CHAPTER 24

WEB APPLICATIONS

CHAPTER GOALS

To understand the web application concept

To learn the syntactical elements of the JavaServer Faces web application framework

To manage navigation in web applications

To build three-tier web applications

CHAPTER CONTENTS

24.1 The Architecture of a Web Application W1038

24.2 The Architecture of a JSF Application W1040

Special Topic 24.1: Session State and Cookies W1045

24.3 JavaBeans components W1046

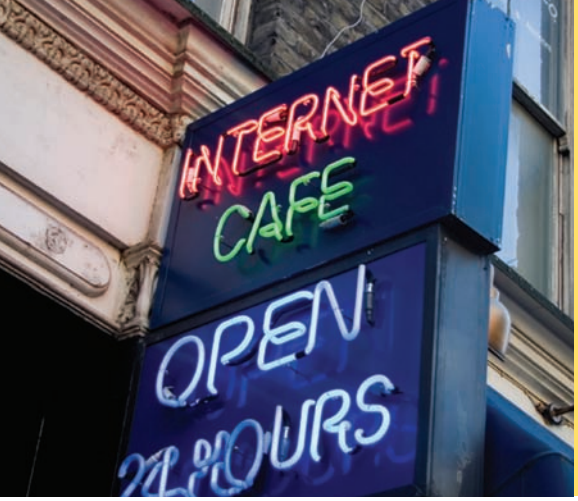
24.4 navigation between pages W1047

How To 24.1: Designing a Managed Bean W1053

24.5 JSF components W1054

24.6 A Three-Tier Application W1056

Special Topic 24.2: AJAX W1063



Web applications for a wide variety of purposes, such as e-mail, banking, shopping, and playing games, run on servers and interact with users through a web browser. Developing web-based user interfaces is more complex and challenging than writing graphical user interfaces. Fortunately, frameworks for web programming have emerged that are roughly analogous to Java's Swing framework for user-interface programming. In this chapter, you will learn how to write web applications using the JavaServer Faces (JSF) framework.

24.1 The Architecture of a Web Application

The user interface of a web application is displayed in a web browser.

A **web application** is an application whose user interface is displayed in a web browser. The application program resides on the web server. The user fills out form elements and clicks on buttons and links. The user inputs are transmitted over the Internet to the server, and the server program updates the web page that the user sees (see Figure 1).

The browser sends a request to the server using a protocol called HTTP (Hypertext transfer protocol). When a user clicks on a link, the request is very simple. The browser simply asks the server for the page with a given address, for example:

```
GET /index.html HTTP/1.1
Host: horstmann.com
```

When the user fills data (such as a user name and password) into a form and then clicks on a button, the HTTP request includes the data that the user provided. Such a request has a slightly different format, like this:

```
POST /login.xhtml HTTP/1.1
Host: horstmann.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 46
blank line
username=jqpublic&passwd=secret&login=Log%20in
```

When a form is submitted, the names and values of the form elements are sent to the web server.

The exact syntax of the request is not important; what matters is that HTTP simply tells the server what the user requested. As a result of the request, the server sends a web page in a format called HTML (Hypertext markup language). An HTML page contains tags that describe the structure of the page: headings, bullets, links, images, input elements, and so on.

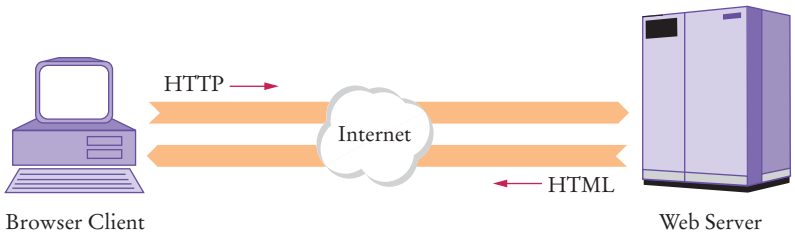


figure 1 The Architecture of a Web Application

For example, here is the HTML code for a simple form that prompts for a user name and password:

```
<html>
   <head>
      <title>A Simple Form</title>
   </head>
   <body>
      <form action="login.xhtml" method="POST">
         <p>
            User name:
            <input type="text" name="username" />
            Password:
            <input type="password" name="passwd" />
            <input type="submit" name="login" value="Log in"/>
         </p>
      </form>
   </body>
</html>
```

Figure 2 shows the form. Note that there are three input elements: a text field, a password field, and a submit button. (The HTML tags are summarized in Appendix J.)

Upon receiving the form data, the web server sends a new web page to the browser.

When a submit button is pressed, the form data is submitted to the server. The web server analyzes the request and sends a new HTML page to the browser. The new page might tell the user that the login was successful and ask the user to specify another action. Alternatively, the new page might tell the user that the login failed.

This simple example illustrates why it is difficult to implement a web application. Imagine what the server program has to do. At any time, it might receive a request with form data. At that point, the server program has to remember which form it has last sent to the client. It then needs to analyze the submitted data, decide what form to show next, and produce the HTML tags for that form.

There are multiple challenges. As described in Special Topic 24.1, the HTTP protocol is *stateless*—there is no memory of which form was last sent when a new request is received. Generating the HTML tags for a form is tedious. Perhaps most importantly, an application that consists of response strategies for a large number of request types is very hard to comprehend without additional structure.

In order to overcome these challenges, various web application frameworks have been developed. A web application framework hides the low-level details of analyzing HTTP and generating HTML from the application programmer. In this chapter, you will learn about the **JavaServer Faces** (JSF) framework, the web framework that is a part of the Java Enterprise Edition. You can think of JSF as "Swing for the Web".

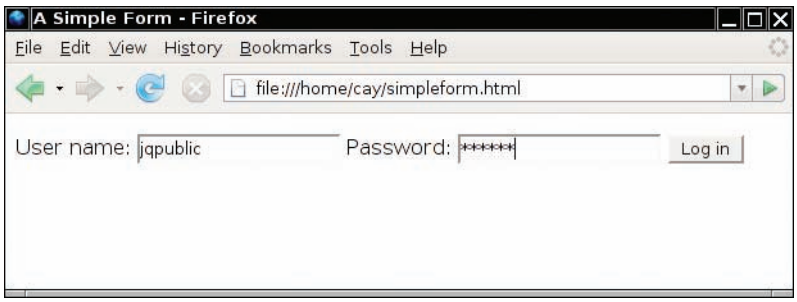


figure 2 A Simple Form

Both Swing and JSF handle the tedious details of capturing user input and painting text fields and buttons. Swing captures mouse and keyboard events and paints pixels in a frame. JSF handles form-posting events and paints by emitting HTML code. This chapter describes JSF 2.0, an improved version of the original JSF framework, that became available in 2009.

1. Why are two different protocols (HTML and HTTP) required by a web application?
2. How can a web application know which user is trying to log in when the information of the sample login screen is submitted?

practice it Now you can try these exercises at the end of the chapter: R24.1, R24.2.

24.2 The Architecture of a JSF Application

In the following sections, we give an overview of the architecture of a JSF application and show a very simple sample application.

24.2.1 JSF Pages

The user interface of a JSF application is described by a set of *JSF pages*. Each JSF page has the following structure:

```
<?xml version="1.0" encoding="UTF-8"?>
<html xmlns="http://www.w3.org/1999/xhtml"
   xmlns:h="http://java.sun.com/jsf/html">
   <h:head>
      <title>Page title</title>
   </h:head>
   <h:body>
      <h:form>
         Page contents
      </h:form>
   </h:body>
</html>
```

A JavaServer Faces (JSF) page contains HTML and JSF tags.

You can think of this as the required "plumbing", similar to the `public static void main` incantation that is required for every Java program. If you compare this page with the HTML page from the preceding section, you will notice that the main elements are very similar to a regular HTML page, but several elements (`head`, `body`, and `form`) are JSF tags with an `h:` prefix.

