Beginning JSF™ 2 APIs and JBoss® Seam

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

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 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.

CHAPTER 1

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.

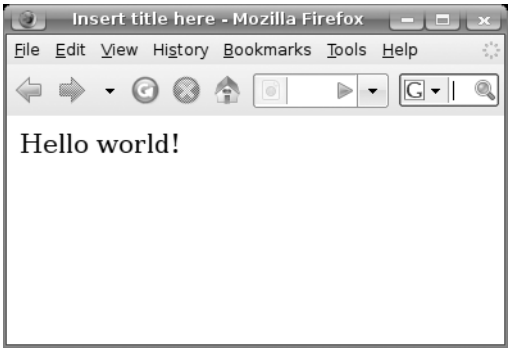

Figure 1-1. *A simple "Hello world!" application with a single page*

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.

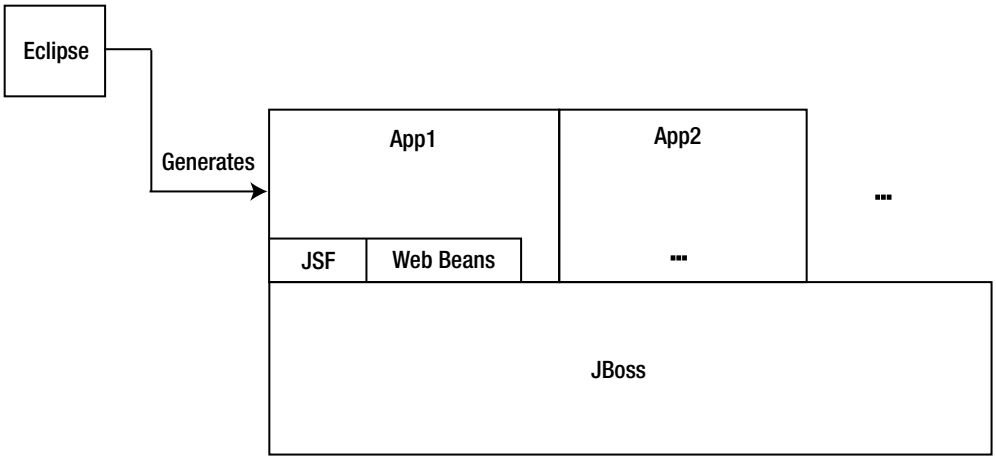

Figure 1-2. *The software that you'll need*

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.

