If any man will draw up his case, and put his name at the foot of the first page, I will give him an immediate reply. Where he compels me to turn over the sheet, he must wait my leisure.

—Lord Sandwich

Rule One: Our client is always right
Rule Two: If you think our client is wrong, see Rule One.

—Anonymous

A fair question should be followed by a deed in silence.

—Dante Alighieri

You will come here and get books that will open your eyes, and your ears, and your curiosity, and turn you inside out or outside in.

—Ralph Waldo Emerson

**OBJECTIVES**

In this chapter you will learn:

- Web application development using Java Technologies and Java Studio Creator 2.0.
- To create JavaServer Pages with JavaServer Faces components.
- To create web applications consisting of multiple pages.
- To validate user input on a web page.
- To maintain state information about a user with session tracking and cookies.
26.1 Introduction

In this chapter, we introduce web application development with Java-based technology. Web-based applications create web content for web browser clients. This web content includes Extensible HyperText Markup Language (XHTML), client-side scripting, images and binary data. For those who are not familiar with XHTML, we’ve provided on the CD that accompanies this book three chapters from our book Internet & World Wide Web How to Program, 3/e—Introduction to XHTML: Part 1, Introduction to XHTML: Part 2 and Cascading Style Sheets (CSS). In Chapters 26–28, we assume you already know XHTML.

This chapter begins with an overview of multitier application architecture and Java’s web technologies for implementing multitier applications. We then present several examples that demonstrate web application development. The first example introduces you to Java web development. In the second example, we build a web application that simply shows the look-and-feel of several web-application GUI components. Next, we demonstrate how to use validation components and custom validation methods to ensure that user input is valid before it is submitted for processing on the server. The chapter finishes with two examples of customizing a user’s experience with session tracking.
26.2 Simple HTTP Transactions

In Chapter 27, we continue our discussion of web application development with more advanced concepts, including the AJAX-enabled components from Sun’s Java BluePrints. AJAX helps web-based applications provide the interactivity and responsiveness that users typically expect of desktop applications.

Throughout this chapter and Chapter 27, we use Sun Java Studio Creator 2.0—an IDE that helps you build web applications using Java technologies such as JavaServer Pages and JavaServer Faces. To implement the examples presented in this chapter, you must install Java Studio Creator 2.0, which is available for download at developers.sun.com/prodtech/javatools/jscreator/downloads/index.jsp. The features of Java Studio Creator 2.0 are being incorporated into Netbeans 5.5 via an add-on called the Netbeans Visual Web Pack 5.5 (www.netbeans.org/products/visualweb/).

26.2 Simple HTTP Transactions

Web application development requires a basic understanding of networking and the World Wide Web. In this section, we discuss the Hypertext Transfer Protocol (HTTP) and what occurs behind the scenes when a user requests a web page in a browser. HTTP specifies a set of methods and headers that allow clients and servers to interact and exchange information in a uniform and reliable manner.

In its simplest form, a web page is nothing more than an XHTML document—a plain text file containing markup (i.e., tags) that describe to a web browser how to display and format the document’s information. For example, the XHTML markup

```
<title>My Web Page</title>
```

indicates that the browser should display the text between the `<title>` start tag and the `</title>` end tag in the browser’s title bar. XHTML documents also can contain hyper-text data (usually called hyperlinks) that link to different pages or to other parts of the same page. When the user activates a hyperlink (usually by clicking it with the mouse), the requested web page loads into the user’s web browser.

HTTP uses URIs (Uniform Resource Identifiers) to identify data on the Internet. URLs that specify document locations are called URLs (Uniform Resource Locators). Common URLs refer to files, directories or objects that perform complex tasks, such as database lookups and Internet searches. If you know the HTTP URL of a publicly available XHTML document anywhere on the web, you can access it through HTTP.

A URL contains information that directs a browser to the resource that the user wishes to access. Computers that run web server software make such resources available. Let’s examine the components of the URL

```
http://www.deitel.com/books/downloads.html
```

The `http://` indicates that the resource is to be obtained using the HTTP protocol. The middle portion, `www.deitel.com`, is the server’s fully qualified hostname—the name of the server on which the resource resides. This computer usually is referred to as the host, because it houses and maintains resources. The hostname `www.deitel.com` is translated into an IP address (68.236.123.125), which identifies the server in a manner similar to how a telephone number uniquely defines a particular phone line. This translation is performed by a domain name system (DNS) server—a computer that maintains a database of hostnames and their corresponding IP addresses—and the process is called a DNS lookup.
The remainder of the URL (i.e., /books/downloads.html) specifies both the name of the requested resource (the XHTML document downloads.html) and its path, or location (/books), on the web server. The path could specify the location of an actual directory on the web server’s file system. However, for security reasons, the path normally specifies the location of a virtual directory. The server translates the virtual directory into a real location on the server (or on another computer on the server’s network), thus hiding the true location of the resource. Some resources are created dynamically and do not reside anywhere on the server. The hostname in the URL for such a resource specifies the correct server; the path and resource information identify the location of the resource with which to respond to the client’s request.

When given a URL, a web browser performs a simple HTTP transaction to retrieve and display the web page found at that address. Figure 26.1 illustrates the transaction in detail, showing the interaction between the web browser (the client side) and the web server application (the server side).

In Fig. 26.1, the web browser sends an HTTP request to the server. The request (in its simplest form) is

GET /books/downloads.html HTTP/1.1

The word GET is an HTTP method indicating that the client wishes to obtain a resource from the server. The remainder of the request provides the path name of the resource (an XHTML document) and the protocol’s name and version number (HTTP/1.1).

Any server that understands HTTP (version 1.1) can translate this request and respond appropriately. Figure 26.2 depicts the results of a successful request. The server first responds by sending a line of text that indicates the HTTP version, followed by a numeric code and a phrase describing the status of the transaction. For example,

HTTP/1.1 200 OK

indicates success, whereas

HTTP/1.1 404 Not found

informs the client that the web server could not locate the requested resource. A complete list of numeric codes indicating the status of an HTTP transaction can be found at www.w3.org/Protocols/HTTP/HTRESP.html.
The server then sends one or more HTTP headers, which provide additional information about the data that will be sent. In this case, the server is sending an XHTML text document, so the HTTP header for this example reads:

```
Content-type: text/html
```

The information provided in this header specifies the Multipurpose Internet Mail Extensions (MIME) type of the content that the server is transmitting to the browser. MIME is an Internet standard that specifies data formats so that programs can interpret data correctly. For example, the MIME type `text/plain` indicates that the sent information is text that can be displayed directly, without any interpretation of the content as XHTML markup. Similarly, the MIME type `image/jpeg` indicates that the content is a JPEG image. When the browser receives this MIME type, it attempts to display the image.

The header or set of headers is followed by a blank line, which indicates to the client that the server is finished sending HTTP headers. The server then sends the contents of the requested XHTML document (`downloads.html`). The server terminates the connection when the resource transfer is complete. The client-side browser parses the XHTML markup it receives and renders (or displays) the results.

### 26.3 Multitier Application Architecture

Web-based applications are multitier applications (sometimes referred to as n-tier applications) that divide functionality into separate tiers (i.e., logical groupings of functionality). Although tiers can be located on the same computer, the tiers of web-based applications typically reside on separate computers. Figure 26.3 presents the basic structure of a three-tier web-based application.

The bottom tier (also called the data tier or the information tier) maintains the application's data. This tier typically stores data in a relational database management system (RDBMS). We discussed RDBMSs in Chapter 25. For example, a retail store might have an inventory information database containing product descriptions, prices and quantities in stock. The same database also might contain customer information, such as user names, billing addresses and credit card numbers. There could be multiple databases residing on one or more computers, which together comprise the application's data.
The middle tier implements business logic, controller logic and presentation logic to control interactions between the application’s clients and its data. The middle tier acts as an intermediary between data in the information tier and the application’s clients. The middle-tier controller logic processes client requests (such as requests to view a product catalog) and retrieves data from the database. The middle-tier presentation logic then processes data from the information tier and presents the content to the client. Web applications typically present data to clients as XHTML documents.

Business logic in the middle tier enforces business rules and ensures that data is reliable before the server application updates the database or presents the data to users. Business rules dictate how clients can and cannot access application data, and how applications process data. For example, a business rule in the middle tier of a retail store’s web-based application might ensure that all product quantities remain positive. A client request to set a negative quantity in the bottom tier’s product information database would be rejected by the middle tier’s business logic.

The top tier, or client tier, is the application’s user interface, which gathers input and displays output. Users interact directly with the application through the user interface, which is typically a web browser, keyboard and mouse. In response to user actions (e.g., clicking a hyperlink), the client tier interacts with the middle tier to make requests and to retrieve data from the information tier. The client tier then displays the data retrieved for the user. The client tier never directly interacts with the information tier.

Java multitier applications are typically implemented using the features of Java Enterprise Edition (Java EE). The technologies we use to develop web applications in Chapters 26–28 are part of Java EE 5 (java.sun.com/javaee).

### 26.4 Java Web Technologies

Java web technologies continually evolve to provide developers with higher levels of abstraction and greater separation of the application’s tiers. This separation makes web applications more maintainable and extensible. It also allows for an effective division of labor. A graphic designer can build the application’s user interface without concern for the underlying page logic, which will be handled by a programmer. Meanwhile, the programmer is free to focus on the application’s business logic, leaving the details of building an attractive and easy-to-use application to the designer. Java Studio Creator 2 is the latest step in this evolution, allowing you to develop a web application’s GUI in a drag-and-drop design tool, while handling the business logic in separate Java classes.
26.4 Java Web Technologies

26.4.1 Servlets
Servlets are the lowest-level view of web development technologies in Java that we will discuss in this chapter. They use the HTTP request-response model of communication between client and server.

Servlets extend a server’s functionality by allowing the server to generate dynamic content. For instance, servlets can dynamically generate custom XHTML documents, help provide secure access to a website, interact with databases on behalf of a client and maintain unique session information for each client. A web server component called the servlet container executes and interacts with servlets. Packages `javax.servlet` and `javax.servlet.http` provide the classes and interfaces to define servlets. The servlet container receives HTTP requests from a client and directs each request to the appropriate servlet. The servlet processes the request and returns an appropriate response to the client—usually in the form of an XHTML or XML (Extensible Markup Language) document to display in the browser. XML is a language used to exchange structured data on the web.

Architecturally, all servlets must implement the `Servlet` interface of package `javax.servlet`, which ensures that each servlet can execute in the framework provided by the servlet container. Interface `Servlet` declares methods used by the servlet container to manage the servlet’s life cycle. A servlet’s life cycle begins when the servlet container loads it into memory—usually in response to the first request for the servlet. Before the servlet can handle that request, the container invokes the servlet’s `init` method, which is called only once during a servlet’s life-cycle to initialize the servlet. After `init` completes execution, the servlet is ready to respond to its first request. All requests are handled by a servlet’s `service` method, which is the key method in defining a servlet’s functionality. The `service` method receives the request, processes it and sends a response to the client. During a servlet’s life cycle, `service` is called once per request. Each new request is typically handled in a separate thread of execution (managed by the servlet container), so each servlet must be thread safe. When the servlet container terminates the servlet (e.g. when the servlet container needs more memory or when it is shut down), the servlet’s `destroy` method is called to release any resources held by the servlet.

26.4.2 JavaServer Pages
JavaServer Pages (JSP) technology is an extension of servlet technology. Each JSP is translated by the JSP container into a servlet. Unlike servlets, JSPs help you separate presentation from content. JavaServer Pages enable web application programmers to create dynamic content by reusing predefined components and by interacting with components using server-side scripting. JSP programmers can use special software components called JavaBeans and custom tag libraries that encapsulate complex, dynamic functionality. A JavaBean is a reusable component that follows certain conventions for class design. For example, JavaBeans classes that allow reading and writing of instance variables must provide appropriate `get` and `set` methods. The complete set of class design conventions is discussed in the JavaBeans specification (`java.sun.com/products/javabeans/glasgow/index.html`).

Custom Tag Libraries
Custom tag libraries are a powerful feature of JSP that allows Java developers to hide code for database access and other complex operations in custom tags. To use such capabilities,
you simply add the custom tags to the page. This simplicity enables web-page designers who are not familiar with Java to enhance web pages with powerful dynamic content and processing capabilities. The JSP classes and interfaces are located in packages `javax.servlet.jsp` and `javax.servlet.jsp.tagext`.

**JSP Components**

There are four key components to JSPs—directives, actions, scripting elements and tag libraries. **Directives** are messages to the JSP container—the web server component that executes JSPs. Directives enable you to specify page settings, to include content from other resources and to specify custom tag libraries for use in JSPs. **Actions** encapsulate functionality in predefined tags that programmers can embed in JSPs. Actions often are performed based on the information sent to the server as part of a particular client request. They also can create Java objects for use in JSPs. **Scripting elements** enable you to insert Java code that interacts with components in a JSP (and possibly other web application components) to perform request processing. **Tag libraries** are part of the tag extension mechanism that enables programmers to create custom tags. Such tags enable web-page designers to manipulate JSP content without prior Java knowledge. The JavaServer Pages Standard Tag Library (JSTL) provides the functionality for many common web application tasks, such as iterating over a collection of objects and executing SQL statements.

**Static Content**

JSPs can contain other static content. For example JSPs normally include XHTML or XML markup. Such markup is known as fixed-template data or fixed-template text. Any literal text in a JSP is translated to a String literal in the servlet representation of the JSP.

**Processing a JSP Request**

When a JSP-enabled server receives the first request for a JSP, the JSP container translates the JSP into a servlet that handles the current request and future requests to the JSP. JSPs thus rely on the same request-response mechanism as servlets to process requests from and send responses to clients.

**Performance Tip 26.1**

Some JSP containers translate JSPs into servlets at the JSP’s deployment time (i.e., when the application is placed on a web server). This eliminates the translation overhead for the first client that requests each JSP, as the JSP will be translated before it is ever requested by a client.

**26.4.3 JavaServer Faces**

JavaServer Faces (JSF) is a web application framework that simplifies the design of an application’s user interface and further separates a web application’s presentation from its business logic. A framework simplifies application development by providing libraries and sometimes software tools to help you organize and build your applications. Though the JSF framework can use many technologies to define the pages in web applications, this chapter focuses on JSF applications that use JavaServer Pages. JSF provides a set of user interface components, or JSF components that simplify web-page design. These components are similar to the Swing components used to build GUI applications. JSF provides two JSP custom tag libraries for adding these components to a JSP page. JSF also includes APIs for handling component events (such as processing component state changes and val-
idating user input), navigating between web application pages and more. You design the look-and-feel of a page with JSF by adding tags to a JSP file and manipulating their attributes. You define the page’s behavior separately in a related Java source-code file.

Though the standard JSF components are sufficient for most basic web applications, you can also write custom component libraries. Additional component libraries are available through the Java BluePrints project—which shows best practices for developing Java applications. Many other vendors provide JSF component libraries. For example, Oracle provides almost 100 components in its ADF Faces library. We discuss one such component library, the BluePrints AJAX components library (blueprints.dev.java.net/ajax-components.html). We discuss the Java BluePrints components for building AJAX-enabled JSF applications in the next chapter.

26.4.4 Web Technologies in Java Studio Creator 2

Java Studio Creator 2 web applications consist of one or more JSP web pages built in the JavaServer Faces framework. These JSP files have the file-name extension .jsp and contain the web page’s GUI elements. The JSPs can also contain JavaScript to add functionality to the page. JSPs can be customized in Java Studio Creator 2 by adding JSF components, including labels, text fields, images, buttons and other GUI components. The IDE allows you to design pages visually by dragging and dropping these components onto a page; you can also customize a web page by editing the .jsp file manually.

Every JSP file created in Java Studio Creator 2 represents a web page and has a corresponding JavaBean class called the page bean. A JavaBean class must have a default (or no-argument) constructor, and get and set methods for all of the bean’s properties (i.e., instance variables). The page bean defines properties for each of the page’s elements. The page bean also contains event handlers and page life-cycle methods for managing tasks such as page initialization and rendering, and other supporting code for the web application.

Every web application built with Java Studio Creator 2 has three other JavaBeans. The RequestBean object is maintained in request scope—this object exists only for the duration of an HTTP request. A SessionBean object has session scope—the object exists throughout a user’s browsing session or until the session times out. There is a unique SessionBean object for each user. Finally, the ApplicationBean object has application scope—this object is shared by all instances of an application and exists as long as the application remains deployed on a web server. This object is used for application-wide data storage or processing; only one instance exists for the application, regardless of the number of open sessions.

26.5 Creating and Running a Simple Application in Java Studio Creator 2

Our first example displays the web server’s time of day in a browser window. When run, this program displays the text “Current Time on the Web Server”, followed by the web server’s time. The application contains a single web page and, as mentioned previously, consists of two related files—a JSP file (Fig. 26.4) and a supporting page bean file (Fig. 26.6). The application also has the three scoped data beans for request, session, and application scopes. Since this application does not store data, these beans are not used in...
Chapter 26  Web Applications: Part I

this example. We first discuss the markup in the JSP file, the code in the page bean file and the application output, then we provide step-by-step instructions for creating the program. [Note: The markup in Fig. 26.4 and other JSP file listings in this chapter is the same as the markup that appears in Java Studio Creator 2, but we have reformatted these listings for presentation purposes to make the code more readable.]

Java Studio Creator 2 generates all the markup shown in Fig. 26.4 when you set the web page’s title, drag two Static Text components onto the page and set the properties of the Static Text components. Static Text components display text that cannot be edited by the user. We show these steps shortly.

```xml
<?xml version = "1.0" encoding = "UTF-8"?>
<jsp:directive.page contentType = "text/html; charset = UTF-8"
pageEncoding = "UTF-8"/>
<f:view>
  <ui:page binding = "#{Time.page}" id = "page">
    <ui:html binding = "#{Time.html}" id = "html">
      <ui:head binding = "#{Time.head}" id = "head">
        <ui:title binding = "#{Time.html}" id = "title">
          "Web Time: A Simple Example"
        </ui:title>
        <ui:link binding = "#{Time.link}" id = "link">
          url = "/resources/stylesheet.css"/
        </ui:link>
      </ui:head>
      <ui:body binding = "#{Time.body}" id = "body">
        style = "-rave-layout: grid">
          <ui:form binding = "#{Time.form}" id = "form">
            <ui:staticText binding = "#{Time.timeHeader}" id = "timeHeader">
              "Current time on the Web Server:"/
            </ui:staticText>
            <ui:staticText binding = "#{Time.clockText}" id = "clockText">
              "Current time on the Web Server:"/
            </ui:staticText>
          </ui:form>
        </ui:body>
      </ui:head>
    </ui:html>
  </ui:page>
</f:view>
</jsp:root>
```

Fig. 26.4  JSP file generated by Java Studio Creator 2 that displays the current time on the web server.
26.5.1 Examining a JSP File

The JSP files used in this and the following examples are generated almost entirely by Java Studio Creator 2, which provides a Visual Editor that allows you to build a page’s GUI by dragging and dropping components onto a design area. The IDE generates a JSP file in response to your interactions. Line 1 of Fig. 26.4 is the XML declaration, indicating that the JSP is expressed in XML syntax and the version of XML that is used. Lines 3–5 are comments that we added to the JSP to indicate its figure number, file name and purpose.

Line 6 begins the root element for the JSP. All JSPs must have this `jsp:root` element, which has a `version` attribute to indicate the version of JSP being used (line 6) and one or more `xmlns` attributes (lines 7–10). Each `xmlns` attribute specifies a prefix and a URL for a tag library, allowing the page to use tags specified in that library. For example, line 9 allows the page to use the standard JSP elements. To use these elements, each element’s tag must be preceded by the `jsp` prefix. All JSPs generated by Java Studio Creator 2 include the tag libraries specified in lines 7–10 (the JSF core components library, the JSF HTML components library, the JSP standard components library and the JSF user interface components library).

Lines 11–12 are the `jsp:directive.page` element. Its `contentType` attribute specifies the MIME type (`text/html`) and the character set (`UTF-8`) the page uses. The `pageEncoding` attribute specifies the character encoding used by the page source. These attributes help the client (typically a web browser) determine how to render the content.

All pages containing JSF components are represented in a component tree (Fig. 26.5) with the root JSF element `f:view`, which is of type `UIViewRoot`. This component tree

![Diagram of JSF component tree](image-url)
structure is represented in a JSP by enclosing all JSF component tags inside the \texttt{f:view} element (lines \(13-37\)). Lines \(14-20\) begin the definition of the JSP with the \texttt{ui:page}, \texttt{ui:html}, and \texttt{ui:head} tags, all from the ui (JSF user interface components) tag library. These, and many other ui page elements, have a binding attribute. For example, the \texttt{ui:head} element (line \(16\)) has the attribute \texttt{binding = \#\{Time.head\}}. This attribute uses JSF Expression Language notation (i.e., \#\{Time.head\}) to reference the head property in the Time class that represents the page bean (you’ll see this class in Fig. 26.6). It is possible to bind a single attribute of a JSP element to a property in any of the web application’s JavaBeans. For instance, the text attribute of a \texttt{ui:label} component can be bound to a String property in the application’s SessionBean. We will see an example of this in Section 26.7.2.

The \texttt{ui:head} element (lines \(16-20\)) has a \texttt{title} attribute that specifies the page’s title. This element also contains a \texttt{ui:link} element (lines \(18-19\)) that specifies the CSS stylesheets used by the page. The \texttt{ui:body} element (lines \(22-34\)) contains a \texttt{ui:form} element (lines \(24-33\)), which contains two \texttt{ui:staticText} components (lines \(25-28\) and \(29-32\)). These components display the page’s text. The \texttt{timeHeader} component (lines \(25-28\)) has a \texttt{text} attribute (lines \(27-28\)) that specifies the text to display (i.e., “Current Time on the Web Server”). The \texttt{clockText} component (lines \(29-32\)) does not specify a \texttt{text} attribute because this component’s text will be set programmatically.

