side.
        </a4j:support>
        |_ Refresh the question and the
        |_ message panel.
    </h:inputText>
    <a4j:commandButton value="Rate" action="#{faqService.rate}"
        reRender="q,mp">
        |_ You don't need the Rate button
        |_ anymore.
    </a4j:commandButton>
</h:form>
<br />
<h:panelGroup id="p">
    <h:outputText value="#{faqService.answerText}" id="a"
        rendered="#{faqService.showingAnswer}" />
</h:panelGroup>
</body>
</html>

```

`<a4j:support>` will add JavaScript to the `<input>` HTML element generated by its parent component.

Figure 7-10. Using `<a4j:support>`

When you now run the application and type some value as the rating, the value should appear immediately after the question text.

Using a Dialog Box to Get the Rating

Suppose that you'd like to display a dialog box to get the rating: the user clicks a Rate link, which pops up a dialog box, and then the user can input the rating (see Figure 7-11) and click the Rate button to close the dialog box. Finally, the rating for the question will be refreshed.

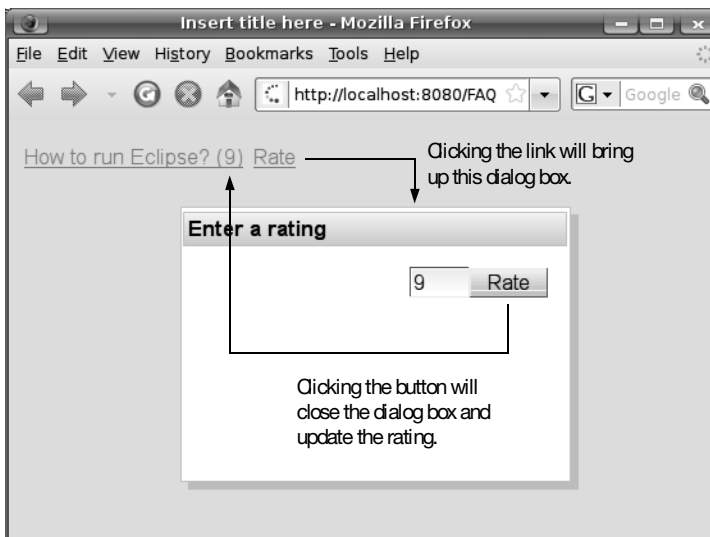


Figure 7-11. *Using a dialog box*

To do that, modify `listfaq.xhtml` as shown in Figure 7-12. Simply put, the `<rich:modalPanel>` tag will create a modal panel that is initially hidden on the HTML page.

```

<!DOCTYPE ...>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:a4j="http://richfaces.org/a4j"
      xmlns:rich="http://richfaces.org/rich">
  ...
  <h:panelGroup id="mp">
    <h:messages id="m" />
  </h:panelGroup>
  <h:form>
    <a4j:commandLink action="#{faqService.trigger}"
      value="#{faqService.questionText} (#{faqService.rating})"
      reRender="p" id="q" />
  </h:form>
  <rich:modalPanel id="myDialog">
    <h:form style="float:right">
      <h:inputText size="2" value="#{faqService.rating}" />
      <a4j:commandButton
        value="Rate"
        action="#{faqService.rate}"
        reRender="q,mp">
      </a4j:commandButton>
    </h:form>
  </rich:modalPanel>
  <br />
  <h:panelGroup id="p">
    <h:outputText value="#{faqService.answerText}" id="a"
      rendered="#{faqService.showingAnswer}" />
  </h:panelGroup>
</body>
</html>

```

It is in the "rich" tag lib in RichFaces.

A dialog box is called a modal panel. Initially, it is in a hidden state in the HTML page in the browser.

Use a button again.

Everything here will appear in the modal panel.

Figure 7-12. Using `<rich:modalPanel>`

But how do you show the modal panel, and how do you hide it later? This is *not* done by Ajax. Instead, it is done using JavaScript (see Listing 7-12). Note the oncomplete property of `<a4j:button>`. It specifies a piece of JavaScript that will be executed after the HTML elements have been updated in the browser. Here, you will hide the modal panel.

Listing 7-12. Showing and Hiding a Modal Panel

```

...
<h:panelGroup id="mp">
  <h:messages id="m" />
</h:panelGroup>

```

```

<h:form>
  <a4j:commandLink action="#{faqService.trigger}"
    value="#{faqService.questionText} ({#{faqService.rating})"
    reRender="p" id="q" />
  <a href="javascript:Richfaces.showModalPanel('myDialog')">Rate</a>
</h:form>
<rich:modalPanel id="myDialog">
  <h:form style="float:right">
    <h:inputText size="2" value="#{faqService.rating}"/>
    <a4j:button
      value="Rate"
      action="#{faqService.rate}"
      reRender="q,mp"
      oncomplete="Richfaces.hideModalPanel('myDialog')">
    </a4j:button>
  </h:form>
</rich:modalPanel>
<br />
<h:panelGroup id="p">
  <h:outputText value="#{faqService.answerText}" id="a"
    rendered="#{faqService.showingAnswer}" />
</h:panelGroup>
</body>
</html>

```

When you run the application now and click the Rate link, it should pop up the modal panel, and you should be able to enter the rating. However, if the user enters some garbage and clicks the Rate button, the code will go ahead and close the modal panel and display the error in the main page. A more proper behavior is to display the error in the modal panel and not close it. To do that, modify the code as shown in Listing 7-13. What you have done is to move the panel group into the modal panel and to check whether there is no error message (with the JavaScript function called `getElementById`) before hiding the modal panel.

Listing 7-13. Hiding a Modal Panel Only If There Is No Error

```

<!DOCTYPE ...>
<html xmlns="http://www.w3.org/1999/xhtml"
  xmlns:h="http://java.sun.com/jsf/html"
  xmlns:f="http://java.sun.com/jsf/core"
  xmlns:a4j="http://richfaces.org/a4j"
  xmlns:rich="http://richfaces.org/rich">

```

```

<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
<title>Insert title here</title>
</head>
<body>
<h:form>
    <a4j:commandLink action="#{faqService.trigger}"
        value="#{faqService.questionText} ({#{faqService.rating})"
        reRender="p" id="q" />
    <a href="javascript:Richfaces.showModalPanel('myDialog')">Rate</a>
</h:form>
<rich:modalPanel id="myDialog">
    <h:panelGroup id="mp">
        <h:messages id="m" />
    </h:panelGroup>
    <h:form style="float:right">
        <h:inputText size="2" value="#{faqService.rating}"/>
        <a4j:commandButton
            value="Rate"
            action="#{faqService.rate}"
            reRender="q,mp"
            onComplete="if (document.getElementById('m')==null)
Richfaces.hideModalPanel('myDialog')">
        </a4j:commandButton>
    </h:form>
</rich:modalPanel>
<br />
<h:panelGroup id="p">
    <h:outputText value="#{faqService.answerText}" id="a"
        rendered="#{faqService.showingAnswer}" />
</h:panelGroup>
</body>
</html>

```

Run the application now, and try to input something invalid as the rating. An error message should be displayed in the modal panel, and the modal panel should remain displayed. Finally, you may want to give the modal panel a title bar so that the user can drag to move it. This is done by giving it a facet named “header” (Listing 7-14). In addition, you may want to set its initial size to a smaller size such as 200 pixels × 140 pixels.

Listing 7-14. *Giving the Modal Panel a Header and Setting Its Initial Size*

```

...
<h:form>
    <a4j:commandLink action="#{faqService.trigger}"
        value="#{faqService.questionText} ({#{faqService.rating})"
        reRender="p" id="q" />
    <a href="javascript:Richfaces.showModalPanel('myDialog')">Rate</a>
</h:form>
<rich:modalPanel id="myDialog" width="200" height="140">
    <f:facet name="header">
        <h:outputText value="Enter a rating"></h:outputText>
    </f:facet>
    <h:panelGroup id="mp">
        <h:messages id="m" />
    </h:panelGroup>
    <h:form style="float:right">
        <h:inputText size="2" value="#{faqService.rating}"/>
        <a4j:commandButton
            value="Rate"
            action="#{faqService.rate}"
            reRender="q,mp"
            oncomplete="if (document.getElementById('m')==null)
Richfaces.hideModalPanel('myDialog')">
        </a4j:commandButton>
    </h:form>
</rich:modalPanel>
<br />
<h:panelGroup id="p">
    <h:outputText value="#{faqService.answerText}" id="a"
        rendered="#{faqService.showingAnswer}" />
</h:panelGroup>
</body>
</html>

```

Run the application, and you'll see that the modal panel should have a header and that it's the right size.

Setting the Look and Feel with Skins

All RichFaces components support so-called skins. For example, a particular skin named `s1` might have been defined as shown in Table 7-1.

Table 7-1. *Definition of a Skin*

Attribute	Value
Text font	Arial
Text color	Blue
Text size	12pt
Background color	Yellow

RichFaces comes with several predefined skins. They're named `blueSky`, `classic`, `deepMarine`, and so on. To choose which one to use, use `web.xml` as shown in Listing 7-15.

Listing 7-15. *Choosing the Skin to Use*

```
<web-app ...>
    ...
    <context-param>
        <param-name>org.richfaces.SKIN</param-name>
        <param-value>blueSky</param-value>
    </context-param>
    ...
</web-app>
```

In addition, by default the selected skin is also applied to the input-related HTML elements such as `<a>` or `<input>` generated by normal JSF components. However, it is not applied to others such as `<span>`. To make sure the whole page uses the skin, group everything into a `<rich:panel>` as shown in Listing 7-16. This way, everything inside that panel will inherit its CSS styles.

Listing 7-16. *Grouping Everything into a `<rich:panel>`*

```
<!DOCTYPE ...>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:rich="http://richfaces.org/rich">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
<title>Insert title here</title>
</head>
<body>
<rich:panel>
```

```

<h:form>
  <a4j:commandLink action="#{faqService.trigger}"
    value="#{faqService.questionText} ({#{faqService.rating})"
    reRender="p" id="q" />
  <a href="javascript:Richfaces.showModalPanel('myDialog')">Rate</a>
</h:form>
<rich:modalPanel id="myDialog" width="200" height="140">
  <f:facet name="header">
    <h:outputText value="Enter a rating"></h:outputText>
  </f:facet>
  <h:panelGroup id="mp">
    <h:messages id="m" />
  </h:panelGroup>
  <h:form style="float:right">
    <h:inputText size="2" value="#{faqService.rating}"/>
    <a4j:commandButton
      value="Rate"
      action="#{faqService.rate}"
      reRender="q,mp"
      oncomplete="if (document.getElementById('m')==null)
Richfaces.hideModalPanel('myDialog')">
    </a4j:commandButton>
  </h:form>
</rich:modalPanel>
<br />
<h:panelGroup id="p">
  <h:outputText value="#{faqService.answerText}" id="a"
    rendered="#{faqService.showingAnswer}" />
</h:panelGroup>
</rich:panel>
</body>
</html>

```

Displaying Multiple Questions

So far, you've simply displayed a single question and answer, but now you're about to display multiple questions. However, because the tags for a single question are getting quite complicated, it is desirable to encapsulate them inside a custom tag/component. Then `listfaq.xhtml` can be simplified as shown in Listing 7-17. It is assumed that `q1` is a `Question` object containing the question text, answer text, and so on.

Listing 7-17. *Using a Custom Tag in listfaq.xhtml*

```

<!DOCTYPE ...>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:rich="http://richfaces.org/rich"
      xmlns:foo="http://foo.com">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
<title>Insert title here</title>
</head>
<body>
<rich:panel>
    <foo:qa question="#{faqService.q1}"/>
</rich:panel>
</body>
</html>

```

To do that, move most of the tags from listfaq.xhtml into qa.xhtml in the src/META-INF folder, and then modify the code as shown in Listing 7-18. The main changes are to take the information from a Question object passed through the question parameter.

Listing 7-18. *qa.xhtml*

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:a4j="http://richfaces.org/a4j"
      xmlns:rich="http://richfaces.org/rich"
      xmlns:ui="http://java.sun.com/jsf/facelets">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
<title>Insert title here</title>
</head>

```

```

<body>
<ui:component>
<h:form>
    <a4j:commandLink
        action="#{question.trigger}"
        value="#{question.questionText} ({#{question.rating})"
        reRender="p" id="q" />
    <a href="javascript:Richfaces.showModalPanel('myDialog')">Rate</a>
</h:form>
<rich:modalPanel id="myDialog" width="200" height="140">
    <f:facet name="header">
        <h:outputText value="Enter a rating"/>
    </f:facet>
    <h:panelGroup id="mp">
        <h:messages id="m" />
    </h:panelGroup>
    <h:form style="float:right">
        <h:inputText size="2" value="#{question.rating}"/>
        <a4j:commandButton
            value="Rate"
            action="#{question.rate}"
            reRender="q,mp"
            oncomplete="if (document.getElementById('m')==null)
Richfaces.hideModalPanel('myDialog')">
        </a4j:commandButton>
    </h:form>
</rich:modalPanel>
<br />
<h:panelGroup id="p">
    <h:outputText id="a"
        value="#{question.answerText}"
        rendered="#{question.showingAnswer}" />
</h:panelGroup>
</ui:component>
</body>
</html>

```

Next, create the tag lib definition file `foo.taglib.xml` in the same META-INF folder as shown in Listing 7-19.

Listing 7-19. *foo.taglib.xml*

```
<!DOCTYPE facelet-taglib PUBLIC
    "-//Sun Microsystems, Inc.//DTD Facelet Taglib 1.0//EN"
    "http://java.sun.com/dtd/facelet-taglib_1_0.dtd">
<facelet-taglib xmlns="http://java.sun.com/JSF/Facelet" >
    <namespace>http://foo.com</namespace>
    <tag>
        <tag-name>qa</tag-name>
        <source>qa.xhtml</source>
    </tag>
</facelet-taglib>
```

Create a `Question` class as shown in Listing 7-20. It is basically a copy of the `FAQService` class except that it has a constructor and is not a web bean (because you should have multiple `Question` objects, it makes no sense to make it a web bean).

Listing 7-20. *Question Class*

```
package faq;

import java.io.Serializable;

public class Question implements Serializable {
    private String questionText;
    private String answerText;
    private boolean isShowingAnswer = false;
    private int rating = 0;

    public Question(String questionText, String answerText) {
        this.questionText = questionText;
        this.answerText = answerText;
    }
    public String getQuestionText() {
        return questionText;
    }
    public String getAnswerText() {
        return answerText;
    }
    public String trigger() {
        isShowingAnswer = !isShowingAnswer;
        return null;
    }
}
```

```

    }
    public boolean isShowingAnswer() {
        return isShowingAnswer;
    }
    public String rate() {
        System.out.println("Setting rating to: " + rating);
        return null;
    }
    public int getRating() {
        return rating;
    }
    public void setRating(int rating) {
        this.rating = rating;
    }
}

```

Modify FAQService as shown in Listing 7-21.

Listing 7-21. *Providing a Single Question Object in FAQService*

```

package faq;

import javax.webbeans.Named;
import javax.webbeans.SessionScoped;

@Named("faqService")
@SessionScoped
public class FAQService implements Serializable {
    private Question q1 = new Question("How to run Eclipse?", "Double-click its
    icon.");
    public Question getQ1() {
        return q1;
    }
}

```

Go ahead and modify listfaq.xhtml as shown in Listing 7-17. Then run the application, and it should continue to work.

Note In Facelets there is a bug preventing *.taglib.xml files in the META-INF folder on the classpath to be discovered in JBoss. To work around it, put the whole META-INF folder into WebContent and then explicitly specify the tag lib in web.xml as shown in Listing 7-22.

Listing 7-22. *Explicitly Specifying a Tag Lib*

```

<?xml version="1.0" encoding="UTF-8"?>
<web-app ...>
    ...
    <servlet>
        <servlet-name>JSF</servlet-name>
        <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>JSF</servlet-name>
        <url-pattern>/faces/*</url-pattern>
    </servlet-mapping>
    <context-param>
        <param-name>facelets.LIBRARIES</param-name>
        <param-value>META-INF/foo.taglib.xml</param-value>
    </context-param>
</web-app>

```

Finally, modify `listfaq.xhtml` to display multiple questions as shown in Listing 7-23.

Listing 7-23. *Using `<a4j:repeat>` to Loop Through the Questions*

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:rich="http://richfaces.org/rich"
    xmlns:a4j="http://richfaces.org/a4j"
    xmlns:foo="http://foo.com">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
<title>Insert title here</title>
</head>
<body>
<rich:panel>
    <a4j:repeat value="#{faqService.questions}" var="q">
        <foo:qa
            question="#{q}" />
        <p />
    </a4j:repeat>
</rich:panel>
</body>
</html>

```

The `<a4j:repeat>` tag is exactly like the `<h:dataTable>` tag, except that it will loop over its body while not generating tags like `<table>`, `<tr>`, or `<td>`. Next, modify `FAQService` to provide the question list as shown in Listing 7-24.

Listing 7-24. *Providing a List of Question Objects in `FAQService`*

```
package faq;
...
@Named("faqService")
@SessionScoped
public class FAQService implements Serializable {
    private List<Question> questions;

    public FAQService() {
        questions = new ArrayList<Question>();
        questions.add(new Question("How to run Eclipse?", "Double-click its
icon."));
        questions.add(new Question("How to auto-complete?", "Press Ctrl-
Space."));
        questions.add(new Question("How to delete a file?", "Press Del."));
    }
    public List<Question> getQuestions() {
        return questions;
    }
}
```

When you run the application now, it should display all three questions.

Note Currently there is a bug in the JSF reference implementation preventing a form from working if it is in a loop.

Summary

In this chapter, you learned how to build pages with Ajax. Ajax means that when a certain event occurs in the browser, a request is sent to the application so that it can perform some action and then only parts of a page are refreshed. You can use an `<a4j:commandLink>` for a link, an `<a4j:commandButton>` for a button, or an `<a4j:support>` for any other events. In these tags, you also specify an action method to execute in the application and a list of component IDs that are to be refreshed.

A component can be excluded from rendering. In that case, it will generate nothing. If you need to show it using Ajax, you can put it inside a panel and refresh that panel instead. Similarly, some components such as the UI Messages component may output nothing in normal use. To update them using Ajax, put them inside a panel.

In addition, you learned how to show or hide a modal panel using JavaScript.

You also learned about skins in this chapter. A skin defines a look and feel including the font family, font size, color, and so on. All RichFaces components support skins. The selected skin by default will also cover HTML code generated by normal JSF components.



Using Conversations

In the previous chapters, you learned how to use web beans to maintain states for a request (request scope), for a user (session scope), or for the whole application (application scope). In this chapter, you'll learn how to use a very powerful type of scope provided by web beans: *conversation scope*. It allows you to maintain a different state on the server for a browser window.

Creating a Wizard to Submit Support Tickets

Suppose you'd like to develop a wizard that allows the user to submit a support ticket, as shown in Figure 8-1. That is, the user enters her customer ID at step 1 and enters the problem description at step 2. After submitting the ticket, a "Thank you!" page is displayed. What is interesting is that the user can use the Next button and Back button to go back and forth between the pages.

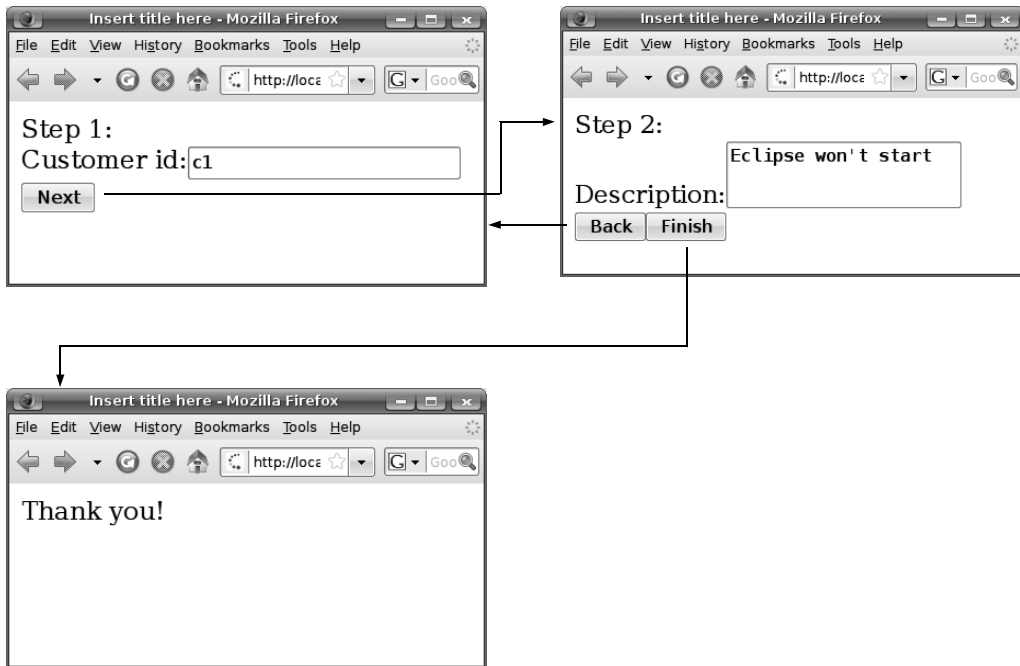


Figure 8-1. *Submitting a ticket using a wizard interface*

To start building this wizard, create a new dynamic web project named Wizard. Then create a Ticket class in the wizard package, as shown in Listing 8-1. Note that it is in the session scope so that it will be available across the two wizard steps.

Listing 8-1. *Ticket Class*

