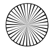
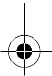
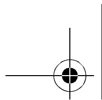
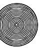
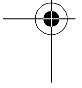
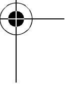
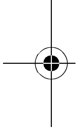
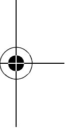
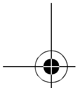
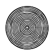
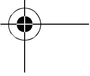
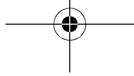
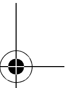
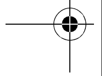
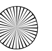
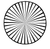
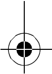
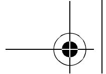
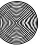



core JavaServer™ Faces

Second Edition



getting-started-VM.fm Page 2 Monday, April 2, 2007 10:39 AM



GETTING STARTED

Topics in This Chapter

- "Why JavaServer Faces?" on page 3
- "Software Installation" on page 4
- "A Simple Example" on page 6
- "Sample Application Analysis" on page 12
- "Development Environments for JSF" on page 21
- "JSF Framework Services" on page 28
- "Behind the Scenes" on page 30

Chapter 1

Why JavaServer Faces?

Judging from the job advertisements at employment web sites, there are two popular techniques for developing web applications:

- The “rapid development” style, in which you use a visual development environment, such as Microsoft ASP.NET
- The “hard-core coding” style, in which you write lots of code to support a high-performance backend, such as Java EE (Java Enterprise Edition)

Development teams face a difficult choice. Java EE is an attractive platform. It is highly scalable, portable to multiple platforms, and supported by many vendors. On the other hand, ASP.NET makes it easy to create attractive user interfaces without tedious programming. Of course, programmers want both: a high-performance backend and easy user interface programming. The promise of JSF (JavaServer Faces) is to bring rapid user interface development to server-side Java.

If you are familiar with client-side Java development, you can think of JSF as “Swing for server-side applications.” If you have experience with JSP (Java-Server Pages), you will find that JSF provides much of the plumbing that JSP developers have to implement by hand, such as page navigation and validation. You can think of servlets and JSP as the “assembly language” under the hood of the high-level JSF framework. If you already know a server-side

framework such as Struts, you will find that JSF uses a similar architecture but provides many additional services.

NOTE: You need *not* know anything about Swing, JSP, or Struts to use this book. We assume basic familiarity only with Java and HTML.

JSF has these parts:

- A set of prefabricated UI (user interface) components
- An event-driven programming model
- A component model that enables third-party developers to supply additional components

Some JSF components are simple, such as input fields and buttons. Others are quite sophisticated—for example, data tables and trees.

JSF contains all the necessary code for event handling and component organization. Application programmers can be blissfully ignorant of these details and spend their effort on the application logic.

Perhaps most important, JSF is part of the Java EE standard. JSF is included in every Java EE application server, and it can be easily added to a standalone web container such as Tomcat.

For additional details, see "JSF Framework Services" on page 28. Many IDEs (integrated development environments) support JSF, with features that range from code completion to visual page designers. See "Development Environments for JSF" on page 21 for more information. In the following sections, we show you how to compose a JSF application by hand, so that you understand what your IDE does under the hood and you can troubleshoot problems effectively.

Software Installation

You need the following software packages to get started:

- JDK (Java SE Development Kit) 5.0 or higher (`http://java.sun.com/j2se`)
- JSF 1.2
- The sample code for this book, available at `http://corejsf.com`

We assume that you have already installed the JDK and that you are familiar with the JDK tools. For more information on the JDK, see Horstmann, Cay, and Cornell, Gary, 2004, 2005. *Core Java™ 2, vol. 2—Advanced Features (7th ed.)*. Santa Clara, CA: Sun Microsystems Press/Prentice Hall.

Since JSF 1.2 is part of the Java EE 5 specification, the easiest way to try out JSF is to use an application server that is compatible with Java EE 5. In this section, we describe the GlassFish application server (`http://glassfish.dev.java.net`). You will find instructions for other application servers on the companion web site (`http://corejsf.com`).

NOTE: As this book is published, there are two major versions of JSF. The most recent version, JSF 1.2, was released as part of Java EE 5 in 2006. The original version, JSF 1.0, was released in 2004, and a bug-fix release, named JSF 1.1, was issued shortly thereafter. This book covers both versions 1.1 and 1.2, but the main focus is on version 1.2.

NOTE: If you do not want to install a complete application server, you can also use Tomcat (`http://tomcat.apache.org`), together with the JSF libraries from Sun Microsystems (`http://javaserverfaces.dev.java.net`). See the book's companion web site (`http://corejsf.com`) for installation instructions.

Install GlassFish, following the directions on the web site. Then start the application server. On Unix/Linux, you use the command

glassfish`/bin/asadmin start-domain`

(See Figure 1–1.) Here, *glassfish* is the directory into which you installed the GlassFish software.

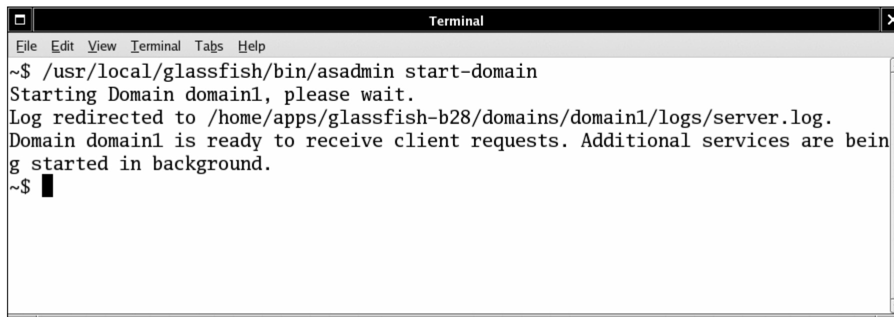


Figure 1–1 Starting GlassFish

On Windows, launch

glassfish`\bin\asadmin start-domain`

To test that GlassFish runs properly, point your browser to `http://localhost:8080`. You should see a welcome page (see Figure 1–2).

getting-started-VM.fm Page 6 Monday, April 2, 2007 10:39 AM

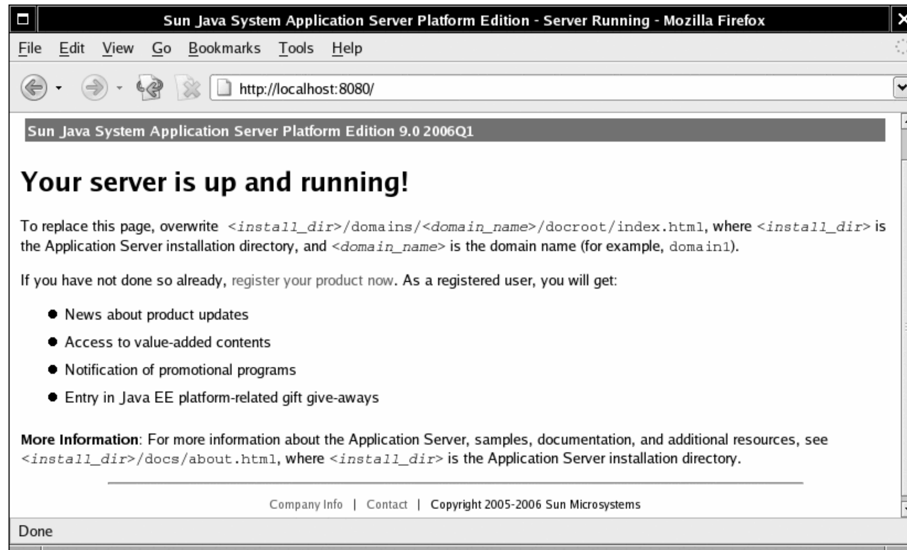


Figure 1–2 GlassFish welcome page

You shut down GlassFish with the command

```
glassfish/bin/asadmin stop-domain
```

or, on Windows,

```
glassfish\bin\asadmin stop-domain
```

A Simple Example

Now we move on to a simple example of a JSF application. Our first example starts with a login screen, shown in Figure 1–3.

Figure 1–3 A login screen

getting-started-VM.fm Page 7 Monday, April 2, 2007 10:39 AM

Of course, in a real web application, this screen would be beautified by a skilled graphic artist.