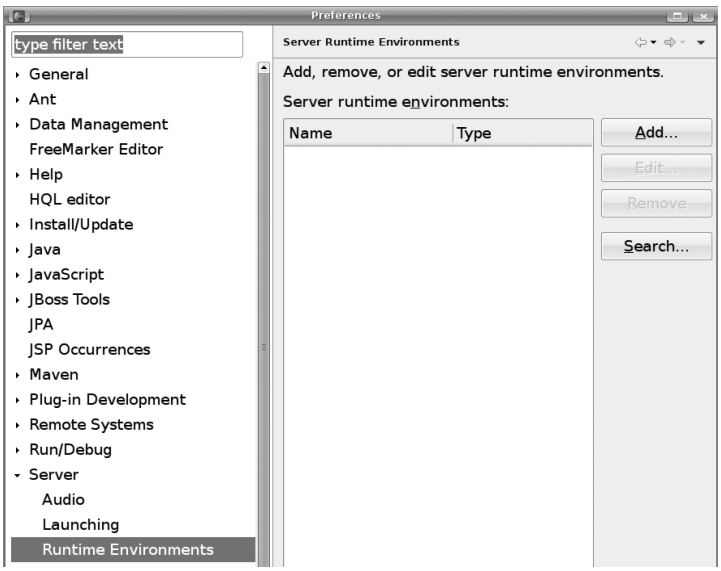

Figure 1-5. *The installed runtime environments*

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

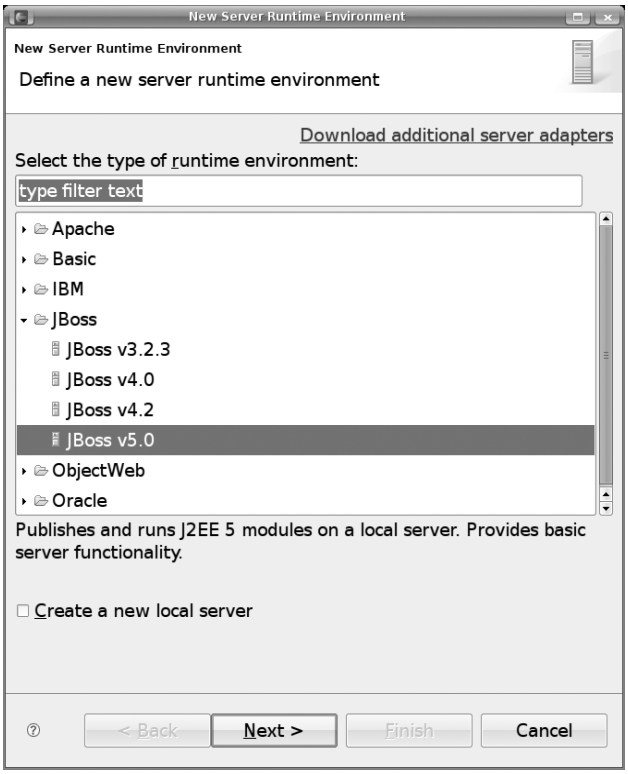

Figure 1-6. *The JBoss 5.0 runtime*

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

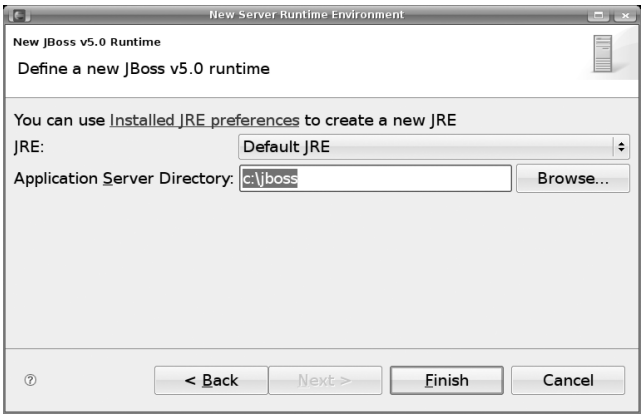

Figure 1-7. *Specifying the JBoss application server directory*

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).