For the markup in this file to be displayed in a web browser, all of the JSP’s elements are automatically mapped to XHTML elements that the browser recognizes. The same web component can map to different XHTML elements, depending on the client browser and the component’s property settings. In this example, the \texttt{ui:staticText} components (lines \(25-28, 29-32\)) map to XHTML \texttt{span} elements. A \texttt{span} element contains text that is displayed on a web page and is typically used to control the formatting of the text. The \texttt{style} attributes of a JSP’s \texttt{ui:staticText} element will be represented as part of the corresponding \texttt{span} element’s \texttt{style} attribute when the browser renders the page. We show momentarily the XHTML document that results when \texttt{Time.jsp} is requested by a browser.

### 26.5.2 Examining a Page Bean File

Figure 26.6 presents the page bean file. Line 3 indicates that this class belongs to package \texttt{webtime}. This line is autogenerated and specifies the project’s name as the package name. Line 17 begins class Time’s declaration and indicates that it inherits from class \texttt{AbstractPageBean} (from package \texttt{com.sun.rave.web.ui.appbase}). All page bean classes that support JSP files with JSF components must inherit from the abstract class \texttt{AbstractPageBean}, which provides page life-cycle methods. Note that the IDE makes the class name match the page name. Package \texttt{com.sun.rave.web.ui.component} includes classes for many of the basic JSF components (see the \texttt{import} statements at lines \(6-11\) and \(13\)).

```java
// Fig. 26.6: Time.java
// Page bean file that sets clockText to the time on the web server.
package webtime;

import com.sun.rave.web.ui.component.Body;

Fig. 26.6 | Page bean file that sets clockText to the time on the web server. (Part 1 of 5.)
```
import com.sun.rave.web.ui.component.Form;
import com.sun.rave.web.ui.component.Head;
import com.sun.rave.web.ui.component.Html;
import com.sun.rave.web.ui.component.Link;
import com.sun.rave.web.ui.component.Page;
import javax.faces.FacesException;
import com.sun.rave.web.ui.component.StaticText;
import java.text.DateFormat;
import java.util.Date;

public class Time extends AbstractPageBean
{
    private int __placeholder;

    // auto-generated component initialization method.
    private void _init() throws Exception
    {
        // empty body
        } // end method _init

    private Page page = new Page();

    public Page getPage()
    {
        return page;
    } // end method getPage

    public void setPage( Page page )
    {
        this.page = page;
    } // end method setPage

    private Html html = new Html();

    public Html getHtml )
    {
        return html;
    } // end method getHtml

    public void setHtml( Html html )
    {
        this.html = html;
    } // end method setHtml

    private Head head = new Head();

    public Head getHead()
    {
        return head;
    } // end method getHead

    private void setPage( Page page )
    {
        this.page = page;
    } // end method setPage

    private Head head = new Head();

    public Head getHead()
    {
        return head;
    } // end method getHead

    public void setHead( Head head )
    {

Fig. 26.6 | Page bean file that sets clockText to the time on the web server. (Part 2 of 5.)
this.head = head;
}
}

private Link link = new Link();

public Link getLink()
{
    return link;
}

public void setLink( Link link )
{
    this.link = link;
}

private Body body = new Body();

public Body getBody()
{
    return body;
}

public void setBody( Body body )
{
    this.body = body;
}

private Form form = new Form();

public Form getForm()
{
    return form;
}

public void setForm( Form form )
{
    this.form = form;
}

private StaticText timeHeader = new StaticText();

public StaticText getTimeHeader()
{
    return timeHeader;
}

public void setTimeHeader( StaticText st )
{
    this.timeHeader = st;
}

private StaticText clockText = new StaticText();

Fig. 26.6 | Page bean file that sets clockText to the time on the web server. (Part 3 of 5.)
26.5 Creating and Running a Simple Application in Java Studio Creator 2

```java
public StaticText getClockText()
{
    return clockText;
} // end method getClockText

public void setClockText( StaticText st )
{
    this.clockText = st;
} // end method setClockText

// Construct a new page bean instance.
public Time()
{
    // empty constructor
} // end constructor

// Return a reference to the scoped data bean.
protected RequestBean getRequestBean()
{
    return (RequestBean) getBean( "RequestBean" );
} // end method getRequestBean

// Return a reference to the scoped data bean.
protected ApplicationBean getApplicationBean()
{
    return (ApplicationBean) getBean( "ApplicationBean" );
} // end method getApplicationBean

// Return a reference to the scoped data bean.
protected SessionBean getSessionBean()
{
    return (SessionBean) getBean( "SessionBean" );
} // end method getSessionBean

// initializes page content
public void init()
{
    super.init();
    try
    {
        _init();
    } // end try
    catch ( Exception e )
    {
        log( "Time Initialization Failure", e );
        throw e instanceof FacesException ? ( FacesException ) e:
            new FacesException( e );
} // end catch
} // end method init

// method called when a postback occurs.
public void preprocess()
{

Fig. 26.6 | Page bean file that sets clockText to the time on the web server. (Part 4 of 5.)
```
This page bean file provides `get` and `set` methods for every element of the JSP file of Fig. 26.4. These methods are generated automatically by the IDE. We included the complete page bean file in this first example, but in future examples these properties and their `get` and `set` methods will be omitted to save space. Lines 99–109 and 111–121 of the page bean file define the two `Static Text` components that we dropped onto the page and their `get` and `set` methods. These components are objects of class `StaticText` in package `com.sun.rave.web.ui.component`.

The only logic required in this page is to set the `clockText` component's text to read the current time on the server. We do this in the `prerender` method (lines 170–174). The meaning of this and other page bean methods will be discussed shortly. Lines 172–173 fetch and format the time on the server and set the value of `clockText` to that time.

### 26.5.3 Event-Processing Life Cycle

Java Studio Creator 2's application model places several methods in the page bean that tie into the JSF event-processing life cycle. These methods represent four major stages—initialization, preprocessing, prerendering and destruction. Each corresponds to a method in the page bean class—`init`, `preprocess`, `prerender` and `destroy`, respectively. Java Studio Creator 2 automatically creates these methods, but you can customize them to handle lifecycle processing tasks, such as rendering an element on a page only if a user clicks a button.

```java
// empty body
}

public void prerender()
{
    clockText.setValue( DateFormat.getTimeInstance( 
        DateFormat.LONG ).format( new Date() ) );
}

// method called after rendering completes, if init was called.

public void destroy()
{
    // empty body
}
} // end class Time
```

**Fig. 26.6** | Page bean file that sets `clockText` to the time on the web server. (Part 5 of 5.)
26.5 Creating and Running a Simple Application in Java Studio Creator 2

The `init` method (Fig. 26.6, lines 148–161) is called by the JSP container the first time the page is requested and on postbacks. A `postback` occurs when form data is submitted, and the page and its contents are sent to the server to be processed. Method `init` invokes its superclass version (line 150) then tries to call the method `_init` (declared in lines 22–25). The `_init` method is also automatically generated and handles component initialization tasks (if there are any), such as setting the options for a group of radio buttons.

The `preprocess` method (lines 164–167) is called after `init`, but only if the page is processing a postback. The `prerender` method (lines 170–174) is called just before a page is rendered (i.e., displayed) by the browser. This method should be used to set component properties; properties that are set sooner (such as in method `init`) may be overwritten before the page is actually rendered by the browser. For this reason, we set the value of `clockText` in the `prerender` method.

Finally, the `destroy` method (lines 177–180) is called after the page has been rendered, but only if the `init` method was called. This method handles tasks such as freeing resources used to render the page.

26.5.4 Relationship Between the JSP and Page Bean Files

The page bean has a property for every element that appears in the JSP file of Fig. 26.4, from the `html` element to the two `Static Text` components. Recall that the elements in the JSP file were explicitly bound to these properties by each element’s `binding` attribute using a JSF Expression Language statement. Because this is a JavaBean class, `get` and `set` methods for each of these properties are also included (lines 27–121). This code is automatically generated by the IDE for every web application project.

26.5.5 Examining the XHTML Generated by a Java Web Application

Figure 26.7 shows the XHTML generated when `Time.jsp` (Fig. 26.4) is requested by a client web browser. To view this XHTML, select `View > Source` in Internet Explorer. [Note: We added the XHTML comments in lines 3–4 and reformatted the XHTML to conform to our coding conventions.]

The XHTML document in Fig. 26.7 is similar in structure to the JSP file of Fig. 26.4. Lines 5–6 are the document type declaration, which declares this document to be an XHTML 1.0 Transitional document. The `ui:meta` tags in lines 9–13 are equivalent to HTTP headers and are used to control browser behavior.

```
<?xml version = "1.0"?
<!-- Fig. 26.7: Time.html -->
<!-- The XHTML response generated when the browser requests Time.jsp. -->
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns = "http://www.w3.org/1999/xhtml">
<head>
  <meta content = "no-cache" http-equiv = "Pragma"/>
  <meta content = "no-cache" http-equiv = "Cache-Control"/>
  <meta content = "no-store" http-equiv = "Cache-Control"/>
</head>
```

Fig. 26.7 | XHTML response generated when the browser requests Time.jsp. (Part 1 of 2.)
Lines 30–43 define the body of the document. Line 31 begins the form, a mechanism for collecting user information and sending it to the web server. In this particular program, the user does not submit data to the web server for processing; however, processing user data is a crucial part of many web applications that is facilitated by forms. We demonstrate how to submit data to the server in later examples.

XHTML forms can contain visual and nonvisual components. Visual components include clickable buttons and other GUI components with which users interact. Nonvisual components, called hidden form elements, store data, such as e-mail addresses, that the document author specifies. One of these hidden inputs is defined in lines 40–41. We discuss the precise meaning of this hidden input later in the chapter. Attribute method of the form element (line 31) specifies the method by which the web browser submits the form to the server. By default, JSPs use the post method. The two most common HTTP request types (also known as request methods) are get and post. A get request gets (or retrieves) information from a server. Such requests often retrieve an HTML document or an image. A post request posts (or sends) data to a server, such as authentication informa-
tion or data from a form that gathers user input. Usually, post requests are used to post a message to a news group or a discussion forum, pass user input to a data-handling process on the server and store or update the data on a server. The form's action attribute (line 32) identifies the resource that will be requested when this form is submitted—in this case, /WebTime/faces/Time.jsp.

Note that the two Static Text components (i.e., timeHeader and clockText) are represented by two span elements in the XHTML document (lines 34–36, 37–39) as previously discussed. The formatting options that were specified as properties of timeHeader and clockText, such as the font size and text color in the components, are now specified in each span element's style attribute.

26.5.6 Building a Web Application in Java Studio Creator 2

Now that we have presented the JSP file, the page bean file and the resulting XHTML web page sent to the web browser, we discuss the steps to create this application. To build the WebTime application, perform the following steps in Java Studio Creator 2:

**Step 1: Creating the Web Application Project**

Select File > New Project... to display the New Project dialog. In this dialog, select Web in the Categories pane, JSF Web Application in the Projects pane and click Next. Change the project name to WebTime and use the default project location and Java package. These settings will create a WebTime directory in your My Documents\Creator\Projects directory to store the project's files. Click Finish to create the web application project.

**Step 2: Examining the Visual Editor Window of the New Project**

The next several figures describe important features of the IDE, beginning with the Visual Editor window (Fig. 26.8). Java Studio Creator 2 creates a single web page named Page1 when a new project is created. This page is open by default in the Visual Editor in Design mode when the project first loads. As you drag and drop new components onto the page,
Design mode allows you to see how your page will be rendered in the browser. The JSP file for this page, named `Page1.jsp`, can be viewed by clicking the JSP button at the top of the Visual Editor or by right clicking anywhere in the Visual Editor and selecting Edit JSP Source. As mentioned previously, each web page is supported by a page bean file. Java Studio Creator 2 creates a file named `Page1.java` when a new project is created. To open this file, click the Java button at the top of the Visual Editor or right click anywhere in the Visual Editor and select Edit Page1 Java Source.

The Preview in Browser button at the top of the Visual Editor window allows you to view your pages in a browser without having to build and run the application. The Refresh button redraws the page in the Visual Editor. The Show Virtual Forms button allows you to see which form elements are participating in virtual forms (we discuss this concept in Chapter 27). The Target Browser Size drop-down list lets you specify the optimal browser resolution for viewing the page and lets you see what the page will look like in different screen resolutions.

**Step 3: Examining the Palette in Java Studio Creator 2**

Figure 26.9 shows the Palette displayed in the IDE when the project loads. Part (a) displays the beginning of the Basic list of web components, and part (b) displays the remaining Basic components, as well as the list of Layout components. We discuss specific components in Fig. 26.9 as they are used throughout the chapter.

**Step 4: Examining the Projects Window**

Figure 26.10 displays the Projects window, which appears in the lower-right corner of the IDE. This window displays the hierarchy of all files included in the project. The JSP files for each page are listed under the Web Pages node. This node also includes the resources
folder, which contains the CSS stylesheet for the project and any other files the pages may need to display properly, such as image files. All of the Java source code, including the page bean file for each web page and the application, session and request scope beans, can be found under the Source Packages node. Another useful file displayed in the project window is the Page Navigation file, which defines rules for navigating the project’s pages based on the outcome of some user-initiated event, such as clicking a button or a link. The Page Navigation file can also be accessed by right clicking in the Visual Editor while in Design mode and selecting Page Navigation.

**Step 5: Examining the JSP and Java Files in the IDE**

Figure 26.11 displays Page1.jsp—the JSP file generated by Java Studio Creator 2 for Page1. [Note: We reformatted the code to match our coding conventions.] Click the JSP button at the top of the Visual Editor to open the JSP file. When it is first created, this file contains some tags for setting up the page, including linking to the page’s stylesheet and defining the necessary JSF libraries. Otherwise, the JSP file’s tags are empty, as no components have been added to the page yet.

Figure 26.12 displays part of Page1.java—the page bean file generated by Java Studio Creator 2 for Page1. Click the Java button at the top of the Visual Editor to open the page bean file. This file contains a Java class with the same name as the page (i.e., Page1), which extends the class AbstractPageBean. As previously mentioned, AbstractPageBean has several methods that manage the page’s life cycle. Four of these methods—init, preprocess, prerender and destroy—are overridden by Page1.java. Other than method init, these methods are initially empty. They serve as placeholders for you to customize the behavior of your web application. The page bean file also includes get and set methods for all of the page’s elements—page, html, head, body and link to start. You can view these get and set methods by clicking the plus (+) sign on the line that says Creator-managed Component Definition.
Fig. 26.11 | JSP file generated for Page1 by Java Studio Creator 2.

Fig. 26.12 | Page bean file for Page1.jsp generated by Java Studio Creator 2.
26.5 Creating and Running a Simple Application in Java Studio Creator 2

Step 6: Renaming the JSP and JSF Files
Typically, you’ll want to rename the JSP and Java files in your project, so that their names are relevant to your application. Right click the page1.jsp file in the Projects Window and select Rename to display the Rename dialog. Enter the new file name Time. If Preview All Changes is checked, the Refactoring Window will appear at the bottom of the IDE when you click Next. Refactoring is the process of modifying source code to improve its readability and reusability without changing its behavior—for example, by renaming methods or variables, or breaking long methods into shorter ones. Java Studio Creator 2 has built-in refactoring tools that automate some refactoring tasks. Using these tools to rename the project files updates the name of both the JSP file and the page bean file. The refactoring tool also changes the class name in the page bean file and all of the attribute bindings in the JSP file to reflect the new class name. Note that none of these changes will be made until you click Do Refactoring in the Refactoring Window. If you do not preview the changes, refactoring occurs when you click Next > in the Rename dialog.

Step 7: Changing the Title of the Page
Before designing the content of the web page, we give it the title “Web Time: A Simple Example”. By default, the page does not have a title when it is generated by the IDE. To add a title, open the JSP file in Design mode. In the Properties window, enter the new title next to the Title property and press Enter. View the JSP to see that the attribute title = "Web Time: A Simple Example" was automatically added to the ui:head tag.

Step 8: Designing the Page
Designing a web page is simple in Java Studio Creator 2. To add components to the page, you can drag and drop them from the Palette onto the page in Design mode. The web page itself, each component is an object that has properties, methods and events. You can set these properties and events visually using the Properties window or programmatically in the page bean file. Get and set methods are automatically added to the page bean file for each component you add to the page.

The IDE generates the JSP tags for the components you drag and drop using a grid layout, as specified in the ui:body tag. This means that components will be rendered to the browser using absolute positioning, so that they appear exactly where they are dropped on the page. As you add components to the page, the style attribute in each component’s JSP element will include the number of pixels from the top and left margins of the page at which the component is positioned.

In this example, we use two Static Text components. To add the first one to the web page, drag and drop it from the Palette’s Basic components list to the page in Design mode. Edit the component’s text by typing “Current time on the Web Server:” directly into the component. The text can also be edited by changing the component’s text property in the Properties window. Java Studio Creator 2 is a WYSIWYG (What You See Is What You Get) editor—whenever you make a change to a web page in Design mode, the IDE creates the markup (visible in JSP mode) necessary to achieve the desired visual effects seen in Design mode. After adding the text to the web page, switch to JSP mode. You should see that the IDE added a ui:staticText element to the page body, which is bound to the object staticText1, in the page bean file and whose text attribute matches the text you just entered. Back in Design mode, click the Static Text component to select it. In the Properties window, click the ellipsis button next to the style property to open a dialog box...
to edit the text’s style. Select 18 px for the font size and click OK. Again in the Properties window, change the id property to timeHeader. Setting the id property also changes the name of the component’s corresponding property in the page bean and updates its binding attribute in the JSP accordingly. Notice that font-size: 18 px has been added to style attribute and the id attribute has been changed to timeHeader in the component’s tag in the JSP file. The IDE should now appear as in Fig. 26.13.

Drop a second Static Text component onto the page and set its id to clockText. Edit its style property so that the font size is 18 px, the text color is yellow, and the background color is black. Do not edit the component’s text, as this will be set programmatically in the page bean file. The component will display with the text Static Text in the IDE, but will not display any text at runtime unless the text is set programmatically. Figure 26.14 shows the IDE after the second component is added.

**Step 9: Adding Page Logic**

After designing the user interface, you can modify the page bean file to set the text of the clockText element. In this example, we add a statement to method prerender (lines 170–174 of Fig. 26.6). Recall that we use method prerender to ensure that clockText will be updated each time the page is refreshed. Lines 172–173 of Fig. 26.6 programmatically set the text of clockText to the current time on the server.

We would like this page to refresh automatically to display an up-to-date time. To accomplish this, add the empty tag `<ui:meta content = "60" httpEquiv = "refresh" />` to the JSP file, between the end of the `ui:head` tag and the beginning of the `ui:body` tag. This tag tells browser to reload the page automatically every 60 seconds. You can also add this tag by dragging a Meta component from the Advanced section of the Palette to your page, then setting the component’s content attribute to 60 and its httpEquiv attribute to refresh.

![Fig. 26.13 | Time.jsp after inserting the first Static Text component.](image-url)
26.5 Creating and Running a Simple Application in Java Studio Creator 2

Step 10: Examining the Outline Window
Figure 26.15 displays the Outline window in Java Studio Creator 2. The project’s four Java files are displayed as gray nodes. The Time node representing the page bean file is expanded and shows the contents of the component tree. The request, session and application scope beans are collapsed by default, as we have not added any properties to these beans in this example. Clicking an item in the page’s component tree selects the item in the Visual Editor.

Step 11: Running the Application
After creating the web page, you can view it several ways. First, you can select Build > Build Main Project, and after the build completes, select Run > Run Main Project, to run the application in a browser window. You can run a project that has already been built by pressing the Run Main Project icon ( ) in the toolbar at the top of the IDE. Note that if changes

Fig. 26.14 | Time.jsp after adding the second Static Text component.

Fig. 26.15 | Outline window in Java Studio Creator 2.
are made to a project, the project must be rebuilt before they will be reflected when the application is viewed in a browser. Because this application was built on the local file system, the URL displayed in the address bar of the browser when the application is run will be http://localhost:29080/WebTime/ (Fig. 26.6), where 29080 is the port number on which Java Studio Creator 2’s built-in test server—Sun Application Server 8—runs by default. When you run a program on the test server, a tray icon appears near the bottom-right of your screen to show that the Sun Application Server is running. To shut down the server after you exit Java Studio Creator 2, right-click the tray icon and select Stop Domain creator.

Alternatively, you can press F5 to build the application, then run it in debug mode—the Java Studio Creator 2 built-in debugger can help you troubleshoot applications. If you type <Ctrl> F5, the program executes without debugging enabled.

Error-Prevention Tip 26.1
If you have trouble building your project due to errors in the Java Studio Creator-generated XML files used for building, try cleaning the project and building again. You can do this by selecting Build > Clean and Build Main Project or by pressing <Alt> B.