The file that describes the login screen is essentially an HTML file with a few additional tags (see Listing 1–1). Its visual appearance can be easily improved by a graphic artist who need not have any programming skills.

Listing 1–1 login/web/index.jsp

```
1. <html>
2.    <%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
3.    <%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
4.    <f:view>
5.       <head>
6.          <title>A Simple JavaServer Faces Application</title>
7.       </head>
8.       <body>
9.          <h:form>
10.            <h3>Please enter your name and password.</h3>
11.            <table>
12.               <tr>
13.                  <td>Name:</td>
14.                  <td>
15.                     <h:inputText value="#{user.name}"/>
16.                  </td>
17.               </tr>
18.               <tr>
19.                  <td>Password:</td>
20.                  <td>
21.                     <h:inputSecret value="#{user.password}"/>
22.                  </td>
23.               </tr>
24.            </table>
25.            <p>
26.               <h:commandButton value="Login" action="login"/>
27.            </p>
28.         </h:form>
29.      </body>
30.   </f:view>
31. </html>
```

We discuss the contents of this file in detail later in this chapter, in the section “JSF Pages” on page 13. For now, note the following points:

- A number of the tags are standard HTML tags: `body`, `table`, and so on.
- Some tags have *prefixes*, such as `f:view` and `h:inputText`. These are JSF tags. The two `taglib` declarations declare the JSF tag libraries.

- The `h:inputText`, `h:inputSecret`, and `h:commandButton` tags correspond to the text field, password field, and submit button in Figure 1–3.
- The input fields are linked to object properties. For example, the attribute `value="#{user.name}"` tells the JSF implementation to link the text field with the `name` property of a `user` object. We discuss this linkage in more detail later in this chapter, in the section "Beans" on page 12.

When the user enters the name and password, and clicks the "Login" button, a welcome screen appears (see Figure 1–4). Listing 1–3 on page 14 shows the source code for this screen. The section "Navigation" on page 16 explains how the application navigates from the login screen and the welcome screen.

Figure 1–4 A welcome screen

The welcome message contains the username. The password is ignored for now.

The purpose of this application is, of course, not to impress anyone but to illustrate the various pieces that are necessary to produce a JSF application.

Ingredients

Our sample application consists of the following ingredients:

- Pages that define the login and welcome screens. We call them `index.jsp` and `welcome.jsp`.
- A bean that manages the user data (in our case, username and password). A *bean* is a Java class that exposes properties, usually by following a simple naming convention for the getter and setter methods. The code is in the file `UserBean.java` (see Listing 1–2). Note that the class is contained in the `com.corejsf` package.

- A configuration file for the application that lists bean resources and navigation rules. By default, this file is called `faces-config.xml`.
- Miscellaneous files that are needed to keep the servlet container happy: the `web.xml` file, and an `index.html` file that redirects the user to the correct URL for the login page.

More advanced JSF applications have the same structure, but they can contain additional Java classes, such as event handlers, validators, and custom components.

Listing 1–2 `login/src/java/com/corejsf/UserBean.java`

```
1. package com.corejsf;
2.
3. public class UserBean {
4.    private String name;
5.    private String password;
6.
7.    // PROPERTY: name
8.    public String getName() { return name; }
9.    public void setName(String newValue) { name = newValue; }
10.
11.   // PROPERTY: password
12.   public String getPassword() { return password; }
13.   public void setPassword(String newValue) { password = newValue; }
14. }
```

Directory Structure

A JSF application is deployed as a *WAR file*: a zipped file with extension `.war` and a directory structure that follows a standardized layout:

```
HTML and JSP files
WEB-INF/
 |
 |--- configuration files
 |--- classes/
 |     |
 |     |--- class files
 |--- lib/
       |
       |--- library files
```

For example, the WAR file of our sample application has the directory structure shown in Figure 1–5. Note that the `UserBean` class is in the package `com.corejsf`.

The META-INF directory is automatically produced by the jar program when the WAR file is created.

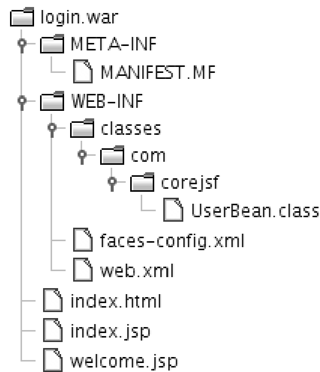


Figure 1–5 Directory structure of the sample WAR file

We package our application source in a slightly different directory structure, following the Java Blueprints conventions (http://java.sun.com/blueprints/code/projectconventions.html). The source code is contained in an src/java directory, and the JSF pages and configuration files are contained in a web directory (see Figure 1–6).

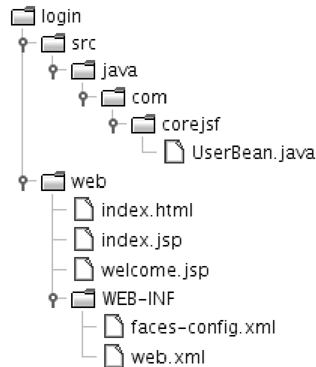


Figure 1–6 Directory structure of the sample application

Build Instructions

We now walk you through the steps required for building JSF applications with your bare hands. At the end of this chapter, we show you how to automate this process.

getting-started-VM.fm Page 11 Monday, April 2, 2007 10:39 AM

1. Launch a command shell.
2. Change to the *corejsf-examples* directory—that is, the directory that contains the sample code for this book.
3. Run the following commands:

```
cd ch1/login/src/java
mkdir ../../web/WEB-INF/classes
javac -d ../../web/WEB-INF/classes com/corejsf/UserBean.java
```

   On Windows, use backslashes instead:

```
cd ch1\login\web
mkdir WEB-INF\classes
javac -d ..\..\web\WEB-INF\classes com\corejsf\UserBean.java
```

4. Change to the `ch1/login/web` directory.
5. Run the following command:

```
jar cvf login.war .
```

   (Note the period at the end of the command, indicating the current directory.)

6. Copy the `login.war` file to the directory *glassfish*`/domains/domain1/autodeploy`.
7. Make sure that GlassFish has been started. Point your browser to

```
http://localhost:8080/login
```

   The application should start up at this point.

The bean classes in more complex programs may need to interact with the JSF framework. In that case, the compilation step is more complex. Your class path must include the JSF libraries. With the GlassFish application server, add a single JAR file:

glassfish`/lib/javaee.jar`

With other systems, you may need multiple JAR files.

A typical compilation command would look like this:

```
javac -classpath .:glassfish/lib/javaee.jar
   -d ../../web/WEB-INF/classes com/corejsf/*.java
```

On Windows, use semicolons to separate the path elements:

```
javac -classpath .;glassfish\lib\javaee.jar
   -d ..\..\web\WEB-INF\classes com\corejsf\*.java
```

Be sure to include the current directory (denoted by a period) in the class path.

getting-started-VM.fm Page 12 Monday, April 2, 2007 10:39 AM

Sample Application Analysis

Web applications have two parts: the *presentation layer* and the *business logic*. The presentation layer is concerned with the look of the application. In the context of a browser-based application, the look is determined by the HTML tags that specify layout, fonts, images, and so on. The business logic is implemented in the Java code that determines the behavior of the application.

Some web technologies intermingle HTML and code. That approach is seductive since it is easy to produce simple applications in a single file. But for serious applications, mixing markup and code poses considerable problems.

Professional web designers know about graphic design, but they typically rely on tools that translate their vision into HTML. They would certainly not want to deal with embedded code. On the other hand, programmers are notoriously unqualified when it comes to graphic design. (The example programs in this book bear ample evidence.)

Thus, for designing professional web applications, it is important to *separate* the presentation from the business logic. This allows both web designers and programmers to focus on their core competences.

In the context of JSF, the application code is contained in beans, and the design is contained in web pages. We look at beans first.

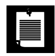
Beans

A Java *bean* is a class that exposes properties and events to an environment such as JSF. A *property* is a named value of a given type that can be read and/or written. The simplest way to define a property is to use a standard naming convention for the reader and writer methods, namely, the familiar `get/set` convention. The first letter of the property name is changed to upper case in the method names.