```
package wizard;

import java.io.Serializable;
import javax.annotation.Named;
import javax.context.SessionScoped;

@Named("ticket")
@SessionScoped
public class Ticket implements Serializable {
    private String customerId;
    private String problemDesc;
```

```

public String getCustomerId() {
    return customerId;
}
public void setCustomerId(String customerId) {
    this.customerId = customerId;
}
public String getProblemDesc() {
    return problemDesc;
}
public void setProblemDesc(String problemDesc) {
    this.problemDesc = problemDesc;
}
}

```

Create the `step1.xhtml`, `step2.xhtml`, and `thankyou.xhtml` files in the `WebContent` folder, as shown in Listing 8-2, Listing 8-3, and Listing 8-4, respectively. There is nothing special about them except the `<h:inputTextarea>` tag used in `step2.xhtml`. The `<h:inputTextarea>` tag is exactly like the `<h:inputText>` tag except that it will generate an HTML input text area so that the user can enter multiple lines of text.

Listing 8-2. *step1.xhtml*

```

<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://java.sun.com/jsf/html">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
<title>Insert title here</title>
</head>
<body>
Step 1:
<h:form>
    Customer id: <h:inputText value="#{ticket.customerId}"/><br/>
    <h:commandButton value="Next" action="next"/>
</h:form>
</body>
</html>

```

Listing 8-3. *step2.xhtml*

```

<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://java.sun.com/jsf/html">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
<title>Insert title here</title>
</head>
<body>
Step 2:
<h:form>
    Description: <h:inputTextarea value="#{ticket.problemDesc}"/><br/>
    <h:commandButton value="Back" action="back"/>
    <h:commandButton value="Finish" action="finish"/>
</h:form>
</body>
</html>

```

Listing 8-4. *thankyou.xhtml*

```

<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://java.sun.com/jsf/html">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
<title>Insert title here</title>
</head>
<body>
Thank you!
</body>
</html>

```

Define the navigation rules in `faces-config.xml`, as shown in Listing 8-5.

Listing 8-5. *Navigation Rules for the Wizard*

```
<faces-config ...>
  <navigation-rule>
    <from-view-id>/step1.xhtml</from-view-id>
    <navigation-case>
      <from-outcome>next</from-outcome>
      <to-view-id>/step2.xhtml</to-view-id>
    </navigation-case>
  </navigation-rule>
  <navigation-rule>
    <from-view-id>/step2.xhtml</from-view-id>
    <navigation-case>
      <from-outcome>back</from-outcome>
      <to-view-id>/step1.xhtml</to-view-id>
    </navigation-case>
    <navigation-case>
      <from-outcome>finish</from-outcome>
      <to-view-id>/thankyou.xhtml</to-view-id>
    </navigation-case>
  </navigation-rule>
</faces-config>
```

Now, start the JBoss instance, and try to access `http://localhost:8080/Wizard/faces/step1.xhtml` in the browser. It should allow you to input the ticket, and the Next and Back buttons should work.

Interference Between Browser Windows

Now, let's run an experiment. Enter `c1` as the customer ID, and then go to step 2. Then open a new browser tab, and display step 1. You'll see "c1" displayed as the customer ID. This means the two browser windows are working on the same ticket object. This is not good, because the application won't allow a customer to work on two or more tickets at the same time.

Assuming that you should really allow customers to work on multiple tickets at the same time, you need to know about something called a *conversation*. Whenever a request arrives, Web Beans will allocate a web beans table in the session for the browser window (see Figure 8-2). This table is the conversation. Such a conversation will be ended (destroyed) automatically after the response is rendered.

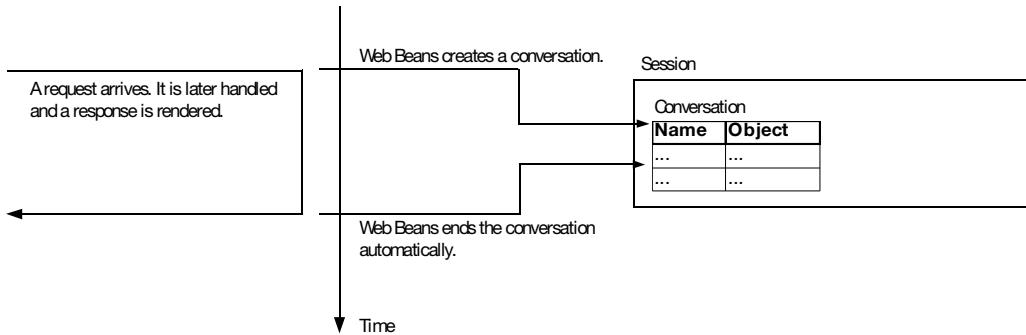


Figure 8-2. *A conversation*

Such a transient conversation is not that useful because it is so short-lived; what you put into it is just like those request-scoped web beans. However, before the response is generated, if you tell web beans to turn the conversation into a long-running one, it will hang in there until when, maybe many requests later, you turn it back into a transient one; then it will be destroyed at the end of the request (see Figure 8-3).

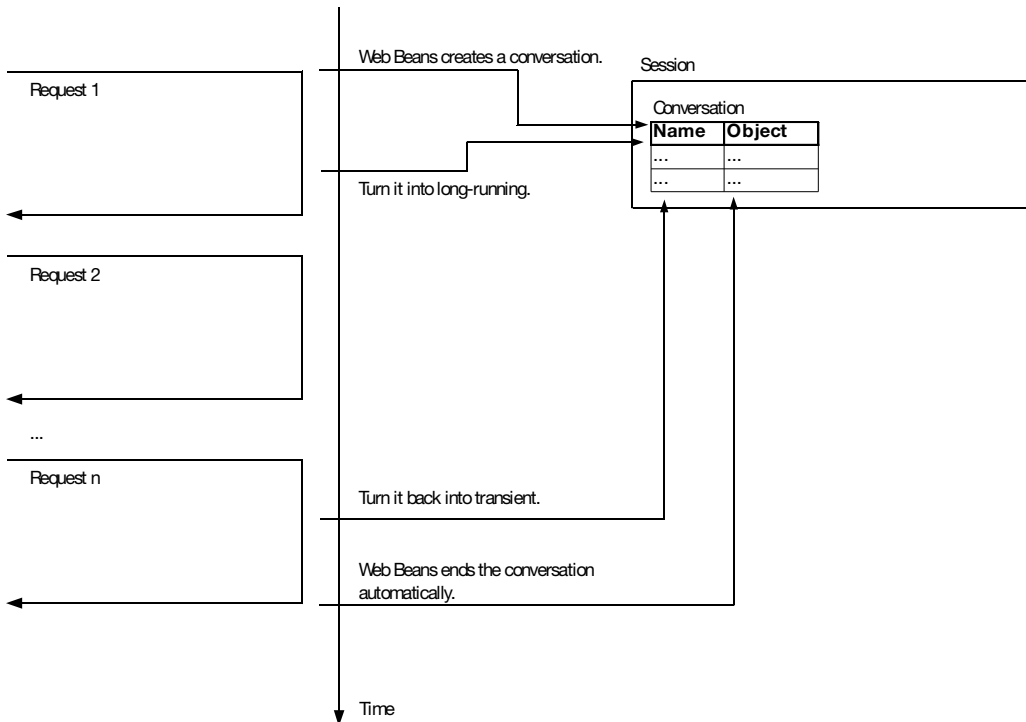


Figure 8-3. *Making a conversation long-running*

The interesting thing about conversations is that if in the middle of the (long-running) conversation you open a new browser tab and access the application, a new conversation will be created, and the two conversations will not interfere with each other (see Figure 8-4).

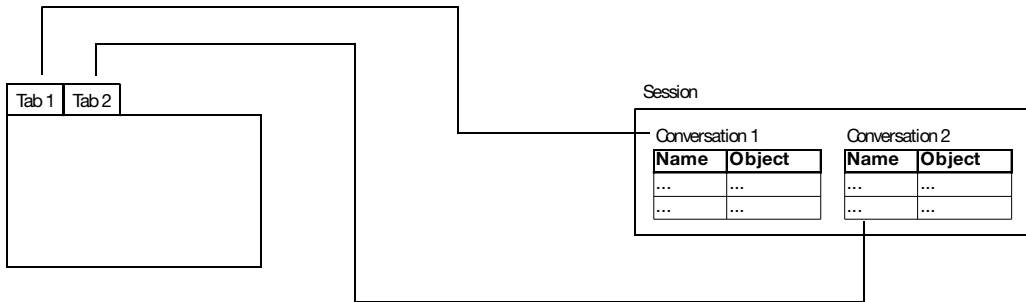


Figure 8-4. *Each tab having a different conversation*

To implement this idea, you need to put the ticket into the conversation scope, as shown in Listing 8-6. Because a conversation is still stored in the session, the ticket still needs to implement `Serializable`.

Listing 8-6. *Putting the Ticket into the Conversation Scope*

```
package wizard;

import java.io.Serializable;
import javax.annotation.Named;
import javax.context.ConversationScoped;

@Named("ticket")
@ConversationScoped
public class Ticket implements Serializable {
    private String customerId;
    private String problemDesc;

    public String getCustomerId() {
        return customerId;
    }

    public void setCustomerId(String customerId) {
        this.customerId = customerId;
    }

    public String getProblemDesc() {
        return problemDesc;
    }
}
```

```

    }
    public void setProblemDesc(String problemDesc) {
        this.problemDesc = problemDesc;
    }
}

```

Next, you need to consider when to turn the conversation into a long-running one. You can do this when the user clicks the Next button at step 1. After that, the ticket will be available across multiple requests. To do that, modify `step1.xhtml` as shown in Listing 8-7.

Listing 8-7. *Invoking a Java Method When Clicking Next*

```

<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://java.sun.com/jsf/html">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
<title>Insert title here</title>
</head>
<body>
Step 1:
<h:form>
    Customer id: <h:inputText value="#{ticket.customerId}"/><br/>
    <h:commandButton value="Next" action="#{step1.next}"/>
</h:form>
</body>
</html>

```

Create a `Step1` class in the `wizard` package to provide the `next()` method, as shown in Listing 8-8.

Listing 8-8. *Step1 Class*

```

package wizard;

import javax.annotation.Named;
import javax.context.RequestScoped;

@Named("step1")
@RequestScoped

```

```
public class Step1 {  
    public String next() {  
        //TURN THE CONVERSATION INTO LONG-RUNNING;  
        return "next";  
    }  
}
```

To turn the conversation into a long-running one, modify the Step1 class as shown in Listing 8-9. First you inject the conversation object (as described in Chapter 4), and then you call `begin()` on it. `begin()` is actually a misnomer because the conversation has already begun; what this method does is turn the conversation into a long-running one.

Listing 8-9. *Turning a Conversation into a Long-Running One*

```
package wizard;  
  
import javax.annotation.Named;  
import javax.context.RequestScoped;  
import javax.context.Conversation;  
import javax.inject.Current;  
  
@Named("step1")  
@RequestScoped  
public class Step1 {  
    @Current  
    private Conversation c;  
  
    public String next() {  
        c.begin();  
        return "next";  
    }  
}
```

Next, you need to consider when to turn the conversation back into transient. You can do this when the user clicks the Finish button at step 2. After that, the ticket will be destroyed along with the conversation at the end of the request. To do that, modify `step2.xhtml` as shown in Listing 8-10.

Listing 8-10. *Invoking a Java Method When Clicking Finish*

```

<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://java.sun.com/jsf/html">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
<title>Insert title here</title>
</head>
<body>
Step 2:
<h:form>
    Description: <h:inputTextarea value="#{ticket.problemDesc}"/><br/>
    <h:commandButton value="Back" action="back"/>
    <h:commandButton value="Finish" action="#{step2.finish}"/>
</h:form>
</body>
</html>

```

Create a `Step2` class in the `wizard` package in order to provide the `finish()` method, as shown in Listing 8-11. The critical part is the call to the `end()` method on the conversation object, which will turn the conversation into transient. Another point to note is that instead of really submitting the ticket (to a database, for example), you simply print its content to the console.

Listing 8-11. *Step2 Class*

```

package wizard;

import javax.annotation.Named;
import javax.context.Conversation;
import javax.context.RequestScoped;
import javax.inject.Current;

@Named("step2")
@RequestScoped

```

```
public class Step2 {
    @Current
    private Conversation c;
    @Current
    private Ticket ticket;

    public String finish() {
        submit(ticket);
        c.end();
        return "finish";
    }
    private void submit(Ticket ticket) {
        System.out.println(ticket.getCustomerId());
        System.out.println(ticket.getProblemDesc());
    }
}
```

Now, when you run the application again, try to submit two tickets simultaneously in two browser tabs. The two tickets will not interfere with each other.

URL Mismatched?

If you're careful, you may have noticed that after clicking Next at step 1, step 2 is indeed displayed, but the URL is still pointing to step 1. Why? When you first enter the URL to `step1.xhtml` and press Enter (see Figure 8-5), a request is sent, and `step1.xhtml` will generate a response. So, you see step 1 in the browser window. But what is the action attribute of the HTML form? If the user clicks the Next button, you'll definitely want the original component tree of step 1 to handle the form submission so that, for example, the various UI Input components can update their web beans using the user input in the request. Therefore, the action attribute should still point to `step1.xhtml`, which is indeed what is done by the UI Form component. That is, it will set the action attribute to invoke the then-current page. Now, when the user clicks the Next button, the browser will copy that action attribute (for `step1.xhtml`) into the location URL and then send the request to the `step1.xhtml` for handling. `step1.xhtml` will handle it and then pass the rendering (through the JSF navigation system) to `step2.xhtml`. As a result, you'll see the step 2 in the browser window, but the location URL is still for `step1.xhtml`.

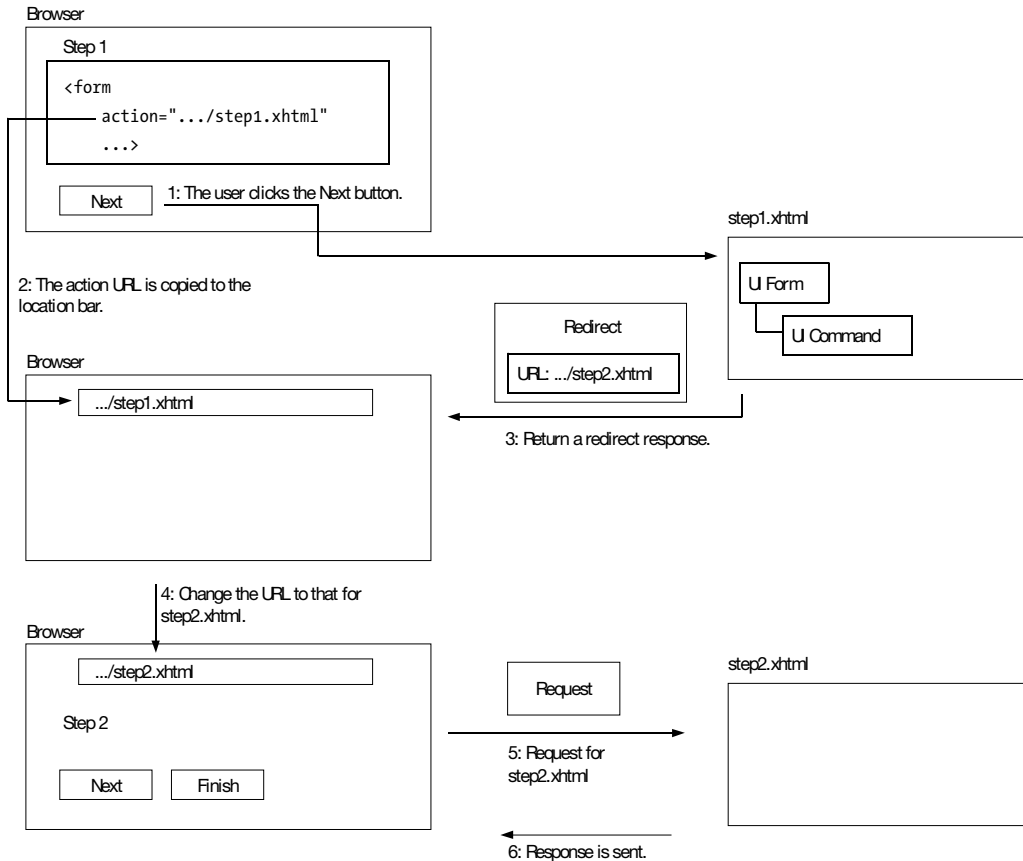


Figure 8-6. *Redirect response*

When applying this technique (“redirect after post”), a very important requirement is that `step2.xhtml` needs to still have access to its data for rendering. If the data were in the request scope, it would have been gone because a new request was triggered by the redirect response. However, because you’re using the conversation scope, the data (the ticket) will still be there.

To tell `step1.xhtml` to send a redirect for `step2.xhtml`, all you need to do is modify the navigation case as shown in Listing 8-12.

Listing 8-12. *Using Redirect in Navigation Case*

```
<faces-config ...>
  <navigation-rule>
    <from-view-id>/step1.xhtml</from-view-id>
    <navigation-case>
      <from-outcome>next</from-outcome>
```

```

        <to-view-id>/step2.xhtml</to-view-id>
        <redirect/>
    </navigation-case>
</navigation-rule>
<navigation-rule>
    <from-view-id>/step2.xhtml</from-view-id>
    <navigation-case>
        <from-outcome>back</from-outcome>
        <to-view-id>/step1.xhtml</to-view-id>
    </navigation-case>
    <navigation-case>
        <from-outcome>finish</from-outcome>
        <to-view-id>/thankyou.xhtml</to-view-id>
    </navigation-case>
</navigation-rule>
</faces-config>

```

Restart the application, and run it again. Go to step 2, and observe that the URL should also be showing `step2.xhtml`.

Note In Web Beans beta 1, there is a bug preventing a long-running conversation from spanning across a redirect response.

Similarly, you can apply the same technique for the Back button and the Finish button, as shown in Listing 8-13.

Listing 8-13. *Using Redirect for Other Buttons*

```

<faces-config ...>
    <navigation-rule>
        <from-view-id>/step1.xhtml</from-view-id>
        <navigation-case>
            <from-outcome>next</from-outcome>
            <to-view-id>/step2.xhtml</to-view-id>
            <redirect/>
        </navigation-case>
    </navigation-rule>
    <navigation-rule>
        <from-view-id>/step2.xhtml</from-view-id>
        <navigation-case>

```

```
<from-outcome>back</from-outcome>
<to-view-id>/step1.xhtml</to-view-id>
<redirect/>
</navigation-case>
<navigation-case>
  <from-outcome>finish</from-outcome>
  <to-view-id>/thankyou.xhtml</to-view-id>
  <redirect/>
</navigation-case>
</navigation-rule>
</faces-config>
</faces-config>
```

Run the application again, and the URL should change correctly with each button click.

Summary

Web Beans provides a very powerful scope: conversation scope. As you learned in this chapter, a conversation is a table in the session for a given browser tab/window. You can make a conversation long-running and later turn it back into Java code. During its lifetime, all conversation-scoped web beans will remain available across different requests.

To render the next page, you can choose between a simple render or a redirect. A redirect will show the new URL in the browser and will work fine with the Reload/Refresh button. To allow a redirect, if one page needs to pass a web bean to the next page, it should be in the conversation scope.



Supporting Other Languages

In this chapter, you'll learn how to develop an application that can appear in two or more different languages to suit users in different countries.

Displaying the Current Date and Time

Suppose that you have an application that displays the current date and time, as shown in Figure 9-1.

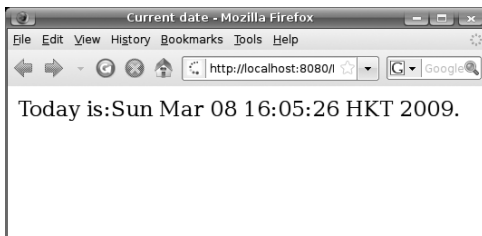


Figure 9-1. *Displaying the current date and time*

To do that, create a new dynamic web project as usual named MultiLang. Then create a `showdate.xhtml` file, as shown in Listing 9-1.

Listing 9-1. *showdate.xhtml*

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<?xml version="1.0" encoding="UTF-8" ?>
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://java.sun.com/jsf/html">
```

```

<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
<title>Current date</title>
</head>
<body>
Today is: <h:outputText value="#{showDate.today}"/>.
</body>
</html>

```

Create the `ShowDate` class in the `multilang` package, and create a web bean from it, as shown in Listing 9-2.