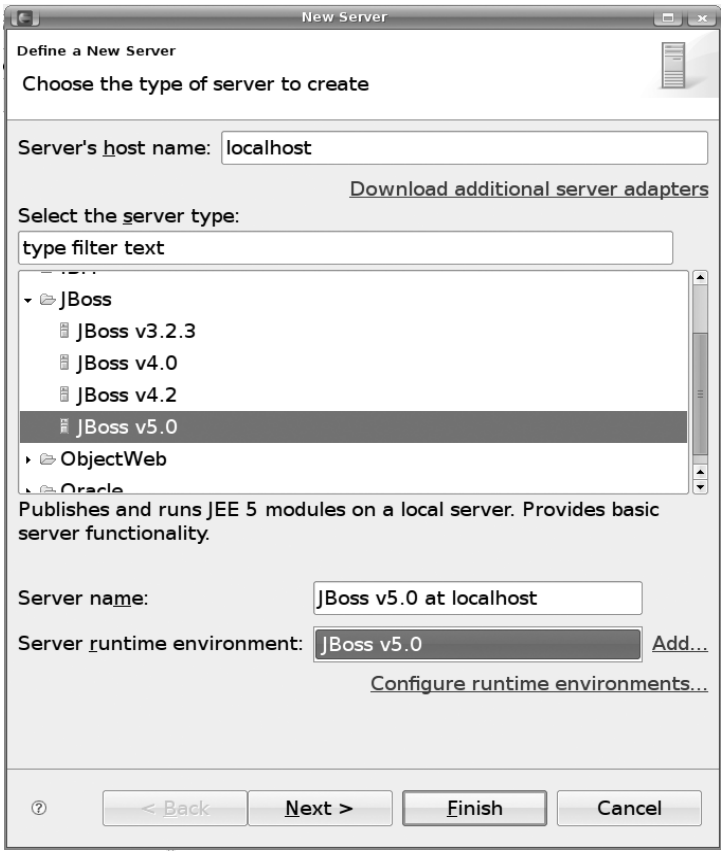

Figure 1-8. *Choosing the JBoss runtime environment*

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

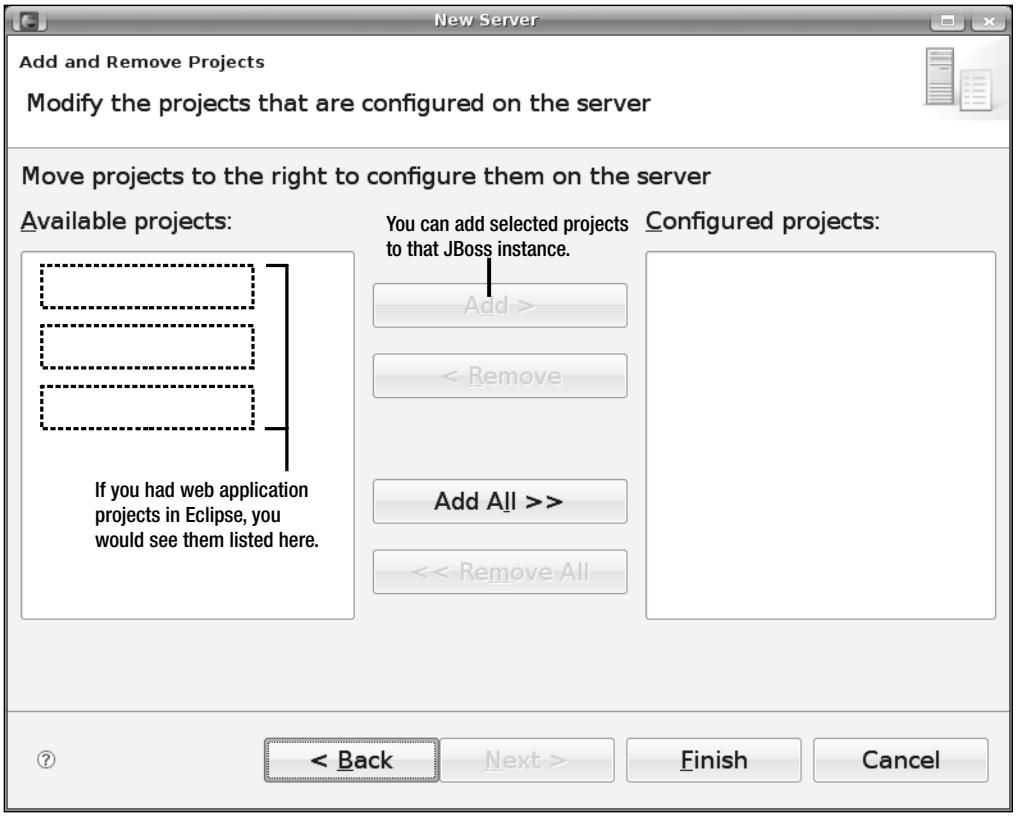

Figure 1-9. *Adding web applications*

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://javaserverfaces.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).

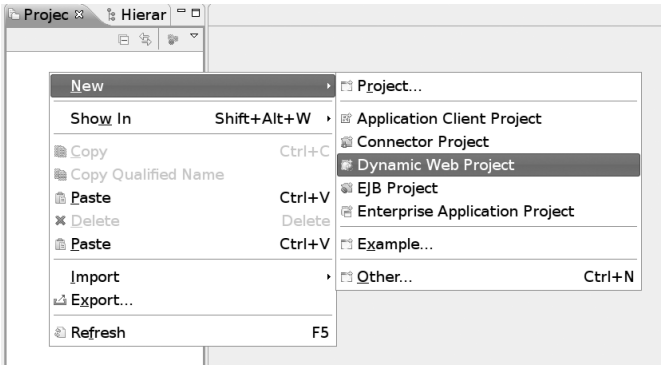

Figure 1-11. *Creating a dynamic web project*

Enter the information shown in Figure 1-12.

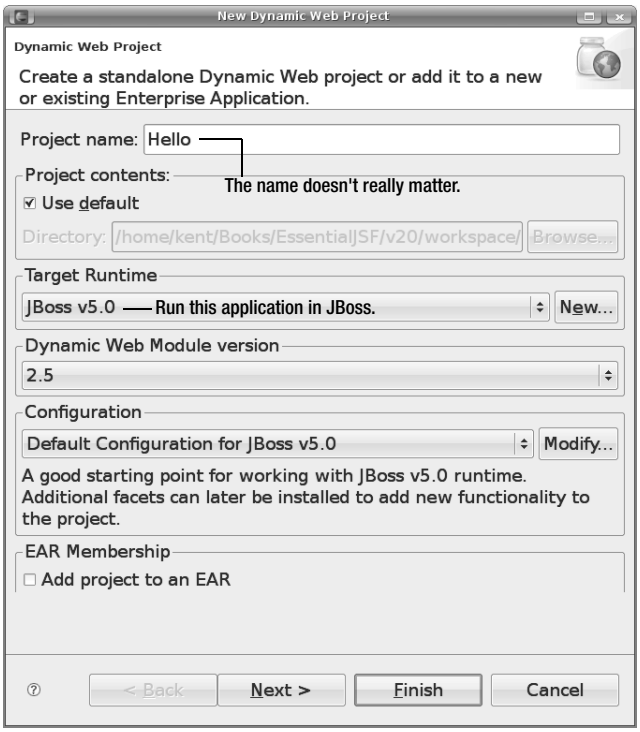

Figure 1-12. *Entering the project information*

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

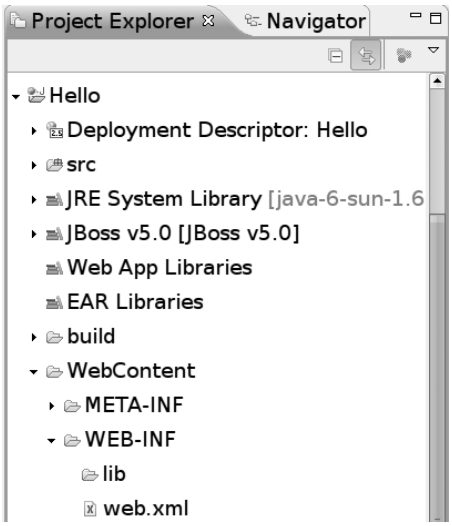

Figure 1-13. *Project structure*

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

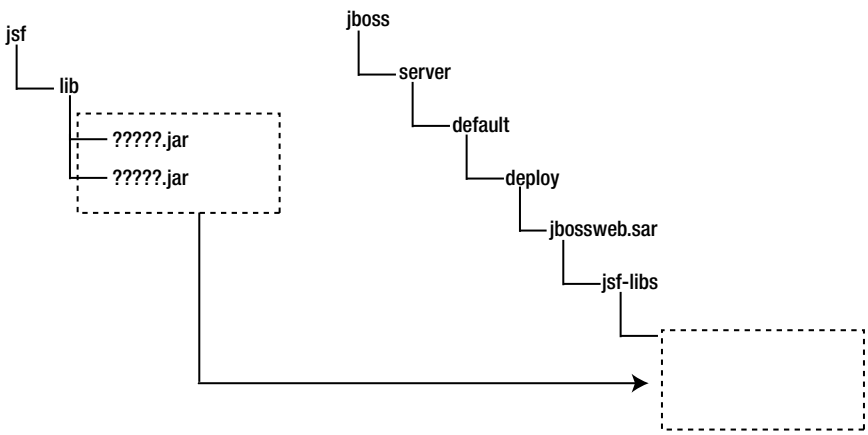

Figure 1-14. *Copying the JAR files into the JBoss*

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.