Here is a complete example of a JSF page:

section_2/time/index.xhtml

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <html xmlns="http://www.w3.org/1999/xhtml"
3     xmlns:h="http://java.sun.com/jsf/html">
4     <h:head>
5        <title>The time application</title>
6     </h:head>
```

```
7     <h:body>
8        <h:form>
9           <p>
10             The current time is #{timeBean.time}
11          </p>
12       </h:form>
13    </h:body>
14 </html>
```

Figure 3 shows the result of executing the program.

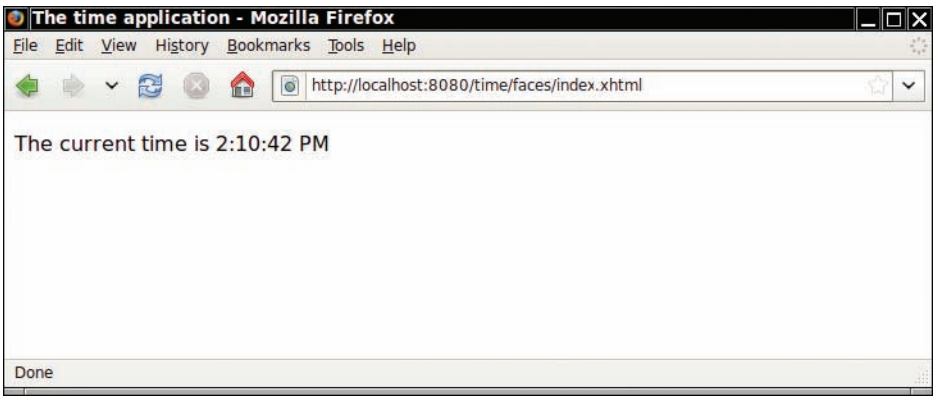


figure 3 Executing the `time` Web Application

The JSF container converts a JSF page to an HTML page, replacing all JSF tags with text and HTML tags.

The purpose of a JSF page is to *generate* an HTML page. The basic process is as follows:

- The HTML tags that are present in the JSF page (such as `title` and `p`) are retained. These are the *static* part of the page: the formatting instructions that do not change.
- The JSF tags are translated into HTML. This translation is *dynamic:* it depends on the state of Java objects that are associated with the tags. In our example, the expression `#{timeBean.time}` has been replaced by dynamically generated text, namely the current time.

Figure 4 shows the basic process. The browser requests a JSF page. The page is processed by the **JSF container**, the server-side software that implements the JSF framework. The JSF container translates all JSF tags into text and HTML tags, yielding a pure HTML page. That page is transmitted to the client browser. The browser displays the page.

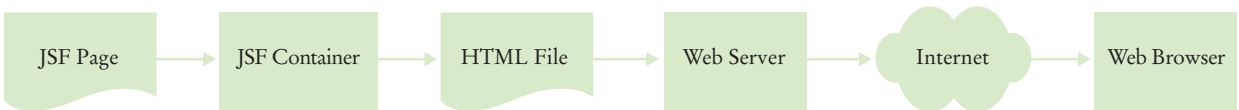


figure 4 The JSF Container Rewrites the Requested Page

24.2.2 Managed Beans

A managed bean is an object that is controlled by the JSF container.

The expression `#{timeBean.time}` is called a **value expression.** Value expressions invoke method calls on Java objects, which are called *managed beans*.

These objects are called "managed" because they are controlled by the JSF container. The container creates a managed bean when it is first used in a value expression. The *scope* of the managed bean determines which clients can access the object and how long the object stays alive.

A bean with session scope is available for multiple requests by the same browser.

In this chapter, we only consider managed beans with *session scope*. A session-scoped object can be accessed by all requests from the same browser. If multiple users are simultaneously accessing a JSF application, each of them is given a separate object. This is a good default for simple web applications.

Below is the code for the `TimeBean` class. Note the following:

- You declare a session-scoped managed bean with the annotations `@ManagedBean` and `@SessionScoped`.
- The name of the bean in a value expression is the class name with the first letter changed to lowercase, e.g., `timeBean`.
- The value expression `timeBean.time` calls the `getTime` method. You will see the reason in the next section.
- The `getTime` method uses the `DateFormat` class to format the current time, producing a string such as `9:00:00 AM`.
- When deploying the application, all class files must be placed inside the `WEB-INF/classes` directory. Because many application servers also require that classes be contained in a package, we place our classes inside the `bigjava` package. For that reason, the class is contained in the `WEB-INF/classes/bigjava` directory.

section_2/time/Web-inf /classes/bigjava/Timebean.java

```
1  package bigjava;
2
3  import java.text.DateFormat;
4  import java.util.Date;
5  import java.util.TimeZone;
6  import javax.faces.bean.ManagedBean;
7  import javax.faces.bean.SessionScoped;
8
9  @ManagedBean
10 @SessionScoped
11 public class TimeBean
12 {
13    private DateFormat timeFormatter;
14
15    /**
16       Initializes the formatter.
17    */
18    public TimeBean()
19    {
20       timeFormatter = DateFormat.getTimeInstance();
21    }
22
23    /**
24       Read-only time property.
25       @return the formatted time
```

```
26      */
27      public String getTime()
28      {
29         Date time = new Date();
30         String timeString = timeFormatter.format(time);
31         return timeString;
32      }
33   }
```

24.2.3 Separation of Presentation and Business Logic

The JSF technology enables the separation of presentation and business logic.

We will look at value expressions and managed beans in more detail in the next section. The key observation is that every JSF application has two parts: *presentation* and *business logic*.

The term "presentation" refers to the user interface of the web application: the arrangement of the text, images, buttons, and so on. The *business logic* is the part of the application that is independent of the visual presentation. In commercial applications, it contains the rules that are used for business decisions: what products to offer, how much to charge, to whom to extend credit, and so on. In our example, we simulated the business logic with a `TimeBean` object.

JSF pages define the presentation logic. Managed beans define the business logic. Value expressions tie the two together.

The separation of presentation logic and business logic is very important when designing web applications. Some web technologies place the code for the business logic right into the web page. However, this quickly turns into a serious problem. Programmers are rarely skilled in web design (as you can see from the boring web pages in this chapter). Graphic designers don't usually know much about programming and find it very challenging to improve web pages that contain a lot of code. JSF solves this problem. In JSF, the graphic designer only sees the elements that make up the presentation logic. It is easy to take a boring JSF page and make it pretty by adding banners, icons, and so on.

24.2.4 Deploying a JSF Application

To run a JSF application, you need a server with a **JSF container**. We suggest that you use the GlassFish application server, `http://glassfish.java.net`, which has, together with many other features that you can ignore, a JSF container and a convenient administration interface.

To deploy a JSF application, follow these steps:

1. Make a separate directory tree for each web application.
2. Place JSF pages (such as `index.xhtml`) into the root directory of the application's directory tree.
3. Create a `WEB-INF` subdirectory in your application directory.
4. Place all Java classes inside a `classes` subdirectory of the `WEB-INF` directory. Note that you should place your classes into a package. Compile with

```
cd WEB-INF/classes
javac -classpath glassfish/modules/jsf-api.jar bigjava/*.java
```

5. Place the file `web.xml` (which is shown below) inside the `WEB-INF` subdirectory. Some servers need the `web.xml` file to configure the JSF container. We also turn on development mode, which gives better error messages.
6. Zip up all application files into a file with extension `.war` (Web Archive). This is easily achieved by running the `jar` command from the command line, after changing to the application directory. For example,

   ```
   cd time
   jar cvf time.war .
   ```

   The period (`.`) denotes the current directory. The `jar` command creates an archive `time.war` consisting of all files in all subdirectories of the current directory.
7. Make sure the application server is started. The application server listens to web requests, typically on port 8080.
8. Deploy the application to the application server. With GlassFish, this can be achieved either through the administrative interface or simply by copying the WAR file into a special deployment directory. By default, this is the subdirectory `domains/domain1/autodeploy` inside the GlassFish installation directory.
9. Point your browser to a URL such as `http://localhost:8080/time/faces/index.xhtml`. Note the `faces` part in the URL. If you forget this part, the file will not be processed by the JSF container.

Figure 5 shows the directory structure for the application.

figure 5
The Directory Structure of the `time` Application

section_2/time/Web-inf /web.xml

```
<?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"
   version="2.5">
   <servlet>
      <servlet-name>Faces Servlet</servlet-name>
      <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
   </servlet>
   <servlet-mapping>
      <servlet-name>Faces Servlet</servlet-name>
      <url-pattern>/faces/*</url-pattern>
   </servlet-mapping>
   <welcome-file-list>
      <welcome-file>faces/index.xhtml</welcome-file>
   </welcome-file-list>
   <context-param>
      <param-name>javax.faces.PROJECT_STAGE</param-name>
```