Finally, you can run your built application by opening a browser window and typing the web page’s URL in the Address field. Since your application resides on the local file system, you must first start the Sun Application Server. If you have previously run the application using one of the methods above, the server will already be started. Otherwise, you can start the server from the IDE by opening the Servers tab (located in the same panel as the Palette), right-clicking the Deployment Server, selecting Start/Stop Server, and clicking Start in the dialog that appears. Then you can type the URL (including the port number for the application server, 29080) in the browser to execute the application. For this example it is not necessary to type the entire URL, http://localhost:29080/WebTime/faces/Time.jsp. The path to the file Time.jsp (i.e., faces/Time.jsp) can be omitted, because this file was set by default as the project’s start page. For projects with multiple pages, you can change the start page by right clicking the desired page in the Projects window and selecting Set As Start Page. The start page is indicated by a green arrow next to the page’s name in the Projects window. [Note: If you use the Netbeans Visual Web Pack 5.5, the port number will depend on the server to which you deploy your web application. Also, the Servers tab is called Runtime in Netbeans.]

26.6 JSF Components
This section introduces some of the JSF components featured in the Palette (Fig. 26.9). Figure 26.16 summarizes some of the JSF components used in the chapter examples.

<table>
<thead>
<tr>
<th>JSF Components</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Label</td>
<td>Displays text that can be associated with an input element.</td>
</tr>
<tr>
<td>Static Text</td>
<td>Displays text that the user cannot edit.</td>
</tr>
<tr>
<td>Text Field</td>
<td>Gathers user input and displays text.</td>
</tr>
</tbody>
</table>

Fig. 26.16 | Commonly used JSF components. (Part 1 of 2.)
26.6 JSF Components

26.6.1 Text and Graphics Components

Figure 26.17 displays a simple form for gathering user input. This example uses all the components listed in Fig. 26.16, except Label, which you will see in later examples. All the code in Fig. 26.17 was generated by Java Studio Creator 2 in response to actions performed in Design mode. This example does not perform any tasks when the user clicks Register. We ask you to add functionality to this example as an exercise. In successive examples, we demonstrate how to add functionality to many of these JSF components.

**Fig. 26.17** Registration form that demonstrates JSF components. (Part 1 of 3.)

```xml
<xml version = "1.0" encoding = "UTF-8"?>
  <!-- Fig. 26.17: WebComponents.jsp -->
  <!-- Registration form that demonstrates JSF components. -->
  <jsp:root version = "1.2" xmlns:f = "http://java.sun.com/jsf/core"
           xmlns:h = "http://java.sun.com/jsf/html"
           xmlns:jsp = "http://java.sun.com/JSP/Page"
           xmlns:ui = "http://www.sun.com/web/ui">
    <jsp:directive.page contentType = "text/html; charset = UTF-8"
                        pageEncoding = "UTF-8"/>
    <f:view>
      <ui:page binding = "#{WebComponents.page}" id = "page">
        <ui:html binding = "#{WebComponents.html}" id = "html">
          <ui:head binding = "#{WebComponents.head}" id = "head">
            <ui:link binding = "#{WebComponents.link}" id = "link"
                     url = "http://www.sun.com/web/ui">
            <ui:body binding = "#{WebComponents.body}" id = "body"
                     style = "-rave-layout: grid">
              <ui:form binding = "#{WebComponents.form}" id = "form">
                <ui:staticText binding = "#{WebComponents.header}"
                                id = "header" style = "font-size: 18px; left: 24px; top: 24px; position: absolute" text = "This is a sample registration form."/>
                <ui:staticText binding = "#{WebComponents.instructions}"
                                id = "instructions" style = "font-style: italic; left: 24px; top: 72px; position: absolute" text = "Please fill in all fields and click Register."/>
            </ui:body>
          </ui:head>
        </ui:body>
      </ui:html>
    </f:view>
  </jsp:root>
</xml>

**Fig. 26.17** Registration form that demonstrates JSF components. (Part 1 of 3.)
Fig. 26.17  |  Registration form that demonstrates JSF components. (Part 2 of 3.)
26.6 JSF Components

This is a sample registration form.

Please fill in all fields and click Register.

- User Information
- Publications
- Which operating system are you using?
- Which book would you like information about?
- Drop Down List
- Text Field
- Image
- Radio Button Group
- Button

Fig. 26.17 | Registration form that demonstrates JSF components. (Part 3 of 3.)
Before discussing the JSF components used in this JSP file, we explain the XHTML that creates the layout in Fig. 26.17. As discussed previously, Java Studio Creator 2 uses absolute positioning, so components are rendered wherever they were dropped in the Visual Editor. In this example, in addition to absolute positioning, we use a Grid Panel component (lines 31–52) from the Palette’s Layout component group. The prefix indicates that it can be found in the JSF HTML tag library. This component, an object of class htmlPanelGrid in package javax.faces.component.html, controls the positioning of the components it contains. The Grid Panel component allows the designer to specify the number of columns the grid should contain. Components may then be dropped anywhere inside the panel, and they will automatically be repositioned into evenly spaced columns in the order in which they are dropped. When the number of components exceeds the number of columns, the panel moves the additional components to a new row. In this way, the Grid Panel behaves like an XHTML table, and is in fact rendered to the browser as an XHTML table. In this example, we use the Grid Panel to control the positions of the Image and Text Field components in the user information section of the page.

Adding a Formatting Component to a Web Page
To create the layout for the User Information section of the form shown in Fig. 26.17, drag a Grid Panel component onto the page. In the Properties window, set the component’s columns property to 4. The component also has properties to control the cell padding, cell spacing and other elements of the component’s appearance. In this case, accept the defaults for these properties. Now you can simply drag the Images and Text Fields for user information into the Grid Panel. The Grid Panel will manage their spacing and their organization into rows and columns.

Examining Web Components on a Sample Registration Form
Lines 28–30 of Fig. 26.17 define an Image component, an object of class Image which inserts an image into a web page. The images used in this example are located in this chapter’s examples directory. Images to be displayed on a web page must be placed in the project’s resources folder. To add images to the project, drop an Image component onto the page and click the ellipsis button next to the url property in the Properties window. This opens a dialog in which you can select the image to display. Since no images have been added to the resources folder yet, click the Add File button, locate the image on your computer’s file system and click Add File. This copies the file you selected into the project’s resources directory. Now you can select the image from the list of files in the resources folder and click OK to insert the image into the page.

Lines 31–52 contain an h:panelGrid element representing the Grid Panel component. Within this element, there are eight Text Field components. Text Fields allow you to obtain text input from the user. For example, lines 37–38 define a Text Field control used to collect the user’s first name. Lines 49–51 define a Text Field with the label property set to “Must be in the form (555) 555-5555”. Setting the label property of a Text Field places text directly above the Text Field. Alternatively, you can label a Text Field by dragging and dropping a Label component onto the page, which allows you to customize the Label’s position and style.

The order in which Text Fields are dragged to the page is important, because their JSP tags are added to the JSP file in that order. When a user presses the Tab key to navigate between input fields, they will navigate the fields in the order in which the JSP tags occur.
in the JSP file. To specify the navigation order, you should drag components onto the page in that order. Alternatively, you can set each input field's Tab Index property in the Properties window to control the order in which the user will tab through the fields. A component with a tab index of 1 will be the first in the tab sequence.

Lines 62–65 define a Drop Down List. When a user clicks the drop-down list, it expands and displays a list from which the user can make a selection. This component is an object of class DropDownList and is bound to the object booksDropDown, a SingleSelectOptionsList object that controls the list of options. This object can be configured automatically by right clicking the drop-down list in Design mode and selecting Configure Default Options, which opens the Options Customizer dialog box to add options to the list. Each option consists of a display String that will represent the option in the browser and a value String that will be returned when programatically retrieving the user's selection from the drop-down list. Java Studio Creator 2 constructs the SingleSelectOptionsList object in the page bean file based on the display-value pairs entered in the Options Customizer dialog box. To view the code that constructs the object, close the dialog box by clicking OK, open the page bean file, and expand the Creator-managed Component Definition node near the top of the file. The object is constructed in the _init method, which is called from method init the first time the page loads.

The Hyperlink component (lines 66–70) of class Hyperlink adds a link to a web page. The url property of this component specifies the resource (http://www.deitel.com in this case) that is requested when a user clicks the hyperlink. Setting the target property to _blank specifies that the requested web page should open in a new browser window. By default, Hyperlink components cause pages to open in the same browser window.

Lines 78–82 define a Radio Button Group component of class RadioButtonGroup, which provides a series of radio buttons from which the user can select only one. Like Drop Down List, a Radio Button Group is bound to a SingleSelectOptionList object. The options can be edited by right clicking the component and selecting Configure Default Options. Also like the drop-down list, the SingleSelectOptionsList constructor is automatically generated by the IDE and placed in the _init method of the page bean class.

The final web control in Fig. 26.17 is a Button (lines 83–85), a JSF component of class button that triggers an action when clicked. A Button component typically maps to an input XHTML element with attribute type set to submit. As stated earlier, clicking the Register button in this example does not do anything.

### 26.6.2 Validation Using Validator Components and Custom Validators

This section introduces form validation. Validating user input is an important step in collecting information from users. Validation helps prevent processing errors due to incomplete or improperly formatted user input. For example, you may perform validation to ensure that all required fields have been filled out or that a ZIP-code field contains exactly five digits. Java Studio Creator 2 provides three validator components. A Length Validator determines whether a field contains an acceptable number of characters. Double Range Validators and Long Range Validators determine whether numeric input falls within acceptable ranges. Package javax.faces.validators contains the classes for these validators. Studio Creator 2 also allows custom validation with validator methods in the page bean file. The following example demonstrates validation using both a validator component and custom validation.
Validating Form Data in a Web Application

The example in this section prompts the user to enter a name, e-mail address and phone number. After the user enters any data, but before the data is sent to the server, validation ensures that the user entered a value in each field, that the entered name does not exceed 30 characters, and that the e-mail address and phone number values are in an acceptable format. In this example, (555) 123-4567, 555-123-4567 and 123-4567 are all considered valid phone numbers. Once the data is submitted, the web server responds by displaying an appropriate message and a Grid Panel component repeating the submitted information. Note that a real business application would typically store the submitted data in a database or in a file on the server. We simply send the data back to the page to demonstrate that the server received the data.

Building the Web Page

This web application introduces two additional JSF components—Label and Message from the Basic section of the Palette. Each of the page’s three text fields should have its own label and message. Label components describe other components and can be associated with user input fields by setting their for property. Message components display error messages when validation fails. This page requires three Text Fields, three Labels and three Messages, as well as a submit Button. To associate the Label components and Message components with their corresponding Text Field components, hold the Ctrl and Shift keys, then drag the label or message to the appropriate Text Field. In the Properties window, notice that each Label and Message component’s for property is set to the appropriate Text Field.

You should also add a Static Text component to display a validation success message at the bottom of the page. Set the text to “Thank you for your submission.” We received the following information:” and change the component’s id to resultText. In the Properties window, unset the component’s rendered and escaped properties. The rendered property controls whether the component will be displayed the first time the page loads. Setting escaped to false enables the browser to recognize the <br/> tag so it can start a new line of text rather than display the characters “<br/>” in the web page.

Finally, add a Grid Panel component below the resultText component. The panel should have two columns, one for displaying Static Text components that label the user’s validated data, and one for displaying Static Text components that echo back that data.

The JSP file for this page is displayed in Fig. 26.18. Lines 30–34, 35–39 and 40–44 define ui:textFields for retrieving the user’s name, e-mail address and phone number, respectively. Lines 45–48, 49–53, and 54–58 define ui:labels for each of these text fields. Lines 63–74 define the text fields’ ui:message elements. Lines 59–62 define a Submit ui:button. Lines 75–80 create a ui:staticText named resultText that displays the response from the server when the user successfully submits the form, and lines 81–101 define a ui:panelGrid that contains components for echoing validated user input to the browser.

Setting the Required Property of an Input Component

Ensuring that the user has made a selection or entered some text in a required input element is a basic type of validation. This is accomplished by checking the required box in the element’s Properties window. If you add a validator component or custom validator method to an input field, the field’s required property must be set to true for validation.
to occur. Notice that the three input `ui:textFields` in this example (Fig. 26.18, lines 30–44) all have their `required` property set to `true`. Also note in the Visual Editor that the label for a required field is automatically marked by a red asterisk. If a user submits this form with empty text fields, the default error message for a required field will be displayed in the empty field’s associated `ui:message` component.

```xml
<?xml version = "1.0" encoding = "UTF-8"?>
<!-- Fig. 26.18: Validation.jsp -->
<!-- JSP that demonstrates validation of user input. -->
<jsp:root version = "1.2" xmlns:f = "http://java.sun.com/jsf/core"
xmlns:h = "http://java.sun.com/jsf/html" xmlns:jsp =
<jsp:directive.page contentType = "text/html; charset = UTF-8"
pageEncoding = "UTF-8"/>
<f:view>
<!-- page binding = 
"#{Validation.page}" id = "page">
<!-- html binding = 
"#{Validation.html}" id = "html">
<!-- head binding = 
"#{Validation.head}" id = "head">
<title = "Validation">
<!-- link binding = 
"#{Validation.link}" id = "link">
<url = "/resources/stylesheet.css"/>
</ui:head>
<ui:body binding = 
"#{Validation.body}" focus = "form1:nameTF"
id = "body" style = "-rave-layout: grid">
<!-- form binding = 
"#{Validation.form}" id = "form">
<!-- staticText binding = 
"#{Validation.header}" id = "header">
<!-- style = "font-size: 16px; height: 22px;
left: 24px; top: 24px; position: absolute; width: 456px" text = "Please fill out the following form."/>
<!-- staticText binding = 
"#{Validation.instructions}" id = "instructions">
<!-- font-style: italic; left: 24px; top: 48px; position: absolute; width: 406px" text = "All fields are required and must contain valid information.” />
<!-- textField binding = 
"#{Validation.nameTF}" columns = "30" id = "nameTF" required = "true" style = "left: 168px; top: 96px; position: absolute; width: 216px" validator = "#{Validation.nameTF_validate}"/>
<!-- textField binding = 
"#{Validation.emailTF}" columns = "28" id = "emailTF" required = "true" style = "left: 168px; top: 144px; position: absolute; width: 216px" validator = "#{Validation.emailTF_validate}"/>
<!-- textField binding = 
"#{Validation.phoneTF}" columns = "30" id = "phoneTF" required = "true" style = "left: 168px; top: 192px; position: absolute; width: 216px" validator = "#{Validation.phoneTF_validate}"/>
<!-- label binding = "#{Validation.nameLabel}" for = "nameTF" id = "nameLabel" style = "font-weight:
"Fig. 26.18 | JSP that demonstrates validation of user input. (Part 1 of 4.)
Fig. 26.18 | JSP that demonstrates validation of user input. (Part 2 of 4.)
Fig. 26.18 | JSP that demonstrates validation of user input. (Part 3 of 4.)
Fig. 26.18 | JSP that demonstrates validation of user input. (Part 4 of 4.)
26.6 JSF Components

Using the LengthValidator Component

In this example, we use the Length Validator component (found in the Validators section of the Palette) to ensure that the length of the user’s name does not exceed 30 characters. This might be useful to ensure that a value will fit in a particular database field.

To add a Length Validator to a component, simply drag the validator from the Palette and drop it onto the field to validate. A lengthValidator node will appear in the Validation section of the Outline window. To edit the validation component’s properties, click this node and set the maximum and minimum properties to the desired number of characters in the Properties window. Here, we set only the maximum property to 30. We also changed the component’s id to nameLengthValidator. Notice that the nameTF input field in the JSP file has been bound to the validate method of the property nameLengthValidator in the page bean file (lines 33–34).

This validator allows users to type as much text in the field as they wish, and if they exceed the limit, the default length validation error message will be displayed in the field’s ui:message component after the user clicks the Submit button. It is possible to limit the length of user input without using validation. By setting a Text Field’s maxLength property, the Text Field’s cursor will not advance beyond the maximum allowable number of characters, so the user cannot submit data that exceeds the length limit.

Using Regular Expressions to Perform Custom Validation

Some of the most common validation tasks involve checking user input for appropriate formatting. For instance, it may be necessary to check user-entered email addresses and telephone numbers to ensure that they conform to the standard formatting for valid email addresses and phone numbers. Matching user input against a regular expression is an effective way to ensure that the input is properly formatted. (We discuss regular expressions in Section 30.7.) Java Studio Creator 2 does not provide components for validation using regular expressions, so we will add our own custom validator methods to the page bean file. To add a custom validator to an input component, right-click the component and select Edit Event Handler > validate. This creates a validation method for the component with an empty body in the page bean file. We’ll add code to this method shortly. Note that both emailTF and phoneTF’s validate attributes are bound to their respective custom validation methods in the page bean file (lines 38–39 and 43–44).

Examining the Page Bean File for a Form That Receives User Input

Figure 26.19 contains the page bean file for the JSP file in Fig. 26.18. Line 33 sets the maximum length for the nameLengthValidator, which is a property of this page bean. Recall that the name text field was bound to this property in the JSP file. Method emailTF_validate (lines 398–410) and phoneTF_validate (lines 414–426) are the custom validator methods that verify the user-entered email address and phone number, respectively. The submitButton_action method (lines 429–440) echoes the entered data back to the user if validation is successful. The validator methods are called before the event handler, so if validation fails, submitButton_action will not be called and the user input will not be echoed.

The two custom validator methods in this page bean file validate a text field’s contents against a regular expression using the String method match, which takes a regular expression as an argument and returns true if the String conforms to the specified format.
Chapter 26  Web Applications: Part I

Fig. 26.19  |  Page bean for validating user input and redisplaying that input if valid. (Part 1 of 3.)

// Fig. 26.19: Validation.java
// Page bean for validating user input and redisplaying that input if valid.
package validation;

import com.sun.rave.web.ui.component.Body;
import com.sun.rave.web.ui.component.Form;
import com.sun.rave.web.ui.component.Head;
import com.sun.rave.web.ui.component.Html;
import com.sun.rave.web.ui.component.Link;
import com.sun.rave.web.ui.component.Page;
import com.sun.rave.web.ui.component.StaticText;
import com.sun.rave.web.ui.component.TextField;
import com.sun.rave.web.ui.component.TextArea;
import com.sun.rave.web.ui.component.Label;
import com.sun.rave.web.ui.component.Button;
import com.sun.rave.web.ui.component.Message;
import javax.faces.FacesException;
import javax.faces.component.UIComponent;
import javax.faces.context.FacesContext;
import javax.faces.validator.ValidatorException;
import javax.faces.application.FacesMessage;
import javax.faces.component.html.HtmlPanelGrid;
import javax.faces.validator.LengthValidator;

public class Validation extends AbstractPageBean {
    private int __placeholder;

    private void _init() throws Exception {
        nameLengthValidator.setMaximum(30);
    }

    public Validation() {
        // empty constructor
    }

    protected RequestBean getRequestBean() {
        return (RequestBean) getBean("RequestBean");
    }

    protected ApplicationBean getApplicationBean() {
        return (ApplicationBean) getBean("ApplicationBean");
    }

    // To save space, we omitted the code in lines 36-345. The complete
    // source code is provided with this chapter's examples.

    // Fig. 26.19  |  Page bean for validating user input and redisplaying that input if valid. (Part 1 of 3.)
protected SessionBean getSessionBean()
{
  return (SessionBean) getBean("SessionBean");
} // end method getSessionBean

public void init()
{
  super.init();
  try
  {
    _init();
  } // end try
  catch (Exception e)
  {
    log("Validation Initialization Failure", e);
    throw e instanceof FacesException ? (FacesException) e:
    new FacesException(e);
  } // end catch
} // end method init

public void preprocess()
{
  // empty body
} // end method preprocess

public void prerender()
{
  // empty body
} // end method prerender

public void destroy()
{
  // empty body
} // end method destroy

// validates entered email address against the regular expression
// that represents the form of a valid email address.
public void emailTF_validate(FacesContext context,
  UIComponent component, Object value)
{
  String email = String.valueOf(value);
  if (!email.matches("\w+(\[-+.'\]\w+)*@\w+(\[-.\]\w+)*\.|\w+(\[-.\]\w+)*")
  {
    throw new ValidatorException(new FacesMessage("Enter a valid email address, e.g. user@domain.com"));
  } // end if
} // end method emailTF_validate

Fig. 26.19  Page bean for validating user input and redisplaying that input if valid. (Part 2 of 3.)
For the `emailTF_validate` method, we use the validation expression

```
\w+([-+.']\w+)*@\w+([-.]\w+)*.\w+([-.]\w+)*
```

Note that each backslash in the regular expression `String` (line 405) must be escaped with another backslash (as in `\`), because the backslash character normally represents the beginning of an escape sequence. This regular expression indicates that an e-mail address is valid if the part before the `@` symbol contains one or more word characters (i.e., alphanumeric characters or underscores), followed by zero or more `String`s comprised of a hyphen, plus sign, period or apostrophe and additional word characters. After the `@` symbol, a valid e-mail address must contain one or more groups of word characters potentially separated by hyphens or periods, followed by a required period and another group of one or more word characters potentially separated by hyphens or periods. For example, the e-mail addresses `bob's-personal.email@white.email.com`, `bob-white@my-email.com` and `bob.white@email.com` are all valid. If the user enters text in `emailTF` that does not have the correct format and attempts to submit the form, lines 408–408 throw a `ValidatorException`. The `emailMessage` component will catch this exception and display the message in red.