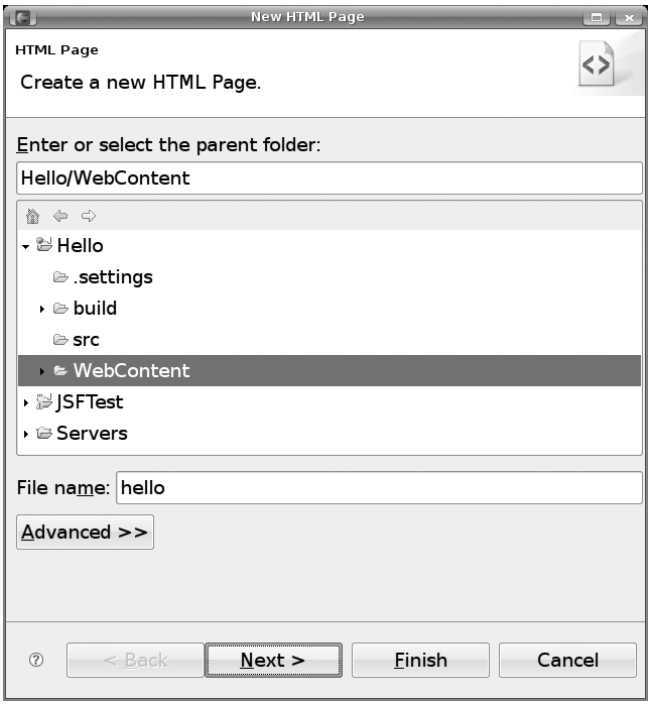

Figure 1-15. *Creating the "Hello world!" page*

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

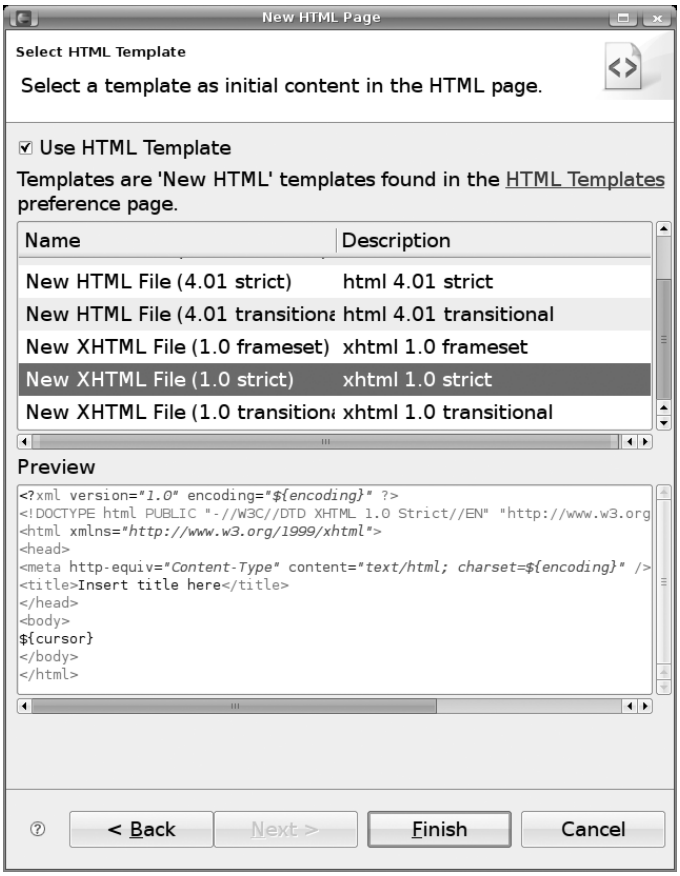

Figure 1-16. *Using the XHTML strict template*

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).