```
21         <param-value>Development</param-value>
22      </context-param>
23   </web-app>
```

3. What steps are required to add the image of a clock to the `time` application? (The clock doesn't have to show the correct time.)
4. Does a Swing program automatically separate presentation and business logic?
5. Why does the WAR file need to be deployed to the application server?

practice it Now you can try these exercises at the end of the chapter: R24.1, R24.7, P24.1.

Special Topic 24.1

Session State and cookies

Recall that HTTP is a *stateless* protocol. A browser sends a request to a web server. The web server sends the reply and then disconnects. This is different from other protocols, such as POP, where the mail client logs into the mail server and stays connected until it has retrieved all e-mail messages. In contrast, a browser makes a new connection to the web server for each web page, and the web server has no way of knowing that those connections originate from the same browser. This makes it difficult to implement web applications. For example, in a shopping application, it is essential to track which requests came from a particular shopper.

Cookies were invented to overcome this restriction. A cookie consists of a small string that the web server sends to a browser, and that the browser sends back to the same server with all further requests. That way, the server can tie the stream of requests together. The JSF container matches up the cookies with the beans that have `session` scope. When a browser request contains a cookie, the value expressions in the JSF page refer to the matching beans.

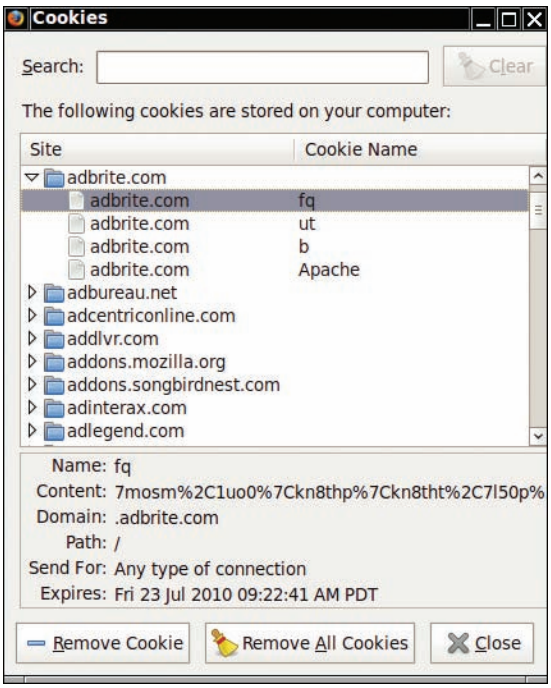


figure 6 Viewing the Cookies in a Browser

You may have heard some privacy advocates complaining about cookies. Cookies are not inherently evil. When used to establish a session or to remember login information, they can make web applications more user-friendly. But when cookies are used to track your identity while you surf the Web, there can be privacy concerns. For example, Figure 6 shows some of the cookies that my browser held on a particular day. I have no recollection of visiting the advertising sites, so it is a bit disconcerting to see that my browser communicated with them.

Some people turn off cookies, and then web applications need to use another scheme to establish a session, typically by embedding a session identifier in the request URL or in a hidden field of a form. The JSF session mechanism automatically switches to URLs with session identifiers if the client browser doesn't support cookies.

24.3 JavaBeans Components

Properties of a software component can be accessed without having to write Java code.

A *software component* is an entity that encapsulates functionality and can be plugged into a software system without programming. A managed bean is an example of a software component. When we added the `timeBean` object to the web application, we did not write Java code to construct the object or to call its methods.

Some programming languages have explicit support for components, but Java does not. Instead, in Java, you use a programming convention to implement components. A **JavaBean** is a Java class that follows this convention. A JavaBean exposes **properties**—values of the component that can be accessed without programming.

A JavaBean is a class that exposes properties through its get and set methods.

Just about any Java class can be a JavaBean—there are only two requirements:

- A JavaBean must have a constructor with no arguments.
- A JavaBean must have methods for accessing the component properties that follow the `get/set` naming convention. For example, to get or set a property named `city`, the methods must be called `getCity` and `setCity`.

In general, if the name of the property is *propertyName*, and its type is *Type*, then the associated methods must be of the form

```
public Type getPropertyName()
public void setPropertyName(Type newValue)
```

Note that the name of a property starts with a lowercase letter (such as `city`), but the corresponding methods have an uppercase letter (`getCity`). The only exception is that property names can be all capitals, such as `ID` or `URL`, with corresponding methods `getID` or `setURL`.

If a property has only a `get` method, then it is a *read-only* property. If it has only a `set` method, then it is a *write-only* property.

A JavaBean can have additional methods, but they are not connected with properties.

Here is a simple example of a bean class that formats the time for a given city, which we will further develop in the next section:

```
public class TimeZoneBean
{
   // Instance variables
   . . .
   // Required constructor with no arguments
   public TimeZoneBean() { . . . }
```

```
   // city property
   public String getCity() { . . . }
   public void setCity(String newValue) { . . . }

   // Read-only time property
   public String getTime() { . . . }

   // Other methods
   . . .
}
```

This bean has two properties: `city` and `time`.

You should *not* make any assumptions about the internal representation of properties in the bean class. The getter and setter methods may simply read or write an instance variable. But they may also do other work. An example is the `getTime` method from the `TimeBean` in the preceding section; it formats the current time.

In the value expression of an output tag, only the property getter is called.

When a property name is used in a value expression that is included in the JSF page, then the get method is involved. For example, when the string

```
The current time is #{timeBean.time}
```

is rendered, the JSF container calls the `getTime` method of the session's `TimeBean` instance.

In the value expression of an input tag, the property setter is called when the page is submitted.

When a property name is used in an `h:inputText` tag (that, is the equivalent of an HTML `input` field or a `JTextField`), the situation is more complex. Consider this example:

```
<h:inputText value="#{timeZoneBean.city}"/>
```

When the JSF page is first displayed, the `getCity` method is called, and the current value of the `city` property is displayed. But after the user submits the page, the `setCity` method is called. It sets the `city` property to the value that the user typed into the input field.

6. Is the `Scanner` class a JavaBean?
7. What work does the `setCity` method of the `TimeZoneBean` do?

practice it Now you can try these exercises at the end of the chapter: R24.5, R24.6, P24.2.

24.4 Navigation Between Pages

In most web applications, users will want to move between different pages. For example, a shopping application might have a login page, a page to show products for sale, and a checkout page that shows the shopping cart. In this section, you will learn how to enable users to navigate from one page to another.

Consider a sample time zone program that displays the current time. If the time computation uses the time zone *at the server location*, it will not be very useful when the user is in another time zone. Therefore, the program will prompt for the city in which the user is located. When the user clicks a submit button, the program moves to the page `next.xhtml` and display the time in the user's time zone (see Figure 7). However, if no time zone is available for the city, the program displays the page `error.xhtml`.

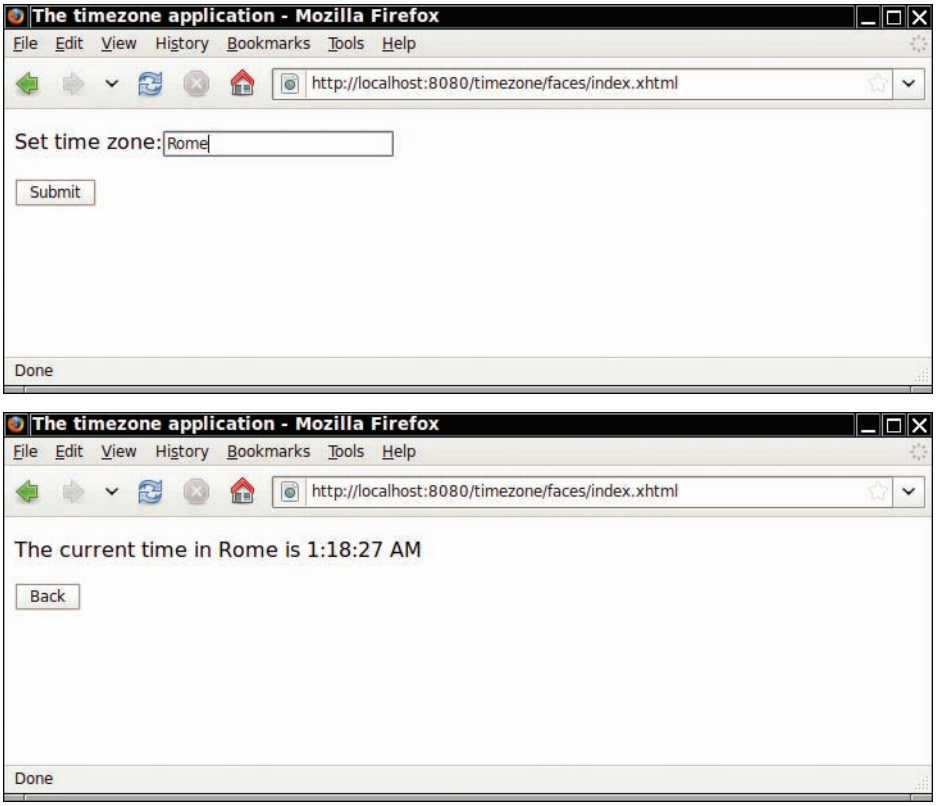


figure 7 The `timezone` Application

The outcome string of an action determines the next page that the JSF container sends to the browser.

A button yields an *outcome*, a string that determines the next page. Unless specified otherwise, the next page is the outcome string with the `.xhtml` extension added. For example, if the outcome string is `error`, the next page is `error.xhtml`. (It is possible to specify a different mapping from outcomes to pages, but there is no need to do so for a simple application.)

In many situations, the next page depends on the result of some computation. In our example, we need different outcomes depending on the city that the user entered. To achieve this flexibility, you specify a **method expression** as the `action` attribute:

```
<h:commandButton value="Submit" action="#{timeZoneBean.checkCity}"/>
```

A method expression specifies a bean and a method that should be invoked on the bean.

A method expression consists of the name of a bean and the name of a method. When the form is submitted, the JSF container calls `timeZoneBean.checkCity()`. The `checkCity` method returns the outcome string:

```
public class TimeZoneBean
{
   . . .
   public String checkCity()
   {
      zone = getTimeZone(city);
      if (zone == null) { return "error"; }
      return "next";
   }
}
```

If the next page does not depend on a computation, then you set the `action` attribute of the button to a fixed outcome string, like this:

```
<h:commandButton value="Back" action="index"/>
```

If a button has no `action` attribute, or if the action outcome is `null`, then the current page is redisplayed.

We can now complete our time zone application. The Java library contains a convenient `TimeZone` class that knows about time zones across the world. A time zone is identified by a string such as `"America/Los_Angeles"` or `"Asia/Tokyo"`. The static method `getAvailableIDs` returns a string array containing all IDs:

```
String[] ids = TimeZone.getAvailableIDs();
```

There are several hundred time zone IDs. (We are using time zones in this example because the `TimeZone` class gives us an interesting data source with lots of data. Later in this chapter, you will see how to access data from a database, but of course that's more complex.)

The static `getTimeZone` method returns a `TimeZone` object for a given ID string:

```
String id = "America/Los_Angeles";
TimeZone zone = TimeZone.getTimeZone(id);
```

Once you have a `TimeZone` object, you can use it in conjunction with a `DateFormat` object to get a time string in that time zone.

```
DateFormat timeFormatter = DateFormat.getTimeInstance();
timeFormatter.setTimeZone(zone);
Date now = new Date();
// Suppose the server is in New York, and it's noon there
System.out.println(timeFormatter.format(now));
// Prints 9:00:00 AM
```

Of course, we don't expect the user to know about time zone ID strings, such as `"America/Los_Angeles"`. Instead, we assume that the user will simply enter the city name. The time zone bean will check whether that string, with spaces replaced by underscores, appears at the end of one of the valid time zone IDs.

Here is the code for the bean class:

section_4/timezone/Web-inf /classes/bigjava/TimeZonebean.java

```
1  package bigjava;
2
3  import java.text.DateFormat;
4  import java.util.Date;
5  import java.util.TimeZone;
6  import javax.faces.bean.ManagedBean;
7  import javax.faces.bean.SessionScoped;
8
9  /**
10    This bean formats the local time of day for a given city.
11 */
12 @ManagedBean
13 @SessionScoped
14 public class TimeZoneBean
15 {
16    private DateFormat timeFormatter;
17    private String city;
18    private TimeZone zone;
19
20    /**
```

```
21          Initializes the formatter.
22       */
23       public TimeZoneBean()
24       {
25          timeFormatter = DateFormat.getTimeInstance();
26       }
27
28       /**
29          Setter for city property.
30          @param aCity the city for which to report the local time
31       */
32       public void setCity(String aCity)
33       {
34          city = aCity;
35       }
36
37       /**
38          Getter for city property.
39          @return the city for which to report the local time
40       */
41       public String getCity()
42       {
43          return city;
44       }
45
46       /**
47          Read-only time property.
48          @return the formatted time
49       */
50       public String getTime()
51       {
52          if (zone == null) { return "not available"; }
53          timeFormatter.setTimeZone(zone);
54          Date time = new Date();
55          String timeString = timeFormatter.format(time);
56          return timeString;
57       }
58
59       /**
60          Action for checking a city.
61          @return "next" if time zone information is available for the city,
62          "error" otherwise
63       */
64       public String checkCity()
65       {
66          zone = getTimeZone(city);
67          if (zone == null) { return "error"; }
68          return "next";
69       }
70
71       /**
72          Looks up the time zone for a city.
73          @param aCity the city for which to find the time zone
74          @return the time zone or null if no match is found
75       */
76       private static TimeZone getTimeZone(String aCity)
77       {
78          String[] ids = TimeZone.getAvailableIDs();
79          for (int i = 0; i < ids.length; i++)
80          {
```

```
81            if (timeZoneIDmatch(ids[i], aCity))
82            {
83               return TimeZone.getTimeZone(ids[i]);
84            }
85         }
86         return null;
87      }
88
89      /**
90         Checks whether a time zone ID matches a city.
91         @param id the time zone ID (e.g., "America/Los_Angeles")
92         @param aCity the city to match (e.g., "Los Angeles")
93         @return true if the ID and city match
94      */
95      private static boolean timeZoneIDmatch(String id, String aCity)
96      {
97         String idCity = id.substring(id.indexOf('/') + 1);
98         return idCity.replace('_', ' ').equals(aCity);
99      }
100  }
```

Following is the JSF page for setting the city. The `h:inputText` tag produces an input field and the `h:commandButton` tag produces a button. (We discuss its `action` attribute in the next section.) When the user clicks the button, the browser sends the form values (that is, the contents of the input field) back to the web application. The web application calls the `setCity` method on the bean because the input field has a `#{timeZoneBean.city}` value expression.

section_4/timezone/index.xhtml

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <html xmlns="http://www.w3.org/1999/xhtml"
3     xmlns:h="http://java.sun.com/jsf/html">
4     <h:head>
5        <title>The timezone application</title>
6     </h:head>
7     <h:body>
8        <h:form>
9           <p>
10             Set time zone:
11             <h:inputText value="#{timeZoneBean.city}"/>
12          </p>
13          <p>
14             <h:commandButton value="Submit"
15                action="#{timeZoneBean.checkCity}"/>
16          </p>
17       </h:form>
18    </h:body>
19 </html>
```

The next JSF page shows the result, using two value expressions that display the `city` and `time` properties. These expressions invoke the `getCity` and `getTime` methods of the bean class.

section_4/timezone/next.xhtml

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <html xmlns="http://www.w3.org/1999/xhtml"
```

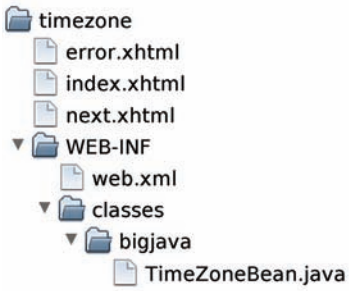
```
3     xmlns:h="http://java.sun.com/jsf/html">
4     <h:head>
5        <title>The timezone application</title>
6     </h:head>
7     <h:body>
8        <h:form>
9           <p>
10             The current time in #{timeZoneBean.city} is #{timeZoneBean.time}
11          </p>
12          <p>
13             <h:commandButton value="Back" action="index"/>
14          </p>
15       </h:form>
16    </h:body>
17 </html>
```

section_4/timezone/error.xhtml

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <html xmlns="http://www.w3.org/1999/xhtml"
3     xmlns:h="http://java.sun.com/jsf/html">
4     <h:head>
5        <title>The timezone application</title>
6     </h:head>
7     <h:body>
8        <h:form>
9           <p>
10             Sorry, no information is available for #{timeZoneBean.city}
11          </p>
12          <p>
13             <h:commandButton value="Back" action="index"/>
14          </p>
15       </h:form>
16    </h:body>
17 </html>
```

Figure 8 shows the directory structure of the `timezone` application.

figure 8
The Directory Structure of the `timezone` Application



8. What tag would you need to add to `error.xhtml` so that the user can click on a button labeled "Help" and see `help.xhtml`?
9. Which page would be displayed if the `checkCity` method returned `null` instead of "error"?

practice it Now you can try these exercises at the end of the chapter: R24.10, P24.4, P24.5.