Listing 9-2. *ShowDate Class*

```

package multilang;
...
import java.util.Date;

@Named("showDate")
@RequestScoped
public class ShowDate {
    public Date getToday() {
        return new Date();
    }
}

```

Start the JBoss instance, and try to access `http://localhost:8080/MultiLang/faces/showdate.xhtml` in the browser. It should display the current date and time correctly. (Note that your application may display the date differently than in Figure 9-1, because this depends on the default language configured on your computer.)

Supporting Chinese

Suppose that some of your users are Chinese. They would like to see the application in Chinese when they run the application. To do that, create a file called `msgs.properties` (the file name is not really important as long as it ends with `.properties`) in the `multilang` package, as shown in Listing 9-3.

Listing 9-3. *Providing Text Messages in a .properties File*

```

currentDate=Current date
todayIs=Today is:

```

To support Chinese, create another file, called `msgs_zh.properties`. The `zh` part represents Chinese. Usually people use the Big5 encoding to encode Chinese. However, Java requires that such files be in a special encoding called *escaped Unicode encoding*. For example, the Chinese equivalent of “Current date” consists of four Unicode characters, as shown in Figure 9-2. The figure also shows their Unicode values (hexadecimal). The properties file should be written as their Unicode values.

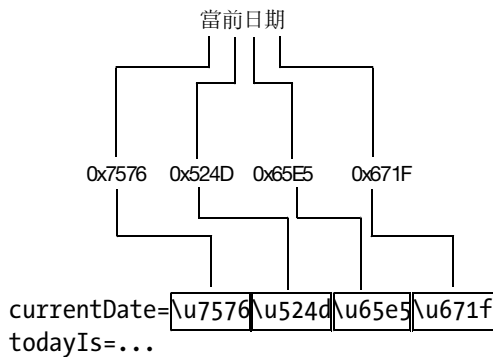


Figure 9-2. Characters written as their Unicode values

Obviously, this is not very convenient when you have lots of strings to encode. Fortunately, there is a free properties file editor available for Eclipse that allows you to type in the text while it saves the text in the escaped Unicode encoding automatically. To install this editor, select **Help ► Software Updates** in Eclipse, choose the Available Software tab, click **Add Site**, and enter **<http://propedit.sourceforge.jp/eclipse/updates>** as the location. Choose the Properties Editor, and click **Install** to start the installation.

To use it, right-click the `msgs_zh.properties` file, and choose **Open With ► PropertiesEditor**. For those readers who don’t know how to input Chinese, you can simply type some random text and pretend it’s Chinese. Listing 9-4 shows the authentic Chinese text messages as would be displayed by the Properties Editor.

Listing 9-4. Chinese Text Messages That Would Be Displayed by the Properties Editor

```
currentDate=當前日期
todayIs=今日是：
```

To make use of the `.properties` files, modify `showdate.xhtml` as shown in Figure 9-3. That is, the `<f:loadBundle>` tag will create a UI Load Bundle component. When the UI View Root component asks the UI Load Bundle component to render, instead of generating HTML code, it will find out the preferred language as set in the HTTP request (suppose that it is `zh`). Then, because the `basename` attribute in the tag specified is `multilang.msgs`, it will go into the `multilang` folder on the classpath and look for the file `msgs_<language>.properties`,

which is `msgs_zh.properties` in this case. Then it will load the text messages into a resource bundle. Finally, as specified by the `var` attribute in the tag, it will store that resource bundle into a request-scoped attribute named `b`.

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core">
<head>
<f:loadBundle basename="multilang.msgs" var="b" />
<title>Current date</title>
</head>
<body>
Today is: <h:outputText .../>.
</body>
</html>
```

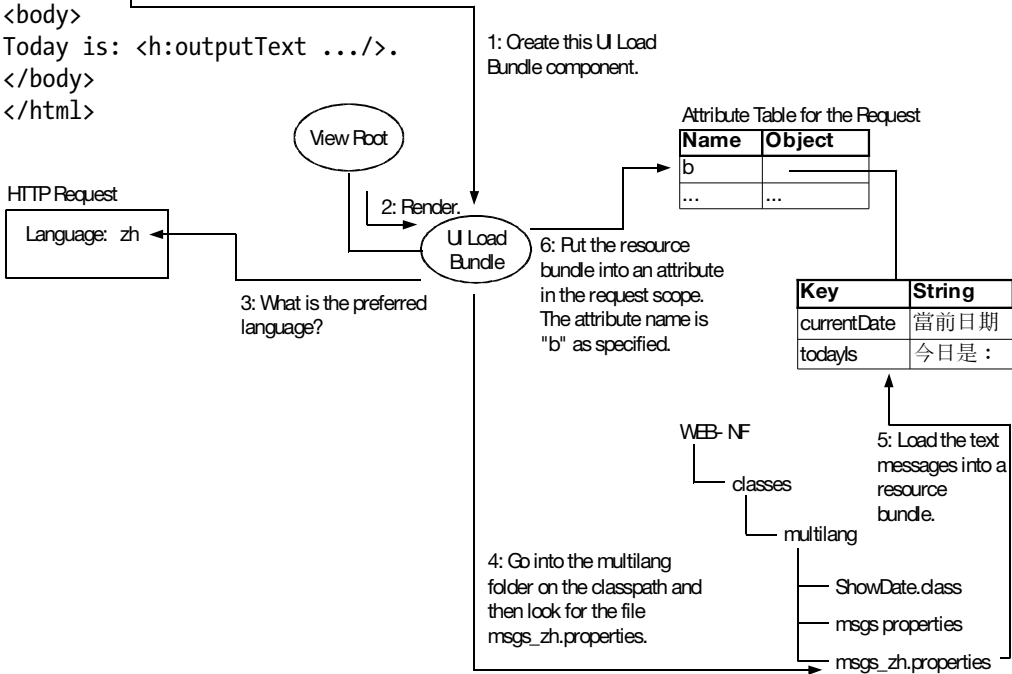


Figure 9-3. Loading a `.properties` file into a resource bundle

To read messages from the resource bundle, access it like a map, as shown in Listing 9-5.

Listing 9-5. Changes to `web.xml` in Order to Use RichFaces

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
```

```

<f:loadBundle basename="multilang.msgs" var="b" />
<title>
<h:outputText value="#{b['currentDate']}" />
</title>
</head>
<body>
<h:outputText value="#{b['todayIs']}" />
<h:outputText value="#{showDate.today}" />
</body>
</html>

```

When you run the application, it should display the current date and time in English.

How do you make the page use the Chinese version of the resource bundle (`msgs_zh.properties`)? For example, in Firefox, choose Tools ► Options ► Content, add Chinese, and move it to the top. However, that is still not enough: JSF will screen the preferred language as set in the HTTP request to see whether it is supported by your application (see Figure 9-4). For example, in the figure, if the preferred language set in the HTTP request were `en` or `zh`, then it would be supported and used. However, if it were, say, `de` (German), then it would not be supported, and the default language would be used (`en` here). No matter which case it is, the language chosen to be used will be stored into the UI View Root. The UI Load Bundle will consult the UI View Root instead of the HTTP request for the language.

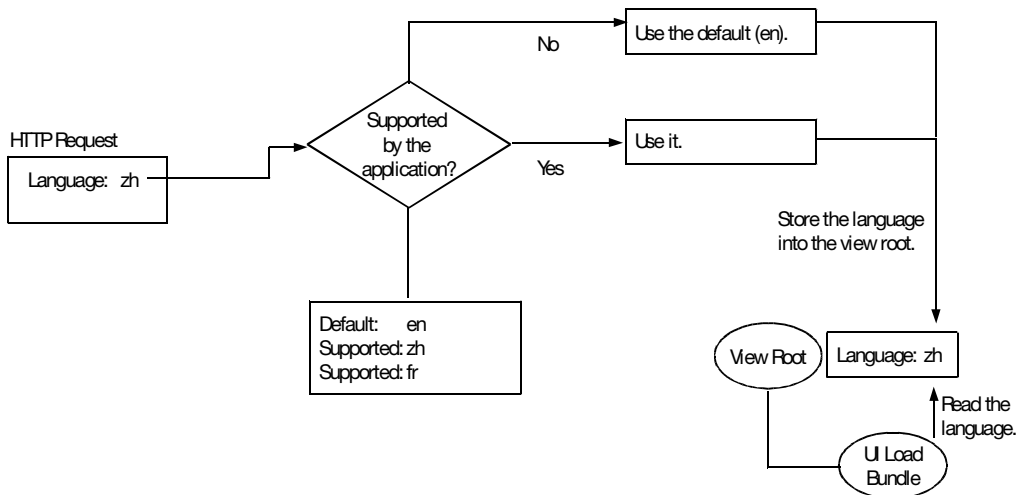


Figure 9-4. JSF screening the preferred language

To specify the default language and supported languages, modify `faces-config.xml` as shown in Listing 9-6. Here, you are supporting both English and Chinese with English as the default.

Listing 9-6. *Listing the Supported and Default Language in faces-config.xml*

```

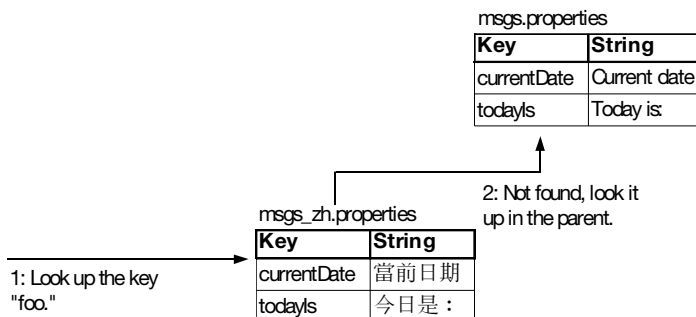
<faces-config ...>
  <application>
    <locale-config>
      <default-locale>en</default-locale>
      <supported-locale>zh</supported-locale>
    </locale-config>
  </application>
</faces-config>

```

Save the file, restart the JBoss instance, and then reload the page. You should see the Chinese version. If you don't see the Chinese characters, make sure your computer has a font that supports Chinese. For example, log in as the administrator, open the Control Panel, choose Regional Settings, and ensure that traditional Chinese support is enabled.

You may be wondering what happens when the preferred language is en and there is no `msgs_en.properties` file, as is the case in our example. You might think the request will get past the JSF screening and that the UI Load Bundle component will try to load the `msgs_en.properties` file and fail miserably because that file doesn't exist.

To understand how it works, first consider the case when the preferred language is zh. In that case, it will load `msgs_zh.properties` and then use `msgs.properties` as the parent resource bundle (see Figure 9-5). When a child resource bundle is looked up for a key but it is not found, the child will look for it in the parent.

**Figure 9-5.** *Parent-child relationship between resource bundles*

Now, consider the case when the preferred language is en. In that case, it tries to load `msgs_en.properties`, but it is not found (see Figure 9-6). Then you can consider it will use the nonexistent `msgs_en.properties` as the child resource bundle, and effectively only the parent resource bundle will be used.

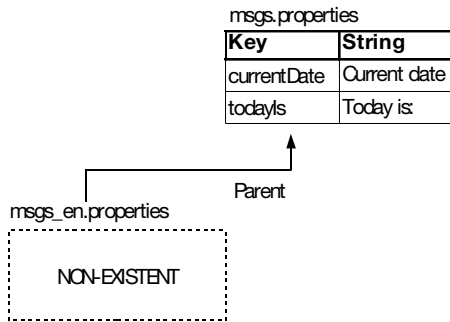


Figure 9-6. A nonexistent child resource bundle

Anyway, now you have internationalized this page (let it use a resource bundle) and localized it to Chinese (provided `msgs_zh.properties`). If in the future you need to add support for, say, French, you will not need to internationalize it again but just need to localize it to French (provide `msgs_fr.properties`).

Easier Way to Access Map Elements

Before moving on, let's introduce an easier way to access a map element if the key is a literal string. Instead of writing `b['currentDate']`, you could write `b.currentDate`. After failing to find the `getCurrentDate()` method on the `b` object, it will try to perform a map lookup or a resource bundle lookup. Therefore, `showdate.xhtml` can be slightly simplified, as shown in Listing 9-7.

Listing 9-7. Accessing Map Elements Using the Dot Notation

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://java.sun.com/jsf/html"
    xmlns:f="http://java.sun.com/jsf/core">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
<f:loadBundle basename="multilang.msgs" var="b" />
<title>
<h:outputText value="#{b.currentDate}"/>
</title>
</head>
<body>
```

```
<h:outputText value="#{b.todayIs}"/>
<h:outputText value="#{showDate.today}"/>.
</body>
</html>
```

Internationalizing the Date Display

For the moment, the current date and time is still displayed in the default language of your OS account (probably English). To solve the problem, modify `showdate.xhtml` as shown in Listing 9-8. It shows that not only can UI Input components take a converter (see Chapter 2 for a review on converters) but that UI Output components can do that too. While the former will use a converter for both rendering and handling form submissions, the latter will use it only for rendering. Without a converter, a UI Output component will simply call `toString()` on the value to convert it to a string. With a converter, it will be used to do the conversion. But more important, the date-time converter will use the language code stored in the UI View Root to format the `Date` object. Setting the date style to `long` is not really required; it is done so that you can see Chinese characters in the date display.

Listing 9-8. *Specifying a Language-Aware Converter for a UI Output Component*

```
...
<body>
<h:outputText value="#{b.todayIs}"/>
<h:outputText value="#{showDate.today}">
    <f:convertDateTime dateStyle="long"/>
</h:outputText>.
</body>
</html>
```

Run the application, and it should display the current date in Chinese. If you change the preferred language to English in the browser and refresh the page, the current date should appear in English.

Letting the User Change the Language Used

Suppose that a user is using a browser that prefers Chinese but he would like to show the application to his friend who doesn't understand Chinese but understands English. To support this, you should enhance the application to allow the user to explicitly choose the language, as shown in Figure 9-7.

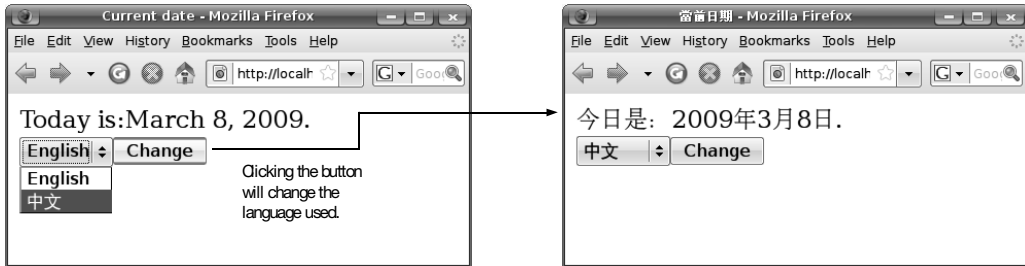


Figure 9-7. *Letting the user change the language used*

To do that, modify `showdate.xhtml` as shown in Listing 9-9.

Listing 9-9. *Letting the User Choose the Language in a Form*

```
...
<body>
<h:outputText value="#{b.todayIs}"/>
<h:outputText value="#{showDate.today}">
    <f:convertDateTime dateStyle="long"/>
</h:outputText>.
<h:form>
    <h:selectOneMenu value="#{showDate.langCode}">
        <f:selectItems value="#{showDate.langCodes}"/>
    </h:selectOneMenu>
    <h:commandButton action="#{showDate.changeLangCode}" value="Change"/>
</h:form>
</body>
```

Define the properties required in the `ShowDate` class as in Listing 9-10. Note how you get the default language and the supported language from the JSF Application object. What you get is not a language code, but a `Locale` object. A `Locale` object contains a language code (such as `en` or `zh`) and optionally a country code (such as `US`, `UK`, or `CN`). Even though a `Locale` object is not a string, you can call `toString()` on it to convert it to a string of the pattern `<LANGUAGE_CODE>_<COUNTRY_CODE>`, such as `en_US` or `zh` (if the country code is not specified).

Listing 9-10. *Letting the User Choose the Language in a Form*

```
package multilang;

import java.util.ArrayList;
```

```

import java.util.Iterator;
import java.util.List;
import java.util.Locale;

import javax.faces.application.Application;
import javax.faces.context.FacesContext;
import javax.faces.model.SelectItem;
...
@Named("showDate")
@RequestScoped
public class ShowDate {
    private String langCode;

    public String getLangCode() {
        return langCode;
    }

    public void setLangCode(String langCode) {
        this.langCode = langCode;
    }

    public List<SelectItem> getLangCodes() {
        List<SelectItem> items = new ArrayList<SelectItem>();
        Application app = FacesContext.getCurrentInstance().getApplication();
        Locale locale = app.getDefaultLocale();
        items.add(new SelectItem(locale.toString()));
        Iterator<Locale> iter = app.getSupportedLocales();
        while (iter.hasNext()) {
            locale = iter.next();
            items.add(new SelectItem(locale.toString()));
        }
        return items;
    }
    public Date getToday() {
        return new Date();
    }
}

```

Define the action method as shown in Listing 9-11. The critical part is creating the `Locale` object from the language code and then setting it into the UI View Root. Finally, it returns null so that the current page is redisplayed.

Listing 9-11. *Changing the Locale in the UI View Root*

```
public class ShowDate {
    private String langCode;

    public String getLangCode() {
        return langCode;
    }
    public void setLangCode(String langCode) {
        this.langCode = langCode;
    }
    public List<SelectItem> getLangCodes() {
        ...
    }
    public String changeLangCode() {
        UIViewRoot viewRoot = FacesContext.getCurrentInstance().getViewRoot();
        viewRoot.setLocale(new Locale(langCode));
        return null;
    }
    public Date getToday() {
        return new Date();
    }
}
```

Run the application, and try to change the language; it should appear in the chosen language. However, it is displaying the language codes such as `en` and `zh` in the combo box, which aren't that user-friendly. Instead, you probably want to display "English" and "中文" (the word "Chinese" as displayed in Chinese) in the combo box. To do that, modify the `ShowDate` class as in Listing 9-12. The critical part is the `getDisplayName()` method that returns the name of the locale suitable for display (for example, "English" or "Chinese"). If you specify a locale, it will return the name of the language in that locale (for example, "中文" if the locale is Chinese).

Listing 9-12. *Getting the Display Name of the Language*