For example, the `UserBean` class has two properties, `name` and `password`, both of type `String`:

```
public class UserBean {
   public String getName() { . . . }
   public void setName(String newValue) {. . . }
   public String getPassword() { . . . }
   public void setPassword(String newValue) { . . . }
   . . .
}
```

The `get/set` methods can carry out arbitrary actions. In many cases, they simply get or set an instance field. But they might also access a database or a JNDI (Java Naming and Directory Interface) directory.



NOTE: According to the bean specification, it is legal to omit a read or write method. For example, if `getPassword` is omitted, then `password` is a write-only property. That might indeed be desirable for security reasons. However, JSF deals poorly with this situation and throws an exception instead of taking a default action when a read or write method is absent. Therefore, it is best to give read/write access to all bean properties.

In JSF applications, you use beans for all data that needs to be accessible from a page. The beans are the conduits between the user interface and the backend of the application.

JSF Pages

You need a JSF page for each browser screen. Depending on your development environment, JSF pages typically have the extension `.jsp` or `.jsf`. At the time of this writing, the extension `.jsp` requires less configuration effort. For that reason, we use the `.jsp` extension in the examples of this book.

NOTE: The extension of the page *files* is `.jsp` or `.jsf`, whereas in the preferred configuration, the extension of the page *URLs* is `.faces`. For example, when the browser requests the URL `http://localhost:8080/login/index.faces`, the URL extension `.faces` is *mapped* to the file extension `.jsp` and the servlet container loads the file `index.jsp`. This process sounds rather byzantine, but it is a consequence of implementing JSF on top of the servlet technology.

Now we take another look at the first page of our sample application in Listing 1–1.

The page starts out with the tag library declarations:

```
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
```

The JSF implementation defines two sets of tags. The HTML tags generate HTML-specific markup. If you want your web application to render pages for an alternative client technology, you must use a different tag library. The core tags are independent of the rendering technology. For example, you need the `f:view` tag both for HTML pages and for pages that are rendered by a cell phone.

getting-started-VM.fm Page 14 Monday, April 2, 2007 10:39 AM

NOTE: You can choose any prefixes for tags, such as `faces:view` and `html:inputText`. In this book, we use `f` for the core tags and `h` for the HTML tags.

Much of the page is similar to an HTML form. Note the following differences:

- All JSF tags are contained in an `f:view` tag.
- Instead of using an HTML `form` tag, you enclose all the JSF components in an `h:form` tag.
- Instead of using the familiar `input` HTML tags, use `h:inputText`, `h:inputSecret`, and `h:commandButton`.

We discuss all standard JSF tags and their attributes in Chapters 4 and 5. In the first three chapters, we can get by with input fields and command buttons.

The input field values are bound to properties of the bean with name `user`:

```
<h:inputText value="#{user.name}"/>
```

You will see the declaration of the `user` variable in "Navigation" on page 16. The `#{...}` delimiters are explained in "The Syntax of Value Expressions" on page 64 of Chapter 2.

When the page is displayed, the framework calls the `getName` method to obtain the current property value. When the page is submitted, the framework invokes the `setName` method to set the value that the user entered.

The `h:commandButton` tag has an `action` attribute whose value is used when specifying navigation rules:

```
<h:commandButton value="Login" action="login"/>
```

We discuss navigation rules in "Navigation" on page 16. The `value` attribute is the string that is displayed on the button.

The second JSF page of our application is even simpler than the first. It uses the `h:outputText` tag to display the username (see Listing 1–3).

Listing 1–3 login/web/welcome.jsp

```
1. <html>
2.    <%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
3.    <%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
4.
5.    <f:view>
6.       <head>
7.          <title>A Simple JavaServer Faces Application</title>
8.       </head>
```

Listing 1–3 login/web/welcome.jsp (cont.)

```
 9.       <body>
10.          <h:form>
11.             <h3>
12.                Welcome to JavaServer Faces,
13.                <h:outputText value="#{user.name}"/>!
14.             </h3>
15.          </h:form>
16.       </body>
17.    </f:view>
18. </html>
```

NOTE: We use a plain and old-fashioned format for our JSF pages so that they are as easy to read as possible.

XML-savvy readers will want to do a better job. First, it is desirable to use proper XML for the tag library declarations, eliminating the <%...%> tags. Moreover, you will want to emit a proper DOCTYPE declaration for the generated HTML document.

The following format solves both issues:

```
<?xml version="1.0" ?>
<jsp:root version="2.0"
      xmlns:jsp="http://java.sun.com/JSP/Page"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:h="http://java.sun.com/jsf/html">
   <jsp:directive.page contentType="text/html"/>
   <jsp:output omit-xml-declaration="no"
         doctype-root-element="html"
         doctype-public="-//W3C//DTD XHTML 1.0 Transitional//EN"
         doctype-system="http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd"/>
    <f:view>
      <html xmlns="http://www.w3.org/1999/xhtml">
         <head>
            <title>A Simple Java Server Faces Application</title>
         </head>
         <body>
            <h:form>
            . . .
            </h:form>
         </body>
      </html>
   </f:view>
</jsp:root>
```

If you use an XML-aware editor, you should seriously consider this form.

getting-started-VM.fm Page 16 Monday, April 2, 2007 10:39 AM

CAUTION: You sometimes see naive page authors produce documents that start with an HTML DOCTYPE declaration, like this:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
   <%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
   <%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
   <f:view>
   . . .
```

This may have been acceptable at one time, but nowadays, it is quite reprehensible. Plainly, this document is *not* an "HTML 4.01 Transitional" document. It merely aims to produce such a document. Many XML editors and tools do not take it kindly when you lie about the document type. Therefore, either omit the DOCTYPE altogether or follow the outline given in the preceding note.

Navigation

To complete our JSF application, we need to specify the navigation rules. A navigation rule tells the JSF implementation which page to send back to the browser after a form has been submitted.

In this case, navigation is simple. When the user clicks the login button, we want to navigate from the index.jsp page to welcome.jsp. You specify this navigation rule in the faces-config.xml file:

```
<navigation-rule>
   <from-view-id>/index.jsp</from-view-id>
   <navigation-case>
      <from-outcome>login</from-outcome>
      <to-view-id>/welcome.jsp</to-view-id>
   </navigation-case>
</navigation-rule>
```

The from-outcome value matches the action attribute of the command button of the index.jsp page:

```
<h:commandButton value="Login" action="login"/>
```

In addition to the navigation rules, the faces-config.xml file contains the bean definitions. Here is the definition of the user bean:

```
   <managed-bean>
      <managed-bean-name>user</managed-bean-name>
      <managed-bean-class>
         com.corejsf.UserBean
```

getting-started-VM.fm Page 17 Monday, April 2, 2007 10:39 AM


```
    </managed-bean-class>
    <managed-bean-scope>session</managed-bean-scope>
  </managed-bean>
```

You can use the bean name, `user`, in the attributes of the user interface components. For example, `index.jsp` contains the tag

```
<h:inputText value="#{user.name}"/>
```

The `value` attribute refers to the `name` property of the `user` bean.

The `managed-bean-class` tag specifies the bean class, in our case, `com.corejsf.UserBean`. Finally, the scope is set to `session`. This means that the bean object is available for one user across multiple pages. Different users who use the web application are given different instances of the bean object.

Listing 1–4 shows the complete `faces-config.xml` file.

NOTE: JSF 1.2 uses a schema declaration to define the syntax of a configuration file. The configuration tags are enclosed in

```
<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_1_2.xsd"
   version="1.2">
   . . .
</faces-config>
```

JSF 1.1 uses a `DOCTYPE` declaration instead:

```
<!DOCTYPE faces-config PUBLIC
   "-//Sun Microsystems, Inc.//DTD JavaServer Faces Config 1.0//EN"
   "http://java.sun.com/dtd/web-facesconfig_1_0.dtd">
<faces-config>
   . . .
</faces-config>
```

We recommend that you use an XML editor that understands XML Schema declarations. If you use Eclipse, a good choice is the XMLBuddy plugin (`http://xmlbuddy.com`).

Listing 1–4 login/web/WEB-INF/faces-config.xml