HOW TO 24.1 Designing a managed bean

A managed bean is just a regular Java class, with these three special characteristics:

- The bean must have a constructor with no arguments.
- Methods of the form

  Type `get`*PropertyName*`()`
  `void set`*PropertyName*`(`*Type* `x)`

  define properties that can be accessed from JSF pages.
- Methods of the form

  `String` *methodName*`()`

  can be used to specify command actions.

This How To provides step-by-step instructions for designing a managed bean class.

Step 1 Decide on the responsibility of the bean.

When designing a JSF application, it is tempting to stuff all code into a single bean class. Some development environments even encourage this approach. However, from a software engineering perspective, it is best to come up with different beans for different responsibilities. For example, a shopping application might have a `UserBean` to describe the current user, a `SiteBean` to describe how the user visits the shopping site, and a `ShoppingCartBean` that holds the items that the user is purchasing.

Step 2 Discover the properties that the bean should expose.

A property is an entity that you want to access or modify from your JSF pages. For example, a `UserBean` might have properties `firstName`, `lastName`, and `password`.

Sometimes, you have to resort to a bit of trickery. For example, consider adding an item to the shopping cart. You could use a property `items`, but it would be cumbersome to access all items in a JSF page and then set `items` to a new collection that contains one additional element. Instead, you can design a property `addedItem`. When that property is set, the `setAddedItem` method of your bean adds its value to the collection of items.

Step 3 Settle on the type and access permissions for each property.

Properties that are only used to generate output can be read-only. Properties that are used in `h:inputText` and other input tags must have read-write access.

Step 4 Define action methods for navigation.

Your action methods can carry out arbitrary tasks in order to react to the user inputs. The only limitation is that they don't have access to the form data. Everything that the user entered on the form must have already been set as a bean property.

The action method's return value is the name of the next page to be displayed, or `null` if you want to redisplay the current page.

Step 5 Implement the constructor with no arguments.

The constructor initializes any instance variables that are reused whenever the bean's computation is executed. Examples are formatters, random number generators, and so on.

Step 6 Implement the `get` and `set` methods for all properties.

Most `get` and `set` methods simply get or set an instance variable. However, you can carry out arbitrary computations in these methods if it is convenient. For example, a `get` method may retrieve information from a database instead of an instance variable.

Step 7 Supply any needed helper methods.

Your bean can have methods that are not property getters and setters. For example, the `TimeZoneBean` has helper methods to look up the time zone for a city.

24.5 JSF Components

There are JSF components for text input, choices, buttons, and images.

The value attribute of an input component denotes the value that the user supplies.

In this section, you will see the most useful user-interface components that you can place on a JSF form. Table 1 shows a summary. (For a comprehensive discussion of all JSF components, see *Core JavaServer Faces*, 3rd ed., by David Geary and Cay Horstmann (Sun Microsystems Press/Prentice Hall, 2010)).

Each component has a `value` attribute that allows you to connect the component value with a bean property, for example

```
<h:inputSecret value="#{user.password}"/>
```

The `h:inputTextArea` component has attributes to specify the rows of text and columns of characters, such as

```
<h:inputTextArea value="#{user.comment}" rows="10" cols="40"/>
```

Table 1 Common JSF Components

Component	JSF Tag	Common Attributes	Example
Text Field	`h:inputText`	`value`	12345678901234567890
Password Field	`h:inputSecret`	`value`	**********

Text Area	`h:inputTextArea`	`value`
		`rows`
		`cols`
			line one
			line two
			line three

Radio Button Group	`h:selectOneRadio`	`value` `layout`	Cheese Pickle Mustard Lettuce Onions
Checkbox	`h:selectOneCheckbox`	`value`	Receive email:
Checkbox Group	`h:selectManyCheckbox`	`value` `layout`	Cheese Pickle Mustard Lettuce Onions
Menu	`h:selectOneMenu` `h:selectManyMenu`	`value`	Cheese Pickle Mustard Lettuce
Image	`h:graphicImage`	`value`	
Submit Button	`h:commandButton`	`value` `action`	press me

The radio button and checkbox groups allow you to specify horizontal or vertical layout:

```
<h:selectOneRadio value="#{burger.topping}" layout="lineDirection">
```

In European languages, `lineDirection` means horizontal and `pageDirection` means vertical. However, in some languages, lines are written top-to-bottom, and the meanings are reversed.

Button groups and menus are more complex than the other user-interface components. They require you to specify two properties:

- the collection of possible choices
- the actual choice

Use an `f:selectItems` tag to specify all choices for a component that allows selection from a list of choices.

The `value` attribute of the component specifies the actual choice to be displayed. The collection of possible choices is defined by a nested `f:selectItems` tag, like this:

```
<h:selectOneRadio value="#{creditCardBean.expirationMonth}"
      layout="pageDirection">
   <f:selectItems value="#{creditCardBean.monthChoices}"/>
</h:selectOneRadio>
```

When you use the `f:selectItems` tag, you need to add the namespace declaration

```
xmlns:f="http://java.sun.com/jsf/core"
```

to the `html` tag at the top of your JSF page.

The value of the `f:selectItems` tag must have a type that can describe a list of choices. There are several types that you can use, but the easiest—and the only one that we will discuss—is a `Map`. The keys of the map are the *labels*—the strings that are displayed next to each choice. The corresponding map values are the *label values*—the values that correspond to the selection. For example, a choice map for months would map January to 1, February to 2, and so on:

```
public class CreditCardBean
{
   . . .
   public Map<String, Integer> getMonthChoices()
   {
      Map<String, Integer> choices = new LinkedHashMap<String, Integer>();
      choices.put("January", 1);
      choices.put("February", 2);
      . . .
      return choices;
   }
}
```

Here, we use a `LinkedHashMap` because we want to visit entries in the order in which they are inserted. This is more useful than a `HashMap`, which would visit the labels in random order or a `TreeMap`, which would visit them in alphabetical order (starting with April!).

The type of the `value` property of the component enclosing the `f:selectItems` tag must match the type of the map value. For example, `creditCardBean.expirationMonth` must be an integer, not a string. If multiple selections are allowed, the type of the `value` property must be a list or array of matching types. For example, if one could choose multiple months, a `selectManyRadio` component would have a `value` property with a type such as `int[]` or `ArrayList<Integer>`.

10. Which JSF components can be used to give a user a choice between "AM/PM" and "military" time?
11. How would you supply a set of choices for a credit card expiration year to a `h:selectOneMenu` component?

practice it Now you can try these exercises at the end of the chapter: R24.11, P24.3, P24.9.

24.6 A Three-Tier Application

A three-tier application has separate tiers for presentation, business logic, and data storage.