```
...
public class ShowDate {
    ...

    public List<SelectItem> getLangCodes() {
        List<SelectItem> items = new ArrayList<SelectItem>();
        Application app = FacesContext.getCurrentInstance().getApplication();
```

```

        Locale locale = app.getDefaultLocale();
        String display = locale.getDisplayName(locale);
        items.add(new SelectItem(locale.toString(), display));
        Iterator<Locale> iter = app.getSupportedLocales();
        while (iter.hasNext()) {
            locale = iter.next();
            display = locale.getDisplayName(locale);
            items.add(new SelectItem(locale.toString(), display));
        }
        return items;
    }
}

```

Run the application again, and the language names will be displayed in the combo box. Finally, the Change button should also be internationalized and localized. That's easy. Just modify `showdate.xhtml` as shown in Listing 9-13.

Listing 9-13. *Getting the Display Name of the Locale*

```

...
<body>
<h:outputText value="#{b.todayIs}"/>
<h:outputText value="#{showDate.today}">
    <f:convertDateTime dateStyle="long"/>
</h:outputText>.
<h:form>
    <h:selectOneMenu value="#{showDate.langCode}">
        <f:selectItems value="#{showDate.langCodes}"/>
    </h:selectOneMenu>
    <h:commandButton action="#{showDate.changeLangCode}"
        value="#{b.change}"/>
</h:form>
</body>
</html>

```

Define the entry in the resource bundles as shown in Listing 9-14 and Listing 9-15.

Listing 9-14. *Entry for the Change Button in `msgs.properties`*

```

currentDate=Current date
todayIs=Today is:
change=Change

```

Listing 9-15. *Entry for the Change Button in `msgs_zh.properties`*

```
currentDate=當前日期
todayIs=今日是：
change=變更
```

Now run the application, and the Change button will appear in the correct language.

Localizing the Full Stop

There is still a minor issue here. The full stop used at the end of the sentence so far is the English one, not the Chinese one (yes, there is a Chinese full stop). To solve this problem, you could add a new entry to your properties files for the full stop, but then you would be outputting the sentence in three separate parts, as shown in Figure 9-8.

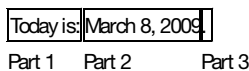


Figure 9-8. *Outputting a sentence in three separate parts*

This is getting too complicated; more important, you're hard-coding the order of these parts and whether or not there is a space between them in `showdate.xhtml`, while in fact these should depend on the language. For example, in English there should be a space after the colon, but in Chinese there should be none. To solve this problem, you should adopt the best practice of outputting the whole sentence in one UI Output component, not two or three. To do that, modify the resource bundles as shown in Listing 9-16 and Listing 9-17. You will fill in the placeholder `{0}` later.

Listing 9-16. *Using a Single Entry for the Whole Sentence in `msgs.properties`*

```
currentDate=Current date
todayIs=Today is: {0}.
change=Change
```

Listing 9-17. *Using a Single Entry for the Whole Sentence in `msgs_zh.properties`*

```
currentDate=當前日期
todayIs=今日是：{0}。
change=變更
```

To output the whole sentence while filling out the {0} placeholder, modify `showdate.xhtml` as shown in Listing 9-18.

Listing 9-18. *Outputting the Whole Sentence with the `<h:outputFormat>` Tag*

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://java.sun.com/jsf/html"
    xmlns:f="http://java.sun.com/jsf/core">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
<f:loadBundle basename="multilang.msgs" var="b" />
<title>
<h:outputText value="#{b.currentDate}"/>
</title>
</head>
<body>
<h:outputFormat value="#{b.todayIs}">
    <f:param value="#{showDate.today}"/>
</h:outputFormat>
<h:form>
    <h:selectOneMenu value="#{showDate.langCode}">
        <f:selectItems value="#{showDate.langCodes}"/>
    </h:selectOneMenu>
    <h:commandButton action="#{showDate.changeLangCode}" value="#{b.change}"/>
</h:form>
</body>
</html>
```

As shown in Figure 9-9, the `<h:outputFormat>` tag will create a UI Output component (just like the `<h:outputText>` tag does) but will associate it with a format renderer. The `<f:param>` will create a UI Parameter component and add it as a child of the UI Output component. The UI Parameter component by itself has no meaning at all. It is entirely up to the parent component (the UI Output component here) how to make use of it. Here, the UI Output component will let the format renderer do the work, which will use the value of the UI Output component as a format pattern. Then it will find out the value of the 0th UI Parameter component and substitute it for the {0} placeholder in the pattern. Of course, if you had more than one placeholder, you would use {0}, {1}, and so on, in the pattern.

```
...
<h:outputFormat value="#{b.todayIs}">
  <f:param value="#{showDate.today}"/>
</h:outputFormat>
```

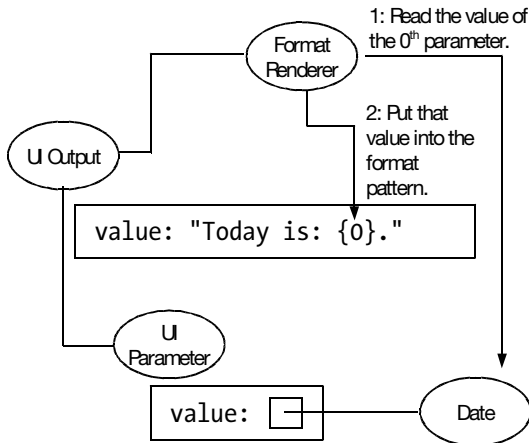


Figure 9-9. How `<h:outputFormat>` and `<f:param>` work

Now when you run the application, the sentence, including the full stop, should appear just fine. However, the date display is no longer in the long style. To fix it, specify the format and style in the placeholder as shown in Listing 9-19 and Listing 9-20.

Listing 9-19. *Specifying the Format and Style for {0} in `msgs.properties`*

```
currentDate=Current date
todayIs=Today is: {0, date, long}.
change=Change
```

Listing 9-20. *Specifying the Format and Style for {0} in `msgs_zh.properties`*

```
currentDate=當前日期
todayIs=今日是：{0, date, long}.
change=變更
```

Now when you run the application, the date should appear in the long style. Table 9-1 shows some more examples for date, time, and currency.

Table 9-1. *Different Formats and Styles*

Placeholder	Meaning
{0, date, short}	Format it as a date using the short style.
{0, date, full}	Format it as a date using the full style.
{0, time, short}	Format it as a time using the short style.
{0, time, long}	Format it as a time using the long style.
{0, number, currency}	Format it as a number using the currency style.
{0, number, integer}	Format it as a number using the integer style.

Displaying a Logo

Now, suppose that you'd like to display a logo on the page, as shown in Figure 9-10.

**Figure 9-10.** *Using a logo*

Note the character “4” in the logo, meaning “for” in this case. In Chinese, “4” doesn’t mean “for” or “four” at all. In fact, it is pronounced just like the word *death* in Chinese, so people tend to avoid it in names. So, say you’d like to have a Chinese version of the logo. Suppose that you have the English version in the file `logo_en.gif` and the Chinese version in `logo_zh.gif`. (You can use any image files; how the images look is not really important as long as they look different from each other.)

First, put them in the `WebContent` folder. Then, modify `showdate.xhtml` as shown in Listing 9-21. The `<h:graphicImage>` tag will create a UI Graphic component, which will generate an HTML `<img>` tag on render. If the value attribute starts with a slash (as is the case now), it will be treated as a relative path from the `WebContent` folder. Without the leading slash, it would be treated as relative to the `showdate.xhtml` file itself (that would work fine too in this case).

Listing 9-21. *Using the `<h:graphicImage>` Tag*

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
```

```

"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
<f:loadBundle basename="multilang.msgs" var="b" />
<title>
<h:outputText value="#{b.currentDate}"/>
</title>
</head>
<body>
<h:graphicImage value="/logo_en.gif"/><p/>
<h:outputFormat value="#{b.todayIs}">
    <f:param value="#{showDate.today}"/>
</h:outputFormat>
<h:form>
    <h:selectOneMenu value="#{showDate.langCode}">
        <f:selectItems value="#{showDate.langCodes}"/>
    </h:selectOneMenu>
    <h:commandButton action="#{showDate.changeLangCode}" value="#{b.change}"/>
</h:form>
</body>
</html>

```

Now run the application, and the English version of the logo should appear (regardless of the language chosen). To make it depend on the language chosen, modify `showdate.xhtml` as shown in Listing 9-22. The view variable is a special variable that will return the UI View Root. Such a special variable is called an *implicit object* provided by JSF. Then you call `getLocale()` on it and let the EL expression evaluator convert it to a string and append it to the file name.

Listing 9-22. *Using the `<h:graphicImage>` Tag*

```

<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />

```

```

<f:loadBundle basename="multilang.msgs" var="b" />
<title>
<h:outputText value="#{b.currentDate}"/>
</title>
</head>
<body>
<h:graphicImage value="/logo_#{view.locale}.gif"/><p/>
<h:outputFormat value="#{b.todayIs}">
    <f:param value="#{showDate.today}"/>
</h:outputFormat>
<h:form>
    <h:selectOneMenu value="#{showDate.langCode}">
        <f:selectItems value="#{showDate.langCodes}"/>
    </h:selectOneMenu>
    <h:commandButton action="#{showDate.changeLangCode}" value="#{b.change}"/>
</h:form>
</body>
</html>

```

When you run the application now, it should display the right version of the logo. As an alternative, you might store the image file name as an entry in the `.properties` file and look it up in the `value` attribute of the `<h:graphicImage>` tag.

Making the Locale Change Persistent

Suppose that your most preferred locale is Chinese in the browser. Let's do an experiment: change the locale to English using the Change button, and then press Enter in the location bar in the browser. The page will be displayed in Chinese again. This means the locale change is temporary: it's set for the current view root and thus for the current request only.

To make the change persistent for, say, the current session, you can store the chosen language code into the session as shown in Listing 9-23. Whenever the user changes the language, the code will check whether there is already a session. If there is, it will store the chosen language code as an attribute in the session under the name `multilang.langCode`.

Listing 9-23. Storing the Language Code into the Session

```

<?xml version="1.0" encoding="UTF-8" ?>
public class ShowDate {
    private String langCode;
    ...
    public String changeLangCode() {

```

```

    HttpSession session = (HttpSession) FacesContext.getCurrentInstance()
        .getExternalContext().getSession(false);
    if (session != null) {
        session.setAttribute("multilang.langCode", langCode);
    }
    UIViewRoot viewRoot = FacesContext.getCurrentInstance().getViewRoot();
    viewRoot.setLocale(new Locale(langCode));
    return null;
}
}

```

The next step is to let the UI View Root use the language code if it exists. To do that, you need to know that it is the so-called view handler in JSF that creates and initializes the UI View Root. This includes retrieving the preferred language in the HTTP request, performing the screening, and then setting the locale into the UI View Root. Now, you need to provide your own view handler that will check whether a language code is in the session and, if there is, use it without looking at the HTTP request.

To do that, create a `MyViewHandler` class in the `multilang` package as shown in Listing 9-24. It extends the `MultiViewHandler` class, which is the default view handler in JSF. When it needs to find out the locale to store into the UI View Root, JSF will call the `calculateLocale()` method. Here you will try to use the language code stored in the session and use it to create a `Locale` object to return. If it doesn't exist, you'll let the base class do what it does (that is, check the preferred language in the HTTP request).

Listing 9-24. *Using the Language Code in the Session to Create the Locale to Use*

```

package multilang;

import java.util.Locale;
import javax.faces.context.FacesContext;
import javax.servlet.http.HttpSession;
import com.sun.faces.application.view.MultiViewHandler;

public class MyViewHandler extends MultiViewHandler {
    public Locale calculateLocale(FacesContext context) {
        HttpSession session = (HttpSession) context.getExternalContext()
            .getSession(false);
        if (session != null) {
            String langCode = (String) session
                .getAttribute("multilang.langCode");
            if (langCode != null) {
                return new Locale(langCode);
            }
        }
    }
}

```

```

        }
    }
    return super.calculateLocale(context);
}
}

```

To tell the JSF to use your own view handler, modify `faces-config.xml` as shown in Listing 9-25.

Listing 9-25. *Using Your Own View Handler*

```

<faces-config ...>
  <application>
    <view-handler>multilang.MyViewHandler</view-handler>
    <locale-config>
      <default-locale>en</default-locale>
      <supported-locale>zh</supported-locale>
    </locale-config>
  </application>
</faces-config>

```

Save the file, restart the JBoss instance, and run the application again. This time the locale change should persist until you end the session (such as by restarting the browser).

Localizing Validation Messages

Remember that you can customize the validation messages using a resource bundle. For example, you may have a file such as `MyApp.properties` in the `multilang` package, as shown in Listing 9-26.

Listing 9-26. *Custom Validation Messages*

```

javax.faces.converter.DateTimeConverter.DATE={0} is an invalid {2}!
javax.faces.component.UIInput.REQUIRED=You must input {0}!
javax.faces.validator.LongRangeValidator.MINIMUM={1} must be at least {0}!
javax.faces.validator.LongRangeValidator.MINIMUM_detail={1} is invalid!

```

To use it, you need to say so in `faces-config.xml`, as shown in Listing 9-27.

Listing 9-27. *Specifying the Application Resource Bundle*

```
<faces-config ...>
  <application>
    <view-handler>multilang.MyViewHandler</view-handler>
    <message-bundle>multilang.MyApp</message-bundle>
    <locale-config>
      <default-locale>en</default-locale>
      <supported-locale>zh</supported-locale>
    </locale-config>
  </application>
</faces-config>
```

It is simply a global resource bundle used by the built-in components. To localize it for, say, Chinese, all you need is to create `MyApp_zh.properties`, just like you would for any other resource bundle.

Summary

You learned the following in this chapter:

- To internationalize a page, you can extract the strings into resource bundles, one for each supported language and a default resource bundle to act as the parent. To look up a string for a certain key in a resource bundle, use the `<loadBundle>` tag to load a resource bundle as a request-scoped attribute and access it like accessing a map.
- To determine the locale to use, the view handler will check the preferred locale as specified in the HTTP request and check whether it is supported by your application. If it is, it will store it into the view root. If it's not, it will use the default locale specified in your application.
- If you'd like to let the user specify a particular locale to use, overriding the preferred locale set in the browser, you may want to store it into the session and provide a view handler subclass to retrieve it later.
- It is best practice to output the whole sentence using one UI Output component. To fill in various placeholders in a pattern before outputting it, you can use `<outputFormat>` and specify the value for each placeholder using a UI Parameter component (created by a `<param>`). The meaning of UI Parameter is entirely determined by its parent component.
- To display an image, use a `<graphicImage>` tag. Its value attribute is a relative path from the `WebContent` folder (if it has a leading slash) or from the `.xhtml` file (otherwise). To internationalize an image, just internationalize its value attribute.



Using JBoss Seam

If your application uses Enterprise JavaBeans (EJBs), you may want to use a framework called Seam (from JBoss) so that your JSF pages will work with the EJBs more easily. For example, Seam will allow you to access EJBs in EL expressions, just like you access web beans. In addition, Seam also provides some nice features to speed up your JSF development. For example, it can generate pages to perform the CRUD (create, retrieve, update, and delete) operations for your business object class without you writing a single line of code. In this chapter, you'll learn about such features.

Don't worry if you don't know about EJBs; you should still be able to go through this chapter easily.

Installing Seam

To install Seam, go to <http://seamframework.org> to download a binary package of JBoss Seam, such as `jboss-seam-2.1.1.GA.zip`. Unzip it into a folder such as `c:\jboss-seam`. As the download is a library, you can't run it yet.

Seam version 2.1 doesn't support JSF 2.0 yet. As you've installed JSF 2.0 into the JBoss application server in Chapter 1, you can't run a Seam application on top of it. To solve this problem, you may want to download a clean copy of the JBoss application server again and unzip it into, say, `c:\jboss`.

Re-creating the E-shop Project

Suppose that you'd like to re-create the e-shop using EJBs and Seam. To do that, open a command prompt and issue the commands shown in Listing 10-1. Note that these commands assume that your JDK has been installed into `c:\Program Files\Java\jdk1.6.0`; if not, you must use the actual path on your computer.

Listing 10-1. *Starting Seam Setup*

```
c:\>cd \jboss-seam
c:\>set JAVA_HOME=c:\Program Files\Java\jdk1.6.0
c:\>seam setup
```

The `seam setup` command will prompt you for some information about your Seam project and save it for later use. The answers that you must enter yourself are highlighted in Listing 10-2. For the rest of the questions, you can simply accept the defaults.

Listing 10-2. *Telling Seam About Your Seam Project*

```
SEAM_HOME: c:/jboss-seam
Using seam-gen sources from: c:/jboss-seam/seam-gen
Buildfile: c:/jboss-seam/seam-gen/build.xml

init:

setup:
    [echo] Welcome to seam-gen :-)
    [input] Enter your Java project workspace (the directory that contains your Seam
projects) [C:/Projects] [C:/Projects]
c:/Documents and Settings/kent/workspace
    [input] Enter your JBoss AS home directory [C:/Program Files/jboss-4.2.3.GA]
[C:/Program Files/jboss-4.2.3.GA]
c:/jboss
    [input] Enter the project name [myproject] [myproject]
SeamShop
    [echo] Accepted project name as: SeamShop
    [input] Do you want to use ICEfaces instead of RichFaces [n] (y, [n])
```

[input] skipping input as property icefaces.home.new has already been set.

[input] Select a RichFaces skin [classic] (blueSky, [classic], deepMarine, DEFAULT, emeraldTown, japanCherry, ruby, wine)

[input] Is this project deployed as an EAR (with EJB components) or a WAR (with no EJB support) [ear] ([ear], war)

[input] Enter the Java package name for your session beans
[com.mydomain.SeamShop] [com.mydomain.SeamShop]

shop

[input] Enter the Java package name for your entity beans [shop] [shop]

[input] Enter the Java package name for your test cases [shop.test] [shop.test]

[input] What kind of database are you using? [hsql] ([hsql], mysql, oracle, postgres, mssql, db2, sybase, enterprisedb, h2)

[input] Enter the Hibernate dialect for your database
[org.hibernate.dialect.HSQLDialect] [org.hibernate.dialect.HSQLDialect]

[input] Enter the filesystem path to the JDBC driver jar [c:/jboss-seam/lib/hsqldb.jar] [c:/jboss-seam/lib/hsqldb.jar]

[input] Enter JDBC driver class for your database [org.hsqldb.jdbcDriver]
[org.hsqldb.jdbcDriver]

[input] Enter the JDBC URL for your database [jdbc:hsqldb:.] [jdbc:hsqldb:.]

[input] Enter database username [sa] [sa]

[input] Enter database password [] []

[input] Enter the database schema name (it is OK to leave this blank) [] []

[input] Enter the database catalog name (it is OK to leave this blank) [] []

[input] Are you working with tables that already exist in the database? [n] (y, [n])

```
[input] Do you want to drop and recreate the database tables and data in
import.sql each time you deploy? [n] (y, [n])
```

```
[propertyfile] Creating new property file: c:/jboss-seam/seam-gen/build.properties
[echo] Installing JDBC driver jar to JBoss AS
[echo] Type 'seam create-project' to create the new project
```

```
BUILD SUCCESSFUL
```

```
Total time: 7 minutes 33 seconds
```

The `seam setup` command only saves the information into a setting file so that it can be used later. To create an Eclipse project (using the saved information), issue the command `seam new-project` as the second step. Then, the tool will print some output and say `BUILD SUCCESSFUL` as shown in Listing 10-3.

Listing 10-3. *Creating the Seam Project for the IDE*

```
...
create-project:
    [echo] A new Seam project named 'SeamShop' was created in the c:/Documents and
    Settings/kent/workspace directory
    [echo] Type 'seam explode' and go to http://localhost:8080/SeamShop
    ...
new-project:

BUILD SUCCESSFUL
Total time: 5 seconds
```

Now, go to Eclipse. Right-click anywhere in the Project Explorer window, and choose **Import** followed by “Existing Projects into Workspace,” and then browse to the `c:/Documents and Settings/kent/workspace/SeamShop` folder to import the project. Note that you do not need to add the project to the JBoss instance, as the project has been set up so that, whenever you change any file in it, Eclipse will update the corresponding application in JBoss.

To see if everything is working, start the JBoss instance (create a new one if you did install a clean copy of JBoss) in Eclipse, and try to access `http://localhost:8080/SeamShop/home.seam`. You should see a welcome page like the one shown in Figure 10-1. As you may have noted in the URL, by default, a Seam application maps `*.seam` to the JSF engine instead of the `/faces` prefix.

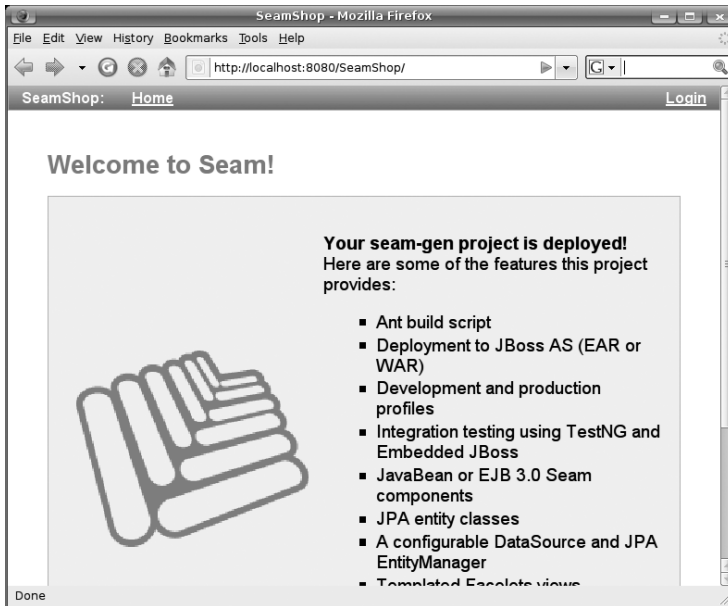
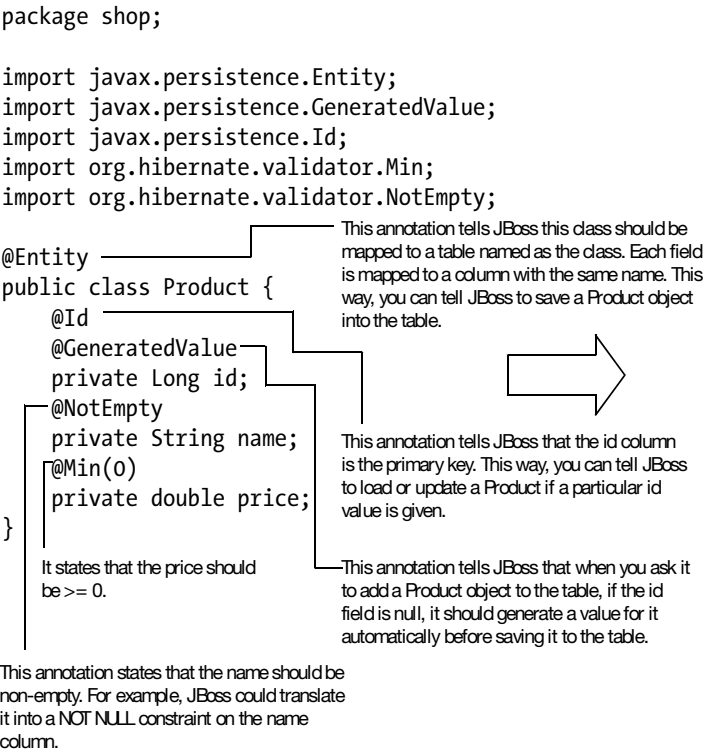


Figure 10-1. The welcome page displayed in a Seam application

Note Keep the command prompt open, because you'll need it in the next section.

Allowing the User to Add Products

For the moment, you have nothing in the products database (no tables and no data). Therefore, you'd like to allow the user to add some products. You don't need to create the products table, as it will be created automatically. All you need is to create the `Product` class in the `src/main` folder (let's put it into the `shop` package), as shown in Figure 10-2. In summary, you are mapping the `Product` class to a table using `@Entity`, specifying the primary key using `@Id` and some validation constraints on the fields (`@NotEmpty` for a non-null field and `@Min` specifying the minimum valid value).



Product

id*	name	price
...	...	...
...	...	...

Figure 10-2. Mapping the Product class to a table

Add the getters and setters for access to the fields, as shown in Listing 10-4.

Listing 10-4. Adding Getters and Setters

```
package shop;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import org.hibernate.validator.Min;
import org.hibernate.validator.NotEmpty;

@Entity
public class Product {
    @Id
    @GeneratedValue
    private Long id;
    @NotEmpty
    private String name;
    private double price;
}
```

```
private String name;
@Min(0)
private double price;

public Long getId() {
    return id;
}
public void setId(Long id) {
    this.id = id;
}
public String getName() {
    return name;
}
public void setName(String name) {
    this.name = name;
}
public double getPrice() {
    return price;
}
public void setPrice(double price) {
    this.price = price;
}
}
```

Next, you need to create a page to let the user add products. Surprisingly, Seam can do that automatically for you. In the same command prompt you used before, issue `seam generate-ui`. This command will scan the `src/main` folder for classes with the `@Entity` annotation. For each such class, the command will create a page to search and list the objects, a page to view an object, and a page to edit an object.

Now, refresh the project in Eclipse so that it notes the new files. Note that three pages should have been created in the view folder: `Product.xhtml` (for viewing a product), `ProductEdit.xhtml` (for editing a product), and `ProductList.xhtml` (for searching and listing products).

To see if these pages are working, go to `http://localhost:8080/SeamShop/home.seam` again. You should see a Product List link at the top, as shown in Figure 10-3. Click the link to take you to the product list page.

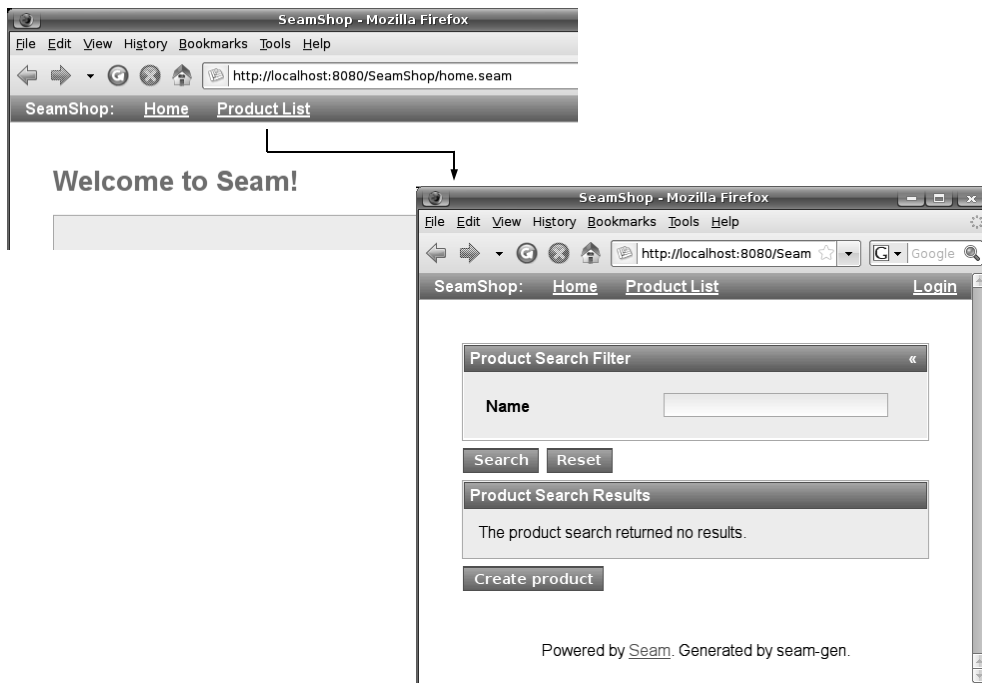


Figure 10-3. *The Product List link*

If you try to create a product by clicking the Create Product button on the product list page, the application will ask you to log in first, because the product-editing page has been declared as requiring a logged-in user. This requirement is set in the `ProductEdit.page.xml` file in the view folder, as shown in Listing 10-5.

Listing 10-5. *Requiring Authenticated Access to a Page*

```
<?xml version="1.0" encoding="UTF-8"?>
<page xmlns="http://jboss.com/products/seam/pages"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://jboss.com/products/seam/pages
http://jboss.com/products/seam/pages-2.1.xsd"
      no-conversation-view-id="/ProductList.xhtml"
      login-required="true">
    ...
</page>
```

As you can see, in Seam, you can have a configuration file for each page. For example, if the page is named `foo.xhtml`, its configuration is named `foo.page.xml` and is stored in the same folder as the page file.

How does Seam check the user name and password? It relies on a web bean named `authenticator`. Actually, Seam versions prior to and including 2.1 predate the Web Beans specification and therefore do not support web beans. However, these versions of Seam do support a similar concept called a component. This authenticator component is in the `src/hot` folder in the `Authenticator` class in the `shop` package, as shown in Listing 10-6. It was generated when `seam new-project` created the Eclipse project. (Are you wondering why the funny name “hot”? Seam can hot deploy components defined in this folder when you make changes to them.)

In Seam, the name of the component is specified using `@Name`, not `@Named`, and injection is done by `@In` instead of `@Current`. `@In` and `@Current` work similarly but with one very important difference: `@In` uses the name of the field (for example, `identity` or `credentials`) to look up the component, while `@Current` uses the class of the field to do that.

When Seam needs to check the user name and password, it will call the `authenticate()` method. That method can obtain the user name and password entered by the user from the `Credentials` component. If the method returns `true`, the authentication will be considered successful; `false` indicates a failure. As Listing 10-6 shows, instead of looking up a user database or an LDAP directory, Seam hard-codes a user named `admin` without checking the password and tells (any parties concerned) that the user has the role of `admin`. You can take that to mean that the user belongs to the `admin` group. You’ll see how to make use of `admin` user group privileges later.

Listing 10-6. *Authenticator Component*

```
package shop;

import org.jboss.seam.annotations.In;
import org.jboss.seam.annotations.Logger;
import org.jboss.seam.annotations.Name;
import org.jboss.seam.log.Log;
import org.jboss.seam.security.Credentials;
import org.jboss.seam.security.Identity;

@Name("authenticator")
public class Authenticator {
    @Logger
    private Log log;
    @In
    Identity identity;
    @In
    Credentials credentials;
```

```

public boolean authenticate() {
    log.info("authenticating {0}", credentials.getUsername());
    //write your authentication logic here,
    //return true if the authentication was
    //successful, false otherwise
    if ("admin".equals(credentials.getUsername())) {
        identity.addRole("admin");
        return true;
    }
    return false;
}
}

```

Now, log in as admin, and try to add a product as shown in Figure 10-4. Note that Seam has figured out that the name is required (from @NotEmpty) and that the price is also required (from the fact that it is a primitive double, not a Double and thus can't be null).

The screenshot shows a web browser window titled "SeamShop - Mozilla Firefox". The address bar displays "http://localhost:8080/SeamShop/ProductEdit.seam". The page has a navigation bar with "SeamShop:" followed by links for "Home" and "Product List". The main content area is a form titled "Add Product". It contains two input fields: "Name *" with the text "Pen" and "Price *" with the value "2.3". Below the price field, there is a note "* required fields". At the bottom of the form are two buttons: "Save" and "Cancel".

Figure 10-4. *Adding a product*

If you try to input a negative price, the page will return an error as shown in Figure 10-5. It means Seam has created a `DoubleRangeValidator` from the `@Min` annotation automatically.

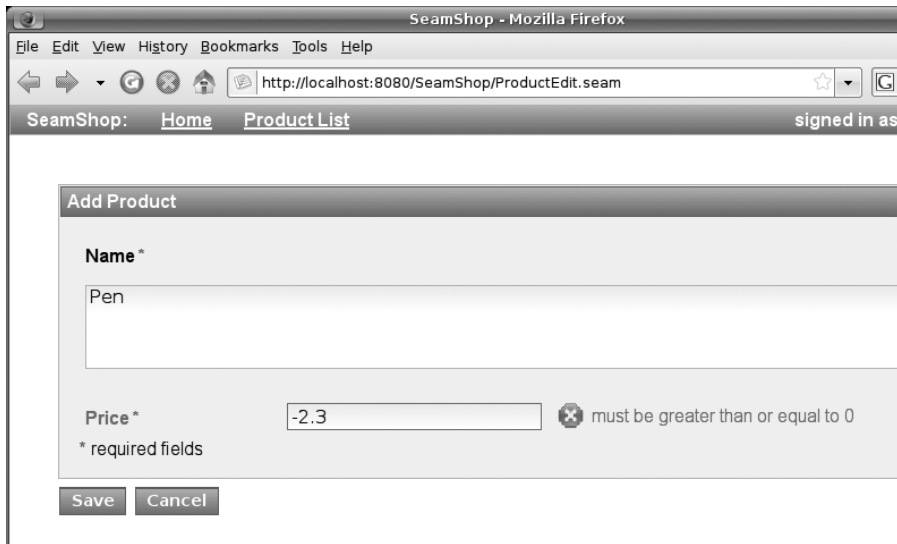


Figure 10-5. *Invalid input caught*

Also in Figure 10-5, note that the invalid field is highlighted, and an error message is displayed to its right. This visual feedback for the user is implemented by the `<s:decorate>` tag in `ProductEdit.xhtml`, as shown in Listing 10-7. This tag is also responsible for adding an asterisk to the label if the field is required. To use the `<s:decorate>` tag, you need to pass a piece of JSF content (representing the field label) to it using the `name` label, and put the input field in the `<s:decorate>` tag's body.

Listing 10-7. *Using the `<s:decorate>` Tag*

```
<!DOCTYPE composition PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<ui:composition xmlns="http://www.w3.org/1999/xhtml"
    xmlns:s="http://jboss.com/products/seam/taglib"
    xmlns:ui="http://java.sun.com/jsf/facelets"
    xmlns:f="http://java.sun.com/jsf/core"
    xmlns:h="http://java.sun.com/jsf/html"
    xmlns:a="http://richfaces.org/a4j"
    xmlns:rich="http://richfaces.org/rich"
    template="layout/template.xhtml">
```

```

<ui:define name="body">
  <h:form id="product" styleClass="edit">
    <rich:panel>
      <f:facet name="header">...</f:facet>
      <s:decorate id="nameField" template="layout/edit.xhtml">
        <ui:define name="label">Name</ui:define>
        <h:inputTextarea id="name"
          cols="80"
          rows="3"
          required="true"
          value="#{productHome.instance.name}"/>
      </s:decorate>
      <s:decorate id="priceField" template="layout/edit.xhtml">
        <ui:define name="label">Price</ui:define>
        <h:inputText id="price"
          required="true"
          value="#{productHome.instance.price}"/>
        <a:support .../>
      </h:inputText>
      </s:decorate>
      ...
    </rich:panel>
    ...
  </h:form>
</ui:define>
</ui:composition>

```

Also note that the page uses a base page (`layout/template.xhtml`). That's why all the pages in the Seam application have the same menu at the top and the same footer at the bottom.

Now, go ahead and add some more products. The product list page should show the products you've added, and you can click the View link to see a particular product, as shown in Figure 10-6.

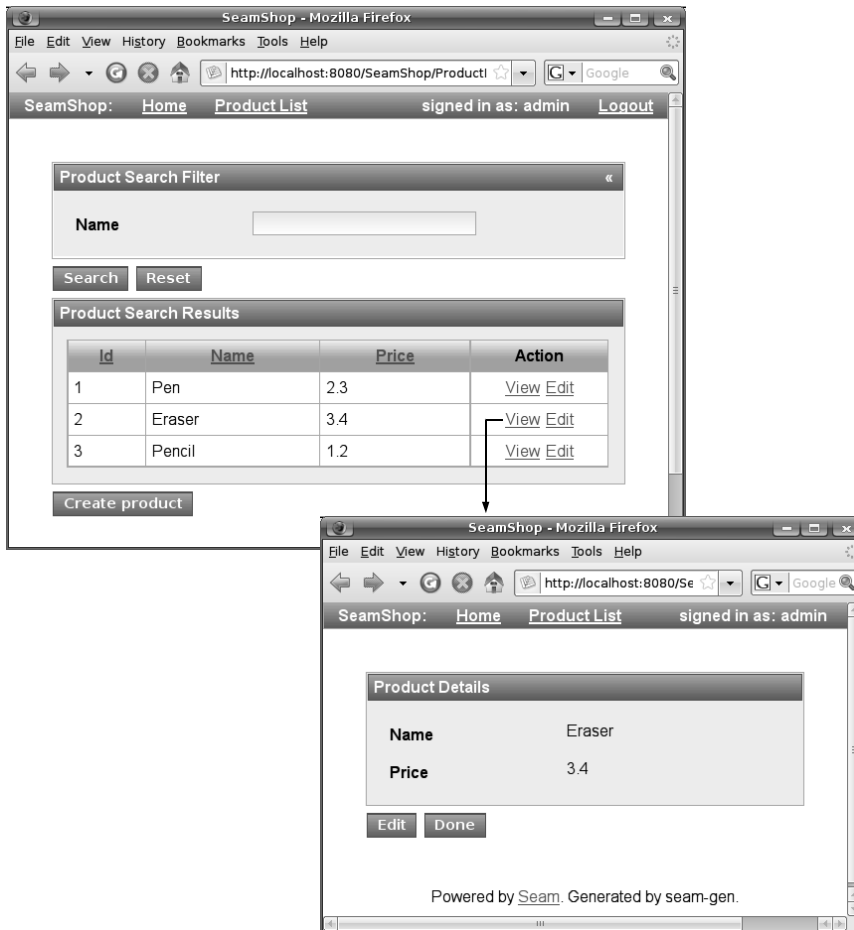


Figure 10-6. *Listing and viewing products*

Restricting Access to the Product-Editing Page

For the moment, any logged-in user can edit the products, and this setup is insecure. Suppose only users with the admin role should be allowed to edit the products. To add this security feature, modify `ProductEdit.page.xml` as shown in Listing 10-8.

Listing 10-8. Restricting Access to Product Editing Page

```
<?xml version="1.0" encoding="UTF-8"?>
<page ...>
  <restrict>SOME EL EXPR RETURNING TRUE ONLY IF USER HAS THE ADMIN
  ROLE</restrict>
  <begin-conversation join="true" flush-mode="MANUAL" />
```

```

<action execute="#{productHome.wire}" />
<param name="productFrom" />
<param name="productId" value="#{productHome.productId}" />
<navigation from-action="#{productHome.persist}">
    <rule>
        <end-conversation />
        <redirect view-id="/Product.xhtml" />
    </rule>
</navigation>
<navigation from-action="#{productHome.update}">
    <rule>
        <end-conversation />
        <redirect view-id="/Product.xhtml" />
    </rule>
</navigation>
<navigation from-action="#{productHome.remove}">
    <rule>
        <end-conversation />
        <redirect view-id="/ProductList.xhtml" />
    </rule>
</navigation>
</page>

```

What EL expression do you put into the body of the `<restrict>` tag? You can do it as shown in Figure 10-7. In summary, the EL expression evaluator will use the prefix “s” and the function name “hasRole” to look up the function and then call it. The return value is the value of the EL expression.

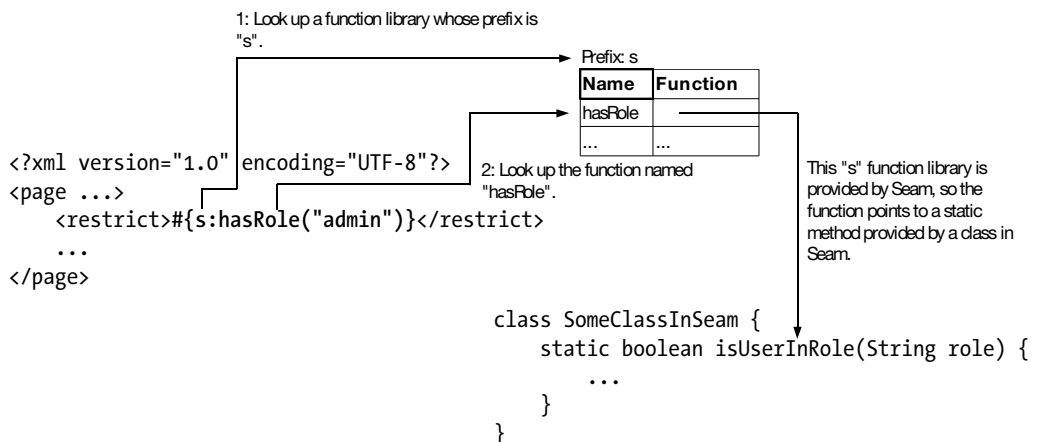


Figure 10-7. Calling a function in an EL expression

Now, you might like to test run the application and log in as an ordinary user, but you can't, because the system has only one (admin) user. To solve the problem, hard-code an ordinary user in the Authenticator class, as shown in Listing 10-9.

Listing 10-9. *Hard-Coding an Ordinary User*