```
1. <?xml version="1.0"?>
2. <faces-config xmlns="http://java.sun.com/xml/ns/javaee"
3.    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4.    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
5.         http://java.sun.com/xml/ns/javaee/web-facesconfig_1_2.xsd"
6.    version="1.2">
7.    <navigation-rule>
8.       <from-view-id>/index.jsp</from-view-id>
9.       <navigation-case>
10.         <from-outcome>login</from-outcome>
11.         <to-view-id>/welcome.jsp</to-view-id>
12.      </navigation-case>
13.   </navigation-rule>
14.
15.   <managed-bean>
16.      <managed-bean-name>user</managed-bean-name>
17.      <managed-bean-class>com.corejsf.UserBean</managed-bean-class>
18.      <managed-bean-scope>session</managed-bean-scope>
19.   </managed-bean>
20. </faces-config>
```

Servlet Configuration

When you deploy a JSF application inside an application server, you need to supply a configuration file named `web.xml`. Fortunately, you can use the same `web.xml` file for most JSF applications. Listing 1–5 shows the file.

Listing 1–5 login/web/WEB-INF/web.xml

```
1. <?xml version="1.0"?>
2. <web-app xmlns="http://java.sun.com/xml/ns/javaee"
3.    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4.    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
5.       http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
6.    version="2.5">
7.    <servlet>
8.       <servlet-name>Faces Servlet</servlet-name>
9.       <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
10.      <load-on-startup>1</load-on-startup>
11.   </servlet>
12.
```

Listing 1–5 login/web/WEB-INF/web.xml (cont.)

```
13.    <servlet-mapping>
14.       <servlet-name>Faces Servlet</servlet-name>
15.       <url-pattern>*.faces</url-pattern>
16.    </servlet-mapping>
17.
18.    <welcome-file-list>
19.       <welcome-file>index.html</welcome-file>
20.    </welcome-file-list>
21. </web-app>
```

The only remarkable aspect of this file is the *servlet mapping*. All JSF pages are processed by a special servlet that is a part of the JSF implementation code. To ensure that the correct servlet is activated when a JSF page is requested, the JSF URLs have a special format. In our configuration, they have an extension `.faces`.

For example, you cannot simply point your browser to `http://localhost:8080/login/index.jsp`. The URL has to be `http://localhost:8080/login/index.faces`. The servlet container uses the servlet mapping rule to activate the JSF servlet, which strips off the `faces` suffix and loads the `index.jsp` page.

NOTE: You can also define a *prefix mapping* instead of the `.faces` extension mapping. Use the following directive in your `web.xml` file:

```
<servlet-mapping>
   <servlet-name>Faces Servlet</servlet-name>
   <url-pattern>/faces/*</url-pattern>
</servlet-mapping>
```

Then use the URL `http://localhost:8080/login/faces/index.jsp`. That URL activates the JSF servlet, which then strips off the `faces` prefix and loads the file `/login/index.jsp`.

NOTE: If you want to use a `.jsf` extension for JSF page files, then you need to configure your web application so that it invokes the JSP servlet for files with that extension. Use the following mapping in the `web.xml` file:

```
<servlet-mapping>
   <servlet-name>jsp</servlet-name>
   <url-pattern>*.jsf</url-pattern>
</servlet-mapping>
```

getting-started-VM.fm Page 20 Monday, April 2, 2007 10:39 AM

You now need to tell the JSF implementation to map the `.faces` extension of the URLs to the `.jsf` extension of the associated files.

```
<context-param>
   <param-name>javax.faces.DEFAULT_SUFFIX</param-name>
   <param-value>.jsf</param-value>
</context-param>
```

Note that this configuration affects only the web developers, not the users of your web application. The URLs still have a `.faces` extension or `/faces` prefix.

NOTE: If you use an older application server that supports version 2.3 of the servlet specification, you use a DTD (`DOCTYPE` declaration) instead of a schema declaration in the `web.xml` file. The DTD is as follows:

```
<!DOCTYPE web-app PUBLIC
   "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
   "http://java.sun.com/dtd/web-app_2_3.dtd">
```

The Welcome File

When a user enters a directory URL such as `http://localhost:8080/login`, the application server automatically loads the `index.jsp` page when it is present. Unfortunately, that mechanism does not work smoothly with JSF pages because the JSF processing phase is skipped.

To overcome this issue, you can supply an `index.html` file that automatically redirects the user to the proper `faces` URL. Listing 1–6 shows such an index file.

Listing 1–6 `login/web/index.html`

```
1. <html>
2.    <head>
3.       <meta http-equiv="Refresh" content= "0; URL=index.faces"/>
4.       <title>Start Web Application</title>
5.    </head>
6.    <body>
7.       <p>Please wait for the web application to start.</p>
8.    </body>
9. </html>
```

Finally, it is a good idea to specify `index.html` as the welcome file in `web.xml`. See the `welcome-file` tag in Listing 1–5 on page 18.

NOTE: The `index.html` file redirects the browser to the `index.faces` URL. It is slightly more efficient to use a JSP forward action instead. Create a page, say, `start.jsp`, that contains the line