In this chapter's final JSF example, you will see a web application with a very common structure. In this example, we will use a database for information storage. We will enhance the time zone example by storing additional cities that are not known to the `TimeZone` class in a database. Such an application is called a **three-tier application** because it consists of three separate layers or tiers (see Figure 9):

- The presentation tier: the web browser
- The "business logic" tier: the JSF container, the JSF pages, and the JavaBeans
- The storage tier: the database

Contrast the three-tier architecture with the more traditional *client-server* or *two-tier architecture* that you saw in the database programs of Chapter 22. In that architecture, one of the tiers is the database server, which is accessed by multiple client programs on desktops. Each client program has a presentation layer—usually with a specially programmed graphical user interface—and business logic code. (See Figure 10.) When the business logic changes, a new client program must be distributed over all desktops. In contrast, in a three-tier application, the business logic resides on a server. When the logic changes, the server code is updated, but the presentation tier—the browser—remains unchanged. That is much simpler to manage than updating multiple desktops.

In our example, we will have a single database table, `CityZone`, with city and time zone names (see Figure 11).

section_6/multizone/sql/cityZone.sql

```
1  CREATE TABLE CityZone (City VARCHAR(40), Zone VARCHAR(40))
2  INSERT INTO CityZone VALUES ('San Francisco', 'America/Los_Angeles')
3  INSERT INTO CityZone VALUES ('Hamburg', 'Europe/Rome')
4  SELECT * FROM CityZone
```

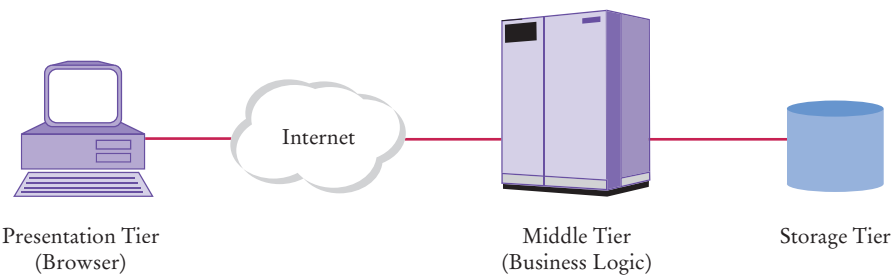


figure 9 Three-Tier Architecture

figure 10 Two-Tier Client-Server Architecture

If the `TimeZoneBean` can't find the city among the standard time zone IDs, it makes a database query:

```
SELECT Zone FROM CityZone WHERE City = the requested city
```

If there is a matching entry in the database, that time zone is returned.

To query the database, the bean needs a `Connection` object. In Chapter 22, we used the static `getConnection` method of the `DriverManager` class to obtain a database connection. However, JSF containers have a better mechanism for configuring a database in one central location so that multiple web applications can access it.

The GlassFish application server includes the Derby database. It has a predefined data source with the resource name `jdbc/__default`. In your bean code, you declare an instance variable of type `DataSource` and tag it with a `@Resource` annotation, like this:

```
@Resource(name="jdbc/__default")
private DataSource source;
```

You can use the administrative interface of GlassFish to define other data sources.

When the application server loads the web application, it automatically initializes this instance variable. Whenever you need a database connection, call

```
Connection conn = source.getConnection();
try
{
   Use the connection.
}
finally
{
   conn.close();
}
```

You define data sources in the JSF container and use resource annotations to initialize them.

The application server provides an additional service: it *pools* database connections. When a pooled connection is closed, it is not physically terminated but instead

cityZone

City	Zone
San Francisco	America/Los_Angeles
Hamburg	Europe/Rome
. . .	. . .

figure 11 The `CityZone` Table

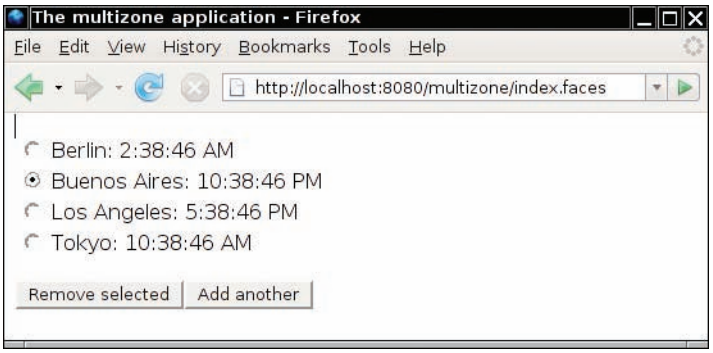


figure 12 The `multizone` Application Shows a List of Cities

returned to a queue and given out again to another caller of the `getConnection` method. Pooling avoids the overhead of creating new database connections. In a web application, it would be particularly inefficient to connect to the database with every web request. Connection pooling is completely automatic.

In order to make the application more interesting, we enhanced the `TimeZoneBean` so that it manages a list of cities. You can add cities to the list and remove a selected city (see Figure 12).

You will find the code for this web application below. Figure 13 shows the directory structure of the application.

figure 13
The Directory Structure of the `multizone` Application

You have now seen how to use the JavaServer Faces technology to build web applications. JSF takes care of low-level details so that you don't have to think about HTML forms and the HTTP protocol. Instead, you can focus on the presentation and business logic of your application.

section_6/multizone/index.xhtml

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <html xmlns="http://www.w3.org/1999/xhtml"
3     xmlns:h="http://java.sun.com/jsf/html">
4     <h:head>
5        <title>The multizone application</title>
6     </h:head>
7     <h:body>
```

```
8         <h:form>
9            <p>
10              Enter city:
11              <h:inputText value="#{timeZoneBean.cityToAdd}"/>
12           </p>
13           <p>
14              <h:commandButton value="Submit"
15                    action="#{timeZoneBean.addCity}"/>
16           </p>
17        </h:form>
18     </h:body>
19  </html>
```

section_6/multizone/next.xhtml

```
1   <?xml version="1.0" encoding="UTF-8"?>
2   <html xmlns="http://www.w3.org/1999/xhtml"
3      xmlns:f="http://java.sun.com/jsf/core"
4      xmlns:h="http://java.sun.com/jsf/html">
5      <h:head>
6         <title>The multizone application</title>
7      </h:head>
8      <h:body>
9         <h:form>
10           <p>
11              <h:selectOneRadio value="#{timeZoneBean.cityToRemove}"
12                 layout="pageDirection">
13                 <f:selectItems value="#{timeZoneBean.citiesAndTimes}"/>
14              </h:selectOneRadio>
15           </p>
16           <p>
17              <h:commandButton value="Remove selected"
18                    action="#{timeZoneBean.removeCity}"/>
19              <h:commandButton value="Add another" action="index"/>
20           </p>
21        </h:form>
22     </h:body>
23  </html>
```

section_6/multizone/error.xhtml

```
1   <?xml version="1.0" encoding="UTF-8"?>
2   <html xmlns="http://www.w3.org/1999/xhtml"
3      xmlns:h="http://java.sun.com/jsf/html">
4      <h:head>
5         <title>The multizone application</title>
6      </h:head>
7      <h:body>
8         <h:form>
9              <p>
10               Sorry, no information is available for #{timeZoneBean.cityToAdd}.
11             </p>
12             <p>
13                <h:commandButton value="Back" action="index"/>
14             </p>
15        </h:form>
16     </h:body>
17  </html>
```

section_6/multizone/Web-inf /classes/bigjava/TimeZonebean.java

```
1  package bigjava;
2
3  import java.sql.Connection;
4  import java.sql.PreparedStatement;
5  import java.sql.ResultSet;
6  import java.sql.SQLException;
7  import java.text.DateFormat;
8  import java.util.ArrayList;
9  import java.util.Date;
10 import java.util.Map;
11 import java.util.TimeZone;
12 import java.util.TreeMap;
13 import java.util.logging.Logger;
14 import javax.annotation.Resource;
15 import javax.faces.bean.ManagedBean;
16 import javax.faces.bean.SessionScoped;
17 import javax.sql.DataSource;
18
19 /**
20    This bean formats the local time of day for a given date
21    and city.
22 */
23 @ManagedBean
24 @SessionScoped
25 public class TimeZoneBean
26 {
27    @Resource(name="jdbc/__default")
28    private DataSource source;
29
30    private DateFormat timeFormatter;
31    private ArrayList<String> cities;
32    private String cityToAdd;
33    private String cityToRemove;
34
35    /**
36       Initializes the formatter.
37    */
38    public TimeZoneBean()
39    {
40       timeFormatter = DateFormat.getTimeInstance();
41       cities = new ArrayList<String>();
42    }
43
44    /**
45       Setter for cityToAdd property.
46       @param city the city to add to the list of cities
47    */
48    public void setCityToAdd(String city)
49    {
50       cityToAdd = city;
51    }
52
53    /**
54       Getter for cityToAdd property.
55       @return the city to add to the list of cities
56    */
57    public String getCityToAdd()
58    {
```