```
package shop;
...
@Name("authenticator")
public class Authenticator {
    @Logger
    private Log log;
    @In
    Identity identity;
    @In
    Credentials credentials;

    public boolean authenticate() {
        log.info("authenticating {0}", credentials.getUsername());
        // write your authentication logic here,
        // return true if the authentication was
        // successful, false otherwise
        if ("u1".equals(credentials.getUsername())) {
            return true;
        }
        if ("admin".equals(credentials.getUsername())) {
            identity.addRole("admin");
            return true;
        }
        return false;
    }
}
```

Now, run the application, log in as u1, and try to access the product-editing page. You should be rejected. If you log out and try again as the admin user, you should get through.

Creating a Shopping Cart

Next, you'd like to allow users to add products to a shopping cart. To do that, you need to add a button on the product-viewing page like the one shown in Figure 10-8. After adding the product, the IDs of the products in the shopping cart are displayed.

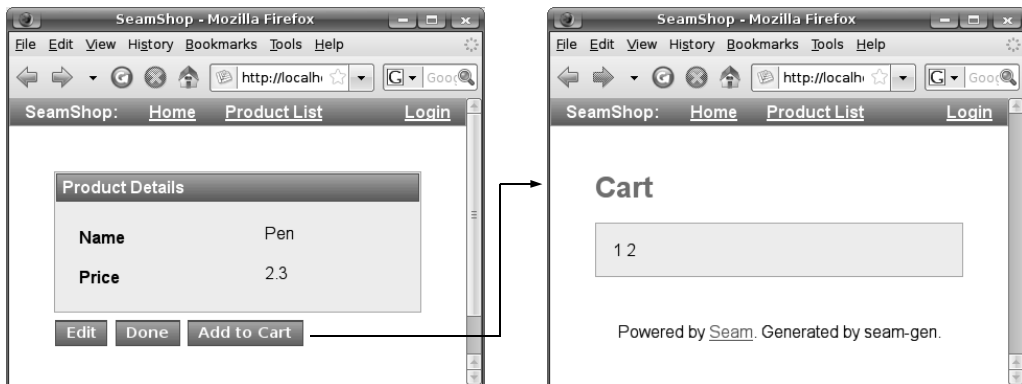


Figure 10-8. Adding a product to the shopping cart

To add the shopping cart button, modify `product.xhtml` as shown in Listing 10-10. The Seam `<s:button>` is very much like an `<h:commandButton>` except that you can specify the next view ID directly using the `view` attribute: using `<s:button>` means you don't need to define a navigation rule. In addition, the Seam button is quite smart—it will work even if it is not inside an `<h:form>`.

Listing 10-10. Using `<s:button>` for the “Add to Cart” Button

```
<!DOCTYPE composition PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<ui:composition xmlns="http://www.w3.org/1999/xhtml"
    xmlns:s="http://jboss.com/products/seam/taglib"
    xmlns:ui="http://java.sun.com/jsf/facelets"
    xmlns:f="http://java.sun.com/jsf/core"
    xmlns:h="http://java.sun.com/jsf/html"
    xmlns:rich="http://richfaces.org/rich"
    template="layout/template.xhtml">
<ui:define name="body">
    <rich:panel>
        <f:facet name="header">Product Details</f:facet>
        <s:decorate id="name" template="layout/display.xhtml">
            <ui:define name="label">Name</ui:define>
            <h:outputText value="#{productHome.instance.name}"/>
        </s:decorate>
        <s:decorate id="price" template="layout/display.xhtml">
            <ui:define name="label">Price</ui:define>
            <h:outputText value="#{productHome.instance.price}"/>
        </s:decorate>
    </rich:panel>
</ui:define>
```

```

        <div style="clear:both"/>
    </rich:panel>
    <div class="actionButtons">
        <s:button view="/ProductEdit.xhtml"
            id="edit"
            value="Edit"/>
        <s:button view="/#{empty productFrom ? 'ProductList' : productFrom}.xhtml"
            id="done"
            value="Done"/>
        <s:button view="/cart.xhtml"
            action="#{cart.add}"
            value="Add to Cart"/>
    </div>
</ui:define>

```

Now that you have a button, you need to create the cart component to provide the `add()` action method to use it. So, create the `Cart` class in the `src/hot` folder in the `shop` package as shown in Listing 10-11. Here, you inject the `ProductHome` object using the `@In` annotation. This `ProductHome` class was generated by Seam when you ran `seam generate-ui` and is very much like the `ProductHolder` class in the e-shop from the plain JSF example in Chapter 4. That is, it is used to hold a product ID, and it can use that to load a `Product` object.

The `Cart.add()` method gets the product ID from the `ProductHome` object. Then you just print the product ID to the console to verify that it is working. Note that the `add()` method returns `void` instead of an outcome, which is possible because the view ID has been specified in the `<s:button>` tag.

Listing 10-11. *Adding the Product ID to the Shopping Cart*

```

package shop;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;

import org.jboss.seam.ScopeType;
import org.jboss.seam.annotations.Name;
import org.jboss.seam.annotations.Scope;

@Name("cart")
@Scope(ScopeType.EVENT)
public class Cart implements Serializable {

```

```

private List<Long> pids;
@Inject
private ProductHome productHome;

public Cart() {
    pids = new ArrayList<Long>();
}
public void add() {
    Long pid = productHome.getProductId();
    System.out.println(pid);
    pids.add(pid);
}
public List<Long> getPids() {
    return pids;
}
}

```

Now that you have the method to handle the cart, create the `cart.xhtml` file in the view folder to display the contents of the shopping cart. For the moment, the content is static, as shown in Listing 10-12.

Listing 10-12. *cart.xhtml*

```

<!DOCTYPE composition PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<ui:composition xmlns="http://www.w3.org/1999/xhtml"
    xmlns:s="http://jboss.com/products/seam/taglib"
    xmlns:ui="http://java.sun.com/jsf/facelets"
    xmlns:f="http://java.sun.com/jsf/core"
    xmlns:h="http://java.sun.com/jsf/html"
    xmlns:rich="http://richfaces.org/rich"
    template="layout/template.xhtml">
    <ui:define name="body">
        <h1>Cart</h1>
    </ui:define>
</ui:composition>

```

Now, run the application, and try to add a product to the shopping cart. Unfortunately, it will fail with the exception shown in Figure 10-9.

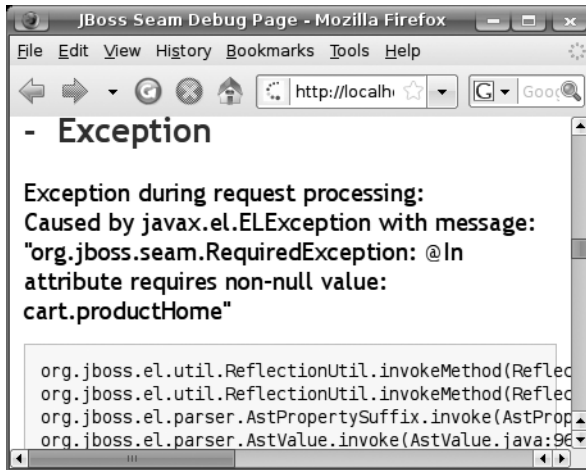


Figure 10-9. An error received when injecting a nonexistent component

The exception in Figure 10-9 is thrown because when Seam tries to inject the `productHome` component into the `cart` component, it finds that there is no such `productHome` component. Why? Because the `ProductHome` class extends the `EntityHome` class (provided by JBoss), which is in the conversation scope. However, the application did not start a long-running conversation, so the `productHome` component will be gone when the “Add to Cart” button is submitted. What happens when the component being injected doesn’t exist? If the component were being referenced in an EL expression, Seam would create it. But when the injected component is referenced by another Seam component, Seam will simply throw an exception.

Simply creating a new `productHome` component again will not resolve the exception, as the component will have lost the product ID. Figure 10-10 shows a solution: When `<s:button>` is generating the URL (to invoke the `cart.xhtml` page), it reads the product ID from the `productHome` component and stores the ID into a query parameter in the URL. When the user clicks the “Add to Cart” button, Seam intercepts the request, notes that it is loading the `cart.xhtml` page, reads that query parameter, and stores the product ID into the (new) `productHome` component.

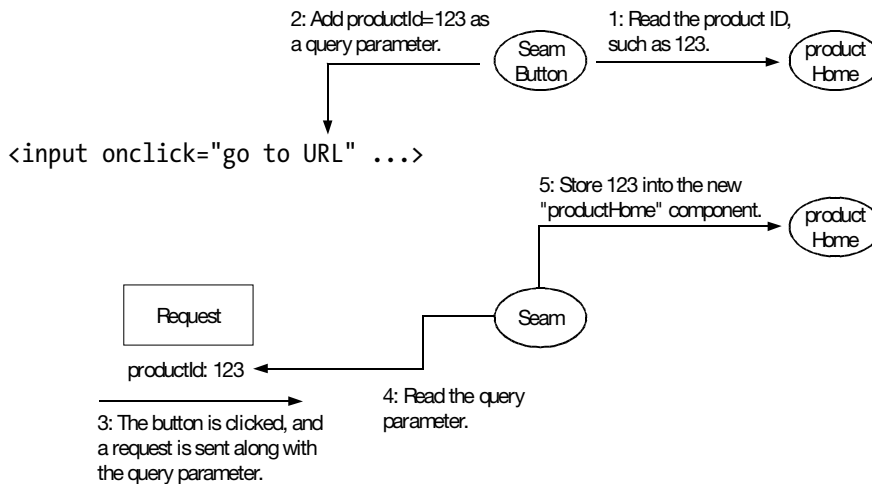


Figure 10-10. Using a query parameter to maintain the product ID

To implement this idea, create the `cart.page.xml` file alongside the `cart.xhtml` page (in the `view` folder). The content is shown in Listing 10-13. The key is the `<param>` tag. It is read by both the `<s:button>` when generating the URL and Seam when handling the request. The former reads the EL expression and stores the result into the specified query parameter. The latter does the opposite: it reads the specified query parameter and stores the result into the EL expression.

Listing 10-13. Using `<param>` to Maintain the Product ID with a Query Parameter

```
<?xml version="1.0" encoding="UTF-8"?>
<page xmlns="http://jboss.com/products/seam/pages"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://jboss.com/products/seam/pages
http://jboss.com/products/seam/pages-2.1.xsd">
  <param name="productId" value="#{productHome.productId}"/>
</page>
```

Run the application, and try to add a product to the shopping cart. The correct product ID should print in the console.

Next, modify `cart.xhtml` to show the cart contents, as shown in Listing 10-14.

Listing 10-14. Displaying the Contents of the Shopping Cart

```
<!DOCTYPE composition PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<ui:composition xmlns="http://www.w3.org/1999/xhtml"
```

```

xmlns:s="http://jboss.com/products/seam/taglib"
xmlns:ui="http://java.sun.com/jsf/facelets"
xmlns:f="http://java.sun.com/jsf/core"
xmlns:h="http://java.sun.com/jsf/html"
xmlns:rich="http://richfaces.org/rich"
template="layout/template.xhtml">
<ui:define name="body">
    <h1>Cart</h1>
    <rich:panel>
        <ui:repeat value="#{cart.pids}" var="pid">
            <h:outputText value="#{pid}" />
        </ui:repeat>
    </rich:panel>
</ui:define>
</ui:composition>

```

There is nothing special here. If you run the application and add some products to the shopping cart, their IDs will be displayed.

Turning the Shopping Cart into a Stateful Session Bean

For the moment, you have a shopping cart for each user. If you have a huge number of users, many shopping carts will exist in the memory, while many of them are not being actively used. To save memory, you may let JBoss save some of them to the hard disk when they are idle. When they are needed again, JBoss will load them from the hard disk. This way, you can support a huge number of users, and your application will become very scalable.

To implement this idea, you can turn the shopping cart into a stateful session bean. To do that, first change the `Cart` class in the `src/hot` folder into an interface, as shown in Listing 10-15.

Listing 10-15. *The Cart Interface*

```

package shop;

import java.util.List;

public interface Cart {
    void add();
    List<Long> getPids();
}

```

You'll still need to create a class to implement that interface. So, create a `CartBean` class in the `src/hot` folder in the `shop` package, as shown in Figure 10-11. Simply put, `@Stateful` tells JBoss to create a stateful session bean from this class. As a result, different bean instances will be created, saved to disk, loaded from disk, and destroyed by JBoss automatically. The application will access a bean instance through a proxy, so that the bean instance can be saved, loaded, or moved around in memory without the application noticing. Because the session bean is stateful, once the application obtains a proxy to one instance (say, `Cart 1`), all subsequent method calls on the proxy will be delegated to the same instance (`Cart 1`). In contrast, if the session bean was stateless, different method calls on the same proxy might be delegated to different bean instances, because all the bean instances are considered identical.

```
package shop;
```

```
import java.util.List;
import javax.ejb.Stateful;
```

```
@Stateful
public class CartBean implements Cart {

    The bean instances (and the proxies) will implement
    the Cart interface, and the application will access
    the proxy through the Cart interface too.

    @Override
    public void add() {

    }

    @Override
    public List<Long> getPids() {
        return null;
    }
}
```

This annotation tells JBoss to create a stateful session bean from this class. The effect is that JBoss will create, manage, and destroy instances of that stateful session bean.

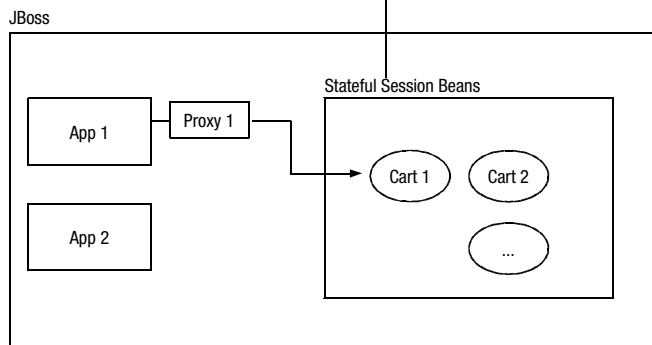


Figure 10-11. *Creating a stateful session bean for the shopping cart*

To inject the `productHome` component into the shopping cart (so that the latter can find out the ID of the selected product), you need to turn the session bean into a Seam component. To do that, modify the `CartBean` class as shown in Listing 10-16.

Listing 10-16. *Turning a Session Bean into a Seam Component*

```

package shop;
...
@Stateful
@Name("cart")
@Scope(ScopeType.SESSION)
public class CartBean implements Cart {
    @In
    private ProductHome productHome;

    @Override
    public void add() {

    }
    @Override
    public List<Long> getPids() {
        return null;
    }
}

```

How does the Seam component work with the session bean? When Seam needs to create the cart component, it notes that the `CartBean` class is a stateful session bean (as specified by the `@Stateful` annotation). So, Seam will obtain a proxy (from JBoss) to access the bean instance and let the component delegate all method calls to it as shown in Figure 10-12.

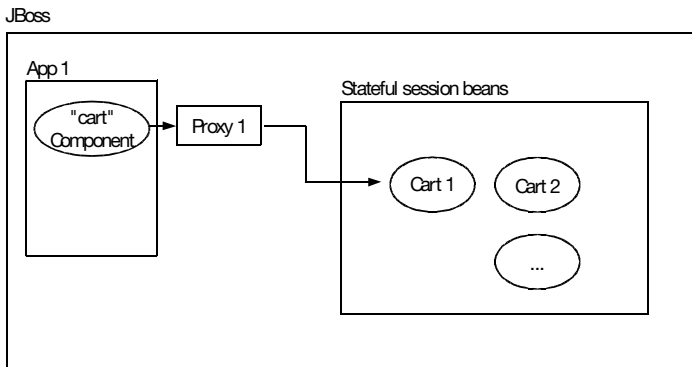


Figure 10-12. *A Seam component delegating method calls to a stateful session bean*

In addition, when the session is about to be destroyed, Seam will need to destroy the cart component and the corresponding bean instance. To allow Seam to do that, you can

modify the `CartBean` class as shown in Listing 10-17. The `@Remove` annotation tells JBoss to remove the bean instance after the `destroy()` method returns. Noting the existence of this annotation, Seam will call the `destroy()` method when it needs to destroy the component.

Listing 10-17. *Providing a Destroy Method to Seam*

```
package shop;
...
import javax.ejb.Remove;

@Stateful
@Name("cart")
@Scope(ScopeType.SESSION)
public class CartBean implements Cart {
    @In
    private ProductHome productHome;

    @Override
    public void add() {

    }
    @Override
    public List<Long> getPids() {
        return null;
    }
    @Remove
    public void destroy() {

    }
}
```

Finally, implement the methods shown in Listing 10-18. They implement the business functionality of the shopping cart, enabling it to store product IDs and retrieve them later.

Listing 10-18. *Implementing the Methods for the Shopping Cart*

```
package shop;
...
@Stateful
@Name("cart")
@Scope(ScopeType.SESSION)
```

```

public class CartBean implements Cart {
    private List<Long> pids;
    @In
    private ProductHome productHome;

    public CartBean() {
        pids = new ArrayList<Long>();
    }
    @Override
    public void add() {
        Long pid = productHome.getProductId();
        pids.add(pid);
    }
    @Override
    public List<Long> getPids() {
        return pids;
    }
    @Remove
    public void destroy() {

    }
}

```

Now, restart the browser so that a new session is created, run the application, and try to add some products to the shopping cart. The application should continue to work.

Creating the Checkout Page

Next, you'd like to allow the user to check out as shown in Figure 10-13.

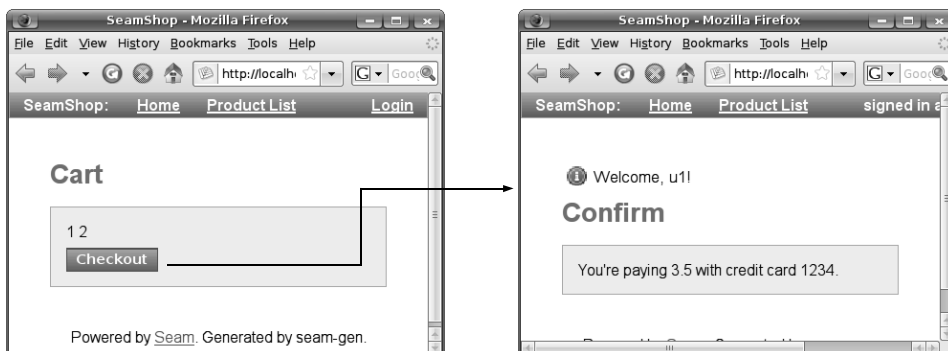


Figure 10-13. *Checking out*

To do that, modify `cart.xhtml` as shown in Listing 10-19.

Listing 10-19. *Checkout Button*