```
<jsp:forward page="/index.faces"/>
```

Then set this page as the `welcome-file` in the `web.xml` configuration file.

Development Environments for JSF

You can produce the pages and configuration files for a simple JSF application with a text editor. However, as your applications become more complex, you will want to use more sophisticated tools. In the next three sections, we discuss JSF support in integrated development environments, visual builder tools, and build automation with Ant.

Integrated Development Environments

IDEs, such as Eclipse or NetBeans, are deservedly popular with programmers. Support for autocompletion, refactoring, debugging, and so on, can dramatically increase programmer productivity, particularly for large projects.

As this book is written, Eclipse has an experimental JSF plug-in that plainly needs more work. Several commercial Eclipse derivatives (such as MyEclipse, Exadel Studio, BEA Workshop Studio, and Rational Application Developer) have better JSF support, but some of them are expensive. They all have trial versions that you can download.

NetBeans, on the other hand, is free and has very good JSF support out of the box. If you are not satisfied with the JSF support in your favorite IDE, we suggest that you give NetBeans a try.

NetBeans gives you autocompletion in JSF pages and configuration files. With NetBeans, it is very easy to launch or debug JSF applications just by clicking toolbar buttons. Figure 1–7 shows the NetBeans debugger, stopped at a breakpoint in the `UserBean` class.

NOTE: Since the user interfaces for IDEs can change quite a bit between versions, we put a guide for getting started with NetBeans on the web (`http://corejsf.com`) rather than in the printed book.

getting-started-VM.fm Page 22 Monday, April 2, 2007 10:39 AM

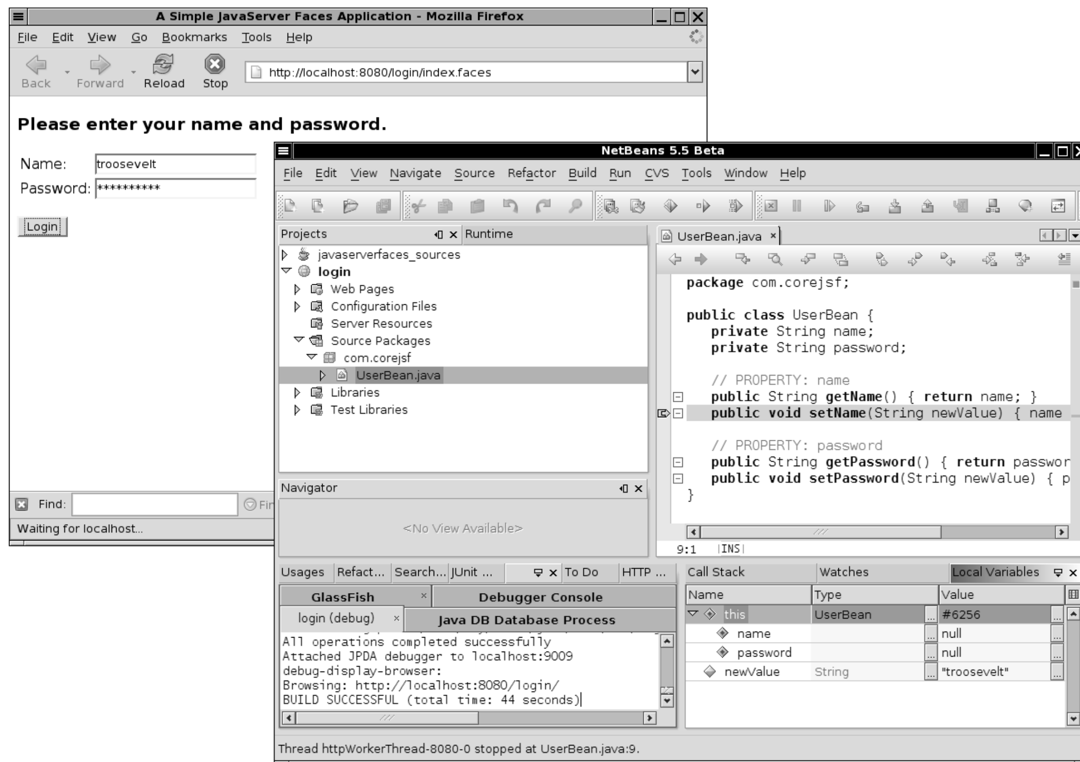


Figure 1–7 Using NetBeans for JSF debugging

Visual Builder Tools

A visual builder tool displays a graphical representation of the components and allows a designer to drag and drop components from a palette. Builder tools can be standalone programs such as Sun Java Studio Creator, or they can be modules of integrated development environments.

Figure 1–8 shows Sun Java Studio Creator (`http://www.sun.com/software/products/jscreator`). The component palette is in the lower-left corner. You drag the components onto the center of the window and customize them with the property sheet in the upper-right corner. The environment produces the corresponding JSF tags automatically (see Figure 1–9).

Moreover, visual builders give you graphical interfaces for specifying the navigation rules and beans (see Figure 1–10). The `faces-config.xml` file is produced automatically.

getting-started-VM.fm Page 23 Monday, April 2, 2007 10:39 AM

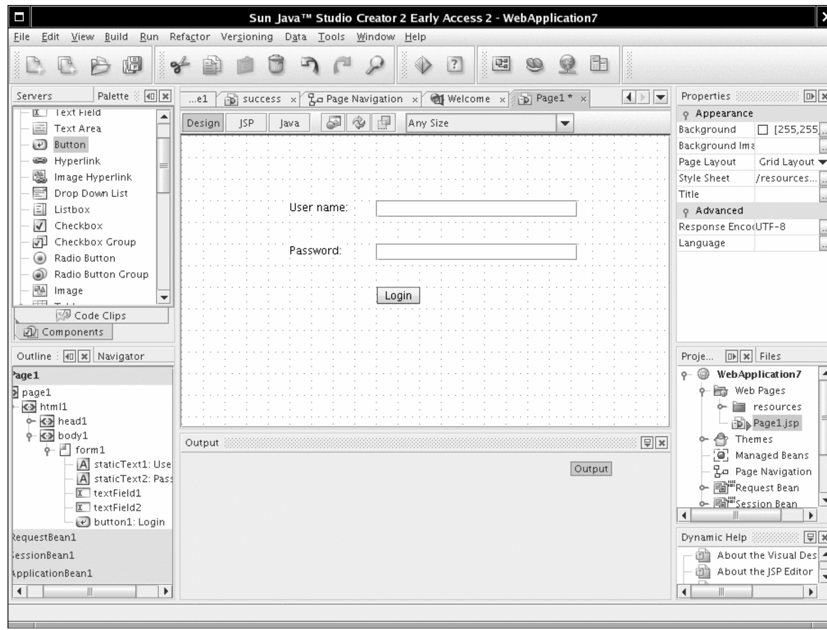


Figure 1–8 Visual JSF development environment

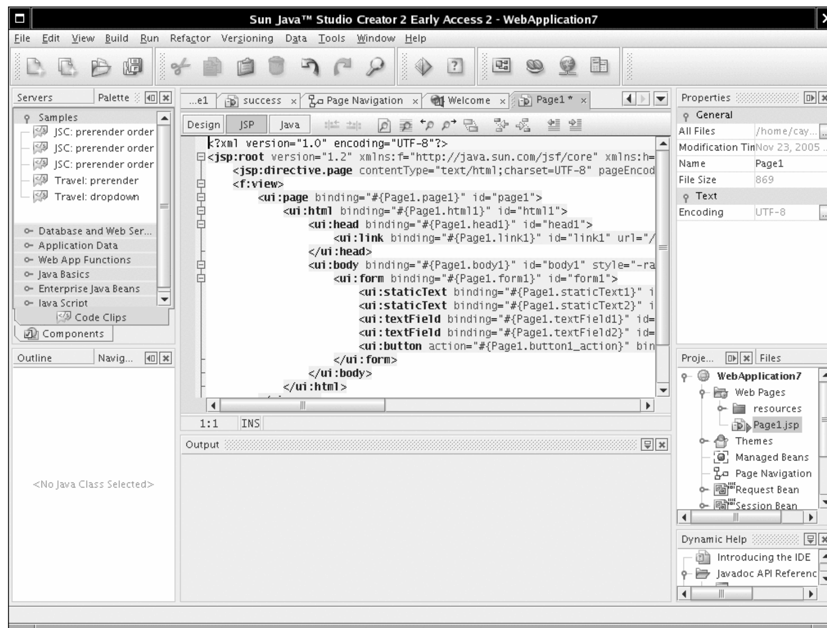


Figure 1–9 Automatically generated JSF markup

getting-started-VM.fm Page 24 Monday, April 2, 2007 10:39 AM

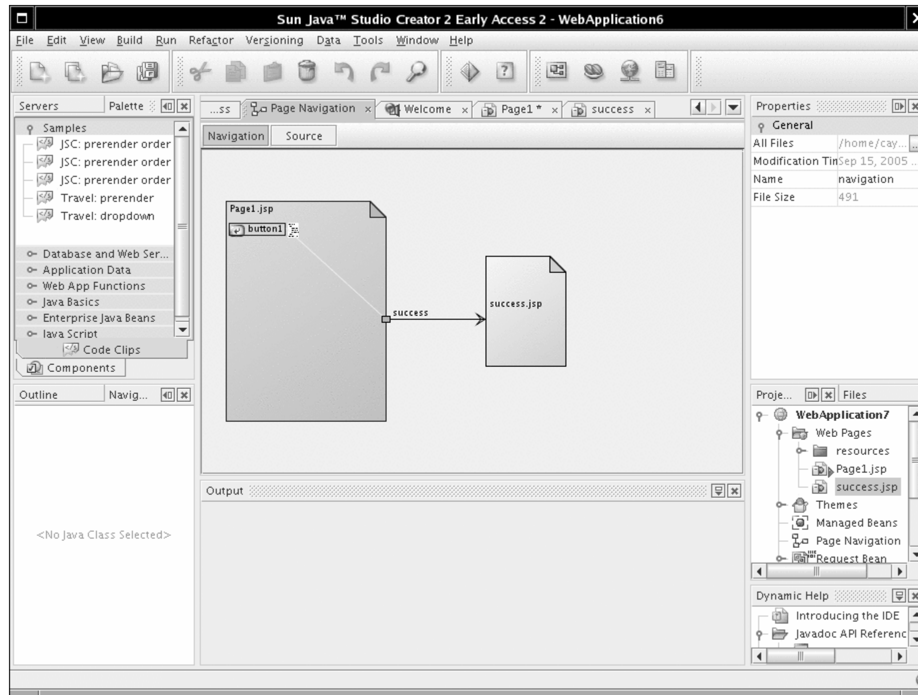


Figure 1–10 Visually specifying navigation rules

Unfortunately, Java Studio Creator has a rather rigid page structure that is not optimal for learning about JSF. We recommend that you use another environment for working through the book examples. After studying the book examples, you will know enough about JSF to use Java Studio Creator effectively for your own projects.

Sun has announced that visual tools from Java Studio Creator will be integrated into future versions of NetBeans. Future versions of Eclipse are also expected to include visual builder features.

Automation of the Build Process with Ant

Many programmers prefer to stick with their favorite text editor or IDE, even if it has little support for JSF. The manual build process that we described earlier in this chapter can become tedious if you need to do it over and over. In this section, we describe how you can automate the process with Ant. The material in this section is not required for working with JSF—feel free to skip it if your IDE has good JSF support or if the manual build process does not bother you.

Fortunately, you need not know much about Ant if you want to use the build script that we prepared. Start by downloading Ant from `http://ant.apache.org` and install it in a directory of your choice. Or, if you use GlassFish, use the `asant` tool that is included in the *glassfish*/`bin` directory.

Ant takes directions from a *build file*. By default, the build file is named `build.xml`. We provide a `build.xml` file for building JSF applications. This file is contained in the root of the *corejsf-examples* directory. The `build.xml` file contains the instructions for compiling, copying, zipping, and deploying to an application server, described in XML syntax (see Listing 1–7).

Listing 1–7 `build.xml`