```
59          return cityToAdd;
60       }
61
62       /**
63          Setter for the cityToRemove property.
64          @param city the city to remove from the list of cities
65       */
66       public void setCityToRemove(String city)
67       {
68          cityToRemove = city;
69       }
70
71       /**
72          Getter for the cityToRemove property.
73          @return the city to remove from the list of cities
74       */
75       public String getCityToRemove()
76       {
77          return cityToRemove;
78       }
79
80       /**
81          Read-only citiesAndTimes property.
82          @return a map containing the cities and formatted times
83       */
84       public Map<String, String> getCitiesAndTimes()
85       {
86          Date time = new Date();
87          Map<String, String> result = new TreeMap<String, String>();
88          for (int i = 0; i < cities.size(); i++)
89          {
90             String city = cities.get(i);
91             String label = city + ": ";
92             TimeZone zone = getTimeZone(city);
93             if (zone != null)
94             {
95                timeFormatter.setTimeZone(zone);
96                String timeString = timeFormatter.format(time);
97                label = label + timeString;
98             }
99             else
100            {
101               label = label + "unavailable";
102            }
103            result.put(label, city);
104         }
105
106         return result;
107      }
108
109      /**
110         Action for adding a city.
111         @return "next" if time zone information is available for the city,
112         "error" otherwise
113      */
114      public String addCity()
115      {
116         TimeZone zone = getTimeZone(cityToAdd);
117         if (zone == null) { return "error"; }
118         cities.add(cityToAdd);
```

```
119        cityToRemove = cityToAdd;
120        cityToAdd = "";
121        return "next";
122     }
123
124     /**
125        Action for removing a city.
126        @return null if there are more cities to remove, "index" otherwise
127     */
128     public String removeCity()
129     {
130        cities.remove(cityToRemove);
131        if (cities.size() > 0) { return null; }
132        else return "index";
133     }
134
135     /**
136        Looks up the time zone for a city.
137        @param city the city for which to find the time zone
138        @return the time zone or null if no match is found
139     */
140     private TimeZone getTimeZone(String city)
141     {
142        String[] ids = TimeZone.getAvailableIDs();
143        for (int i = 0; i < ids.length; i++)
144        {
145           if (timeZoneIDmatch(ids[i], city))
146           {
147              return TimeZone.getTimeZone(ids[i]);
148           }
149        }
150        try
151        {
152           String id = getZoneNameFromDB(city);
153           if (id != null)
154           {
155              return TimeZone.getTimeZone(id);
156           }
157        }
158        catch (Exception ex)
159        {
160           Logger.global.info("Caught in TimeZone.getTimeZone: "
161                 + ex);
162        }
163        return null;
164     }
165
166     private String getZoneNameFromDB(String city)
167        throws SQLException
168     {
169        if (source == null)
170        {
171           Logger.global.info("No database connection");
172           return null;
173        }
174        Connection conn = source.getConnection();
175        try
176        {
```

```
177             PreparedStatement stat = conn.prepareStatement(
178                   "SELECT Zone FROM CityZone WHERE City=?");
179             stat.setString(1, city);
180             ResultSet result = stat.executeQuery();
181             if (result.next()) { return result.getString(1); }
182             else { return null; }
183          }
184          finally
185          {
186             conn.close();
187          }
188       }
189
190       /**
191          Checks whether a time zone ID matches a city.
192          @param id the time zone ID (e.g., "America/Los_Angeles")
193          @param city the city to match (e.g., "Los Angeles")
194          @return true if the ID and city match
195       */
196       private static boolean timeZoneIDmatch(String id, String city)
197       {
198          String idCity = id.substring(id.indexOf('/') + 1);
199          return idCity.replace('_', ' ').equals(city);
200       }
201    }
```

Self check

12. Why don't we just keep a database connection as an instance variable in the `TimeZoneBean`?
13. Why does the `removeCity` method of the `TimeZoneBean` return `null` or `"index"`, depending on the size of the `cities` instance variable?

practice it Now you can try these exercises at the end of the chapter: R24.12, P24.6, P24.7.

Special Topic 24.2

AJAX

In Section 24.1, you learned that a web application receives an HTTP request from the browser and then sends back an HTML form. The cycle repeats when the user submits the next form data. Web application designers and users dislike the "page flip"—the visual discontinuity between pages that is often accompanied by a significant delay, as the browser waits for the new form tags.

The AJAX (Asynchronous JavaScript and XML) technology, invented in 2005, aims to solve this problem. In an AJAX application, the browser does not merely display an HTML page, but it also executes code written in the JavaScript language. The JavaScript code continuously communicates with the server program and updates parts of the HTML page.

One example of an AJAX application is the Google Maps™ mapping service—see Figure 14. In a traditional map application, the user might click on a "move North" button and then wait until the browser receives the new map image and displays it in a new page. The Google Maps application uses AJAX to fetch only the needed tiles, and it fluidly rearranges the tiles in the current page, without the dreaded page flip.

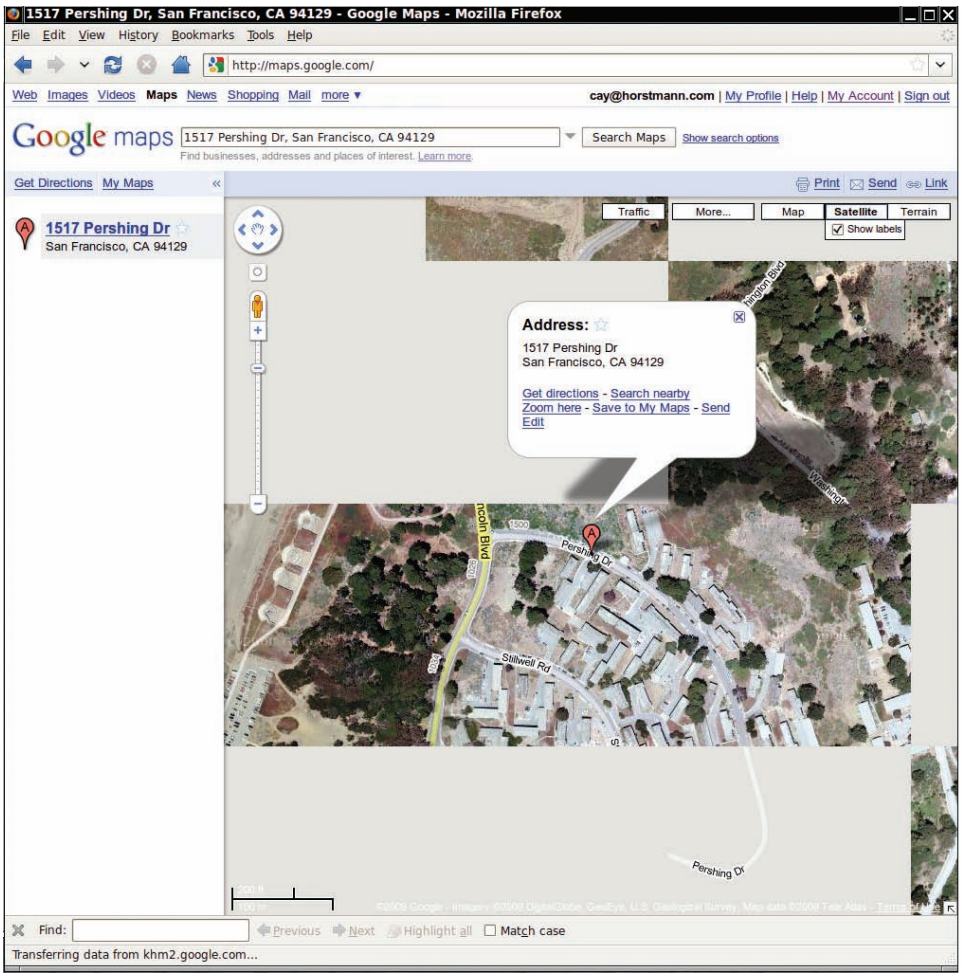


figure 14 A Google Maps Image with Partially-Fetched Tiles

full code example

Go to `wiley.com/go/javacode` to download the multizone application code using AJAX.

AJAX applications are much more difficult to program than regular web applications. Frameworks are being proposed to handle these additional challenges. JSF 2 supports AJAX, giving the web application programmer the benefit of producing a pleasant user experience without having to worry about the intricate details of the JavaScript communication channel. The book's companion code contains a modification of the `multizone` application that uses AJAX. When you click one of the buttons, the page is updated without a page flip.

CHAPTER SUMMARY

Describe the architecture of a web application.

- The user interface of a web application is displayed in a web browser.
- When a form is submitted, the names and values of the form elements are sent to the web server.
- Upon receiving the form data, the web server sends a new web page to the browser.

Describe the architecture of a JSF application.

- A JavaServer Faces (JSF) page contains HTML and JSF tags.
- The JSF container converts a JSF page to an HTML page, replacing all JSF tags with text and HTML tags.
- A managed bean is an object that is controlled by the JSF container.
- A bean with session scope is available for multiple requests by the same browser.
- The JSF technology enables the separation of presentation and business logic.

explain how properties are defined in managed beans and accessed in value expressions.

- Properties of a software component can be accessed without having to write Java code.
- A JavaBean is a class that exposes properties through its `get` and `set` methods.
- In the value expression of an output tag, only the property getter is called.
- In the value expression of an input tag, the property setter is called when the page is submitted.

implement navigation between pages.

- The outcome string of an action determines the next page that the JSF container sends to the browser.
- A method expression specifies a bean and a method that should be invoked on the bean.

use common JSF components for designing a user interface.

- There are JSF components for text input, choices, buttons, and images.
- The value attribute of an input component denotes the value that the user supplies.
- Use an `f:selectItems` tag to specify all choices for a component that allows selection from a list of choices.

Develop applications that use JSF and a database.

- A three-tier application has separate tiers for presentation, business logic, and data storage.
- You define data sources in the JSF container and use resource annotations to initialize them.

STANDARD LIBRARY ITEMS INTRODUCED IN THIS CHAPTER