The regular expression in `phoneTF_validate` ensures that the `phoneTextBox` contains a valid phone number before the form is submitted. The user input is matched against the regular expression

```
((\(\d{3}\) ?)|\d{3}-)\d{3}-\d{4}
```

Note that each backslash in the regular expression `String` (line 414) must be escaped with another backslash (as in `\`), because the backslash character normally represents the beginning of an escape sequence. This regular expression indicates that a phone number is valid if it matches the format `(555) 555-1234`. If the entered phone number does not match this format, lines 419–419 throw a `ValidatorException`. The `phoneMessage` component will catch this exception and display the message in red.
26.7 Session Tracking

In the early days of the internet, e-businesses could not provide the kind of customized service typically experienced in “brick-and-mortar” stores. To address this problem, e-businesses began to establish mechanisms by which they could personalize users' browsing experiences, tailoring content to individual users while enabling them to bypass irrelevant information. Businesses achieve this level of service by tracking each customer’s movement through their websites and combining the collected data with information provided by the consumer, including billing information, personal preferences, interests and hobbies.

Personalization

Personalization makes it possible for e-businesses to communicate effectively with their customers and also improves the user’s ability to locate desired products and services. Companies that provide content of particular interest to users can establish relationships with customers and build on those relationships over time. Furthermore, by targeting consumers with personal offers, recommendations, advertisements, promotions and services, e-businesses create customer loyalty. Websites can use sophisticated technology to allow visitors to customize home pages to suit their individual needs and preferences. Similarly, online shopping sites often store personal information for customers, tailoring notifications and special offers to their interests. Such services encourage customers to visit sites and make purchases more frequently.

Privacy

A trade-off exists, however, between personalized e-business service and protection of privacy. Some consumers embrace the idea of tailored content, but others fear the possible adverse consequences if the information they provide to e-businesses is released or collected by tracking technologies. Consumers and privacy advocates ask: What if the e-business to which we give personal data sells or gives that information to another organization without our knowledge? What if we do not want our actions on the Internet—a supposedly anonymous medium—to be tracked and recorded by unknown parties? What if unauthorized parties gain access to sensitive private data, such as credit-card numbers or medical history?
All of these are questions that must be debated and addressed by programmers, consumers, e-businesses and lawmakers alike.

**Recognizing Clients**

To provide personalized services to consumers, e-businesses must be able to recognize clients when they request information from a site. As we have discussed, the request/response system on which the web operates is facilitated by HTTP. Unfortunately, HTTP is a stateless protocol—it does not support persistent connections that would enable web servers to maintain state information regarding particular clients. So, web servers cannot determine whether a request comes from a particular client or whether a series of requests comes from one or several clients. To circumvent this problem, sites can provide mechanisms to identify individual clients. A session represents a unique client on a website. If the client leaves a site and then returns later, the client will still be recognized as the same user. To help the server distinguish among clients, each client must identify itself to the server. Tracking individual clients, known as **session tracking**, can be achieved in a number of ways. One popular technique uses **cookies** (Section 26.7.1); another uses the **SessionBean** object (Section 26.7.2). Additional session-tracking techniques include using **input form elements** of type "hidden" and URL rewriting. With "hidden" form elements, a web form can write session-tracking data into a **form** in the web page that it returns to the client in response to a prior request. When the user submits the form in the new web page, all the form data, including the "hidden" fields, is sent to the form handler on the web server. With URL rewriting, the web server embeds session-tracking information directly in the URLs of hyperlinks that the user clicks to send subsequent requests to the web server.

### 26.7.1 Cookies

**Cookies** provide web developers with a tool for personalizing web pages. A cookie is a piece of data typically stored in a text file on the user’s computer. A cookie maintains information about the client during and between browser sessions. The first time a user visits the website, the user’s computer might receive a cookie; this cookie is then reactivated each time the user revisits that site. The collected information is intended to be an anonymous record containing data that is used to personalize the user’s future visits to the site. For example, cookies in a shopping application might store unique identifiers for users. When a user adds items to an online shopping cart or performs another task resulting in a request to the web server, the server receives a cookie from the client containing the user’s unique identifier. The server then uses the unique identifier to locate the shopping cart and perform any necessary processing.

In addition to identifying users, cookies also can indicate clients’ shopping preferences. When a web server receives a request from a client, the server can examine the cookie(s) it sent to the client during previous communications, identify the client’s preferences and immediately display products of interest to the client.

Every HTTP-based interaction between a client and a server includes a header containing information either about the request (when the communication is from the client to the server) or about the response (when the communication is from the server to the client). When a page receives a request, the header includes information such as the request type (e.g., `GET` or `POST`) and any cookies that have been sent previously from the server to be stored on the client machine. When the server formulates its response, the header infor-
Session Tracking

Information contains any cookies the server wants to store on the client computer and other information, such as the MIME type of the response.

The expiration date of a cookie determines how long the cookie remains on the client’s computer. If you do not set an expiration date for a cookie, the web browser maintains the cookie for the duration of the browsing session. Otherwise, the web browser maintains the cookie until the expiration date occurs. When the browser requests a resource from a web server, cookies previously sent to the client by that web server are returned to the web server as part of the request formulated by the browser. Cookies are deleted when they expire.

Portability Tip 26.1
Clients may disable cookies in their web browsers for more privacy. When such clients use web applications that depend on cookies to maintain state information, the applications will not execute correctly.

Using Cookies to Provide Book Recommendations
The next web application shows how to use cookies. The example contains two pages. In the first page (Figs. 26.20 and 26.22), users select a favorite programming language from a group of radio buttons and submit the form to the web server for processing. The web server responds by creating a cookie that stores the selected language and the ISBN number for a recommended book on that topic. The server then renders new components in the browser that allow the user either to select another favorite programming language or to view the second page in our application (Figs. 26.23–26.24), which lists recommended books pertaining to the programming language(s) that the user selected. When the user clicks the hyperlink, the cookies previously stored on the client are read and used to form the list of book recommendations.

The JSP file in Fig. 26.20 contains a Radio Button Group (lines 26–39) with the options Java, C, C++, Visual Basic 2005, and Visual C# 2005. Recall that you can set the display and value Strings of radio buttons by right clicking the Radio Button Group and selecting Configure Default Options. The user selects a programming language by clicking one of the radio buttons. When the user presses the Submit button, the web application creates a cookie containing the selected language. This cookie is added to the HTTP response header and sent to the client as part of the response.

When Submit is clicked, the ui:label, ui:radioButtonGroup and ui:button elements used to select a language are hidden, and a ui:staticText and two ui:hyperlink elements are displayed. Each ui:staticText and ui:hyperlink initially has its rendered property set to false (lines 31, 37, and 43). This indicates that these components are not visible the first time the page loads, as we want the user’s first view of the page to include only the components for selecting a programming language and submitting the selection.

```xml
<?xml version = "1.0" encoding = "UTF-8"?>
<!-- Fig. 26.20: Options.jsp -->
<!-- JSP file that allows the user to select a programming language. -->
```

Fig. 26.20 | JSP file that allows the user to select a programming language. (Part 1 of 4.)
<jsp:root version="1.2" xmlns:f="http://java.sun.com/jsf/core"
  <jsp:directive.page contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"/>
  <f:view>
    <ui:page binding="#{Options.page}" id="page">
      <ui:html binding="#{Options.html}" id="html">
        <ui:head binding="#{Options.head}" id="head" title="Options">
          <ui:link binding="#{Options.link}" id="link" url="/resources/stylesheet.css"/>
        </ui:head>
        <ui:body binding="#{Options.body}" id="body" style="-rave-layout: grid">
          <ui:form binding="#{Options.form}" id="form">
            <ui:label binding="#{Options.languageLabel}" for="languageList" id="languageLabel" style="font-size: 16px; font-weight: bold; left: 24px; top: 24px; position: absolute" text="Select a programming language:">
              <ui:radioButtonGroup binding="#{Options.languageList}" id="languageList" items="#{Options.languageListOptions.options}" style="left: 24px; top: 48px; position: absolute"/>
            </ui:label>
            <ui:staticText binding="#{Options.responseLabel}" id="responseLabel" rendered="false" style="font-size: 16px; font-weight: bold; height: 24px; left: 24px; top: 24px; position: absolute; width: 216px"/>
            <ui:hyperlink action="#{Options.languagesLink_action}" binding="#{Options.languagesLink}" id="languagesLink" rendered="false" style="left: 24px; top: 96px; position: absolute" text="Click here to choose another language."/>
            <ui:hyperlink action="#{Options.recommendationsLink_action}" binding="#{Options.recommendationsLink}" id="recommendationsLink" rendered="false" style="left: 24px; top: 120px; position: absolute" text="Click here to get book recommendations." url="/faces/Recommendations.jsp"/>
            <ui:button action="#{Options.submit_action}" binding="#{Options.submit}" id="submit" style="left: 24px; top: 192px; position: absolute" text="Submit"/>
          </ui:form>
        </ui:body>
      </ui:html>
    </ui:page>
  </f:view>
</jsp:root>

Fig. 26.20 | JSP file that allows the user to select a programming language. (Part 2 of 4.)
Fig. 26.20 | JSP file that allows the user to select a programming language. (Part 3 of 4.)
Chapter 26  Web Applications: Part 1

The first hyperlink (lines 35–39) requests this page, and the second (lines 40–47) requests Recommendations.jsp. The url property is not set for the first link; we discuss this momentarily. The second link’s url property is set to /faces/Recommendations.jsp. Recall that earlier in the chapter, we set a url property to a remote website (http://www.deitel.com). To set this property to a page within the current application, click the ellipsis button next to the url property in the Properties window to open a dialog. Use this dialog to select a page within your project as the destination for the link.

Adding and Linking to a New Web Page
Setting the url property to a page in the current application requires that the destination page already exists. To set the url property of a link to Recommendations.jsp, you must first create this page. Right click the Web Pages node in the Projects window and select New > Page from the menu that appears. In the New Page dialog, change the name of the page to Recommendations and click Finish to create the files Recommendations.jsp and Recommendations.java. (We discuss the contents of these files shortly.) Once the Recommendations.jsp file exists, you can select it as the url value for recommendationsLink.

For Options.jsp, rather than setting the languagesLink’s url property, we will add an action handler for this component to the page bean. The action handler will enable us to show and hide components of the page without redirecting the user to another page. Specifying a destination url would override the component’s action handler and redirect the user to the specified page, so it is important that we do not set the url property in this case. Since we use this link to reload the current page, we simply return null from the action handler, causing Options.jsp to reload.

To add an action handler to a hyperlink that should also direct the user to another page, you must add a rule to the Page Navigation file (Fig. 26.21). To edit this file, right click anywhere in the Visual Designer and select Page Navigation…. Locate the link whose navigation rule you would like to set and drag it to the destination page. Now the link can direct the user to a new page without overriding its action handler. Editing the Page Navigation file is also useful when you would like action elements that cannot specify a url property, such as buttons, to direct users to another page.
26.7 Session Tracking

Figure 26.22 contains the code that writes a cookie to the client machine when the user selects a programming language. The file also determines which components appear on the page, displaying either the components for choosing a language or the links for navigating the application, depending on the user’s actions.

```java
// Fig. 26.22: Options.java
// Page bean that stores the user's language selection as a cookie on the client.
package cookies;

import com.sun.rave.web.ui.component.Body;
import com.sun.rave.web.ui.component.Form;
import com.sun.rave.web.ui.component.Head;
import com.sun.rave.web.ui.component.Html;
import com.sun.rave.web.ui.component.Link;
import com.sun.rave.web.ui.component.Page;
import javax.faces.FacesException;
import com.sun.rave.web.ui.component.StaticText;
import com.sun.rave.web.ui.component.Label;
import com.sun.rave.web.ui.model.SingleSelectOptionsList;
import com.sun.rave.web.ui.component.Hyperlink;
import com.sun.rave.web.ui.component.Button;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.Cookie;
import java.util.Properties;

public class Options extends AbstractPageBean {
    private int __placeholder;
    ...
    public class Options extends AbstractPageBean {
        private int __placeholder;
        ...
    }
```
private void _init() throws Exception {
    languageListOptions.setOptions(
        new com.sun.rave.web.ui.model.Option[] {
            new com.sun.rave.web.ui.model.Option("Java", "Java"),
            new com.sun.rave.web.ui.model.Option("C", "C"),
            new com.sun.rave.web.ui.model.Option("C++", "C++"),
        }
    );
} // end method _init

// To save space, we omitted the code in lines 46-203. The complete
// source code is provided with this chapter's examples.

private Properties books = new Properties();

// Construct a new page bean instance and initialize the properties
// that map languages to ISBN numbers of recommended books.
public Options() {
    // initialize the Properties object of values to be stored as
    // cookies.
    books.setProperty("Java", "0-13-222220-5");
    books.setProperty("C", "0-13-142644-3");
    books.setProperty("C++", "0-13-183757-6");
    books.setProperty("Visual/Basic/2005", "0-13-186900-0");
} // end Options constructor

protected ApplicationBean getApplicationBean() {
    return (ApplicationBean) getBean("ApplicationBean");
} // end method getApplicationBean

protected RequestBean getRequestBean() {
    return (RequestBean) getBean("RequestBean");
} // end method getRequestBean

protected SessionBean getSessionBean() {
    return (SessionBean) getBean("SessionBean");
} // end method getSessionBean

Fig. 26.22 | Page bean that stores the user’s language selection in a cookie on the client. (Part 2
of 4.)
26.7 Session Tracking

public void init()
{
    super.init();
    try
    {
        _init();
    } // end try
    catch ( Exception e )
    {
        log( "Options Initialization Failure", e );
        throw e instanceof FacesException ? ( FacesException ) e:
        new FacesException( e );
    } // end catch
} // end method init

public void preprocess()
{
    // empty body
} // end method preprocess

public void prerender()
{
    // empty body
} // end method prerender

public void destroy()
{
    // empty body
} // end method destroy

// Action handler for the Submit button. Checks to see if a language
// was selected and if so, registers a cookie for that language and
// sets the responseLabel to indicate the chosen language.
public String submit_action()
{
    String msg = "Welcome to Cookies! You ";
    // if the user made a selection
    if ( languageList.getSelected() != null )
    {
        String language = languageList.getSelected().toString();
        String displayLanguage = language.replace( '/', ' ' );
        msg += "selected " + displayLanguage + "."
        // get ISBN number of book for the given language.
        String ISBN = books.getProperty( language );
        // create cookie using language-ISBN name-value pair
        Cookie cookie = new Cookie( language, ISBN );
    }

Fig. 26.22 | Page bean that stores the user's language selection in a cookie on the client. (Part 3
of 4.)
As mentioned previously, the _init method handles component initialization. Since this page contains a RadioButtonGroup object that requires initialization, method _init (lines 30–44) constructs an array of Option objects to be displayed by the buttons. The option's names contain slashes rather than spaces in lines 38 and 40, because we later use these names as cookie names and Java does not allow cookie names to contain spaces.

Lines 212–216 in the constructor initialize a Properties object—a data structure that stores String key-value pairs. The application uses the key to store and retrieve the associated value in the Properties object. In this example, the keys are Strings containing the programming language names, and the values are Strings containing the ISBN numbers for the recommended books. Class Properties provides method setProperty, which takes as arguments a key and a value. A value that is added via method setProperty is placed in the Properties at a location determined by the key. The value for a specific Properties entry can be determined by invoking the method getProperty on the Properties object with that value's key as an argument.
Software Engineering Observation 26.1

Java Studio Creator 2 can automatically import any missing packages your Java file needs. For example, after adding the Properties object to Options.java, you can right click in the Java editor window and select Fix Imports to automatically import java.util.Properties.

Clicking Submit invokes the event handler submit_action (lines 267–301), which display a message indicating the selected language in the responseLabel element and adds a new cookie to the response. If a language was selected (line 272), the selected value is retrieved (line 274). Line 275 converts the selection to a String that can be displayed in the responseLabel, replacing the slashes with spaces. Line 276 adds the selected language to the results message.

Line 279 retrieves the ISBN for the selected language from the books Properties. Then line 282 creates a new cookie object (of class Cookie in package javax.servlet.http), using the selected language as the cookie’s name and a corresponding ISBN number as the cookie’s value. This cookie is added to the HTTP response header in lines 286–288. An object of class HttpServletResponse (from package javax.servlet.http) represents the response. This object can be accessed by invoking the method getExternalContext on the page bean then invoking getResponse on the resulting object. If a language was not selected, line 291 sets the results message to indicate that no selection was made.

Lines 293–299 control the appearance of the page after the user clicks Submit. Line 293 sets the responseLabel to display the String msg. Since the user has just submitted a language selection, the components used to collect the selection are hidden (lines 294–296) and responseLabel and the links used to navigate the application are displayed (lines 297–299). The action handler returns null at line 300, which reloads options.jsp.

Lines 305–311 contain the languagesLink’s event handler. When the user clicks this link, responseLabel and the two links are hidden (lines 307–309), and the components that allow the user to select a language are redisplayed (lines 310–312). The method returns null at line 313, causing options.jsp to reload.

Displaying Book Recommendations Based on Cookie Values

After clicking Submit, the user may request a book recommendation. The book recommendations hyperlink forwards the user to Recommendations.jsp (Fig. 26.23) to display recommendations based on the user’s language selections.

```xml
<?xml version = "1.0" encoding = "UTF-8"?>
<!-- Fig. 26.23: Recommendations.jsp -->
<!-- JSP file that displays book recommendations based on cookies. -->
<jsp:root version = "1.2" xmlns:f = "http://java.sun.com/jsf/core"
xmlns:ui = "http://www.sun.com/web/ui">
<jsp:directive.page contentType = "text/html; charset = UTF-8"
pageEncoding = "UTF-8"/>
<f:view>
  <ui:page binding = "#{Recommendations.page}" id = "page">

Fig. 26.23 | JSP file that displays book recommendations based on cookies. (Part 1 of 2.)
Recommendations.jsp contains a Label (lines 21–24), a List Box (lines 25–29) and a Hyperlink (lines 30–33). The Label displays the text Recommendations at the top of the page. A List Box component displays a list of options from which a user can make multiple selections. The List Box in this example displays the recommendations created by the Recommendation.java page bean (Fig. 26.24), or the text "No Recommendations. Please
select a language." The Hyperlink allows the user to return to Options.jsp to select additional languages.

**Page Bean That Creates Book Recommendations from Cookies**

In Recommendations.java (Fig. 26.24), method prerender (lines 192–223) retrieves the cookies from the client, using the request object’s get cookies method (lines 195–197). An object of class HttpServletRequest (from package javax.servlet.http) represents the request. This object can be obtained by invoking method getExternalContext on the page bean, then invoking getRequest on the resulting object. The call to get cookies returns an array of the cookies previously written to the client. Cookies can be read by an application only if they were created by a server in the domain in which the application is running—a web server cannot access cookies created by servers in other domains. For example, a cookie created by a web server in the deitel.com domain cannot be read by a web server in any other domain.

```java
// Fig. 26.24: Recommendations.java
// Page bean that displays book recommendations based on cookies storing user's selected languages.
package cookies;

import com.sun.rave.web.ui.component.Body;
import com.sun.rave.web.ui.component.Form;
import com.sun.rave.web.ui.component.Head;
import com.sun.rave.web.ui.component.Html;
import com.sun.rave.web.ui.component.Link;
import com.sun.rave.web.ui.component.Page;
import com.sun.rave.web.ui.component.Listbox;
import com.sun.rave.web.ui.component.Label;
import com.sun.rave.web.ui.component.Hyperlink;
import com.sun.rave.web.ui.component.HiddenField;

public class Recommendations extends AbstractPageBean
{
    private int __placeholder;

    private void _init() throws Exception
    {
        // empty body
    } // end method _init()

    // To save space, we omitted the code in lines 32-151. The complete
    // source code is provided with this chapter's examples.
```

Fig. 26.24 | Page bean that displays book recommendations based on cookies storing user’s selected languages. (Part 1 of 3.)
public Recommendations()
{
    // empty body
} // end constructor

protected RequestBean getRequestBean()
{
    return (RequestBean) getBean("RequestBean");
} // end method getRequestBean

protected ApplicationBean getApplicationBean()
{
    return (ApplicationBean) getBean("ApplicationBean");
} // end method getApplicationBean

protected SessionBean getSessionBean()
{
    return (SessionBean) getBean("SessionBean");
} // end method getSessionBean

public void init()
{
    super.init();
    try
    {
        _init();
    } // end try
    catch (Exception e)
    {
        log("Recommendations Initialization Failure", e);
        throw e instanceof FacesException ? (FacesException) e:
        new FacesException(e);
    } // end catch
} // end method init

public void preprocess()
{
    // empty body
} // end method preprocess

public void prerender()
{
    //retrieve client's cookies
    HttpServletRequest request =
    (HttpServletRequest)getExternalContext().getRequest();
    Cookie[] cookies = request.getCookies();
    // if there are cookies, store the corresponding books and ISBN
    Option[] recommendations;
}

Fig. 26.24 | Page bean that displays book recommendations based on cookies storing user’s
selected languages. (Part 2 of 3.)
Session Tracking

Line 203 determines whether at least one cookie exists. (We ignore the first cookie in the array which contains information that is not specific to our application.) Lines 205–213 add the information in the cookie(s) to an `Option` array. Arrays of `Option` objects can be displayed as a list of items in a `ListBox` component. The loop retrieves the name and value of each cookie using the control variable to determine the current value in the cookie array. If no language was selected, lines 215–220 add to an `Options` array a message instructing the user to select a language. Line 222 sets `booksListBox` to display the resulting `Options` array. We summarize commonly used `Cookie` methods in Fig. 26.25.