```
1. <project default="install">
2.
3.    <property environment="env"/>
4.    <property file="build.properties"/>
5.    <property name="appdir" value="${basedir}/${app}"/>
6.    <basename property="appname" file="${appdir}"/>
7.    <property name="builddir" value="${appdir}/build"/>
8.    <property name="warfile" value="${builddir}/${appname}.war"/>
9.
10.   <path id="classpath">
11.      <pathelement location="${javaee.api.jar}"/>
12.      <fileset dir="${appdir}">
13.        <include name="web/WEB-INF/**/*.jar"/>
14.      </fileset>
15.   </path>
16.
17.   <target name="init">
18.      <fail unless="app" message="Run ant -Dapp=..."/>
19.   </target>
20.
21.   <target name="prepare" depends="init"
22.         description="Create build directory.">
23.      <mkdir dir="${builddir}"/>
24.      <mkdir dir="${builddir}/WEB-INF"/>
25.      <mkdir dir="${builddir}/WEB-INF/classes"/>
26.   </target>
27.
28.   <target name="copy" depends="prepare"
29.         description="Copy files to build directory.">
30.      <copy todir="${builddir}" failonerror="false" verbose="true">
31.         <fileset dir="${appdir}/web"/>
32.      </copy>
33.      <copy todir="${builddir}/WEB-INF/classes"
34.           failonerror="false" verbose="true">
```

getting-started-VM.fm Page 26 Monday, April 2, 2007 10:39 AM

Listing 1–7 build.xml (cont.)

```
35.         <fileset dir="${appdir}/src/java">
36.            <exclude name="**/*.java"/>
37.         </fileset>
38.      </copy>
39.      <copy todir="${builddir}/WEB-INF" failonerror="false" verbose="true">
40.         <fileset dir="${appdir}">
41.            <include name="lib/**"/>
42.         </fileset>
43.      </copy>
44.   </target>
45.
46.   <target name="compile" depends="copy"
47.         description="Compile source files.">
48.      <javac
49.         srcdir="${appdir}/src/java"
50.         destdir="${builddir}/WEB-INF/classes"
51.         debug="true"
52.         deprecation="true">
53.         <compilerarg value="-Xlint:unchecked"/>
54.         <include name="**/*.java"/>
55.         <classpath refid="classpath"/>
56.      </javac>
57.   </target>
58.
59.   <target name="war" depends="compile"
60.         description="Build WAR file.">
61.      <delete file="${warfile}"/>
62.      <jar jarfile="${warfile}" basedir="${builddir}"/>
63.   </target>
64.
65.   <target name="install" depends="war"
66.         description="Deploy web application.">
67.      <copy file="${warfile}" todir="${deploy.dir}"/>
68.   </target>
69.
70.   <target name="clean" depends="init"
71.         description="Clean everything.">
72.      <delete dir="${builddir}"/>
73.   </target>
74. </project>
```

To use this build file, you must customize the `build.properties` file that is contained in the same directory. The default file looks like what is shown in Listing 1–8.

getting-started-VM.fm Page 27 Monday, April 2, 2007 10:39 AM

Listing 1–8 `build.properties`

```
1. appserver.dir=${env.GLASSFISH_HOME}
2. javaee.api.jar=${appserver.dir}/lib/javaee.jar
3. deploy.dir=${appserver.dir}/domains/domain1/autodeploy
```

You need to change the directory for the application server to match your local installation. Edit the first line of `build.properties`.

Now you are ready to build the sample application (see Figure 1–11).

1. Open a command shell and change into the *corejsf-examples* directory.
2. Run the command

   apache-ant`/bin/ant -Dapp=ch1/login`

   Here, *apache-ant* is the directory into which you installed Ant, such as `c:\apache-ant-1.6.5`. With GlassFish, you can also use

   glassfish`/bin/asant -Dapp=ch1/login`

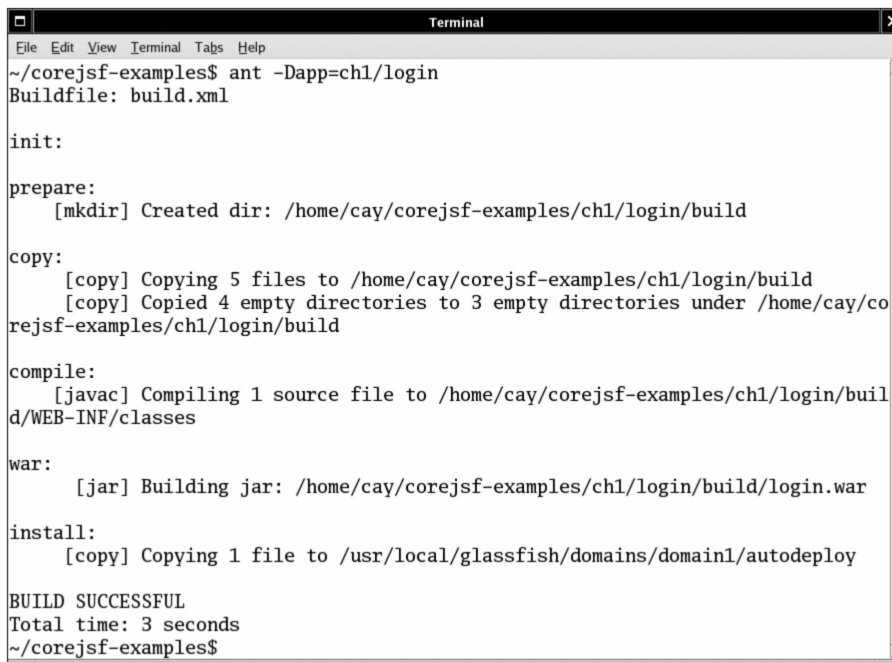


Figure 1–11 Installing a web application with Ant

getting-started-VM.fm Page 28 Monday, April 2, 2007 10:39 AM

NOTE: Our Ant script is a bit different from the scripts that you often find with sample applications. We use a single script that can build all applications in the book. You use the `-Dapp=...` flag to specify the name of the application that you want to build. We think that approach is better than supplying lots of nearly identical scripts. Note that you call the script from the *corejsf-examples* directory, not the directory of the application.

JSF Framework Services

Now that you have seen your first JSF application, it is easier to explain the services that the JSF framework offers to developers. Figure 1–12 gives a high-level overview of the JSF architecture. As you can see, the JSF framework is responsible for interacting with client devices, and it provides tools for tying together the visual presentation, application logic, and business logic of a web application. However, the scope of JSF is restricted to the presentation tier. Database persistence, web services, and other backend connections are outside the scope of JSF.

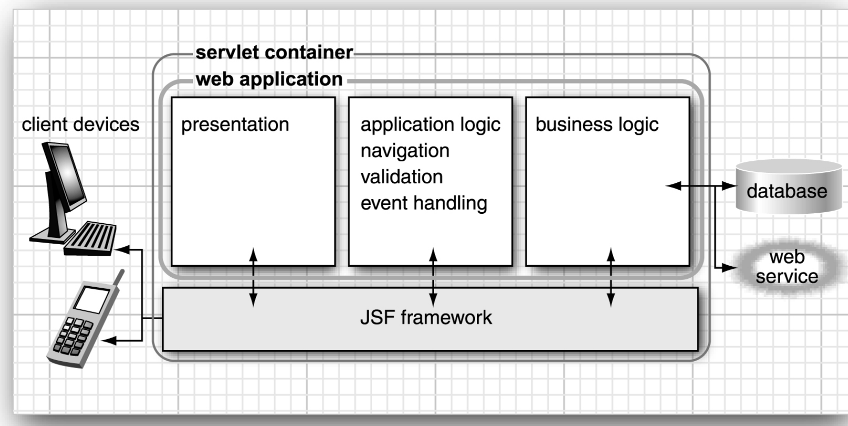


Figure 1–12 High-level overview of the JSF framework

Here are the most important services that the JSF framework provides:

Model-view-controller architecture—All software applications let users manipulate certain data, such as shopping carts, travel itineraries, or whatever data is required in a particular problem domain. This data is called the *model*. Just as an artist creates a painting of a model in a studio,

getting-started-VM.fm Page 29 Monday, April 2, 2007 10:39 AM

a software developer produces *views* of the data model. In a web application, HTML (or a similar rendering technology) is used to paint these views.

JSF connects the view and the model. As you have seen, a view component can be wired to a bean property of a model object, such as