```
java.text.DateFormat
   format
   getTimeInstance
   setTimeZone
java.util.LinkedHashMap
java.util.TimeZone
   getAvailableIDs
   getTimeZone
javax.sql.DataSource
   getConnection
```

REVIEW QUESTIONS

- **R24.1** Most web browsers have a command to "view the source" of a web page. Load the page `http://horstmann.com` into your browser and view the source. What is the "language" used for formatting the source? What images, links, bullets, and input elements can you find?
- **R24.2** Have a closer look at the HTTP POST request on page W1038. Where is the data that the user provided? What does `login=Log%20in` mean? (The code `%20` denotes a space in the "URL encoding" scheme.)
- **R24.3** What is the difference between a JSF page and a JSF container?
- **R24.4** What is a bean?
- **R24.5** What is a bean property?
- **R24.6** Is a `JButton` a bean? Why or why not?
- **R24.7** What is the software engineering purpose of using beans in conjunction with JSF pages?
- **R24.8** How are variables in the JSF expression language different from variables in Java programs?
- **R24.9** When is a bean constructed in a JSF application? Can you have two different instances of a bean that are active at the same time?
- **R24.10** How can you implement error checking in a JSF application? Explain, using a login page as an example.
- **R24.11** What input elements can you place on a JSF form? What are their Swing equivalents?
- **R24.12** What is the difference between a client-server application and a three-tier application?

PROGRAMMING EXERCISES

- **P24.1** Write a JSF application that reports the values of the following system properties of the web server:
  - The Java version (`java.version`)
  - The operating system name (`os.name`)
  - The operating system version (`os.version`)

  Supply a bean that uses the `getProperties` method of the `System` class.

■ **P24.2** Write a JSF application that simulates two rolls of a die, producing an output such as "Rolled a 4 and a 6". When the user reloads the page, a new pair of values should be displayed. Provide a bean that yields random numbers.

■■ **P24.3** Enhance Exercise P24.2 by producing a web page that shows images of the rolled dice. Find GIF images of dice with numbers 1 through 6 on the front, and generate an HTML page that references the appropriate images. *Hint:* Use the tag `<h:graphicImage value=`*imageURL*`/>` and take advantage of the fact that you can embed a value expression into regular text, such as `"/image#{`*expression*`}.gif"`.

■ **P24.4** Write a web application that allows a user to specify six lottery numbers. Generate your own combination on the server, and then print out the user's and the server's combinations together with a count of matches.

■■ **P24.5** Add error checking to Exercise P24.4. If the lottery numbers are not within the correct range, or if there are duplicates, show an appropriate message and allow the user to fix the error.

■■■ **P24.6** Personalize the time zone application of Section 24.3. Prompt the user to log in and specify a city to be stored in a profile. The next time the user logs in, the time of their favorite city is displayed automatically. Store users, passwords, and favorite cities in a database. You need a logout button to switch users.

■■■ **P24.7** Extend Exercise P24.6 so that a user can choose multiple cities and all cities chosen by the user are remembered on the next login.

■■■ **P24.8** Write a web version of the `ExecSQL` utility of Chapter 22. Allow users to type arbitrary SQL queries into a text area. Then submit the query to the database and display the result.

■■■ **P24.9** Produce a web front end for the ATM program in Worked Example 11.1.

■■■ **P24.10** Produce a web front end for the appointment calendar application of Exercise P11.8.

■■■ **P24.11** Produce a web front end for the airline reservation program of Exercise P11.12.

■■■ **Business P24.12** Write a shopping cart application. A database contains items that can be purchased and their prices, descriptions, and available quantities. If the user wants to check out, ask for the user account. If the user does not yet have an account, create one. The user name and address should be stored with the account in the database. Display an invoice as the last step in the check out process. When the user has confirmed the purchase, update the quantities in the warehouse.

■■■ **P24.13** Write a web-based grade book application that your instructor might use to manage student grades in this course. Your application should have one account for the instructor, and one account for each student. Instructors can enter and view grades for all students. Students can only see their own grades and their ranking within the course. Implement the features that your instructor uses for determining the course grade (such as dropping the lowest quiz score, counting homework as 30 percent of the total grade, and so on.) All information should be stored in a database.

ANSWERS TO SELF-CHECK QUESTIONS

1. Each protocol has a specific purpose. HTML describes the appearance of a page; it would be useless for sending requests from a browser to a server. HTTP describes a request; it cannot describe the appearance of a page.
2. The data of the `POST` request contain a portion `username=`*the name supplied by the user*`&password=`*the password supplied by the user*.
3. Place an image file, say `clock.gif`, into the `time` directory, and add a tag `<img src="clock.gif"/>` to the `index.xhtml` file.
4. No—it is possible (and sadly common) for programmers to place the business logic into the frame and component classes of the user interface.
5. The application server knows nothing about the files on your computer. You need to hand it the WAR file with all the application's pages, code, and configuration files so that it can execute the application when it receives a web request.
6. No. The `Scanner` class does not have a constructor with no arguments.
7. There is no way of knowing without looking at the source code. Perhaps it simply executes a statement `city = newValue`, setting an instance variable of the bean class. But the method may also do other work, for example checking whether the city name is valid or storing the name in a database.
8. Add the tag `<h:commandButton value="Help" action="help"/>` to `error.xhtml`.
9. The current page would be redisplayed.
10. `h:selectOneRadio`, `h:selectOneMenu`, or `h:selectOneCheckbox`
11. You would need a bean with a property such as the following:

```
public Map<String, Integer> getYearChoices()
{
   Map<String, Integer> choices =
         new TreeMap<String, Integer>();
   choices.put("2003", 2003);
   choices.put("2004", 2004);
   . . .
   return choices;
}
```

   Then supply a tag `<f:selectItems value="#{creditCard.yearChoices}"/>`.
12. Then the database connection would be kept open for the entire session.
13. As long as there are cities, the same page (`next.xhtml`) page is redisplayed. If all cities are removed, it is pointless to display the `next.xhtml` page, so the application navigates to the `index.xhtml` page.