```java
202
203    if ( cookies.length > 1 )
204    {
205        recommendations = new Option[ cookies.length - 1 ];
206        for ( int i = 0; i < cookies.length - 1 ; i++ )
207            {
208                String language =
209                        cookies[ i ].getName().replace( '/','' );
210                recommendations[ i ] = new Option( language + " How to "
211                        "Program. ISBN#:" + cookies[ i ].getValue() );
212            } // end for
213    } // end if
214    // otherwise store a message indicating no language was selected
215    else
216        {
217            recommendations = new Option[ 1 ];
218            recommendations[ 0 ] = new Option( "No recommendations. " +
219                        "Please select a language." ) ;
220        } // end else
221
222    booksListBox.setItems( recommendations );
223} // end method prerender
224
225    public void destroy()
226    {
227        // empty body
228    } // end method destroy
229} // end class Recommendations
```

Fig. 26.24 | Page bean that displays book recommendations based on cookies storing user's selected languages. (Part 3 of 3.)

Line 203 determines whether at least one cookie exists. (We ignore the first cookie in the array which contains information that is not specific to our application.) Lines 205–213 add the information in the cookies to an `Option` array. Arrays of `Option` objects can be displayed as a list of items in a `ListBox` component. The loop retrieves the name and value of each cookie using the control variable to determine the current value in the cookie array. If no language was selected, lines 215–220 add to an `Options` array a message instructing the user to select a language. Line 222 sets `booksListBox` to display the resulting `Options` array. We summarize commonly used `Cookie` methods in Fig. 26.25.

```
Methods       Description
getDomain     Returns a `String` containing the cookie's domain (i.e., the domain from which the cookie was written). This determines which web servers can receive the cookie. By default, cookies are sent to the web server that originally sent the cookie to the client. Changing the `Domain` property causes the cookie to be returned to a web server other than the one that originally wrote it.
```

Fig. 26.25 | `javax.servlet.http.Cookie` methods. (Part 1 of 2.)
26.7.2 Session Tracking with the SessionBean Object

You can also perform session tracking with the SessionBean class that is provided in each web application created with Java Studio Creator 2. When a web page in the project is requested, a SessionBean object is created. Properties of this object can be accessed throughout a browser session by invoking the method getSessionBean on the page bean. To demonstrate session-tracking techniques using the SessionBean, we modified the page bean files in Figs. 26.22 and 26.24 so that they use the SessionBean to store the user’s selected languages. We begin with the updated Options.jsp file (Fig. 26.27). Figure 26.29 presents the SessionBean.java file, and Fig. 26.30 presents the modified page bean file for Options.jsp.

The Options.jsp file in Fig. 26.26 is similar to that presented in Fig. 26.20 for the cookies example. Lines 38–45 define two ui:staticText elements that were not present in the cookies example. The first element displays the text "Number of selections so far:". The second element’s text attribute is bound to property numSelections in the SessionBean (lines 44–45). We discuss how to bind the text attribute to a SessionBean property momentarily.

```
<?xml version = "1.0" encoding = "UTF-8"?>
<!-- Fig. 26.26: Options.jsp -->
<!-- JSP file that allows the user to select a programming language. -->
<jsp:root version = "1.2" xmlns:f = "http://java.sun.com/jsf/core"
  <jsp:directive.page contentType = "text/html; charset = UTF-8"
      pageEncoding = "UTF-8"/>
  ...
</jsp:root>
```

Fig. 26.26 | JSP file that allows the user to select a programming language. (Part 1 of 4.)
Fig. 26.26 | JSP file that allows the user to select a programming language. (Part 3 of 4.)
Adding Properties to the SessionBean

In this example, we use session tracking to store not only the user’s selected languages, but also the number of selections made. To store this information in the SessionBean, we add properties to the SessionBean class.

To add a property that will store the number of selections so far, right click the SessionBean node in the Outline window and select Add > Property to display the New Property Pattern dialog (Fig. 26.27). This dialog allows you to add primitive, String or primitive type-wrapper properties to the SessionBean. Add an int property named numSelections and click OK to accept the default settings for this property. Open the SessionBean file and you will see a new property definition, a get and set method for numSelections.

Fig. 26.26 | JSP file that allows the user to select a programming language. (Part 4 of 4.)

Fig. 26.27 | New Property dialog for adding a property to the SessionBean.
26.7 Session Tracking

Fig. 26.26 | JSP file that allows the user to select a programming language. (Part 2 of 4.)
Property `numSelections` will be manipulated in the page bean file to store the number of languages the user selected. To display the value of this property in the `numSelected` `Static Text` element in the JSP file, right click the `Static Text` component in the Outline window in Design mode and select Bind to Data.... In the Bind to Data dialog (Fig. 26.28), select the Bind to an Object tab, locate property `numSelections` under the SessionBean node and click OK. The `Static Text` element will now always display the value of SessionBean's `numSelections` property. If the property's value changes, the text changes as well, so that you need not programatically set the text in the page bean file.

Now that we have added a property to store the number of selections in the SessionBean, we must add a second property to store the selections themselves. We would like to store selections as key-value pairs of the selected language and the ISBN number of a related book, similar to the way selections were stored using cookies. To do this, we add a `Properties` object named `selectedLanguages` to the `SessionBean`. We manually added this property to the `SessionBean` file, but you can add it using the New Property dialog in Fig. 26.27. Simply type `java.util.Properties` in the Type drop down list's field, configure the property and click OK. The final `SessionBean` file after the two properties have been added is displayed in Fig. 26.29.

```
// Fig. 26.29: SessionBean.java

// SessionBean file for storing language selections.
package session;

import com.sun.rave.web.ui.appbase.AbstractSessionBean;
import java.util.Properties;
import javax.faces.FacesException;
```

Fig. 26.29 | SessionBean file for storing language selections. (Part 1 of 3.)
public class SessionBean extends AbstractSessionBean
{
  private int __placeholder;
  private void _init() throws Exception
  {
    // empty body
    } // end method _init
  public SessionBean()
  {
    // empty constructor
    } // end constructor
  protected ApplicationBean getApplicationBean()
  {
    return (ApplicationBean) getBean("ApplicationBean");
  } // end method getApplicationBean
  public void init()
  {
    super.init();
    try
    {
      _init();
    } // end try
    catch (Exception e)
    {
      log("SessionBean Initialization Failure", e);
      throw e instanceof FacesException ? (FacesException) e:
          new FacesException( e);
    } // end catch
    } // end method init
  public void passivate()
  {
    // empty body
    } // end method passivate
  public void activate()
  {
    // empty body
    } // end method activate
  public void destroy()
  {
    // empty body
    } // end method destroy
  private int numSelections = 0; // stores number of unique selections
  public int getNumSelections()
  {

Fig. 26.29 | SessionBean file for storing language selections. (Part 2 of 3.)
1302  Chapter 26  Web Applications: Part I

Fig. 26.29  |  SessionBean file for storing language selections. (Part 3 of 3.)

Line 58 declares the numSelections property, and lines 60–63 and 65–68 declare its get and set methods, respectively. This portion of the code was generated automatically when we used the New Property dialog. Line 71 defines the Properties object selectedLanguages that will store user selections. Lines 73–76 and 78–81 are the get and set methods for this property.

**Manipulating SessionBean Properties in a Page Bean File**

The page bean file for the Options.jsp page is displayed in Fig. 26.30. Because much of this example is identical to the preceding one, we concentrate on the new features.

```
// Fig. 26.30: Options.java
// Page bean that stores language selections in a SessionBean property.
package session;

import com.sun.rave.web.ui.component.Body;
import com.sun.rave.web.ui.component.Form;
import com.sun.rave.web.ui.component.Head;
import com.sun.rave.web.ui.component.Html;
import com.sun.rave.web.ui.component.Link;
import com.sun.rave.web.ui.component.Page;
import javax.faces.FacesException;
import com.sun.rave.web.ui.component.Hyperlink;
import com.sun.rave.web.ui.component.Button;
import com.sun.rave.web.ui.component.Label;
```

Fig. 26.30  |  Page bean that stores language selections in a SessionBean property. (Part 1 of 4.)
import com.sun.rave.web.ui.component.StaticText;
import com.sun.rave.web.ui.model.SingleSelectOptionsList;
import java.util.Properties;
import javax.servlet.http.Cookie;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpSession;

public class Options extends AbstractPageBean {
    private int __placeholder;

    private void _init() throws Exception {
        languageListOptions.setOptions(new com.sun.rave.web.ui.model.Option[]{
            new com.sun.rave.web.ui.model.Option( "Java", "Java" ),
            new com.sun.rave.web.ui.model.Option( "C", "C" ),
            new com.sun.rave.web.ui.model.Option( "C++", "C++" ),
        });
    }

    private Properties books = new Properties();

    public Options() {
        // initialize the Properties object of values to be stored in
        // the session bean.
        books.setProperty( "Java", "0-13-148398-6" );
        books.setProperty( "C", "0-13-142644-3" );
        books.setProperty( "C++", "0-13-185757-6" );
        books.setProperty( "Visual Basic 2005", "0-13-186900-0" );
        books.setProperty( "Visual C# 2005", "0-13-152523-9" );
    }

    protected ApplicationBean getApplicationBean() {
        return (ApplicationBean) getBean( "ApplicationBean" );
    }

    protected RequestBean getRequestBean() {
        return (RequestBean) getBean( "RequestBean" );
    }

    // To save space, we omitted the code in lines 44-219. The complete
    // source code is provided with this chapter's examples.

    Fig. 26.30 | Page bean that stores language selections in a SessionBean property. (Part 2 of 4.)
protected SessionBean getSessionBean()
{
    return (SessionBean) getBean("SessionBean");
} // end method getSessionBean

public void init()
{
    super.init();
    try
    {
        _init();
    } // end try
    catch ( Exception e )
    {
        log("Options Initialization Failure", e);
        throw e instanceof FacesException ? (FacesException) e:
            new FacesException( e );
    } // end catch
} // end method init

public void preprocess()
{
    // empty body
} // end method preprocess

public void prerender()
{
    // empty body
} // end method prerender

public void destroy()
{
    // empty body
} // end method destroy

// action handler for the submit button, stores selected languages
// in session scope for retrieval when making book recommendations.
public String submit_action()
{
    String msg = "Welcome to sessions! You ";
    // if the user made a selection
    if ( getLanguageList().getSelected() != null )
    {
        String language = languageList.getSelected().toString();
        msg += "selected " + language + ".";
        // get ISBN number of book for the given language.
        String ISBN = books.getProperty( language );
        // add the selection to the SessionBean's Properties object
        Properties selections = getSessionBean().getSelectedLanguages();
        Object result = selections.setProperty( language, ISBN );
    }
} // end method submit_action

Fig. 26.30 | Page bean that stores language selections in a SessionBean property. (Part 3 of 4.)
The submitButton's action handler (lines 280–319) stores the user's selections in the SessionBean and increments the number of selections made, if necessary. Line 294 retrieves from the SessionBean the Properties object that contains the user's selections. Line 295 adds the current selection to the Properties object. Method setProperty returns the value previously associated with the new key, or null if this key was not already stored in the Properties object. If adding the new property returns null, then the user has made a new selection. In this case, lines 302–303 increment the numSelections property in the SessionBean. Lines 309–317 and the languagesLink action handler (lines 323–334) control the components that will be displayed on the page, just as in the cookies examples.
Software Engineering Observation 26.2

A benefit of using SessionBean properties (rather than cookies) is that they can store any type of object (not just Strings) as attribute values. This provides you with increased flexibility in maintaining client state information.

Displaying Recommendations Based on Session Values

As in the cookies example, this application provides a link to Recommendations.jsp (Fig. 26.31), which displays a list of book recommendations based on the user’s language selections. It is identical to recommendation.jsp from the cookies example (Fig. 26.23).

```xml
<?xml version = "1.0" encoding = "UTF-8"?>
<jsp:root version = "1.2" xmlns:f = "http://java.sun.com/jsf/core"
xmlns:ui = "http://www.sun.com/web/ui">
<jsp:directive.page contentType = "text/html; charset = UTF-8"
pageEncoding = "UTF-8"/>
<f:view>
<ui:page binding = "#{Recommendations.page}" id = "page">
<ui:head binding = "#{Recommendations.html}" id = "html">
<ui:link binding = "#{Recommendations.link}" id = "link"
url = "/resources/stylesheet.css"/>
</ui:head>
<ui:body binding = "#{Recommendations.body}" id = "body"
style = "-rave-layout: grid">
<ui:form binding = "#{Recommendations.form}" id = "form">
<ui:label binding = "#{Recommendations.languageLabel}"
for = "booksListBox" id = "languageLabel" style = "font-size: 20px; font-weight: bold; left: 24px; top: 24px; position: absolute" text = "Recommendations"/>
<ui:listbox binding = "#{Recommendations.booksListBox}" id = "booksListBoxDefaultOptions.options" rows = "6"
style = "left: 24px; top: 72px; position: absolute; width: 360px"/>
<ui:hyperlink action = "case1" binding = "#{Recommendations.optionsLink}" id = "optionsLink"
style = "left: 24px; top: 192px; position: absolute" text = "Click here to choose another language."/>
</ui:form>
</ui:body>
</ui:html>
</ui:page>
</f:view>
</jsp:root>
```

Fig. 26.31 | JSP file that displays book recommendations based on language selections stored in session scope. (Part 1 of 2.)
26.7 Session Tracking

Figure 26.32 presents the page bean for Recommendations.jsp. Again, much of it is similar to the page bean used in the cookies example. We discuss only the new features.

```java
// Fig. 26.32: Recommendations.java
// Page bean that displays book recommendations based on a SessionBean property.
package session;

import com.sun.rave.web.ui.component.Body;
import com.sun.rave.web.ui.component.Form;
import com.sun.rave.web.ui.component.Head;
import com.sun.rave.web.ui.component.Html;
import com.sun.rave.web.ui.component.Link;
import com.sun.rave.web.ui.component.Page;
import javax.faces.FacesException;
import com.sun.rave.web.ui.component.Listbox;
import com.sun.rave.web.ui.component.Label;
import com.sun.rave.web.ui.component.Hyperlink;
import com.sun.rave.web.ui.model.DefaultOptionsList;
import java.util.Enumeration;
import com.sun.rave.web.ui.model.Option;
import java.util.Properties;

public class Recommendations extends AbstractPageBean {
    private int __placeholder;

    // Fig. 26.32: Recommendations.java
    // Page bean that displays book recommendations based on a SessionBean property.
    public class Recommendations extends AbstractPageBean {
        private int __placeholder;
    }
```
private void _init() throws Exception
{
    // empty body
} // end method _init

// To save space, we omitted the code in lines 31-150. The complete
// source code is provided with this chapter's examples.

public Recommendations()
{
    // empty constructor
} // end constructor

protected RequestBean getRequestBean() {
    return (RequestBean) getBean( "RequestBean" );
} // end method getRequestBean

protected ApplicationBean getApplicationBean() {
    return (ApplicationBean) getBean( "ApplicationBean" );
} // end method getApplicationBean

protected SessionBean getSessionBean() {
    return (SessionBean) getBean( "SessionBean" );
} // end method getSessionBean

public void init()
{
    super.init();
    try
    {
        _init();
    } // end try
    catch ( Exception e )
    {
        log( "Recommendations Initialization Failure", e );
        throw e instanceof FacesException ? (FacesException) e:
        new FacesException( e );
    } // end catch
} // end method init

public void preprocess()
{
    // empty body
} // end method preprocess

public void prerender()
{

Fig. 26.32 | Page bean that displays book recommendations based on a SessionBean property.
(Part 2 of 3.)
26.8 Wrap-Up

In this chapter, we introduced web application development using JavaServer Pages and JavaServer Faces in Java Studio Creator 2. We began by discussing the simple HTTP transactions that take place when you request and receive a web page through a web brows-

```java
//retrieve user's selections and number of selections made
Properties languages = getSessionBean().getSelectedLanguages();
Enumeration selectionsEnum = languages.propertyNames();
int numSelected = getSessionBean().getNumSelections();
Option [] recommendations;