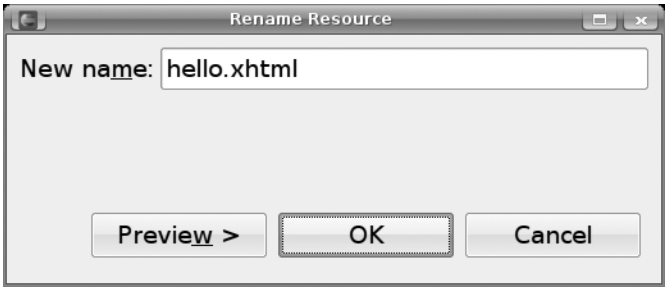

Figure 1-17. *Renaming the file*

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>
```


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.

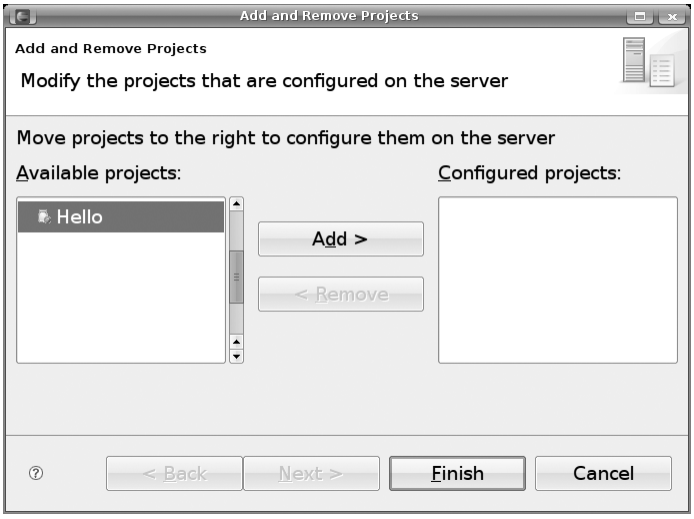

Figure 1-19. *Adding projects to the JBoss instance*

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.

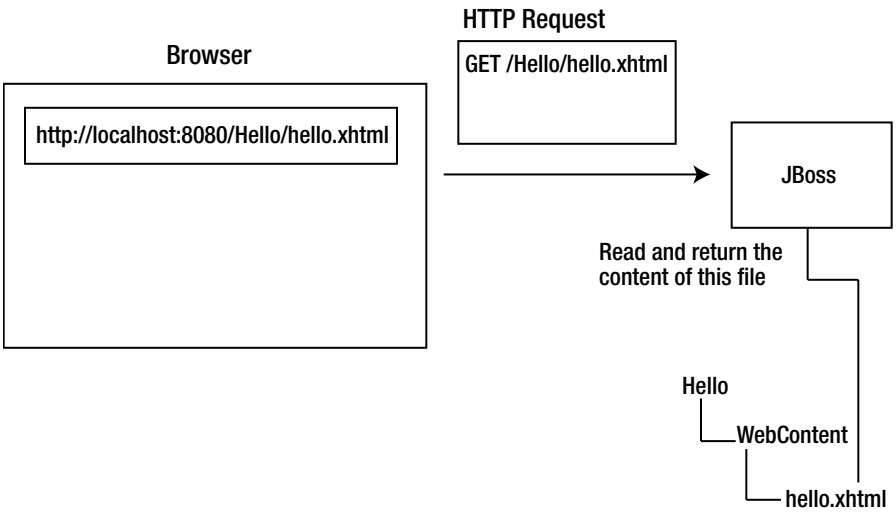

Figure 1-20. *Directly accessing the content of* `hello.xhtml`

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.

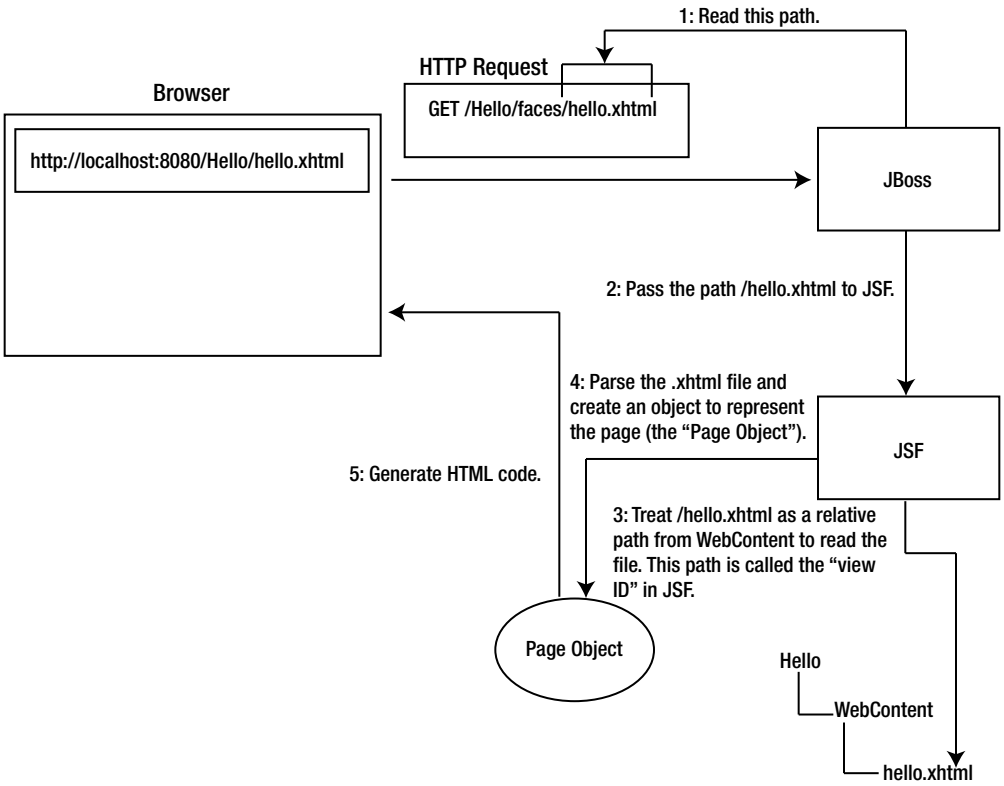

Figure 1-21. *Accessing the* `hello.xhtml` *file through JSF*

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.