```
package shop;
<!DOCTYPE composition PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<ui:composition xmlns="http://www.w3.org/1999/xhtml"
    xmlns:s="http://jboss.com/products/seam/taglib"
    xmlns:ui="http://java.sun.com/jsf/facelets"
    xmlns:f="http://java.sun.com/jsf/core"
    xmlns:h="http://java.sun.com/jsf/html"
    xmlns:rich="http://richfaces.org/rich"
    template="layout/template.xhtml">
    <ui:define name="body">
        <h1>Cart</h1>
        <rich:panel>
            <ui:repeat value="#{cart.pids}" var="pid">
                <h:outputText value="#{pid}" />
            </ui:repeat>
            <div>
                <s:button view="/confirm.xhtml" value="Checkout"/>
            </div>
        </rich:panel>
    </ui:define>
</ui:composition>
```

Create the `confirm.xhtml` file in the view folder. The content is shown in Listing 10-20.

Listing 10-20. *confirm.xhtml*

```
package shop;
<!DOCTYPE composition PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<ui:composition xmlns="http://www.w3.org/1999/xhtml"
    xmlns:s="http://jboss.com/products/seam/taglib"
    xmlns:ui="http://java.sun.com/jsf/facelets"
    xmlns:f="http://java.sun.com/jsf/core"
    xmlns:h="http://java.sun.com/jsf/html"
    xmlns:rich="http://richfaces.org/rich"
    template="layout/template.xhtml">
    <ui:define name="body">
```

```

        <h1>Confirm</h1>
        <rich:panel>
            You're paying #{confirm.total} with credit card #{confirm.creditCardNo}.
        </rich:panel>
    </ui:define>
</ui:composition>

```

Again, to achieve better scalability, you'd like to make the `confirm` component a session bean. To do that, create the `Confirm` interface in the `src/hot` folder in the `shop` package as shown in Listing 10-21.

Listing 10-21. *The Confirm Interface*

```

package shop;

public interface Confirm {
    double getTotal();
    String getCreditCardNo();
}

```

Create a `ConfirmBean` class in the `src/hot` folder in the `shop` package as shown in Listing 10-22. Note that because the class has no state in itself (it simply injects other components into itself), its scope is set to `EVENT`, which is the same as `request`. In addition, the class is annotated with `@Stateless`, meaning that different method calls on the same proxy could delegate to different bean instances. As all `ConfirmBean` instances are considered identical, JBoss will never save or load them to and from the hard disk (and thus has less work to do); all it needs is to create and destroy them.

Listing 10-22. *Seam Component Delegating to Stateless Session Bean*

```

package shop;
...
import javax.ejb.Stateless;

@Stateless
@Name("confirm")
@Scope(ScopeType.EVENT)
public class ConfirmBean implements Confirm {
    @In
    private Credentials credentials;
    @In
    private Cart cart;
}

```

```

    @Override
    public String getCreditCardNo() {
        return null;
    }
    @Override
    public double getTotal() {
        return 0;
    }
    @Remove
    public void destroy() {

    }
}

```

Next, implement the `getTotal()` method as shown in Listing 10-23. Here, you inject the identity manager into the `confirm` bean. This entity manager component is provided by Seam. You can consider the entity manager a connection to your database. Using it, you can issue queries, updates, and so on. Here, you select the `Product` object whose ID is specified, so you can find out the price of the product to add to the total amount.

Listing 10-23. *Injecting the EntityManager*

```

package shop;
...
import javax.persistence.EntityManager;
import javax.persistence.Query;

@Stateless
@Name("confirm")
@Scope(ScopeType.EVENT)
public class ConfirmBean implements Confirm {
    @In
    private Credentials credentials;
    @In
    private Cart cart;
    @In
    private EntityManager entityManager;

    @Override
    public String getCreditCardNo() {
        return null;
    }
}

```

```

@Override
public double getTotal() {
    Query q = entityManager
        .createQuery("select p from Product p where p.id=:id");
    double total = 0;
    for (Long pid : cart.getPids()) {
        q.setParameter("id", pid);
        Product p = (Product) q.getSingleResult();
        total += p.getPrice();
    }
    return total;
}
@Remove
public void destroy() {

}
}

```

To implement the `getCreditCardNo()` method, for simplicity, let's hard-code the credit card number for user `u1` instead of looking up a user database, as shown in Listing 10-24.

Listing 10-24. *getCreditCardNo() Method*

```

package shop;
...
@Stateless
@Name("confirm")
@Scope(ScopeType.EVENT)
public class ConfirmBean implements Confirm {
    @In
    private Credentials credentials;
    @In
    private Cart cart;
    @In
    private EntityManager entityManager;

    @Override
    public String getCreditCardNo() {
        if (credentials.getUsername().equals("u1")) {
            return "1234";
        }
        return "unknown";
    }
}

```

```

    }
    @Override
    public double getTotal() {
        Query q = entityManager
            .createQuery("select p from Product p where p.id=:id");
        double total = 0;
        for (Long pid : cart.getPids()) {
            q.setParameter("id", pid);
            Product p = (Product) q.getSingleResult();
            total += p.getPrice();
        }
        return total;
    }
    @Remove
    public void destroy() {

    }
}

```

Now, run the application, and try to check out. Unfortunately, the attempt will fail with the following error messages:

```

Caused by: org.jboss.seam.RequiredException: @In attribute requires non-null value:
    cart.productHome
    at org.jboss.seam.Component.getValueToInject(Component.java:2297)
    at org.jboss.seam.Component.injectAttributes(Component.java:1703)
    at org.jboss.seam.Component.inject(Component.java:1521)

```

You can't check out yet, because when Seam tries to call `getPids()` on the cart component, Seam will try to inject the `productHome` component into cart component. But there is no `productHome` component, which causes an error. As `getPids()` doesn't really need the `productHome` component, you can tell Seam to not to treat this as an error (see Listing 10-25).

Listing 10-25. *Marking an Injected Field As Unrequired*

```

package shop;
...
@Stateful
@Name("cart")
@Scope(ScopeType.SESSION)

```

```
public class CartBean implements Cart {
    private List<Long> pids;
    @In(required=false)
    private ProductHome productHome;

    public CartBean() {
        pids = new ArrayList<Long>();
    }
    @Override
    public void add() {
        Long pid = productHome.getProductId();
        pids.add(pid);
    }
    @Override
    public List<Long> getPids() {
        return pids;
    }
    @Remove
    public void destroy() {

    }
}
```

If you try again now, you'll get the following error in the console:

```
Caused by: java.lang.NullPointerException
    at shop.ConfirmBean.getCreditCardNo(ConfirmBean.java:27)
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at ...
```

This time, the user hasn't logged in yet, and thus the user name is null. To solve this problem, you should force the user to log in before viewing the confirmation page. To do that, create the `confirm.page.xml` file along with the `confirm.xhtml` file. The content is shown in Listing 10-26.

Listing 10-26. *Requiring Authenticated Access to a Page*

```
<?xml version="1.0" encoding="UTF-8"?>
<page xmlns="http://jboss.com/products/seam/pages"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://jboss.com/products/seam/pages
```