```
<h:inputText value="#{user.name}"/>
```

Moreover, JSF operates as the *controller* that reacts to the user by processing action and value change events, routing them to code that updates the model or the view. For example, you may want to invoke a method to check whether a user is allowed to log on. Use the following JSF tag:

```
<h:commandButton value="Login" action="#{user.check}"/>
```

When the user clicks the button and the form is submitted to the server, the JSF implementation invokes the `check` method of the `user` bean. That method can take arbitrary actions to update the model, and it returns the navigation ID of the next page to be displayed. We discuss this mechanism further in "Dynamic Navigation" on page 73 of Chapter 3.

Thus, JSF implements the classical model-view-controller architecture.

Data conversion—Users enter data into web forms as text. Business objects want data as numbers, dates, or other data types. As explained in Chapter 6, JSF makes it easy to specify and customize conversion rules.

Validation and error handling—JSF makes it easy to attach validation rules for fields such as "this field is required" or "this field must be a number". Of course, when users enter invalid data, you need to display appropriate error messages. JSF takes away much of the tedium of this programming task. We cover validation in Chapter 6.

Internationalization—JSF manages internationalization issues such as character encodings and the selection of resource bundles. We cover resource bundles in "Message Bundles" on page 42 of Chapter 2.

Custom components—Component developers can develop sophisticated components that page designers simply drop into their pages. For example, suppose a component developer produces a calendar component with all the usual bells and whistles. You just use it in your page, with a command such as

```
<acme:calendar value="#{flight.departure}" startOfWeek="Mon"/>
```

Chapter 9 covers custom components in detail.

getting-started-VM.fm Page 30 Monday, April 2, 2007 10:39 AM

Alternative renderers—By default, JSF generates markup for HTML pages. But it is easy to extend the JSF framework to produce markup for another page description language such as WML or XUL. The book's companion site contains a chapter that shows you how to use JSF to communicate with Java ME-powered cell phones.

Tool support—JSF is optimized for use with automated tools. As these tools mature in the coming years, we believe that JSF will be the must-have framework for developing web interfaces with Java.

Behind the Scenes

Now that you have read about the "what" and the "why" of JSF, you may be curious about just how the JSF framework does its job.

Next, we look behind the scenes of our sample application. We start at the point when the browser first connects to `http://localhost:8080/login/index.faces`. The JSF servlet initializes the JSF code and reads the `index.jsp` page. That page contains tags such as `f:form` and `h:inputText`. Each tag has an associated *tag handler* class. When the page is read, the tag handlers are executed. The JSF tag handlers collaborate with each other to build a *component tree* (see Figure 1–13).

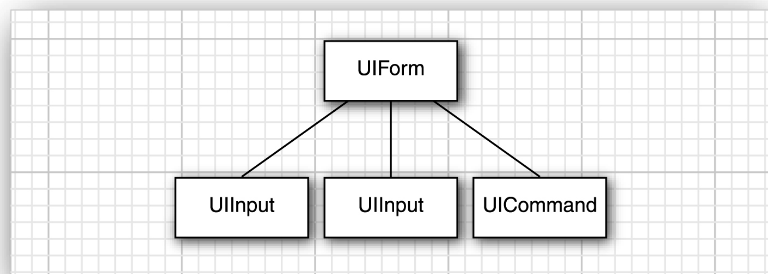


Figure 1–13 Component tree of the sample application

The component tree is a data structure that contains Java objects for all user interface elements on the JSF page. For example, the two `UIInput` objects correspond to the `h:inputText` and `h:inputSecret` fields in the JSF file.

Rendering Pages

Next, the HTML page is *rendered*. All text that is not a JSF tag is passed through. The `h:form`, `h:inputText`, `h:inputSecret`, and `h:commandButton` tags are converted to HTML.

As we just discussed, each of these tags gives rise to an associated component. Each component has a *renderer* that produces HTML output, reflecting the component state. For example, the renderer for the component that corresponds to the `h:inputText` tag produces the following output:

```
<input type="text" name="unique ID" value="current value"/>
```

This process is called *encoding*. The renderer of the `UIInput` object asks the framework to look up the unique ID and the current value of the expression `user.name`. By default, ID strings are assigned by the framework. The IDs can look rather random, such as `_id_id12:_id_id21`.

The encoded page is sent to the browser, and the browser displays it in the usual way (see Figure 1–14).

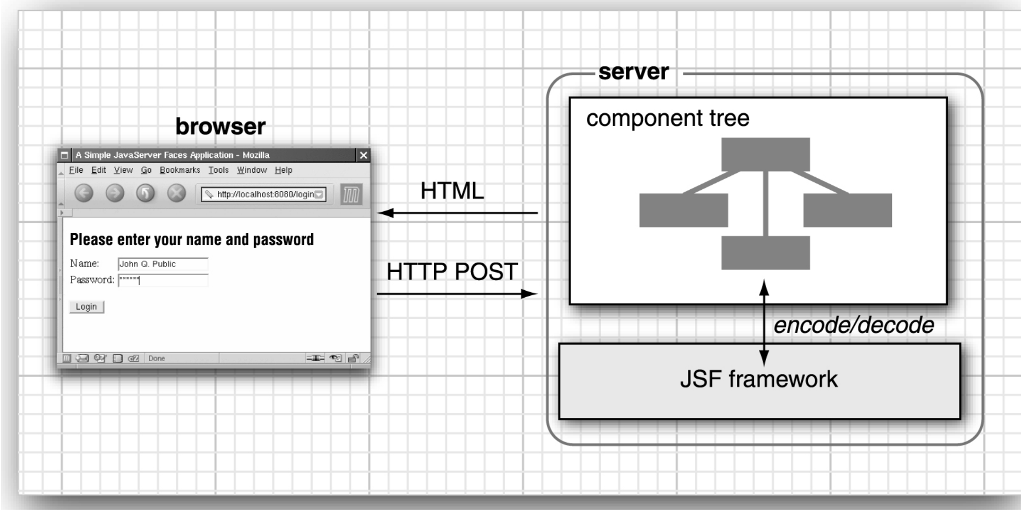
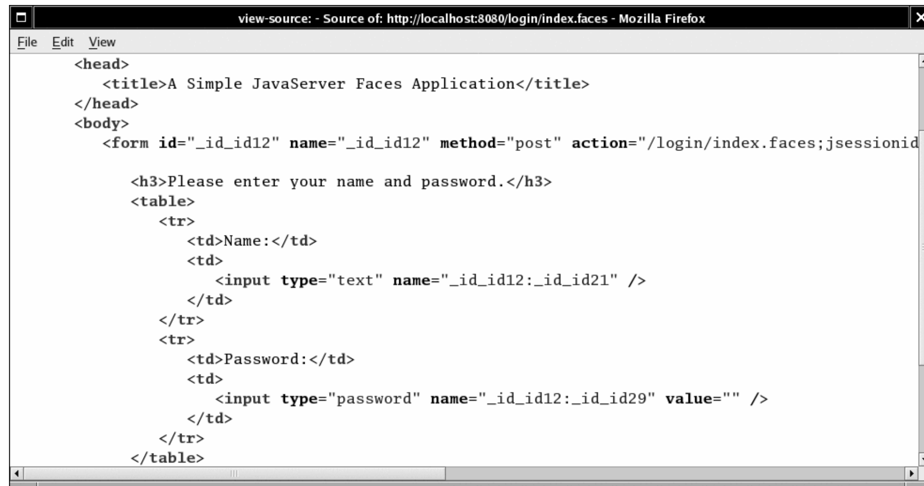


Figure 1–14 Encoding and decoding JSF pages

TIP: Select “View->Page source” from the browser menu to see the HTML output of the rendering process. Figure 1–15 shows a typical output. This is useful for debugging JSF problems.

getting-started-VM.fm Page 32 Monday, April 2, 2007 10:39 AM