```
<?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<h:outputText value="John"></h:outputText>!
</body>
</html>
```

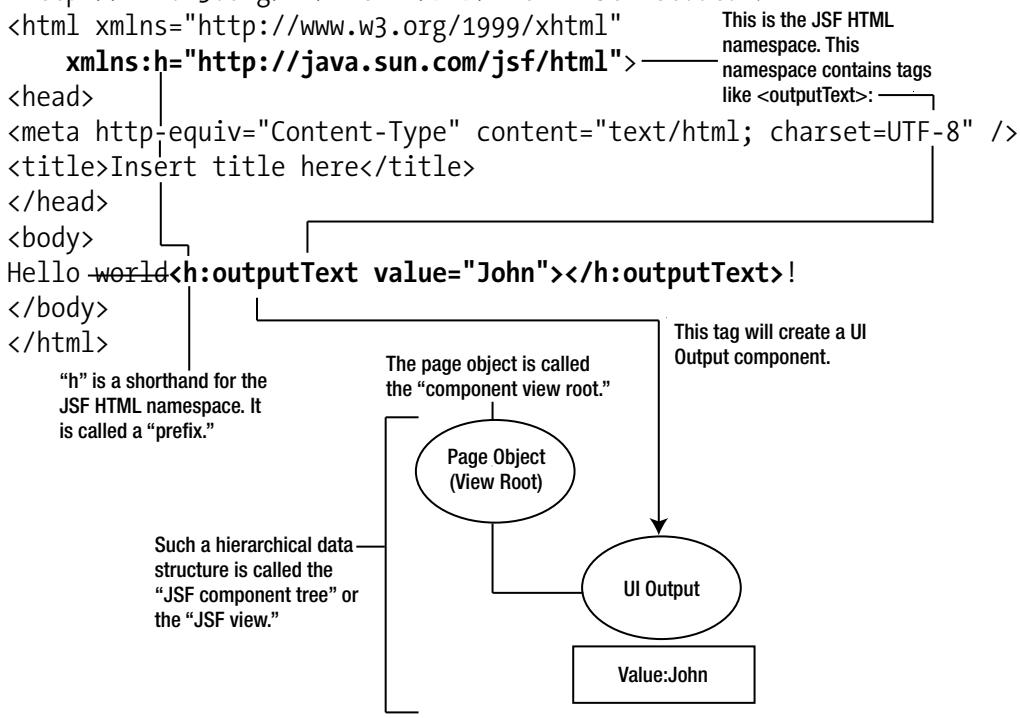

Figure 1-22. *JSF component tree*

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

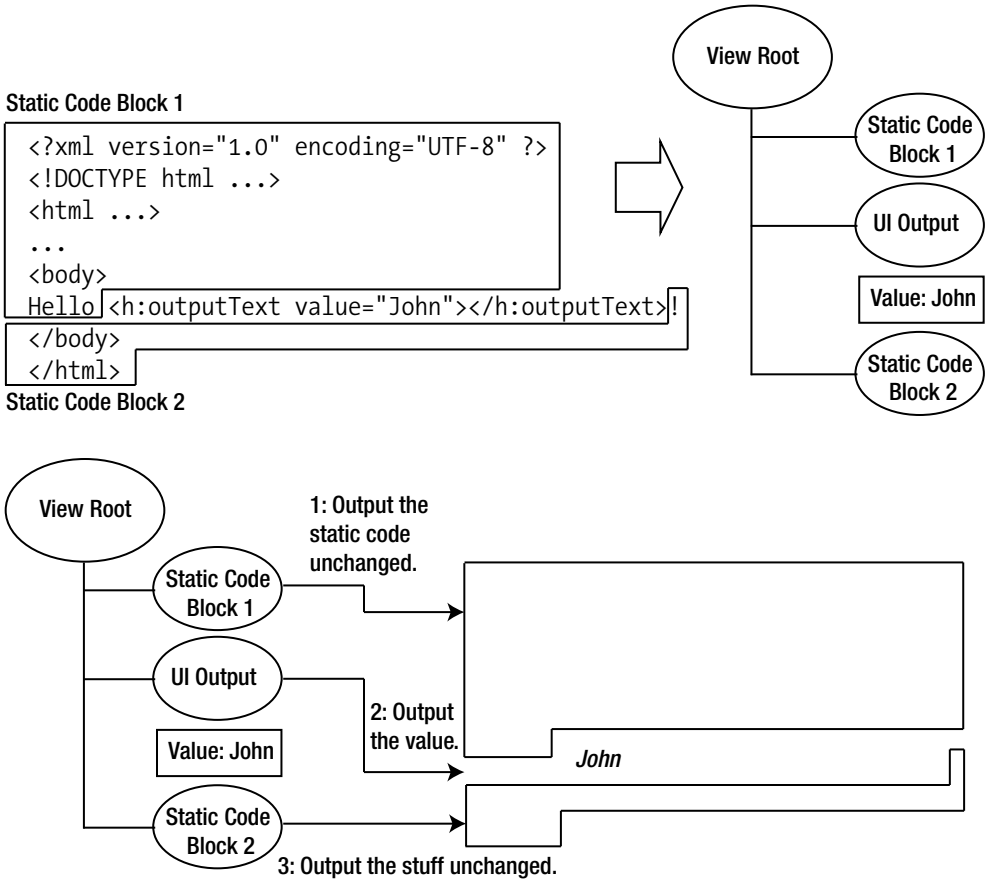

Figure 1-23. *JSF component tree generating HTML code*

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`.

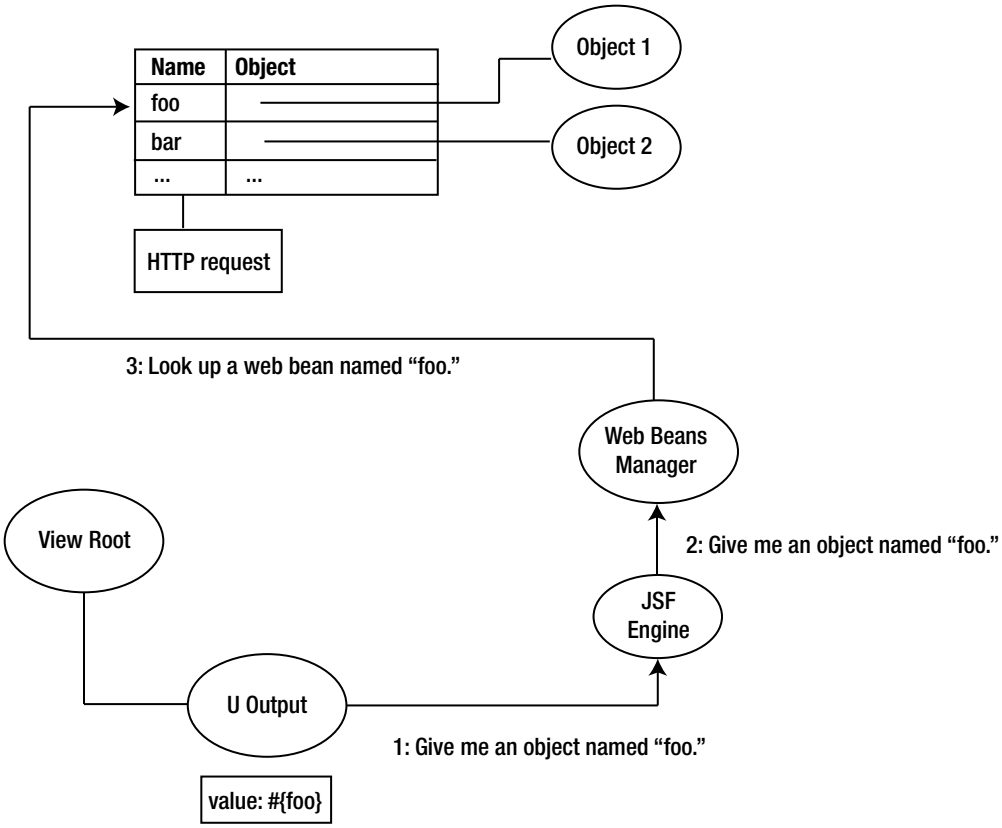

Figure 1-25. *Accessing a web bean*

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.

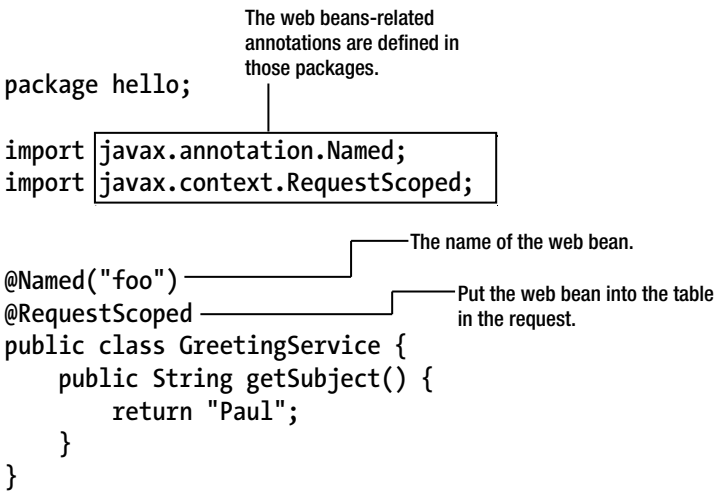