```
http://jboss.com/products/seam/pages-2.1.xsd"
    login-required="true">
</page>
```

Now, try to check out again, and the application should force you to log in. Log in as u1 to see the total amount and the credit card number properly.

Using WebLogic, WebSphere, or GlassFish

Even though you used JBoss in this chapter, the code you developed should run fine on other EJB3 servers such as WebLogic, WebSphere, or GlassFish. However, for each particular type of server, you may need to perform some specific setup adjustments. Consult the Seam documentation for more information.

Summary

In the chapter, you've learned that if a class is annotated with `@Entity`, its instances can be easily saved to or loaded from a database table with JBoss. Seam can generate pages for such a class for searching and listing the instances and for viewing and editing of a particular instance. You can further annotate its fields to specify constraints.

You also learned that you can use session beans to achieve better scalability. If a bean instance contains states, make it a stateful bean; otherwise, make it a stateless bean. Whether your bean is stateful or stateless, JBoss will create and destroy the bean instances, and the application will access a bean instance using a proxy. To save memory when working with stateful beans, JBoss may save bean instances to the hard disk and load them back when they are needed again. JBoss will have less work to do for a stateless bean: it can simply destroy and create them.

You learned, too, that a Seam component is like a web bean. You can inject one component into another by name. In addition, you now know how to turn a session bean (stateful or stateless) into a Seam component, so the component will delegate method calls to the proxy, which will further delegate to the bean instance. You also saw how to annotate a method with `@Remove` to allow Seam to destroy the bean instance when it destroys the component.

You learned that you can let Seam automatically generate a skeleton application, complete with a common layout, login page, and many built-in components such as `Credentials` (user name and password) and `EntityManager` (connection to database). You also explored some of Seam's powerful tags, specifically `<s:decorate>` and `<s:button>`. You can now use the former to set up JSF validators according to the constraints you specified on the field, add an asterisk to indicate a required field, and display the error message (if any). And you can use the latter to specify the next view ID directly without using

a navigation rule and to maintain some information for the next page through a query parameter.

Finally, you learned that, in order to specify what information to maintain for the next page, to require authenticated access to a page, or to restrict access to the users with a particular role, you can create a file `<PAGE-NAME>.page.xml` in the same folder as your page.

Index

A

- `<a4j:button>` tag, 201
- `<a4j:commandButton>` tag, 195
- `<a4j:commandLink>` tag, 189, 194, 199
- `<a4j:repeat>` tag, 212
- `<a4j:support>` tag, 199
- `AbortProcessingException`, 98
- action attribute, 38, 107, 225
- action listener
 - creating for the e-shop project, 109
 - implementing the `ActionListener` interface, 98
 - modifying for the e-shop project, 112
 - moving UI-specific code into, 98
- action provider, 162
- `add()`, 269
- `addToCart()`, 117, 120
- admin user group, 261
- Ajax
 - `<a4j:button>` tag, 201
 - `<a4j:commandButton>` tag, 195
 - `<a4j:commandLink>` tag, 189, 194, 199
 - `<a4j:repeat>` tag, 212
 - `<a4j:support>` tag, 199
 - assigning an ID to a component for identification, 186
 - changing a component's visibility, 191
 - displaying a FAQ, 183
 - `FAQService` class, 184, 195, 209–210, 212
 - `foo.taglib.xml`, 208
 - `<h:dataTable>` tag, 212
 - `<h:messages>` tag, 196
 - `isShortForm` flag, 185
 - `isShowingAnswer()`, 190
 - `listfaq.xhtml`, 183, 189, 193, 196, 200, 206, 210–211
 - onclick event, 186
 - `qa.xhtml`, 207
 - `Question` class, 209
 - question parameter, 207

- rating a question and answer, 194
- refreshing a partial or entire page, 186
- rendered attribute, 190, 192
- `reRender` attribute, 189, 196
- RichFaces component library, 187
- `<rich:modalPanel>` tag, 200
- `<rich:panel>` tag, 205
- `trigger()`, 185
- updating a panel, 192
- updating a rating as the user types, 199
- using a dialog box to get a rating, 200
- Ant, downloading version 1.7.0 or newer, 8
- Application object, 239
- Apply Request Values phase, 56, 74, 110
- `authenticate()`, 261
- Authenticator class, 267
- authenticator component, 261

B

- b1 bean, 32
 - adding the `quoteDate` property, 52–53
 - defining, 35
 - determining the stock value from, 39
- b2 bean, creating, 61
- Back button, 215, 219, 228
- `basename` attribute, 233
- base page, 264
- `base.xhtml`
 - creating a base page, 175
 - inheriting from, 176
- `beans.xml`, 23
- `begin()`, 223
- Big5 encoding, 233
- box component
 - `<box>` tag, defining, 165
 - `box.xhtml`, 165
 - creating, 163

- p1.xhtml, 164
- passing XHTML code as the tag body, 164
- passing XHTML code through a
 - parameter, 163
- <ui:insert> tag, 165
- See also* components

C

- calculateLocale(), 249
- Cart class, 119, 126, 269, 273
- CartBean class, 274–276
- cart.page.xml, 272
- cart.xhtml, 121, 129, 270–272, 278
- Catalog class, 103, 125
- catalog.xhtml, 102, 109, 114, 116, 132, 146, 148
- Change button, 242
- Checkout button, creating, 129
- Chinese language support, 232
 - Big5 encoding, 233
 - localizing the full stop, 243
 - using Chinese fonts, 236
 - using the Chinese version of a resource bundle, 235
- client IDs
 - code example, 46
 - specifying, 89
 - specifying a relative client ID, 94
 - storing with error messages, 88
- combo box, choosing stock symbols from, 60
- components
 - <component> tag, 156
 - creating a box component, 163
 - creating a component library without taglib.xml, 170
 - creating a product editor, 159
 - creating a reusable component library, 168
 - creating custom components, 151
 - displaying a copyright notice on multiple pages, 151
 - grouping multiple components into one, 91
 - passing method parameters to custom tags, 162

- providing parameters to a custom tag, 157
- specifying the validation error message
 - for a single component, 77
- UI Command, 38–39, 41, 107, 186
- UI Data, 105
- UI Form, 31, 97
- UI Input, 32, 35, 39–41, 46–47, 49, 51, 91
- UI Load Bundle, 233, 235
- UI Message, 88, 90, 95
- UI Output, 105, 186, 190, 238, 243–244
- UI Panel, 68, 91
- UI Parameter, 244
- UI View Root, 235, 238, 240, 247, 249
- See also* box component
- <composite:attribute> tag, 170
- <composite:implementation> tag, 170
- <composite:interface> tag, 170
- confirm page, 127
- ConfirmBean class, 279
- confirm.page.xml, 283
- ConfirmService class, 129, 138
- confirm.xhtml, 128, 140, 278, 283
- conversation scope, 215
 - conversation, definition of, 219
 - making a conversation long-running, 222–223
 - transient vs. long-running conversations, 220
 - turning a conversation back into a transient one, 223
- converters, 238
- copyright notice
 - <component> tag, 156
 - copyright.xhtml, 152, 156
 - creating the CustomComp project, 154
 - defining a Facelet tag lib, 154–155
 - defining and naming a tag library, 154
 - defining the namespace, 153
 - developing a custom <copyright> tag, 153
 - displaying on multiple pages, 151
 - extracting common code into a separate XHTML file, 151
 - foo.taglib.xml, 154
 - p1.xhtml, 154

- `<copyright>` tag
 - complete XHTML page for, 156
 - developing, 153
 - outputting the company parameter in
 - `copyright.xhtml`, 157
- Credentials component, 261
- `@Current` annotation, 117, 261
- CustomComp project
 - creating, 154
 - `p2.xhtml`, 169
 - packing the META-INF folder into a JAR file for exporting, 169
- custom tags
 - accepting method parameters, 162
 - accepting two pieces of XHTML code, 166
 - `p1.xhtml`, 166
 - `<pair>` tag, defining in `foo.taglib.xml`, 167
 - `pair.xhtml`, 166
 - `<ui:define>` tag, 166
 - `<ui:insert>` tag, 166

D

- date display
 - internationalizing, 238
 - setting the date style to long, 238
 - table of formats and styles, 245
- Date object
 - converting into a string, 50
 - Date converter and conversion failure, 55
 - Date converter, using, 51
 - inputting, 49
 - specifying the date format, 53–54
- `DateTimeConverter` class, 58
- dependencies
 - dependency injection, 118
 - object, 118
 - pulling, 118
- `destroy()`, 276
- `detail.xhtml`, 108, 113, 117, 124
- `DoubleRangeValidator`, 78, 262

E

- Eclipse
 - debugging a JSF application, 25
 - downloading, 2
 - installing, 2
 - launching JBoss in, 3
- Navigator view, 29
- Properties Editor, installing and using, 233
- setting a breakpoint in Java code, 25
- switching to the Java EE perspective, 3
- EL expression
 - accessing a parameter in, 157
 - definition of, 20
 - EL variable, 157
 - evaluating, 105
 - linking to a variable table, 158
 - using directly in body text, 25
- encoding, definition of, 18
- `end()`, 224
- `@Entity` annotation, 257, 259
- `EntityHome` class, 271
- error messages
 - `AbortProcessingException`, 98
 - creating a `messages.properties` text file, 48, 58
 - customizing error messages using a message bundle, 76
 - displaying in red, 86
 - displaying with a field, 87
 - `<h:messages>` tag, 75
 - specifying a conversion error message, 59
 - specifying for a single UI Input component, 49
 - specifying the validation error message for a single component, 77
 - storing with client IDs, 88
 - `ValidatorException`, 80
 - `validatorMessage` attribute, 92
- escaped Unicode encoding, 233
- e-shop project
 - accessing an attribute like a Web Bean, 105
 - accessing a selected Product object, 111
 - action attribute, 107
 - action listener, 109, 112
 - adding products to the shopping cart, 116
 - `addToCart()`, 117, 120
 - allowing the user to check out, 127
 - appending the row index to the client ID, 110

- Cart class, 119, 126
- cart.xhtml, 121, 129
- Catalog class, 103, 125
- catalog.xhtml, 102, 109, 114, 116, 132, 146, 148
- confirm page, 127, 138, 140, 143, 146
- ConfirmService class, 129, 138
- confirm.xhtml, 128, 140
- creating the Checkout button, 129
- creating the link to show the product details, 106
- @Current annotation, 117
- defining the navigation case for a successful login, 137
- detail.xhtml, 108, 113, 117, 124
- displaying a password as asterisks, 148
- displaying column headers, 115
- displaying the shopping cart's contents, 126
- faces-config.xml, 120, 130, 133, 142
- ForceLoginPhaseListener class, 141
- forcing the user to log in, 139
- getProduct(), 125
- getProductId(), 124
- getProductIds(), 126
- getting the user's credit card number, 128, 131
- <h:commandButton> tag, 107
- <h:commandLink> tag, 106–107, 110
- <h:dataTable> tag, 102, 126
- <h:inputHidden> tag, 123
- <h:inputSecret> tag, 148
- <h:inputText> tag, 123–124, 148
- <h:outputText> tag, 126
- <h:panelGrid> tag, 102
- implementing a firewall as a phase listener, 141
- implementing logout, 146
- implementing Serializable, 119, 136
- listing the products, 102
- Login link, 132, 146, 148
- login page, 132, 135, 137, 140, 143, 146
- login.xhtml, 134, 140, 148
- LoginRequest class, 135
- Logout link, 146, 148
- LogoutActionListener class, 147
- making the grid visible, 106
- NullPointerException, 121

- OnDetailActionListener class, 111, 115
- printing the product ID to the console, 112
- Product class, creating, 104
- ProductHolder class, 112, 117, 120, 124
- providing the List to the dataTable, 104
- putting a shopping cart into the session, 119
- recreating with EJBs and Seam, 254
- removing a session, 146
- returning loggedIn as an outcome, 146
- session timeout, 119
- @SessionScoped, 119
- storing the original view ID in the UserHolder Web Bean, 143
- storing the Product object in a Web Bean, 112
- UI Data, 105
- <ui:repeat> tag, 126
- User class, 136
- User object, 131, 139
- UserHolder class, 136
- UserHolder Web Bean, 135, 143
- var attribute, 105
- view ID, 135, 140, 143, 146

F

- Facelet tag lib, defining, 154–155
- Facelets, downloading and installing, 187
- faces-config.xml, 77, 120, 130, 133, 142, 235, 250
 - configuring the supported languages, 54
 - creating, 15
 - defining a navigation rule, 37
 - enabling Facelets in, 187
 - matching any source view ID, 177
 - modifying to load a properties file, 48
- FacesContext, 98
- FacesMessage, 80, 87, 98
- FAQ project
 - <a4j:button> tag, 201
 - <a4j:commandButton> tag, 195
 - <a4j:commandLink> tag, 189, 194, 199
 - <a4j:repeat> tag, 212
 - <a4j:support> tag, 199
- adding a title bar (header) to a modal panel, 203

- displaying a FAQ, 183
- displaying an invalid-entry error, 196
- displaying multiple questions, 206
- encapsulating questions inside a
 - custom tag, 206
- foo.taglib.xml, 208
- <h:dataTable> tag, 212
- hiding a modal panel if there is no error, 202
- <h:messages> tag, 196
- isShortForm flag, 185
- isShowingAnswer(), 190
- listfaq.xhtml, 183, 189, 193, 196, 200, 206, 210–211
- oncomplete property, 201
- qa.xhtml, 207
- Question class, 209
- question parameter, 207
- Rate link, 200, 202
- rating a question and answer, 194
- refreshing a partial or entire page, 186
- rendered attribute, 190, 192
- reRender attribute, 189, 196
- RichFaces component library, 187
- <rich:modalPanel> tag, 200
- <rich:panel> tag, 205
- showing or hiding a modal panel, 201
- showing or hiding an answer in a Web Bean, 190
- trigger(), 185
- updating a rating as the user types, 199
- using a dialog box to get a rating, 200
- using Ajax to change a component's visibility, 191
- using Ajax to update a panel, 192
- FAQService class, 184, 195, 209–210, 212
- finish(), 224
- Finish button, 223, 228
- Firefox
 - setting the preferred language, 54
 - using the Chinese version of a resource bundle, 235
- <f:loadBundle> tag, 233
- foo.taglib.xml, 83, 154, 159, 208
- foo.v1 validator, 84
- ForceLoginPhaseListener class, 141

forms

- Apply Request Values phase, 56, 74, 110
- Date converter and conversion failure, 55
- form submission process, 33, 35
- Input Processing phase, 39, 41, 50, 56
- inputting a Date object, 49
- Invoke Application phase, 40–41, 98–99
- Process Validations phase, 74
- QuoteRequest class, 64
- Render Response phase, 40–41, 50, 56, 74, 140, 190
- Update Domain Values phase, 43, 50, 99
- See also* input validation
- <f:param> tag, 244
- full stop, localizing, 243
- <f:validateLongRange> tag, 75

G

- getCreditCardNo(), 281
- getDisplayName(), 241
- getLocale(), 247
- getPids(), 282
- getProduct(), 125
- getProductId(), 124
- getProductIds(), 126
- getrequest.xhtml, 67, 75, 86, 94, 96
- getSubject(), 20–21, 23
- getsymbol.xhtml
 - creating, 31
 - modifying, 34, 37, 52
 - redisplaying, 41
 - specifying the label instead of the client ID, 47
 - using a combo box, 60
- getTotal(), 280
- GlassFish, 284
- GreetingService class
 - accessing the subject property, 21
 - creating, 20
 - placing a GreetingService object into the Web Bean table, 22
- group renderer, 91

H

- <h:commandButton> tag, 72, 107, 268
- <h:commandLink> tag, 106–107, 110

- <h:dataTable> tag, 87, 102, 126, 212
- Hello world application, creating with JSF, 1, 9–10, 12–13, 15, 17
- hello.xhtml
 - accessing, 16
 - creating, 12
 - modifying, 17
- <h:form> tag, 94, 268
- <h:graphicImage> tag, 246, 248
- <h:inputHidden> tag, 123
- <h:inputSecret> tag, 148
- <h:inputText> tag, 70, 123–124, 148, 217
- <h:inputTextarea> tag, 217
- <h:message> tag, 88, 94
- <h:messages> tag, 75
 - CSS classes and, 87
 - placing inside a panel, 197
 - refreshing, 196
- home.xhtml, 180
 - creating, 173
 - inheriting from base.xhtml, 176
- hotdeals.xhtml, creating, 181
- <h:outputFormat> tag, 244
- <h:outputText> tag, 126
- <h:panelGrid> tag, 68, 90–91, 102
- <h:panelGroup> tag, 91
- HTML grid renderer, 91
- HTML renderer and components, 68
- HTTP requests, 32, 39



- @Id annotation, 257
- id attribute, 46
- tag, 246
- implicit object, definition of, 247
- @In annotation, 261, 269
- Input Processing phase, 39, 41, 50, 56
- input validation
 - adding validator objects to a UI Input component, 74
 - catching invalid input, 73
 - creating a custom validator for the patron code, 82
 - creating a DoubleRangeValidator, 78
 - creating a LengthValidator, 78
 - creating a long range validator, 75
 - defining the foo.v1 validator, 84
 - <f:validateLongRange> tag, 75

- handling optional input, 74
- <h:messages> tag, 75
- invoking an action method for validation, 96
- moving UI-specific code into an action listener, 98
- null input and validators, 78
- RequestValidatingListener class, 98
- specifying a tag lib, 83
- specifying a validator method, 80
- specifying the validation error message for a single component, 77
- validate(), 84
- validatePatron(), 80, 86
- <validatePatron> tag, 83
- validating a combination of input values, 96
- See also* forms
- installing Seam, 253
- Invoke Application phase, 40–41, 98–99
- isShortForm flag, 185
- isShowingAnswer(), 190

J

- JBoss
 - choosing the JBoss runtime environment, 5
 - downloading the JBoss Application Server 5.x, 3
 - installing, 3–4, 6–7
 - installing Web Beans into, 8
 - launching in debug mode, 26
 - launching in Eclipse, 3
 - registering the Hello world application, 15
 - RichFaces component library, 187
 - setting a longer timeout value, 7
 - stopping, 7
 - telling Web Beans where JBoss is, 8
 - updating a web application, 19
- JBoss Seam. *See* Seam
- JSF
 - displaying the available Web Beans classes, 11
 - downloading Mojarra, 7
 - encoding, definition of, 18
 - faces-config.xml, creating, 15
 - hello.xhtml, creating, 12
 - hello.xhtml, modifying, 17

- installing Sun's JSF implementation, 7
- MyFaces, 7
- Package Explorer, 9
- using the XHTML strict template, 11
- WebContent folder, 11
- web.xml, modifying, 13
- JSF Core tag lib, 156
- JSF HTML tag lib, 156

L

- languages. *See* MultiLang project
- Layout project
 - base.xhtml, 175–176
 - creating a hot deals page, 181
 - creating navigation rules for links, 177
 - creating page-specific navigation cases, 180
 - extracting duplicate XHTML code, 174
 - faces-config.xml, 177
 - home.xhtml, 173, 176, 180
 - hotdeals.xhtml, 181
 - matching any source view ID, 177
 - page inheritance, 175
 - products.xhtml, 173, 176, 180–181
 - providing concrete parts, 180
 - <ui:insert> tag, 175
 - using two abstract parts, 178–179
- LengthValidator, creating, 78
- listfaq.xhtml, 183, 189, 193, 196, 200, 206, 210–211
- Locale object, 239, 249
- localization, 237, 250
- loggedIn, returning as an outcome, 146
- Login link, 132, 146, 148
- login.xhtml, 134, 140, 148
- LoginRequest class, 135
- logo_en.gif, 246
- logo_zh.gif, 246
- Logout link, 146, 148
- LogoutActionListener class, 147
- long range validator, creating, 75
- LongRangeValidator class, 76

M

- Map, accessing the parameters in, 171
- message bundle
 - providing a detail message in, 93
 - specifying, 76

- messages.properties, creating, 48, 58
- method parameters
 - calling, 162
 - passing to custom tags, 162
- @Min annotation, 257, 262
- Mojarra
 - downloading, 7
 - *.taglib.xml files and Mojarra 2.0.0.PR2, 83
- msgs_en.properties, 236
- msgs_zh.properties, 233, 236, 242
- msgs.properties, 232, 236, 242
- MultiLang project
 - accessing map elements using dot notation, 237
 - Application object, 239
 - basename attribute, 233
 - Big5 encoding, 233
 - calculateLocale(), 249
 - Change button, 242
 - changing the preferred language, 238
 - displaying a logo on a page, 246
 - displaying the current date and time, 231
 - escaped Unicode encoding, 233
 - faces-config.xml, 235, 250
 - <f:loadBundle> tag, 233
 - <f:param> tag, 244
 - getDisplayName(), 241
 - getLocale(), 247
 - getting the display name of the locale, 241
 - <h:graphicImage> tag, 246, 248
 - <h:outputFormat> tag, 244
 - tag, 246
 - internationalizing the date display, 238
 - letting users change the displayed language, 238
 - Locale object, 239, 249
 - localizing the full stop, 243
 - localizing validation messages, 250
 - logo_en.gif, 246
 - logo_zh.gif, 246
 - making a locale change persistent, 248
 - msgs_en.properties, 236
 - msgs_zh.properties, 233, 236, 242
 - msgs.properties, 232, 236, 242
 - multilang.msgs, 233

- MultiViewHandler class, 249
- MyApp.properties, 250
- MyApp_zh.properties, 251
- MyViewHandler class, creating, 249
- reading messages from a resource bundle, 234
- setting the date style to long, 238
- ShowDate class, 232, 239, 241
- showdate.xhtml, 231, 233, 238–239, 242–244, 246–247
- specifying the default and supported languages, 235
- storing a language code in a session, 248
- supporting the Chinese language, 232
- table of formats and styles, 245
- toString(), 238–239
- using the Chinese version of a resource bundle, 235
- using the Eclipse Properties Editor, 233
- value attribute, 246, 248
- view variable, 247
- web.xml, 234

- MultiViewHandler class, 249
- MyApp.properties, 250
- MyApp_zh.properties, 251
- MyFaces, 7
- MyViewHandler class, creating, 249

N

- @Name annotation, 261
- name attribute, 46
- namespace, defining, 153
- navigation
 - creating navigation rules for links, 177
 - creating page-specific navigation cases, 180
 - faces-config.xml, 177
 - matching any source view ID, 177
 - navigation rules, defining, 37, 73
 - providing concrete parts, 180
 - using two abstract parts, 178–179
- Navigator view, 29
- next(), 222
- Next button, 215, 219, 222, 225
- @NotEmpty annotation, 257, 262
- NullPointerException, 121

O

- object dependencies, 118
- onclick event, 186
- oncomplete property, 201
- OnDetailActionListener class, 111, 115
- onOK(), 96, 98
- onUpdate(), 161
- outputText tag, 21, 24

P

- p1.xhtml, 154, 164, 166, 171
- p2.xhtml, 169
- Package Explorer, 9
- page inheritance, 175
- <pair> tag, defining in foo.taglib.xml, 167
- pair.xhtml, 166
- <param> tag, 272
- passwords, displaying as asterisks, 148
- patronExists(), 81–82
- PatronValidator class, 84
- <pe> tag
 - defining, 159, 170
 - using in the bar tag lib, 171
- pe.xhtml, 170
- postage calculator application
 - creating a custom validator for the patron code, 82
 - creating the results page, 72
 - developing, 67
 - <f:validateLongRange> tag, 75
 - getrequest.xhtml, 67
 - <h:commandButton> tag, 72
 - <h:inputText> tag, 70
 - <h:messages> tag, 75
 - <h:panelGrid> tag, 68
 - HTML renderer, 68
 - linking the request properties and the UI Input components, 71
 - marking the weight as required, 78
 - navigation rule, defining, 73
 - patronExists(), 81–82
 - PatronValidator class, 84
 - PostageService class, 71
 - Request bean, 71
 - Request class, 70
 - running the application, 73
 - showpostage.xhtml, 72
 - specifying a validator method, 80

- <table> element, 67
 - validatePatron(), 86
 - <validatePatron> tag, 83
 - validating the patron code, 80
- Postage.properties, 76, 86
- PostageService class, creating, 71
- Process Validations phase, 74
- processAction(), 98
- Product class, 104, 257
- product editor
 - currentProduct Web Beans, creating, 160
 - foo.taglib.xml, 159
 - onUpdate(), 161
 - passing an object to a custom tag, 159
 - <pe> tag, 159–160
 - pe.xhtml, creating, 160
 - Product class, creating, 160
- Product List link, 259
- Product.xhtml, 259, 268
- ProductEdit.page.xml, 260, 265
- ProductEdit.xhtml, 259, 263
- ProductHolder bean, 135
- ProductHolder class, 112, 117, 120, 124
- ProductHome class, 269, 271
- ProductList.xhtml, 259
- products.xhtml, 173, 176, 180–181
- pulling dependencies, 118

Q

- qa.xhtml, 207
- Question class, creating, 209
- question parameter, 207
- quoteDate property, 52–53
- QuoteRequest class, 35, 63–64

R

- Rate link, 200, 202
- redirect after post, 226–227
- @Remove annotation, 276
- Render Response phase, 40–41, 50, 56, 74, 140, 190
- rendered attribute, 190, 192
- Request bean, 71
- Request class, 70, 86, 98–99
- request scope, 185
- RequestValidatingListener class, 98
- required attribute, setting to true, 59

- reRender attribute, 189, 196
- resource bundle
 - reading messages from, 234
 - using the Chinese version of a resource bundle, 235
- <restrict> tag, 266
- RichFaces component library
 - downloading and installing, 187
 - JSF 2.0 and, 187
 - modifying web.xml, 188
 - predefined skins, 205
 - support for skins, 204
- <rich:modalPanel> tag, 200
- <rich:panel> tag, 205

S

- <s:button> tag, 268–269, 271–272
- <s:decorate> tag, 263
- Seam
 - accessing EJBs in EL expressions, 253
 - add(), 269
 - adding getters and setters, 258
 - adding products to the shopping cart, 267
 - adding the product ID to the shopping cart, 269
 - admin user group, 261
 - authenticate(), 261
 - Authenticator class, 267
 - authenticator component, 261
 - base page, 264
 - Cart class, 269, 273
 - CartBean class, 274–276
 - cart.page.xml, 272
 - cart.xhtml, 270–272, 278
 - confirm.page.xml, 283
 - confirm.xhtml, 278, 283
 - ConfirmBean class, 279
 - creating a page for adding products, 259
 - creating an Eclipse project, 256
 - creating the cart component, 269
 - creating the checkout page, 277
 - creating the Confirm interface, 279
 - Credentials component, 261
 - @Current annotation, 261
 - destroy(), 276
 - destroying the cart component, 275
 - displaying the welcome page, 256

- DoubleRangeValidator, 262
- @Entity annotation, 257, 259
- EntityHome class, 271
- getCreditCardNo(), 281
- getPids(), 282
- getTotal(), 280
- GlassFish, 284
- having a configuration file for each
 - page, 260
- <h:commandButton> tag, 268
- <h:form> tag, 268
- @Id annotation, 257
- implementing the methods for the
 - shopping cart, 276
- @In annotation, 261, 269
- injecting a nonexistent component,
 - 271
- injecting the EntityManager, 280
- installing, 253
- @Min annotation, 257, 262
- @Name annotation, 261
- @NotEmpty annotation, 257, 262
- <param> tag, 272
- performing CRUD operations, 253
- Product class, mapping to a table,
 - 257
- Product List link, 259
- Product.xhtml, 259, 268
- ProductEdit.page.xml, 260, 265
- ProductEdit.xhtml, 259, 263
- ProductHome class, 269, 271
- ProductList.xhtml, 259
- recreating the e-shop project using EJBs
 - and Seam, 254
- @Remove annotation, 276
- requiring authenticated access to a
 - page, 260, 283
- <restrict> tag, 266
- restricting access to the product-editing
 - page, 265
- <s:button> tag, 268–269, 271–272
- <s:decorate> tag, 263
- seam generate-ui command, 259, 269
- seam new-project command, 256
- seam setup command, 254, 256
- specifying component names, 261
- specifying injections, 261
- @Stateful annotation, 274–275
- @Stateless annotation, 279
 - turning a session bean into a Seam component, 274
 - turning the shopping cart into a stateful session bean, 273
- WebLogic, 284
- WebSphere, 284
- Serializable, 119, 136, 221
- session scope, 185
- session timeout, 119
- @SessionScoped, 119
- ShowDate class, 232, 239, 241
- showdate.xhtml, 231, 233, 238–239,
 - 242–244, 246–247
- showpostage.xhtml, 72
- skins
 - choosing, 205
 - definition of, 204
 - predefined, 205
 - RichFaces component library and, 204
 - web.xml, 205
- @Stateful annotation, 274–275
- @Stateless annotation, 279
- Step1 class, creating, 222
- step1.xhtml, 217, 222, 225–227
- Step2 class, creating, 224
- step2.xhtml, 217, 223, 225–227
- stock quote application
 - b1 bean, 32, 35
 - choosing stock symbols from a combo box, 60
 - determining the stock value from the b1 bean, 39
 - getsymbol.xhtml, 31, 34, 37, 41, 47, 52, 60
 - marking the UI Input component as required, 40
 - QuoteRequest class, 35, 63
 - StockService class, 61, 63–64
 - stockvalue.xhtml, 36, 39
 - sym property, 35
 - updating the project name, 30
- StockService class
 - creating, 61
 - inserting the stock value calculation, 64
 - modifying, 63
- stockvalue.xhtml, 36, 39
- sym property, 35

T

<table> element, 67
 tag library, defining and naming, 154
 taglib.xml, 170
 thankyou.xhtml, 217
 Ticket class, 216
 toString(), 238–239
 trigger(), 185

U

UI Command, 38–39, 41, 107, 186
 UI Data, 105
 UI Form, 31, 97
 UI Input, 32, 35, 39, 41, 51, 91
 adding validator objects to, 74
 client ID and, 46
 displaying a user-friendly description
 for, 47
 marking as required, 40
 specifying an error message for a single
 component, 49
 UI Load Bundle, 233, 235
 UI Message, 88, 90, 95
 UI Output, 20, 105, 186, 190, 238,
 243–244
 UI Panel, 68, 91
 UI Parameter, 244
 UI View Root, 235, 238, 240, 247, 249
 <ui:component> tag, 170
 <ui:define> tag, 166
 <ui:insert> tag, 165–166, 175
 <ui:repeat> tag, 126
 Unicode
 escaped Unicode encoding,
 233
 properties files and, 233
 using the Eclipse Properties Editor,
 233
 Update Domain Values phase, 43,
 50, 99
 User class, 136
 User object, 131, 139
 UserHolder class, 136
 UserHolder Web Bean, 135, 143

V

validate(), 84
 validatePatron(), 80, 86

validator objects
 adding to a UI Input component, 74
 creating a custom validator for the
 patron code, 82
 defining the foo.v1 validator, 84
 null input and validators, 78
 specifying a validator method, 80
 validate(), 84
 validatePatron(), 80
 <validatePatron> tag, 83
 ValidatorException, 80
 validatorMessage attribute, 92
 value attribute, 246, 248
 var attribute, 105
 view ID, 135, 140, 143, 146, 177
 view variable, 247

W

Web Beans
 conversation, definition of, 219
 definition of, 20
 displaying the available Web Beans
 classes, 11
 downloading, 8
 installing into JBoss, 8
 life cycle of, 25
 telling Web Beans where JBoss is, 8
 transient vs. long-running
 conversations, 220
 Web Beans manager, 20, 22–23
 web.xml, 13, 188, 205, 234
 WebContent folder, 11
 WebLogic, 284
 WebSphere, 284
 Wizard project
 action attribute, 225
 Back button, 215, 219, 228
 begin(), 223
 defining the navigation rules for, 218
 end(), 224
 finish(), 224
 Finish button, 223, 228
 <h:inputText> tag, 217
 <h:inputTextarea> tag, 217
 implementing Serializable, 221
 making a conversation long-running,
 222–223
 next(), 222

- Next button, 215, 219, 222, 225
- putting the ticket into the conversation scope, 221
- redirect after post, 226–227
- resolving URL mismatches, 225
- Step1 class, creating, 222
- step1.xhtml, 217, 222, 225–227
- Step2 class, creating, 224
- step2.xhtml, 217, 223, 225–227
- thankyou.xhtml, 217

- Ticket class, creating, 216
- turning a conversation back into a transient one, 223

■ X

XHTML

- extracting common code into a separate XHTML file, 151
- extracting duplicate XHTML code, 174

You Need the Companion eBook

Your purchase of this book entitles you to buy the companion PDF-version eBook for only \$10. Take the weightless companion with you anywhere.

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 (in PDF format) for \$10 to customers who purchase this book now. Convenient and fully searchable, the PDF version of any content-rich, page-heavy Apress book makes a valuable addition to your programming library. You can easily find and copy code—or 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 your 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 \$10.00 eBook.

Apress®
THE EXPERT'S VOICE™



2855 TELEGRAPH AVENUE | SUITE 600 | BERKELEY, CA 94705

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 noncommercial use. The purchaser may place the eBook title on any of their personal computers for their own personal reading and reference.

Offer valid through 11/09.