if ( numSelected > 0 ) {
    recommendations = new Option[ numSelected ];
    for ( int i = 0; i < numSelected; i++ ) {
        String language = (String) selectionsEnum.nextElement();
        recommendations[ i ] = new Option( language +
            "How to Program. ISBN#:" +
            languages.getProperty( language ) );
    }
} else {
    recommendations = new Option[ 1 ];
    recommendations[ 0 ] = new Option( "No recommendations. " +
        "Please select a language." );
}
booksListBox.setItems( recommendations );
```

Fig. 26.32 | Page bean that displays book recommendations based on a SessionBean property. (Part 3 of 3.)

Line 194 retrieves the Properties object containing the user's selections from the SessionBean, and line 195 gets an enumeration of all of the keys in that Properties object. Line 196 retrieves the number of selections made from the SessionBean. If any selections were made, line 208 constructs an appropriately sized Option array to display the selections in the ui:listBox element of Recommendations.jsp. Lines 205–211 add each of the user's selections to this Option array. Line 207 gets the next key from the enumeration of keys, and lines 208–210 add a recommendation to the Option array.
er. We then discussed the three tiers (i.e., the client or top tier, the business logic or middle tier and the information or bottom tier) that comprise most web applications.

You learned the role of JSP files and page bean files, and the relationship between them. You learned how to use Java Studio Creator 2 to compile and execute web applications. You also learned how to build web applications visually using Java Studio Creator 2’s drag-and-drop capabilities.

We demonstrated several common JSF components used for displaying text and images on web pages. We also discussed validation components and custom validator methods, which allow you to ensure that user input satisfies certain requirements.

We discussed the benefits of maintaining user information across multiple pages of a website. We then demonstrated how you can include such functionality in a web application using either cookies or properties in the SessionBean class. In the next chapter, we continue our discussion of web application development. You’ll learn how to access a database from a web application, how to use several of the AJAX-enabled JSF components from Sun’s Java Blueprints and how to use virtual forms.

### 26.9 Web Resources

- [developers.sun.com/prodtech/javatools/jscreator](developers.sun.com/prodtech/javatools/jscreator)
  - Overviews Java Studio Creator 2 and includes articles, forums, product demonstrations and links to useful resources relevant to building web applications in Java Studio Creator 2.

- [developers.sun.com/prodtech/javatools/jscreator/index.jsp](developers.sun.com/prodtech/javatools/jscreator/index.jsp)
  - Sun’s Java Studio Creator center, has everything you need to get started. Download the IDE for free and check out the Learning tab for Java Studio Creator tutorials.

- [developers.sun.com/prodtech/javatools/jscreator/learning/tutorials/index.jsp](developers.sun.com/prodtech/javatools/jscreator/learning/tutorials/index.jsp)
  - Provides dozens of tutorials, ranging from tips on getting started with Java Studio Creator 2 to feature-specific instructions on how to use many facets of the IDE.

- [developers.sun.com/prodtech/javatools/jscreator/reference/docs/apis/](developers.sun.com/prodtech/javatools/jscreator/reference/docs/apis/)
  - The documentation for Java Studio Creator 2.

- [java.sun.com/javase/javaserverfaces/](java.sun.com/javase/javaserverfaces/)
  - This official Sun site provides the documentation for JavaServer Faces and links to relevant articles and tutorials.

- [www.netbeans.org/products/visualweb/](www.netbeans.org/products/visualweb/)
  - Get the Netbeans Visual Web Pack 5.5 for Netbeans 5.5 here.

- [java.sun.com/javase/5/docs/tutorial/doc/JSFPage.html#wp114889](java.sun.com/javase/5/docs/tutorial/doc/JSFPage.html#wp114889)
  - The Java EE 5 JavaServer Faces tutorial.

- [jsftutorials.net/](jsftutorials.net/)
  - Links to tutorials and general articles on JavaServer Faces.

- [javaserverfaces.dev.java.net](javaserverfaces.dev.java.net)
  - Download the latest version of Sun’s JavaServer Faces implementation.

- [java.sun.com/javase/javaserverfaces/reference/api/](java.sun.com/javase/javaserverfaces/reference/api/)
  - Tag Library, API, and Standard RenderKit documentation for all versions of JSF.

- [java.sun.com/javase/5/docs/tutorial/doc/JSFCustom.html](java.sun.com/javase/5/docs/tutorial/doc/JSFCustom.html)
  - A tutorial on building custom JSF components.

- [bpcatalog.dev.java.net/nonav/webtier/index.html](bpcatalog.dev.java.net/nonav/webtier/index.html)
  - The Java BluePrints solution catalog contains reusable code examples for designing web applications using JavaServer Faces and AJAX.
Summary

Section 26.1 Introduction
• Web-based applications create web content for web browser clients.
• AJAX helps web-based applications provide the interactivity and responsiveness that users typically expect of desktop applications.

Section 26.2 Simple HTTP Transactions
• Hypertext Transfer Protocol (HTTP) specifies a set of methods and headers that allow clients and servers to interact and exchange information in a uniform and reliable manner.
• In its simplest form, a web page is nothing more than an XHTML document containing markup that describes to a web browser how to display and format the document’s information.
• XHTML documents can contain hypertext data (hyperlinks) that link to different pages or to other parts of the same page when the user clicks the link.
• HTTP uses URIs (Uniform Resource Identifiers) to identify data on the Internet.
• URIs that specify the locations of documents are called URLs (Uniform Resource Locators). Common URLs refer to files or directories and can reference objects that perform complex tasks.
• A URL contains information that directs a browser to the resource that the user wishes to access. Computers that run web server software make such resources available.
• When given a URL, a web browser performs a simple HTTP transaction to retrieve and display the web page found at that address.
• The HTTP GET method indicates that the client wishes to obtain a resource from the server.
• HTTP headers provide information about the data sent to a client, such as the MIME type.
• Multipurpose Internet Mail Extensions (MIME) is an Internet standard that specifies data formats so that programs can interpret data correctly.

Section 26.3 Multitier Application Architecture
• Web-based applications are multitier (or n-tier) applications that divide functionality into separate tiers that typically reside on separate computers.
• The bottom tier (also called the data tier or the information tier) maintains the application’s data. This tier typically stores data in a relational database management system (RDBMS).
• The middle tier implements business logic, controller logic and presentation logic to control interactions between the application’s clients and the its data. The middle tier acts as an intermediary between data in the information tier and the application’s clients.
• The middle-tier controller logic processes client requests and retrieves data from the database.
• The middle-tier presentation logic processes data from the information tier and presents the content to the client.
• Web applications typically present data to clients as XHTML documents.
• Business logic in the middle tier enforces business rules and ensures that data is reliable before the server application updates the database or presents the data to users.
• Business rules dictate how clients can and cannot access application data, and how applications process data.
• The top tier, or client tier, is the application’s user interface, which gathers input and displays output. Users interact with the application through the user interface, typically in a web browser.
Chapter 26  Web Applications: Part 1

• In response to user actions, the client tier interacts with the middle tier to make requests and to retrieve data from the information tier. The client tier then displays the data retrieved for the user. The client tier never directly interacts with the information tier.

Section 26.4  Java Web Technologies
• Java web technologies continually evolve to provide developers with higher levels of abstraction and greater separation of the application’s tiers. This separation makes web applications more maintainable and extensible.
• Java Studio Creator 2 allows you to develop a web application’s GUI in a drag-and-drop design tool, while handling the business logic in separate Java classes.

Section 26.4.1 Servlets
• Servlets use the HTTP request-response model of communication between client and server.
• Servlets extend a server's functionality by allowing the server to generate dynamic content. A servlet container executes and interacts with servlets.
• Packages javax.servlet and javax.servlet.http contain the servlet classes and interfaces.
• The servlet container receives HTTP requests from a client and directs each request to the appropriate servlet. The servlet processes the request and returns an appropriate response to the client—usually in the form of an XHTML or XML document.
• All servlets implement the Servlet interface of package javax.servlet, which ensures that each servlet can execute in the framework provided by the servlet container. Interface Servlet declares methods used by the servlet container to manage the servlet’s life cycle.
• A servlet’s life cycle begins when the servlet container loads it into memory—usually in response to the first request for the servlet. The container invokes the servlet’s init method, which is called only once during a servlet’s life-cycle to initialize the servlet. After init completes execution, the servlet is ready to respond to its first request. All requests are handled by a servlet’s service method, which receives the request, processes it and sends a response to the client. Method service is called once per request. When the servlet container terminates the servlet, the servlet’s destroy method is called to release any resources held by the servlet.

Section 26.4.2 JavaServer Pages
• JavaServer Pages (JSP) are an extension of servlet technology. Each JSP is translated by the JSP container into a servlet.
• Unlike servlets, JSPs help you separate presentation from content.
• JavaServer Pages enable web application programmers to create dynamic content by reusing predefined components and by interacting with components using server-side scripting.
• JSP programmers can use special software components called JavaBeans and custom tag libraries that encapsulate complex, dynamic functionality.
• Custom tag libraries allow Java developers to hide code for database access and other complex operations in custom tags. To use such capabilities, you simply add the custom tags to the page. This simplicity enables web-page designers who are not familiar with Java to enhance web pages with powerful dynamic content and processing capabilities.
• The JSP classes and interfaces are located in packages javax.servlet.jsp and javax.servlet.jsp.tagext.
• There are four key components to JSPs—directives, actions, scripting elements and tag libraries.
• Directives are messages to the JSP container that enable you to specify page settings, to include content from other resources and to specify custom tag libraries for use in JSPs.
• Actions encapsulate functionality in predefined tags that programmers can embed in JSPs. Actions often are performed based on the information sent to the server as part of a particular client request. They also can create Java objects for use in JSPs.

• Scripting elements enable you to insert Java code that interacts with components in a JSP to perform request processing.

• Tag libraries enable programmers to create custom tags and web-page designers to manipulate JSP content without prior Java knowledge.

• The JavaServer Pages Standard Tag Library (JSTL) provides the functionality for many common web application tasks.

• JSPs can contain static content such as XHTML or XML markup, which is known as fixed-template data or fixed-template text. Any literal text in a JSP is translated to a String literal in the servlet representation of the JSP.

• When a JSP-enabled server receives the first request for a JSP, the JSP container translates the JSP into a servlet that handles the current request and future requests to the JSP.

• JSPs rely on the same request/response mechanism as servlets to process requests from and send responses to clients.

Section 26.4.3 JavaServer Faces

• JavaServer Faces (JSF) is a web application framework that simplifies the design of an application’s user interface and further separates a web application’s presentation from its business logic.

• A framework simplifies application development by providing libraries and sometimes software tools to help you organize and build your applications.

• JSF provides custom tag libraries containing user interface components that simplify web-page design. JSF also includes a set of APIs for handling component events.

• You design the look-and-feel of a page with JSF by adding custom tags to a JSP file and manipulating their attributes. You define the page’s behavior in a separate Java source-code file.

Section 26.4.4 Web Technologies in Java Studio Creator 2

• Java Studio Creator 2 web applications consist of one or more JSPs built in the JavaServer Faces framework. Each has the file-name extension .jsp and contains the web page’s GUI elements.

• Java Studio Creator 2 allows you to design pages visually by dragging and dropping JSF components onto a page; you can also customize a web page by editing its .jsp file manually.

• Every JSP file created in Java Studio Creator 2 represents a web page and has a corresponding JavaBeans class called the page bean.

• A JavaBean class must have a default (or no-argument) constructor, and get and set methods for all of its properties.

• The page bean defines properties for each of the page’s elements, and contains event handlers, page life-cycle methods and other supporting code for the web application.

• Every web application built with Java Studio Creator 2 has a page bean, a RequestBean, a SessionBean and an ApplicationBean.

• The RequestBean object is maintained in request scope—this object exists only for the duration of an HTTP request.

• A SessionBean object has session scope—the object exists throughout a user’s browsing session or until the session times out. There is a unique SessionBean object for each user.

• The ApplicationBean object has application scope—this object is shared by all instances of an application and exists as long as the application remains deployed on a web server. This object is
used for application-wide data storage or processing; only one instance exists for the application, regardless of the number of open sessions.

Section 26.5.1 Examining a JSP File

• Java Studio Creator 2 generates a JSP file in response to your interactions with the Visual Editor.
• All JSPs have a jsp:root element with a version attribute to indicate the version of JSP being used and one or more xmlns attributes. Each xmlns attribute specifies a prefix and a URL for a tag library, allowing the page to use tags specified in that library.
• All JSPs generated by Java Studio Creator 2 include the tag libraries for the JSF core components library, the JSF HTML components library, the JSP standard components library and the JSP user interface components library.
• The jsp:directive.page element's contentType attribute specifies the MIME type and the character set the page uses. The pageEncoding attribute specifies the character encoding used by the page source. These attributes help the client determine how to render the content.
• All pages containing JSF components are represented in a component tree with the root JSF element f:view (of type UIViewRoot). All JSF component elements are placed in this element.
• Many ui page elements have a binding attribute to bind their values to properties in the web application's JavaBeans. JSF Expression Language is used to perform these bindings.
• The ui:head element has a title attribute that specifies the page's title.
• A ui:link element can be used to specify the CSS stylesheet used by a page.
• A ui:body element defines the body of the page.
• A ui:form element defines a form in a page.
• A ui:staticText component displays text that does not change.
• JSP elements are mapped to XHTML elements for rendering in a browser. The same JSP element can map to different XHTML elements, depending on the client browser and the component's property settings.
• A ui:staticText component typically maps to an XHTML span element. A span element contains text that is displayed on a web page and is used to control the formatting of the text. The style attribute of a ui:staticText element will be represented as part of the corresponding span element's style attribute when the browser renders the page.

Section 26.5.2 Examining a Page Bean File

• Page bean classes inherit from class AbstractPageBean (package com.sun.rave.web.ui.appbase), which provides page life-cycle methods.
• Package com.sun.rave.web.ui.component includes classes for many basic JSF components.
• A ui:staticText component is a StaticText object (package com.sun.rave.web.ui.component).

Section 26.5.3 Event-Processing Life Cycle

• Java Studio Creator 2's application model places several methods (init, preprocess, prerender and destroy) in the page bean that tie into the JSF event-processing life cycle. These methods represent four major stages—initialization, preprocessing, prerendering and destruction.
• The init method is called by the JSP container the first time the page is requested and on postbacks. A postback occurs when form data is submitted, and the page and its contents are sent to the server to be processed.
• Method init invokes its superclass version, then tries to call the method _init, which handles component initialization tasks.
• The preprocess method is called after init, but only if the page is processing a postback. The prerender method is called just before a page is rendered by the browser. This method should be used to set component properties; properties that are set sooner (such as in method init) may be overwritten before the page is actually rendered by the browser.

• The destroy method is called after the page has been rendered, but only if the init method was called. This method handles tasks such as freeing resources used to render the page.

Section 26.5.4 Relationship Between the JSP and Page Bean Files
• The page bean has a property for every element that appears in the JSP file.

Section 26.5.5 Examining the XHTML Generated by a Java Web Application
• To create a new web application, select File > New Project... to display the New Project dialog. In this dialog, select Web in the Categories pane, JSF Web Application in the Projects pane and click Next. Specify the project name and location. Click Finish to create the web application project.

• Java Studio Creator 2 creates a single web page named Page1 when you create a new project. This page is open by default in the Visual Editor in Design mode when the project first loads. As you drag and drop new components onto the page, Design mode allows you to see how your page will be rendered in the browser. The JSP file for this page, named Page1.jsp, can be viewed by clicking the JSP button at the top of the Visual Editor or by right clicking anywhere in the Visual Editor and selecting Edit JSP Source.

• To open the corresponding page bean file, click the Java button at the top of the Visual Editor or right click anywhere in the Visual Editor and select Edit Page1 Java Source.

• The Refresh button redraws the page in the Visual Editor.

• The Target Browser Size drop-down list allows you to specify the optimal browser resolution for viewing the page and allows you to see what the page will look like in different screen resolutions.

• The Projects window in the lower-right corner of the IDE displays the hierarchy of all the project’s files. The Web Pages node contains the JSP files and includes the resources folder, which contains the project’s CSS stylesheet and any other files the pages may need to display properly (e.g., images). The Java source code, including the page bean file for each web page and the application, session and request scope beans, can be found under the Source Packages node.

• The Page Navigation file defines rules for navigating the project’s pages based on the outcome of user-initiated events, such as clicking a button or a link. This file can also be accessed by right clicking in the Visual Editor while in Design mode and selecting Page Navigation....

• Methods init, preprocess, prerender and destroy are overridden in each page bean. Other than method init, these methods are initially empty. They serve as placeholders for you to customize the behavior of your web application.

• Typically, you’ll want to rename the JSP and Java files in your project, so that their names are relevant to your application. To do so, right click the JSP file in the Projects Window and select Rename to display the Rename dialog. Enter the new file name. If Preview All Changes is checked, the Refactoring Window will appear at the bottom of the IDE when you click Next >. No changes will be made until you click Do Refactoring in the Refactoring Window. If you do not preview the changes, refactoring occurs when you click Next > in the Rename dialog.

• Refactoring is the process of modifying source code to improve its readability and reusability without changing its behavior—for example, by renaming methods or variables, or breaking long methods into shorter ones. Java Studio Creator 2 has built-in refactoring tools that automate some refactoring tasks.
Chapter 26  Web Applications: Part I

• To add a title, open the JSP file in Design mode. In the Properties window, enter the new title next to the Title property and press Enter.

• To add components to a page, drag and drop them from the Palette onto the page in Design mode. Each component is an object that has properties, methods and events. You can set these properties and events in the Properties window or programmatically in the page bean file. Get and set methods are added to the page bean file for each component you add to the page.

• Components are rendered using absolute positioning, so that they appear exactly where they are dropped on the page.

• Java Studio Creator 2 is a WYSIWYG (What You See Is What You Get) editor—whenever you make a change to a web page in Design mode, the IDE creates the markup (visible in JSP mode) necessary to achieve the desired visual effects seen in Design mode.

• After designing the user interface, you can modify the page bean to add your business logic.

• The Outline window displays the page bean and the request, session and application scope beans. Clicking an item in the page bean’s component tree selects the item in the Visual Editor.

• Select Build > Build Main Project then Run > Run Main Project to run the application.

• You can run a project that has already been built by pressing the Run Main Project icon ( ) in the toolbar at the top of the IDE.

• If changes are made to a project, the project must be rebuilt before the changes will be reflected when the application is viewed in a browser.

• Press F5 to build the application, then run it in debug mode. If you type <Ctrl> F5, the program executes without debugging enabled.

Section 26.5.6 Building a Web Application in Java Studio Creator 2

• The Grid Panel component allows the designer to specify the number of columns the grid should contain. Components may then be dropped anywhere inside the panel, and they will automatically be repositioned into evenly spaced columns in the order in which they are dropped. When the number of components exceeds the number of columns, the panel moves the additional components to a new row.

• An Image component (of class Image) inserts an image into a web page. Images to be displayed on a web page must be placed in the project’s resources folder. To add images to the project, drop an Image component onto the page and click the ellipsis button next to the url property in the Properties window. This opens a dialog in which you can select the image to display.

• Text Fields allow you to obtain text input from the user.

• Note that the order in which components are dragged to the page is important, because their JSP tags will be added to the JSP file in that order. Tabbing between components navigates the components in the order in which the JSP tags occur in the JSP file. If you would like the user to navigate the components in a certain order, you should drag them onto the page in that order. Alternatively, you can set each input field’s Tab Index property in the Properties window. A component with a tab index of 1 will be the first in the tab sequence.

• A Drop Down List displays a list from which the user can make a selection. This object can be configured by right clicking the drop-down list in Design mode and selecting Configure Default Options, which opens the Options Customizer dialog box to add options to the list.

• A Hyperlink component of class Hyperlink adds a link to a web page. The url property of this component specifies the resource that is requested when a user clicks the hyperlink.

• A RadioButton Group component of class RadioButtonGroup provides a series of radio buttons from which the user can select only one. The options can be edited by right clicking the component and selecting Configure Default Options.
Summary

A Button is a JSF component of class Button that triggers an action when clicked. A Button component typically maps to an input XHTML element with attribute type set to submit.

Section 26.6.2 Validation Using Validator Components and Custom Validators

• Validation helps prevent processing errors due to incomplete or improperly formatted user input.
• A Length Validator determines whether a field contains an acceptable number of characters.
• Double Range Validators and Long Range Validators determine whether numeric input falls within acceptable ranges.
• Package javax.faces.validators contains the classes for these validators.
• Label components describe other components and can be associated with user input fields by setting their for property.
• Message components display error messages when validation fails.
• To associate a Label or Message component with another component, hold the Ctrl and Shift keys, then drag the label or message to the appropriate component.
• Set the required property of a component to ensure that the user enters data for it.
• If you add a validator component or custom validator method to an input field, the field’s required property must be set to true for validation to occur.
• In the Visual Editor the label for a required field is automatically marked by a red asterisk.
• If a user submits a form with an empty text field for which a value is required, the default error message for that field will be displayed in its associated ui:message component.
• To edit a Double Range Validator’s or a Long Range Validator’s properties, click its node in the Outline window in Design mode and set the maximum and minimum properties in the Properties window.
• It is possible to limit the length of user input without using validation by setting a TextField’s maxLength property.
• Matching user input against a regular expression is an effective way to ensure that the input is properly formatted.
• Java Studio Creator 2 does not provide components for validation using regular expressions, but you can add your own custom validator methods to the page bean file.
• To add a custom validator method to an input component, right click the component and select Edit Event Handler > validate to create a validation method for the component in the page bean file.

Section 26.7 Session Tracking

• Personalization makes it possible for e-businesses to communicate effectively with their customers and also improves the user’s ability to locate desired products and services.
• A trade-off exists between personalized e-business service and protection of privacy. Some consumers embrace the idea of tailored content, but others fear the possible adverse consequences if the information they provide to e-businesses is released or collected by tracking technologies.
• To provide personalized services to consumers, e-businesses must be able to recognize clients when they request information from a site. Unfortunately, HTTP is a stateless protocol—it does not support persistent connections that would enable web servers to maintain state information regarding particular clients. So, web servers cannot determine whether a request comes from a particular client or whether a series of requests comes from one or several clients.
• To help the server distinguish among clients, each client must identify itself to the server. Tracking individual clients, known as session tracking, can be achieved in a number of ways. One popular technique uses cookies; another uses the SessionBean object.
With “hidden” form elements, a web form can write session-tracking data into a form in the web page that it returns to the client in response to a prior request. When the user submits the form in the new web page, all the form data, including the "hidden" fields, is sent to the form handler on the web server. With URL rewriting, the web server embeds session-tracking information directly in the URLs of hyperlinks that the user clicks to send subsequent requests.

Section 26.7.1 Cookies
• A cookie is a piece of data typically stored in a text file on the user’s computer. A cookie maintains information about the client during and between browser sessions.
• The first time a user visits the website, the user’s computer might receive a cookie; this cookie is then reactivated each time the user revisits that site. The collected information is intended to be an anonymous record containing data that is used to personalize the user’s future visits to the site.
• Every HTTP-based interaction between a client and a server includes a header containing information either about the request (when the communication is from the client to the server) or about the response (when the communication is from the server to the client).
• When a page receives a request, the header includes information such as the request type and any cookies that have been sent previously from the server to be stored on the client machine. When the server formulates its response, the header information contains any cookies the server wants to store on the client computer and other information, such as the MIME type of the response.
• A cookie’s expiration date determines how long the cookie remains on the client’s computer. If you do not set a cookie’s expiration date, the web browser maintains the cookie for the browsing session’s duration. Otherwise, it maintains the cookie until the expiration date.
• Setting the action handler for a **Hyperlink** enables you to respond to a click without redirecting the user to another page.
• To add an action handler to a **Hyperlink** that should also direct the user to another page, you must add a rule to the **Page Navigation** file. To edit this file, right click in the Visual Designer and select **Page Navigation…**, then drag the appropriate **Hyperlink** to the destination page.
• A cookie object is an instance of class **Cookie** in package **javax.servlet.http**.
• An object of class **HttpServletResponse** (from package **javax.servlet.http**) represents the response. This object can be accessed by invoking the method `getExternalContext` on the page bean, then invoking `getResponse` on the resulting object.
• An object of class **HttpServletRequest** (from package **javax.servlet.http**) represents the request. This object can be obtained by invoking method `getExternalContext` on the page bean, then invoking `getRequest` on the resulting object.
• **HttpServletRequest** method `getCookies` returns an array of the cookies previously written to the client.
• A web server cannot access cookies created by servers in other domains.

Section 26.7.2 Session Tracking with the SessionBean Object
• You can perform session tracking with the **SessionBean** class that is provided in each web application created with Java Studio Creator 2. When a new client requests a web page in the project, a **SessionBean** object is created.
• The **SessionBean** can be accessed throughout a session by invoking the method `getSessionBean` on the page bean. You can then use the **SessionBean** object to access stored session properties.
• To store information in the **SessionBean**, add properties to the **SessionBean** class. To add a property, right click the **SessionBean** node in the **Outline** window and select **Add > Property** to display the **New Property Pattern** dialog. Configure the property and click **OK** to create it.
### Terminology

<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>absolute positioning</td>
<td></td>
</tr>
<tr>
<td>AbstractPageBean</td>
<td></td>
</tr>
<tr>
<td>action attribute of XHTML element form</td>
<td></td>
</tr>
<tr>
<td>action in a JSP</td>
<td></td>
</tr>
<tr>
<td>ApplicationBean</td>
<td></td>
</tr>
<tr>
<td>bottom tier</td>
<td></td>
</tr>
<tr>
<td>business logic</td>
<td></td>
</tr>
<tr>
<td>business rule</td>
<td></td>
</tr>
<tr>
<td>Button</td>
<td>JSF component</td>
</tr>
<tr>
<td>client tier</td>
<td></td>
</tr>
<tr>
<td>com.sun.rave.web.ui.component</td>
<td></td>
</tr>
<tr>
<td>component tree</td>
<td></td>
</tr>
<tr>
<td>controller logic</td>
<td></td>
</tr>
<tr>
<td>cookie</td>
<td></td>
</tr>
<tr>
<td>custom tag library</td>
<td></td>
</tr>
<tr>
<td>custom tag</td>
<td></td>
</tr>
<tr>
<td>data tier</td>
<td></td>
</tr>
<tr>
<td>Design mode</td>
<td></td>
</tr>
<tr>
<td>destroy event-processing life-cycle method</td>
<td></td>
</tr>
<tr>
<td>directive in a JSP</td>
<td></td>
</tr>
<tr>
<td>DNS (domain name system) server</td>
<td></td>
</tr>
<tr>
<td>DNS lookup</td>
<td></td>
</tr>
<tr>
<td>Double Range Validator</td>
<td>JSF component</td>
</tr>
<tr>
<td>Drop Down List</td>
<td>JSF component</td>
</tr>
<tr>
<td>end tag</td>
<td></td>
</tr>
<tr>
<td>escaped property</td>
<td></td>
</tr>
<tr>
<td>event-processing life cycle</td>
<td></td>
</tr>
<tr>
<td>expiration date of a cookie</td>
<td></td>
</tr>
<tr>
<td>fixed-template data</td>
<td></td>
</tr>
<tr>
<td>fixed-template text</td>
<td></td>
</tr>
<tr>
<td>framework</td>
<td></td>
</tr>
<tr>
<td>GET HTTP request</td>
<td></td>
</tr>
<tr>
<td>Grid Panel</td>
<td>JSF component</td>
</tr>
<tr>
<td>hidden input in an XHTML form</td>
<td></td>
</tr>
<tr>
<td>host</td>
<td></td>
</tr>
<tr>
<td>hostname</td>
<td></td>
</tr>
<tr>
<td>HTTP header</td>
<td></td>
</tr>
<tr>
<td>HTTP method</td>
<td></td>
</tr>
<tr>
<td>Hyperlink</td>
<td>JSF component</td>
</tr>
<tr>
<td>hyperlink</td>
<td></td>
</tr>
<tr>
<td>hypertext</td>
<td></td>
</tr>
<tr>
<td>Image</td>
<td>JSF component</td>
</tr>
<tr>
<td>information tier</td>
<td></td>
</tr>
<tr>
<td>init event-processing life-cycle method</td>
<td></td>
</tr>
<tr>
<td>IP address</td>
<td></td>
</tr>
<tr>
<td>JavaBeans</td>
<td></td>
</tr>
<tr>
<td>Java BluePrints</td>
<td></td>
</tr>
<tr>
<td>Java Studio Creator 2</td>
<td></td>
</tr>
<tr>
<td>javax.servlet</td>
<td></td>
</tr>
<tr>
<td>javax.servlet.http</td>
<td>package</td>
</tr>
<tr>
<td>JSP (JavaServer Pages)</td>
<td></td>
</tr>
<tr>
<td>JSP container</td>
<td></td>
</tr>
<tr>
<td>JSTL (JSP Standard Tag Library)</td>
<td></td>
</tr>
<tr>
<td>Label</td>
<td>JSF component</td>
</tr>
<tr>
<td>Length Validator</td>
<td>JSF component</td>
</tr>
<tr>
<td>List Box</td>
<td>JSF component</td>
</tr>
<tr>
<td>localhost</td>
<td></td>
</tr>
<tr>
<td>Long Range Validator</td>
<td>JSF component</td>
</tr>
<tr>
<td>Message</td>
<td>JSF component</td>
</tr>
<tr>
<td>method attribute of XHTML element form</td>
<td></td>
</tr>
<tr>
<td>middle tier</td>
<td></td>
</tr>
<tr>
<td>MIME (Multipurpose Internet Mail Extensions)</td>
<td></td>
</tr>
<tr>
<td>multtier application</td>
<td></td>
</tr>
<tr>
<td>n-tier application</td>
<td></td>
</tr>
<tr>
<td>Outline window</td>
<td></td>
</tr>
<tr>
<td>page bean</td>
<td></td>
</tr>
<tr>
<td>Palette</td>
<td></td>
</tr>
<tr>
<td>personalization</td>
<td></td>
</tr>
<tr>
<td>postback</td>
<td></td>
</tr>
<tr>
<td>preprocess event-processing life-cycle method</td>
<td></td>
</tr>
<tr>
<td>prerender event-processing life-cycle method</td>
<td></td>
</tr>
<tr>
<td>presentation logic</td>
<td></td>
</tr>
<tr>
<td>Radio Button Group</td>
<td>JSF component</td>
</tr>
<tr>
<td>refactoring</td>
<td></td>
</tr>
<tr>
<td>rendered property</td>
<td></td>
</tr>
<tr>
<td>rendering XHTML in a web browser</td>
<td></td>
</tr>
<tr>
<td>RequestBean</td>
<td></td>
</tr>
<tr>
<td>required property</td>
<td></td>
</tr>
<tr>
<td>scripting element in a JSP</td>
<td></td>
</tr>
<tr>
<td>service method of Interface Servlet</td>
<td></td>
</tr>
<tr>
<td>servlet</td>
<td></td>
</tr>
<tr>
<td>Servlet interface</td>
<td></td>
</tr>
<tr>
<td>servlet container</td>
<td></td>
</tr>
<tr>
<td>SessionBean</td>
<td></td>
</tr>
<tr>
<td>session tracking</td>
<td></td>
</tr>
<tr>
<td>span element</td>
<td></td>
</tr>
<tr>
<td>start tag</td>
<td></td>
</tr>
<tr>
<td>Static Text</td>
<td>JSF component</td>
</tr>
<tr>
<td>Sun Application Server 8</td>
<td></td>
</tr>
<tr>
<td>tag extension mechanism</td>
<td></td>
</tr>
<tr>
<td>tag library</td>
<td></td>
</tr>
<tr>
<td>Text Field</td>
<td>JSF component</td>
</tr>
<tr>
<td>three-tier web-based application</td>
<td></td>
</tr>
</tbody>
</table>
Self-Review Exercises

26.1 State whether each of the following is true or false. If false, explain why.
   a) Every JSP web page created in Java Studio Creator 2 has its own ApplicationBean, SessionBean, and RequestBean files.
   b) Event-processing life-cycle method init is invoked every time a page loads.
   c) Every component on a JSP web page is bound to a property in the Java page bean file.
   d) A single JSF component may have multiple validation components placed on it.
   e) If no expiration date is set for a cookie, that cookie will be destroyed at the end of the browser session.
   f) Each JSF component maps to exactly one corresponding XHTML element.
   g) Expressions in the JSF Expression Language syntax are delimited by <!-- and -->.
   h) The SessionBean can store only primitive properties and properties of type String.

26.2 Fill in the blanks in each of the following statements:
   a) Web applications contain three basic tiers: ________, ________, and ________.
   b) The ________ JSF component is used to display error messages if validation fails.
   c) A component that checks the input in another component before submitting that input to the server is called a(n) ________.
   d) Every page bean class inherits from class ________.
   e) When a page loads the first time, the ________ event occurs first, followed by the ________ event.
   f) The ________ file contains the functionality for a JSP.
   g) ________ can be used in a custom validator method to validate the format of user input.
   h) The array of Cookie objects stored on the client can be obtained by calling getCookies on the ________ object.
   i) In a multitier application, the ________ tier controls interactions between the application’s clients and the application’s data.

Answers to Self-Review Exercises

26.1 a) False. If an application contains multiple JSPs, those JSPs will share the scoped data beans. b) False. init is invoked the first time the page is requested, but not on page refreshes. c) True. d) True. e) True. f) False. A web component can map to a group of XHTML elements—JSPs can generate complex XHTML markup from simple components. g) False. #{ and } delimit JSF Expression Language statements. h) False. The scoped data beans may store any type of property.

26.2 a) bottom (information), middle (business logic), top (client). b) Message. c) validator. d) AbstractPageBean. e) init, prerender. f) page bean. g) regular expression. h) Request (HttpServletRequest). i) middle.
Exercises

26.3 *(WebTime Modification)* Modify the WebTime example to contain drop-down lists that allow the user to modify such Static Text component properties as background-color, color and font-size. Configure these drop-down lists so that the page refreshes whenever the user makes a selection. When the page reloads, it should reflect the specified changes to the properties of the Static Text displaying the time.

26.4 *(Registration Form Modification)* Modify the WebComponents application to add functionality to the Register button. When the user clicks Submit, validate all input fields to make sure the user has filled out the form completely and entered a valid email address and phone number. Then, direct the user to another page that displays a message indicating successful registration and echoes back the user’s registration information.

26.5 *(Page Hit Counter with Cookies)* Create a JSP that uses a persistent cookie (i.e., a cookie with an expiration date in the future) to keep track of how many times the client computer has visited the page. Use the setMaxAge method to cause the cookie to remain on the client’s computer for one month. Display the number of page hits (i.e., the cookie’s value) every time the page loads.

26.6 *(Page Hit Counter with ApplicationBean)* Create a JSP that uses the ApplicationBean to keep track of how many times a page has been visited. *[Note: if you were to deploy this page on the web, it would count the number of times that any computer requested the page, unlike in the previous exercise.]* Display the number of page hits (i.e., the value of an int property in the ApplicationBean) every time the page loads.
27

Web Applications: Part 2

OBJECTIVES

In this chapter you will learn:

■ To use data providers to access databases from web applications built in Java Studio Creator 2.
■ The basic principles and advantages of Ajax technology.
■ To include Ajax-enabled JSF components in a Java Studio Creator 2 web application project.
■ To configure virtual forms that enable subsets of a form’s input components to be submitted to the server.

Whatever is in any way beautiful hath its source of beauty in itself, and is complete in itself; praise forms no part of it.
—Marcus Aurelius Antoninus

There is something in a face, An air, and a peculiar grace, Which boldest painters cannot trace.
—William Somerville

Cato said the best way to keep good acts in memory was to refresh them with new.
—Francis Bacon

I never forget a face, but in your case I’ll make an exception.
—Groucho Marx

Painting is only a bridge linking the painter’s mind with that of the viewer.
—Eugène Delacroix
27.1 Introduction

This chapter continues our discussion of web application development with several advanced concepts. We discuss accessing, updating and searching databases in a web application, adding virtual forms to web pages to enable subsets of a form’s input components to be submitted to the server, and using Ajax-enabled component libraries to improve application performance and component responsiveness.

We present a single address book application developed in three stages to illustrate these concepts. The application is backed by a JavaDB database for storing the contact names and their addresses.

The address book application presents a form that allows the user to enter a new name and address to store in the address book and displays the contents of the address book in table format. It also provides a search form that allows the user to search for a contact and, if found, display the contact’s address on a map. The first version of this application demonstrates how to add contacts to the database and how to display the list of contacts in a JSF Table component. In the second version, we add an Ajax-enabled Auto Complete Text Field component and enable it to suggest a list of contact names as the user types. The last version allows you to search the address book for a contact and display the corresponding address on a map using the Ajax-enabled MapViewer component that is powered by Google Maps (maps.google.com).

As in Chapter 26, the examples in this chapter were developed in Java Studio Creator 2.0. We installed a supplementary component library—the Java BluePrints Ajax component library—which provides the Ajax-enabled components used in the address book application. Instructions for installing this library are included in Section 27.3.1.
27.2 Accessing Databases in Web Applications

Many web applications access databases to store and retrieve persistent data. In this section, we build a web application that uses a Java DB database to store contacts in the address book and display contacts from the address book on a web page.

The web page enables the user to enter new contacts in a form. This form consists of Text Field components for the contact’s first name, last name, street address, city, state and zip code. The form also has a Submit button to send the data to the server and a Clear button to reset the form’s fields. The application stores the address book information in a database named AddressBook, which has a single table named Addresses. (We provide this database in the examples directory for this chapter. You can download the examples from www.deitel.com/books/jhtp7). This example also introduces the Table JSF component, which displays the addresses from the database in tabular format. We show how to configure the Table component shortly.

27.2.1 Building a Web Application That Displays Data from a Database

We now explain how to build the AddressBook application’s GUI and set up a data binding that allows the Table component to display information from the database. We present the generated JSP file later in the section, and we discuss the related page bean file in Section 27.2.2. To build the AddressBook application, perform the following steps:

Step 1: Creating the Project

In Java Studio Creator 2, create a JSF Web Application project named AddressBook. Rename the JSP and page bean files to AddressBook using the refactoring tools.

Step 2: Creating the Form for User Input

In Design mode, add a Static Text component to the top of the page that reads “Add a contact to the address book:” and use the component’s style property to set the font size to 18px. Add six Text Field components to the page and rename them fnameTextField, lnameTextField, streetTextField, cityTextField, stateTextField and zipTextField. Set each Text Field’s required property to true by selecting the Text Field, then clicking the required property’s checkbox. Label each Text Field with a Label component and associate the Label with its corresponding Text Field. Finally, add a Submit and a Clear button. Set the Submit button’s primary property to true to make it stand out more on the page than the Clear button and to allow the user to submit a new contact by pressing Enter rather than by clicking the Submit button. Set the Clear button’s reset property to true to prevent validation when the user clicks the Clear button. Since we are clearing the fields, we don’t want to ensure that they contain information. We discuss the action handler for the Submit button after we present the page bean file. The Clear button does not need an action handler method, because setting the reset property to true automatically configures the button to reset all of the page’s input fields. When you have finished these steps, your form should look like Fig. 27.1.

Step 3: Adding a Table Component to the Page

Drag a Table component from the Basic section of the Palette to the page and place it just below the two Button components. Name it addressesTable. The Table component formats and displays data from database tables. In the Properties window, change the Table’s
27.2 Accessing Databases in Web Applications

We show how to configure the Table to interact with the AddressBook database shortly.

**Step 4: Adding a Database to a Java Studio Creator 2 Web Application**

For this example, we use a Java DB database named AddressBook with a single database table named Addresses. To make this database available in your projects, copy the AddressBook folder from the chapter’s examples folder into your Java Studio Creator 2 installation folder’s SunAppServer8\derby\databases folder.

To use a database in a Java Studio Creator 2 web application, you must first start the IDE’s bundled database server, which allows database connections to be used in Java Studio Creator 2 projects. The server includes drivers for many databases, including Java DB. Click the Servers tab below the File menu, right click Bundled Database Server at the bottom of the Servers window and select Start Bundled Database Server. You can now use databases that run on this server in your applications.

To add the AddressBook database to this project, right click the Data Sources node at the top of the Servers window and select Add Data Source.... In the Add Data Source dialog (Fig. 27.2), enter AddressBook for the data source name and select Derby for the server type. (Recall from Chapter 25 that Java DB is the Sun-branded version of Apache Derby.) The user ID and password for this database are both jhtp7. For the database

![Fig. 27.1](image1)

*Fig. 27.1 | AddressBook application form for adding a contact.*

![Fig. 27.2](image2)

*Fig. 27.2 | Dialog to add a data source.*
URL, enter `jdbc:derby://localhost:21527/AddressBook`. This URL indicates that the database resides on the local machine and accepts connections on port 21527. Click the Select button to choose a table that will be used to validate the database. In the dialog that appears, choose the `JHTP7.ADDRESSES` table, as this is the only table in the database. Click Select to close this dialog, then click Add to add the database as a data source for the project and close the dialog. [Note: Java Studio Creator 2 displays database and table names in capital letters.]

**Step 5: Binding the Table Component to the Addresses Table of the AddressBook Database**

Now that we’ve configured a data source for the Addresses database table, we can configure the Table component to display the AddressBook data. Simply drag the database table from the Servers tab and drop it on the Table component to create the binding.

If you need more precise control over the columns to display, you can bind to a database table as follows: Right click the Table component and select Bind to Data to display the Table Layout dialog. Click the Add Data Provider... button to display the Add Data Provider dialog, which contains a list of the available data sources. Expand the AddressBook node, expand the Tables node, select ADDRESSES and click Add. The Table Layout dialog now displays a list of the columns in the Addresses database table (Fig. 27.3). All of the items under the Selected heading will be displayed in the Table. To remove a column from the Table, you can select it and click the < button. Since we want to display all of these columns in our Table, simply click OK to exit the dialog.

By default, the Table uses the database table’s column names in all uppercase letters as headings. To change these headings, select a column and edit its headerText property in the Properties window. To select a column, expand the addressesTable node in the Outline window (while in Design mode), then select the appropriate column object. We also changed the id property of each column to make the variable names in the code more readable. In Design mode, your Table’s column heads should appear as in Fig. 27.4.

An address book might contain many contacts, so we’d like to display only a few at a time. Clicking the checkbox next to the table’s paginationControls property in the Properties window configures this Table for automatic pagination. Five rows will be displayed at a time, and buttons for moving forward and backward between groups of five contacts.

Fig. 27.3  |  Dialog for binding to the Addresses table.
will be added to the bottom of the Table. (You may also use the Table Layout dialog’s Options tab to select the pagination and number of rows. To view this tab, right click the Table, select Table Layout..., then click the Options tab.) Next, set the addressesTable’s internalVirtualForm property. Virtual forms allow subsets of a form’s input components to be submitted to the server. Setting this property prevents the pagination control buttons on the Table from submitting the Text Fields on the form every time the user wishes to view the next group of contacts. Virtual forms are discussed in Section 27.4.1.

Notice that binding the Table to a data provider added a new addressesDataProvider object (an instance of class CachedRowSetDataProvider) to the AddressBook node in the Outline window. A CachedRowSetDataProvider provides a scrollable RowSet that can be bound to a Table component to display the RowSet’s data. This data provider is a wrapper for a CachedRowSet object. If you click the addressesDataProvider element in the Outline window, you will see in the Properties window that its CachedRowSet property is set to addressesRowSet, an object that implements interface CachedRowSet.

**Step 6: Modifying addressesRowSet’s SQL Statement**

The CachedRowSet object wrapped by our addressesDataProvider is configured by default to execute a SQL query that selects all the data in the Addresses table of the AddressBook database. You can edit this SQL query by expanding the SessionBean node in the Outline window and double clicking the addressesRowSet element to open the query editor window (Fig. 27.5). We’d like to edit the SQL statement so that records with duplicate last names are sorted by last name, then by first name. To do this, click in the Sort Type column next to the LASTNAME row and select Ascending. Then, repeat this for the FIRSTNAME row. Notice that the expression