Figure 1-26. *Declaring a web bean class*

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.

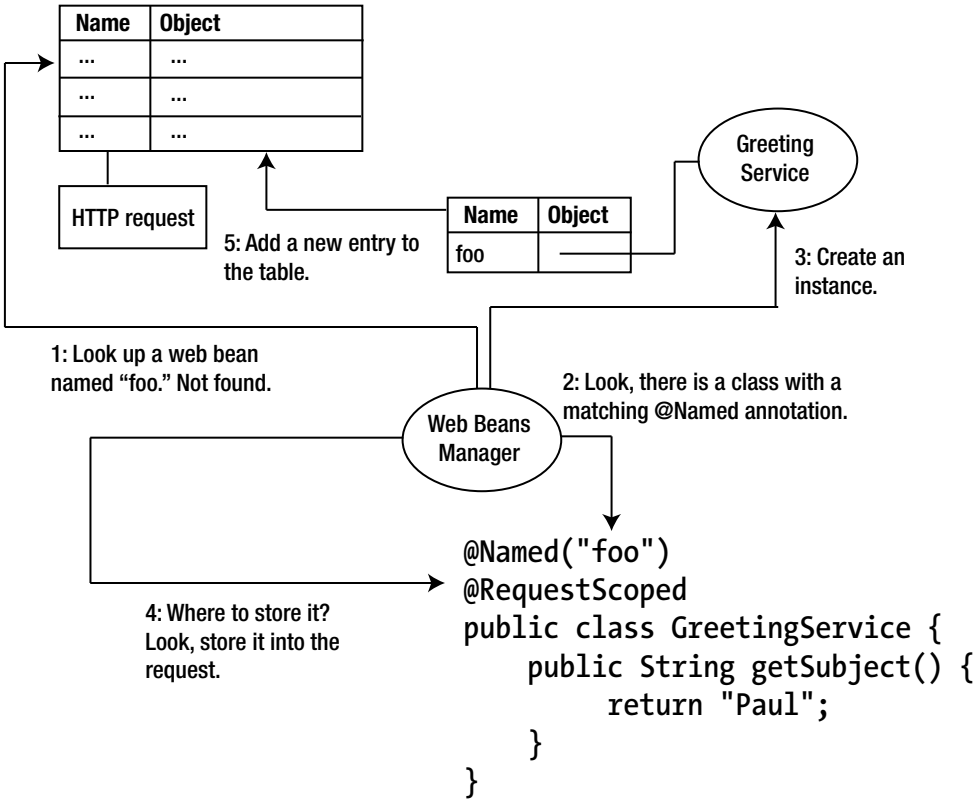

Figure 1-27. *How the Web Beans manager creates the web bean*

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.

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

This part will be evaluated at runtime and is called an "eval expression."

Add a space here. It is treated as static text and will be output as is. It is called a "literal expression."

In general, you can have multiple literal expressions and multiple eval expressions in a single EL expression like:

```
<h:outputText value="... #{...}" ... #{...} ...>
```

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.

```
GreetingService.java
package hello;

import javax.annotation.Named;

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

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.

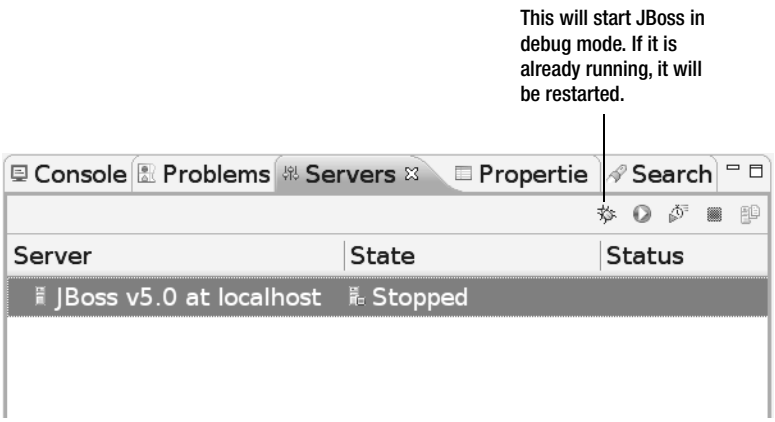

Figure 1-31. *Launching JBoss in debug mode*

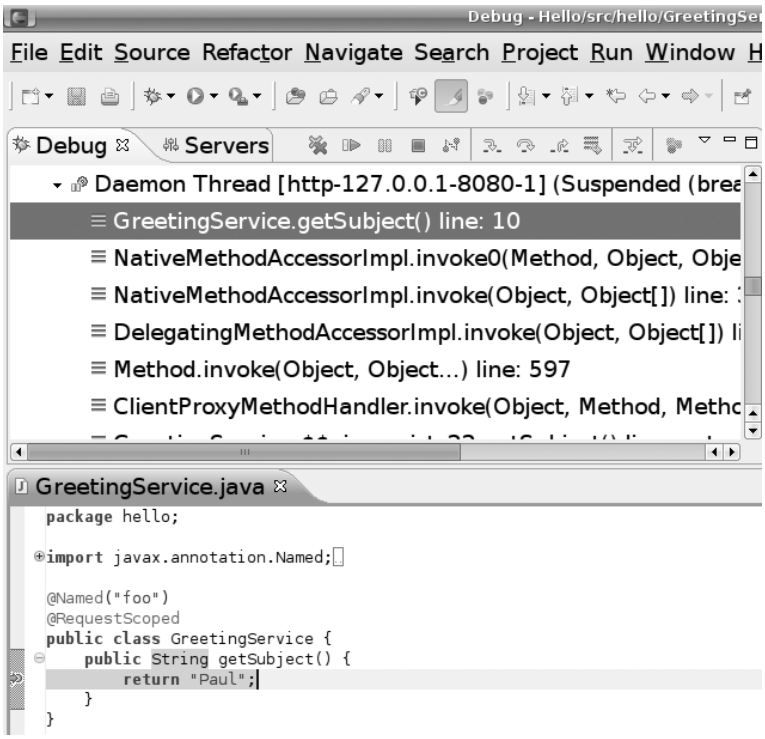

Figure 1-32. *Stopping at a breakpoint*

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.