```
<head>
   <title>A Simple JavaServer Faces Application</title>
</head>
<body>
   <form id="_id_id12" name="_id_id12" method="post" action="/login/index.faces;jsessionid

      <h3>Please enter your name and password.</h3>
      <table>
         <tr>
            <td>Name:</td>
            <td>
               <input type="text" name="_id_id12:_id_id21" />
            </td>
         </tr>
         <tr>
            <td>Password:</td>
            <td>
               <input type="password" name="_id_id12:_id_id29" value="" />
            </td>
         </tr>
      </table>
```

Figure 1–15 Viewing the source of the login page

Decoding Requests

After the page is displayed in the browser, the user fills in the form fields and clicks the login button. The browser sends the *form data* back to the web server, formatted as a *POST request*. This is a special format, defined as part of the HTTP protocol. The POST request contains the URL of the form (`/login/index.faces`), as well as the form data.

NOTE: The URL for the POST request is the same as that of the request that renders the form. Navigation to a new page occurs after the form has been submitted.

The form data is a string of ID/value pairs, such as

id1`=me&`*id2*`=secret&`*id3*`=Login`

As part of the normal servlet processing, the form data is placed in a hash table that all components can access.

Next, the JSF framework gives each component a chance to inspect that hash table, a process called *decoding*. Each component decides on its own how to interpret the form data.

The login form has three component objects: two `UIInput` objects that correspond to the text fields on the form and a `UICommand` object that corresponds to the submit button.

getting-started-VM.fm Page 33 Monday, April 2, 2007 10:39 AM

- The `UIInput` components update the bean properties referenced in the `value` attributes: they invoke the setter methods with the values that the user supplied.
- The `UICommand` component checks whether the button was clicked. If so, it fires an *action event* to launch the `login` action referenced in the `action` attribute. That event tells the navigation handler to look up the successor page, `welcome.jsp`.

Now the cycle repeats.

You have just seen the two most important processing steps of the JSF framework: encoding and decoding. However, the processing sequence (also called the *life cycle*) is a bit more intricate. If everything goes well, you do not need to worry about the intricacies of the life cycle. However, when an error occurs, you will definitely want to understand what the framework does. In the next section, we look at the life cycle in greater detail.

The Life Cycle

The JSF specification defines six distinct *phases*, as shown in Figure 1–16. The normal flow of control is shown with solid lines; alternative flows are shown with dashed lines.

The *Restore View* phase retrieves the component tree for the requested page if it was displayed previously or constructs a new component tree if it is displayed for the first time. If the page was displayed previously, all components are set to their prior state. This means that JSF automatically retains form information. For example, when a user posts illegal data that is rejected during decoding, the inputs are redisplayed so that the user can correct them.

If the request has no query data, the JSF implementation skips ahead to the *Render Response* phase. This happens when a page is displayed for the first time.

Otherwise, the next phase is the *Apply Request Values* phase. In this phase, the JSF implementation iterates over the component objects in the component tree. Each component object checks which request values belong to it and stores them.

NOTE: In addition to extracting request information, the Apply Request Values phase adds events to an event queue when a command button or link has been clicked. We discuss event handling in detail in Chapter 7. As you can see in Figure 1–16, events can be executed after each phase. In specialized situations, an event handler can “bail out” and skip to the Render Response phase or even terminate request processing altogether.

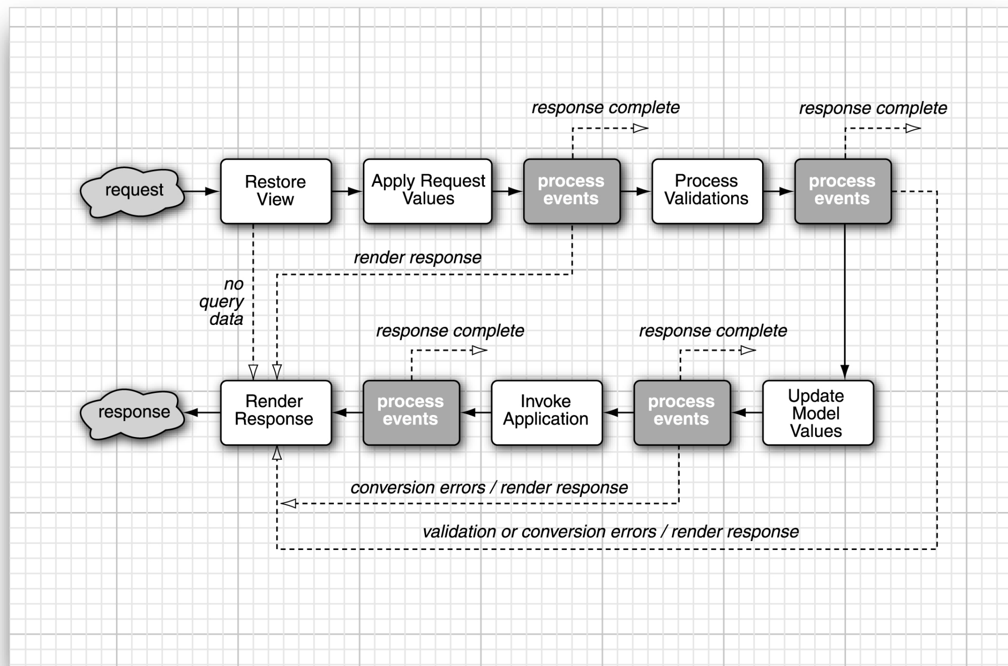
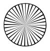
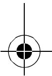
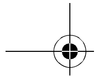
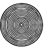


Figure 1–16 The JSF life cycle

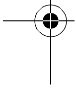
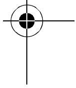
In the *Process Validations* phase, the submitted string values are first converted to "local values," which can be objects of any type. When you design a JSF page, you can attach *validators* that perform correctness checks on the local values. If validation passes, the JSF life cycle proceeds normally. However, when conversion or validation errors occur, the JSF implementation invokes the Render Response phase directly, redisplaying the current page so that the user has another chance to provide correct inputs.

NOTE: To many programmers, this is the most surprising aspect of the JSF life cycle. If a converter or validator fails, the current page is redisplayed. You should add tags to display the validation errors so that your users know why they see the old page again. See Chapter 6 for details.

After the converters and validators have done their work, it is assumed that it is safe to update the model data. During the *Update Model Values* phase, the local values are used to update the beans that are wired to the components.

getting-started-VM.fm Page 35 Monday, April 2, 2007 10:39 AM

In the *Invoke Application* phase, the `action` method of the button or link component that caused the form submission is executed. That method can carry out arbitrary application processing. It returns an outcome string that is passed to the navigation handler. The navigation handler looks up the next page.

Finally, the Render Response phase encodes the response and sends it to the browser. When a user submits a form, clicks a link, or otherwise generates a new request, the cycle starts anew.

You have now seen the basic mechanisms that make the JSF magic possible. In the following chapters, we examine the various parts of the life cycle in more detail.