```
ORDER BY JHTP7.ADDRESSES.LASTNAME ASC,
         JHTP7.ADDRESSES.FIRSTNAME ASC
```

was added to the SQL statement at the bottom of the editor.

**Step 7: Adding Validation**

It is important to validate the form data on this page to ensure that the data can be successfully inserted into the AddressBook database. All of the database’s columns are of type varchar and have length restrictions. For this reason, you should either add a LengthValidator to each Text Field component or set each Text Field component’s maxLength property. We chose to set the maxLength property of each. The first name, last name, street, city, state and zip code Text Field components may not exceed 20, 30, 100, 30, 2 and 5 characters, respectively.
Finally, drag a **Message Group** component onto your page to the right of the **Table**. A **Message Group** component displays system messages. We use this component to display an error message when an attempt to add a contact to the database fails. Set the **Message Group**’s `showGlobalOnly` property to `true` to prevent component-level validation error messages from being displayed here.

**JSP File for a Web Page that Interacts with a Database**

The JSP file for the application is shown in Fig. 27.6. This file contains a large amount of generated markup for components you learned in Chapter 26. We discuss the markup for only the components that are new in this example.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!-- Fig. 27.6: AddressBook.jsp -->
<!-- AddressBook JSP with an add form and a Table JSF component. -->
<jsp:root version="1.2" xmlns:f="http://java.sun.com/jsf/core"
    xmlns:ui="http://www.sun.com/web/ui">
  <jsp:directive.page contentType="text/html;charset=UTF-8"
      pageEncoding="UTF-8"/>
  <f:view>
    <ui:page binding="#{AddressBook.page1}" id="page1">
      <ui:html binding="#{AddressBook.html1}" id="html1">
        <ui:head binding="#{AddressBook.head1}" id="head1">
          <ui:link binding="#{AddressBook.link1}" id="link1"
                  url="/resources/stylesheet.css"/>
        </ui:head>
      </ui:html>
    </ui:page>
  </f:view>
</jsp:root>
```

**Fig. 27.6 | AddressBook JSP with an add form and a Table JSF component (Part 1 of 5.)**
27.2 Accessing Databases in Web Applications

Fig. 27.6 | AddressBook JSP with an add form and a Table JSF component (Part 2 of 5.)
Fig. 27.6 | AddressBook JSP with an add form and a Table JSF component (Part 3 of 5.)
27.2 Accessing Databases in Web Applications

```
<ui:tableColumn binding="#{AddressBook.stateColumn}" headerText="State"
    id="stateColumn" sort="ADDRESSES.STATE">
    <ui:staticText binding="#{AddressBook.stateHeader}" id="stateHeader" text="#{currentRow.value['ADDRESSES.STATE']}"/>
</ui:tableColumn>

<ui:tableColumn binding="#{AddressBook.zipColumn}" headerText="Zip"
    id="zipColumn" sort="ADDRESSES.ZIP">
    <ui:staticText binding="#{AddressBook.zipHeader}" id="zipHeader" text="#{currentRow.value['ADDRESSES.ZIP']}"/>
</ui:tableColumn>
</ui:tableRowGroup>
</ui:table>

<ui:messageGroup binding="#{AddressBook.messageGroup1}" id="messageGroup1" showGlobalOnly="true" style="position: absolute; left: 24px; top: 624px"/>
</ui:form>
</ui:body>
</ui:html>
</jsp:root>
```

Fig. 27.6 | AddressBook JSP with an add form and a Table JSF component (Part 4 of 5.)
Lines 21–72 contain the JSF components that comprise the form that gathers user input. Lines 73–199 define the Table element (ui:table) that displays address information from the database. Lines 79–140 (not shown here) contain JavaScript functions generated by the IDE to handle Table actions, such as a change in the current row’s state. JSF Tables may have multiple groups of rows displaying different data. This Table has a single ui:tableRowGroup with a start tag in lines 142–146. The row group’s sourceData attribute is bound to our addressesDataProvider and given the variable name currentRow. The row group also defines the Table’s columns. Each ui:tableColumn element contains a ui:staticText element with its text attribute bound to a column in the data provider currentRow. These ui:staticText elements enable the Table to display each row’s data.

Session Bean for the AddressBook Application
Figure 27.7 displays the SessionBean1.java file generated by Java Studio Creator 2 for the AddressBook application. The CachedRowSet that the Table component’s data provider uses to access the AddressBook database is a property of this class (lines 31–41).

```java
1 // Fig. 27.7: SessionBean1.java
2 // Session bean that initializes the data source for the
3 // AddressBook database.
```

Fig. 27.6 | AddressBook JSP with an add form and a Table JSF component (Part 5 of 5.)
27.2 Accessing Databases in Web Applications

The _init method (lines 14–29) configures addressesRowSet to interact with the AddressBook database (lines 16–27). Lines 16–17 connect the row set to the database. Lines 18–27 set addressesRowSet's SQL command to the query configured in Fig. 27.5.

Fig. 27.7 | Session Bean that initializes the data source for the AddressBook database. (Part 2 of 2.)

27.2.2 Modifying the Page Bean File for the AddressBook Application

After building the web page and configuring the components used in this example, double click the Submit button to create an action event handler for this button in the page bean
file. The code to insert a contact into the database will be placed in this method. The page bean with the completed event handler is shown in Fig. 27.8 below.

```java
// Fig. 27.8: AddressBook.java
// Page bean for adding a contact to the address book.
package addressbook;

import com.sun.data.provider.RowKey;
import com.sun.rave.web.ui.component.Form;
import com.sun.rave.web.ui.component.Html;
import com.sun.rave.web.ui.component.Link;
import com.sun.rave.web.ui.component.MessageGroup;
import com.sun.rave.web.ui.component.Page;
import com.sun.rave.web.ui.component.Rows;
import com.sun.rave.web.ui.component.Table;
import com.sun.data.provider.impl.CachedRowSetDataProvider;
import javax.faces.FacesException;
import com.sun.rave.web.ui.component.TextField;
import com.sun.rave.web.ui.component.Label;
import com.sun.rave.web.ui.component.Button;
import com.sun.rave.web.ui.component.Table;
import com.sun.rave.web.ui.component.TableRowGroup;
import com.sun.rave.web.ui.component.TableColumn;
import com.sun.data.provider.impl.CachedRowSetDataProvider;

public class AddressBook extends AbstractPageBean
{
    private int __placeholder;

    private void _init() throws Exception
    {
        addressesDataProvider.setCachedRowSet(
            (javax.sql.rowset.CachedRowSet)
                getValue( "#{SessionBean1.addressesRowSet}" ));
        addressesTable.setInternalVirtualForm( true );
    }

    // Lines 36-521 of the autogenerated code were removed to save space.
    // The complete source code is provided in this example's folder.

    public void prerender()
    {
        addressesDataProvider.refresh();
    }

    public void destroy()
    {
        addressesDataProvider.close();
    }
}
```

Fig. 27.8 | Page bean for adding a contact to the address book. (Part 1 of 2.)
27.2 Accessing Databases in Web Applications

```java
// action handler that adds a contact to the AddressBook database
// when the user clicks submit
public String submitButton_action()
{
    if ( addressesDataProvider.canAppendRow() )
    {
        try
        {
            RowKey rk = addressesDataProvider.appendRow();
            addressesDataProvider.setCursorRow( rk );
            addressesDataProvider.setValue( "ADDRESSES.FIRSTNAME", fnameTextField.getValue() );
            addressesDataProvider.setValue( "ADDRESSES.LASTNAME", lnameTextField.getValue() );
            addressesDataProvider.setValue( "ADDRESSES.STREET", streetTextField.getValue() );
            addressesDataProvider.setValue( "ADDRESSES.CITY", cityTextField.getValue() );
            addressesDataProvider.setValue( "ADDRESSES.STATE", stateTextField.getValue() );
            addressesDataProvider.setValue( "ADDRESSES.ZIP", zipTextField.getValue() );
            addressesDataProvider.commitChanges();

            // reset text fields
            lnameTextField.setValue( "" );
            fnameTextField.setValue( "" );
            streetTextField.setValue( "" );
            cityTextField.setValue( "" );
            stateTextField.setValue( "" );
            zipTextField.setValue( "" );
        } // end try
        catch ( Exception ex )
        {
            error( "The address book was not updated. " + ex.getMessage() );
        } // end catch
    } // end if
    return null;
} // end method submitButton_action
```

Fig. 27.8 | Page bean for adding a contact to the address book. (Part 2 of 2.)

Lines 534–573 contain the event-handling code for the Submit button. Line 536 determines whether a new row can be appended to the data provider. If so, a new row is appended at line 540. Every row in a CachedRowSetDataProvider has its own key; method appendRow returns the key for the new row. Line 541 sets the data provider's cursor to the new row, so that any changes we make to the data provider affect that row. Lines 543–554 set each of the row's columns to the values entered by the user in the cor-
responding Text Fields. Line 555 stores the new contact by calling method commitChanges of class CachedRowSetDataProvider to insert the new row into the AddressBook database.

Lines 558–563 clear all of the form’s Text Fields. If these lines are omitted, the fields will retain their current values after the database is updated and the page reloads. Also, the Clear button will not work properly if the Text Fields are not cleared. Rather than emptying the Text Fields, it will reset them to the values they held the last time form was submitted.

Lines 565–569 catch any exceptions that might occur while updating the AddressBook database. Lines 567–568 display a message indicating that the database was not updated as well as the exception’s error message in the page’s MessageGroup component.

In method prerender, line 524 calls CachedRowSetDataProvider method refresh. This re-executes the wrapped CachedRowSet’s SQL statement and re-sorts the Table’s rows so that the new row is displayed in the proper order. If you do not call refresh, the new address is displayed at the end of the Table (since we appended the new row to the end of the data provider). The IDE automatically generated code to free resources used by the data provider (line 529) in the destroy method.

27.3 Ajax-Enabled JSF Components

The term Ajax—short for Asynchronous JavaScript and XML—was coined by Jesse James Garrett of Adaptive Path, Inc. in February 2005 to describe a range of technologies for developing highly responsive, dynamic web applications. Ajax applications include Google Maps, Yahoo’s FlickR and many more. Ajax separates the user interaction portion of an application from its server interaction, enabling both to proceed asynchronously in parallel. This enables Ajax web-based applications to perform at speeds approaching those of desktop applications, reducing or even eliminating the performance advantage that desktop applications have traditionally had over web-based applications. This has huge ramifications for the desktop applications industry—the applications platform of choice is starting to shift from the desktop to the web. Many people believe that the web—especially in the context of abundant open-source software, inexpensive computers and exploding Internet bandwidth—will create the next major growth phase for Internet companies.

Ajax makes asynchronous calls to the server to exchange small amounts of data with each call. Where normally the entire page would be submitted and reloaded with every user interaction on a web page, Ajax allows only the necessary portions of the page to reload, saving time and resources.

Ajax applications are marked up in XHTML and CSS as any other web page and make use of client-side scripting technologies such as JavaScript to interact with page elements. The XMLHttpRequestObject enables the asynchronous exchanges with the web server that make Ajax applications so responsive. This object can be used by most scripting languages to pass XML data from the client to the server and to process XML data sent from the server back to the client.

While using Ajax technologies in web applications can dramatically improve performance, programming in Ajax is complex and error prone. It requires page designers to know both scripting and markup languages. Ajax libraries make it possible to reap Ajax’s benefits in web applications without the labor of writing “raw” Ajax. These libraries provide Ajax-enabled page elements that can be included in web pages simply by adding library-defined tags to the page’s markup. We limit our discussion of building Ajax applications to the use of one such library in Java Studio Creator 2.
27.3 Ajax-Enabled JSF Components

27.3.1 Java BluePrints Component Library

The Java BluePrints Ajax component library provides Ajax-enabled JSF components. These components rely on Ajax technology to deliver the feel and responsiveness of a desktop application over the web. Figure 27.9 summarizes the current set of components that you can download and use with Java Studio Creator 2. We demonstrate the AutoComplete Text Field and Map Viewer components in the next two sections.

To use the Java BluePrints Ajax-enabled components in Java Studio Creator 2, you must download and import them. The IDE provides a wizard for installing this group of components. To access it, choose Tools > Update Center to display the Update Center Wizard dialog. Click Next > to search for available updates. In the Available Updates and New Modules area of the dialog, select BluePrints AJAX Components and click the right arrow (>) button to add it to the list of items you’d like to install. Click Next > and follow the prompts to accept the terms of use and download the components. When the download completes, click Next > then click Finish. Click OK to restart the IDE.

Next, you must import the components into the Palette. Select Tools > Component Library Manager, then click Import…. Click Browse… in the Import Component Library dialog that appears. Select the ui.complib file and click Open. Click OK to import both the BluePrints AJAX Components and the BluePrints AJAX SupportBeans. Close the Component Library Manager to return to the IDE.

You should now see two new nodes at the bottom of the Palette. The first, BluePrints AJAX Components, provides the eight components listed in Fig. 27.9. The second, Blue-

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AutoComplete Text Field</td>
<td>Makes Ajax requests to display a list of suggestions as the user types in the text field.</td>
</tr>
<tr>
<td>Buy Now Button</td>
<td>Initiates a transaction through the PayPal web site.</td>
</tr>
<tr>
<td>Map Viewer</td>
<td>Uses the Google Maps API to display a map that pans, zooms, and can display markers for locations of interest.</td>
</tr>
<tr>
<td>Popup Calendar</td>
<td>Provides a calendar that enables a user to scroll between months and years. Fills a Text Field with a formatted date when the user selects a day.</td>
</tr>
<tr>
<td>Progress Bar</td>
<td>Visually displays the progress of a long-running operation. Uses a programmer-supplied calculation to determine the progress percentage.</td>
</tr>
<tr>
<td>Rating</td>
<td>Provides a customizable five-star rating bar that can display messages as the user moves the mouse over the ratings.</td>
</tr>
<tr>
<td>Rich Textarea Editor</td>
<td>Provides an editable text area that allows the user to format text with fonts, colors, hyperlinks and backgrounds.</td>
</tr>
<tr>
<td>Select Value Text Field</td>
<td>Displays a list of suggestions in a drop-down list as the user types, similar to the AutoComplete Text Field.</td>
</tr>
</tbody>
</table>

Fig. 27.9 | Ajax-enabled components provided by the Java BluePrints Ajax component library.
Chapter 27  Web Applications: Part 2

Prints AJAX Support Beans, includes components that support the Ajax components. You can now build high-performance Ajax web applications by dragging, dropping and configuring the component’s properties, just as you do with other components in the Palette.

27.4 AutoComplete Text Field and Virtual Forms

We demonstrate the AutoComplete Text Field component from the BluePrints catalog by adding a new form to our AddressBook application. The AutoComplete Text Field provides a list of suggestions as the user types. It obtains the suggestions from a data source, such as a database or web service. Eventually, the new form will allow users to search the address book by last name, then first name. If the user selects a contact, the application will display the contact’s name and address on a map of the neighborhood. We build this form in two stages. First, we’ll add the AutoComplete Text Field that will display suggestions as the user types a contact’s last name. Then we’ll add the search functionality and map display in the next step.

Adding Search Components to the AddressBook.jsp Page

Using the AddressBook application from Section 27.2, drop a Static Text component named searchHeader below addressesTable. Change its text to “Search the address book by last name:” and change its font size to 18 px. Now drag an AutoComplete Text Field component to the page and name it nameAutoComplete. Set this field’s required property to true. Add a Label named nameSearchLabel containing the text “Last Name:” to the left of the AutoComplete Text Field. Finally, add a button called lookUpButton with the text Look Up to the right of the AutoComplete Text Field.

27.4.1 Configuring Virtual Forms

Virtual forms are used when you would like a button to submit a subset of the page’s input fields to the server. Recall that the Table’s internal virtual forms were enabled so that clicking the pagination buttons would not submit any of the data in the Text Field used to add a contact to the AddressBook database. Virtual forms are particularly useful for displaying multiple forms on the same page. They allow you to specify a submitter and one or more participants for a form. When the virtual form’s submitter component is clicked, only the values of its participant components will be submitted to the server. We use virtual forms in our AddressBook application to separate the form for adding a contact to the AddressBook database from the form for searching the database.

To add virtual forms to the page, right click the Submit button on the upper form and choose Configure Virtual Forms… from the popup menu to display the Configure Virtual Forms dialog. Click New to add a virtual form, then click in the Name column and change the new form’s name to addForm. Double click the Submit column and change the option to Yes to indicate that this button should be used to submit the addForm virtual form. Click OK to exit the dialog. Next, select all the Text Fields used to enter a contact’s information in the upper form. You can do this by holding the Ctrl key while you click each Text Field. Right click one of the selected Text Fields and choose Configure Virtual Forms....

In the Participate column of the addForm, change the option to Yes to indicate that the values in these Text Fields should be submitted to the server when the form is submitted. Figure 27.10 shows the Configure Virtual Forms dialog after both virtual forms have been added. Click OK to exit.
Repeat the process described above to create a second virtual form named searchForm for the lower form. The Look Up Button should submit the searchForm, and nameAutoComplete should participate in the searchForm. Next, return to Design mode and click the Show Virtual Forms button ( ) at the top of the Visual Designer panel to display a legend of the virtual forms on the page. Your virtual forms should be configured as in Fig. 27.11. The Text Fields outlined in blue participate in the virtual form addForm. Those outlined in green participate in the virtual form searchForm. The components outlined with a

![Fig. 27.10 | Configure Virtual Forms dialog.](image)

![Fig. 27.11 | Virtual forms legend.](image)
dashed line submit their respective forms. A color key is provided at the bottom right of the Design area so that you know which components belong to each virtual form.

### 27.4.2 JSP File with Virtual Forms and an **AutoComplete Text Field**

Figure 27.12 presents the JSP file generated by Java Studio Creator 2 for this stage of the AddressBook application. Notice that a new tag library is specified in the root element (xmlns:bp="http://java.sun.com/blueprints/ui/14"; line 6). This is the BluePrints catalog library that provides Ajax-enabled components such as the **AutoComplete Text Field** component. We focus only on the new features of this JSP.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!-- Fig. 27.12: AddressBook.jsp -->
<!-- AddressBook JSP with an AutoComplete Text Field component -->
<jsp:root version="1.2" xmlns:bp="http://java.sun.com/blueprints/ui/14"
xmlns:ui="http://www.sun.com/web/ui">
  <jsp:directive.page contentType="text/html;charset=UTF-8"
pageEncoding="UTF-8"/>
  <f:view>
    <ui:page binding="#{AddressBook.page1}" id="page1">
      <ui:html binding="#{AddressBook.html1}" id="html1">
        <ui:head binding="#{AddressBook.head1}" id="head1">
          <ui:link binding="#{AddressBook.link1}" id="link1"
url="/resources/stylesheet.css"/>
        </ui:head>
        <ui:body binding="#{AddressBook.body1}" id="body1"
style="-rave-layout: grid">
          <ui:form binding="#{AddressBook.form1}" id="form1"
virtualFormsConfig="addForm | lnameTextField streetTextField fnameTextField cityTextField stateTextField zipTextField | submitButton , searchForm | nameAutoComplete | lookUpButton">
            <ui:staticText binding="#{AddressBook.staticText1}" id="staticText1" style="font-size: 18px; left: 24px; top: 24px; position: absolute; text="Add a contact to the address book:"/>
            <ui:textField binding="#{AddressBook.fnameTextField}" id="fnameTextField" maxLength="20" required="true"
style="left: 120px; top: 72px; position: absolute; width: 192px"/>
            <ui:textField binding="#{AddressBook.lnameTextField}" id="lnameTextField" maxLength="30" required="true"
style="left: 432px; top: 72px; position: absolute; width: 240px"/>
            <ui:textField binding="#{AddressBook.streetTextField}" id="streetTextField" maxLength="100" required="true"
style="left: 120px; top: 96px; position: absolute; width: 552px"/>
          </ui:form>
        </ui:body>
      </ui:html>
    </ui:page>
  </f:view>
</jsp:root>
```

**Fig. 27.12** | AddressBook JSP with an **AutoComplete Text Field** component. (Part 1 of 4.)
27.4 AutoComplete Text Field and Virtual Forms

Fig. 27.12 AddressBook JSP with an AutoComplete Text Field component. (Part 2 of 4.)
Fig. 27.12 | AddressBook JSP with an AutoComplete Text Field component. (Part 3 of 4.)
27.4 **AutoComplete Text Field and Virtual Forms**

```xml
<ui:staticText binding="#{AddressBook.searchHeader}" id="searchHeader" style="font-size: 18px; left: 24px; top: 456px; position: absolute" text="Search the address book by last name:">
<ui:label binding="#{AddressBook.nameSearchLabel}" for="nameAutoComplete" id="nameSearchLabel" requiredIndicator="true" style="position: absolute; left: 24px; top: 504px" text="Last Name:">
<bp:autoComplete binding="#{AddressBook.nameAutoComplete}" completionMethod="#{AddressBook.nameAutoComplete_complete}" id="nameAutoComplete" required="true" style="left: 120px; top: 504px; position: absolute" />
<ui:button binding="#{AddressBook.lookUpButton}" id="lookUpButton" style="position: absolute; left: 312px; top: 504px" text="Look Up" />
</ui:label>
</ui:form>
</ui:body>
</ui:html>
</f:view>
```

*Fig. 27.12* | AddressBook JSP with an **AutoComplete Text Field** component. (Part 4 of 4.)
Lines 21–25 configure the virtual forms for this page. Lines 217–221 define the **AutoComplete Text Field** component. This component's `completionMethod` attribute is bound to the page bean's `nameAutoComplete_complete` method (discussed in Section 27.4.3), which provides the list of options the **AutoComplete Text Field** component should suggest. To create this method, right-click the `nameAutoComplete` component in **Design** view and select **Edit Event Handler > complete**. Notice that the **Look Up** button (lines 222–224) does not specify an action handler method binding; we'll add this in Section 27.5.

### 27.4.3 Providing Suggestions for an AutoComplete Text Field

Figure 27.13 displays the page bean file for the JSP in Fig. 27.12. It includes the method `nameAutoComplete_complete`, which provides the functionality for the **AutoComplete Text Field**. Otherwise, this page bean is identical to the one in Fig. 27.8.

```java
// Fig. 27.8: AddressBook.java
// Page bean that suggests names in the AutoComplete Text Field.
package addressbook;

import com.sun.data.provider.RowKey;
import com.sun.rave.web.ui.component.Body;
import com.sun.rave.web.ui.component.Form;
import com.sun.rave.web.ui.component.Head;
import com.sun.rave.web.ui.component.Html;
import com.sun.rave.web.ui.component.Link;
import com.sun.rave.web.ui.component.Page;
import javax.faces.FacesException;
import com.sun.rave.web.ui.component.StaticText;
import com.sun.rave.web.ui.component.TextField;
import com.sun.rave.web.ui.component.Label;
import com.sun.rave.web.ui.component.Button;
import com.sun.rave.web.ui.component.Table;
import com.sun.rave.web.ui.component.TableRowGroup;
import com.sun.rave.web.ui.component.TableColumn;
import com.sun.data.provider.impl.CachedRowSetDataProvider;
import com.sun.rave.web.ui.component.MessageGroup;
import com.sun.j2ee.blueprints.ui.autocomplete.AutoCompleteComponent;
import com.sun.j2ee.blueprints.ui.autocomplete.CompletionResult;
import javax.faces.context.FacesContext;

public class AddressBook extends AbstractPageBean
{
  private int __placeholder;

  private void _init() throws Exception
  {
    addressesDataProvider.setCachedRowSet((javax.sql.rowset.CachedRowSet)
      getValue("#{SessionBean1.addressesRowSet}" ));
    addressesTable.setInternalVirtualForm( true );
  }

Fig. 27.13 | Page bean that suggests names in the **AutoComplete Text Field**. (Part 1 of 3.)
public void prerender() {
    addressesDataProvider.refresh();
}

public void destroy() {
    addressesDataProvider.close();
}

// action handler that adds a contact to the AddressBook database
// when the user clicks submit
public String submitButton_action() {
    try {
        RowKey rk = addressesDataProvider.appendRow();
        addressesDataProvider.setCursorRow( rk );
        addressesDataProvider.setValue( "ADDRESSES.FIRSTNAME", fnameTextField.getValue() );
        addressesDataProvider.setValue( "ADDRESSES.LASTNAME", lnameTextField.getValue() );
        addressesDataProvider.setValue( "ADDRESSES.STREET", streetTextField.getValue() );
        addressesDataProvider.setValue( "ADDRESSES.CITY", cityTextField.getValue() );
        addressesDataProvider.setValue( "ADDRESSES.STATE", stateTextField.getValue() );
        addressesDataProvider.setValue( "ADDRESSES.ZIP", zipTextField.getValue() );
        addressesDataProvider.commitChanges();

        // reset text fields
        lnameTextField.setValue( "" );
        fnameTextField.setValue( "" );
        streetTextField.setValue( "" );
        cityTextField.setValue( "" );
        stateTextField.setValue( "" );
        zipTextField.setValue( "" );
    } catch ( Exception ex ) {
        error( "The address book was not updated. " + ex.getMessage() );
    }
}

Fig. 27.13 | Page bean that suggests names in the AutoComplete Text Field. (Part 2 of 3.)
return null;

} // end method submitButton_action

// action handler for the autocomplete box that fetches names
// from the address book whose prefixes match the letters typed so far
// and displays them in a suggestion list.
public void nameAutoComplete_complete( FacesContext context, String
prefix, CompletionResult result )
{
    try
    {
        boolean hasNext = addressesDataProvider.cursorFirst();

        while ( hasNext )
        {
            String name =
            (String) addressesDataProvider.getValue(  
            "ADDRESSES.LASTNAME" ) + "," +
            (String) addressesDataProvider.getValue(  
            "ADDRESSES.FIRSTNAME" );

            // if the name in the database starts with the prefix, add it
            // to the list of suggestions
            if ( name.toLowerCase().startsWith( prefix.toLowerCase() ) )
            {
                result.addItem( name );
            } // end if
            else
            {
                // terminate the loop if the rest of the names are
                // alphabetically less than the prefix
                if ( prefix.compareTo( name ) < 0 )
                {
                    break;
                } // end if
            } // end else

            hasNext = addressesDataProvider.cursorNext();
        } // end while
    } // end try

    catch ( Exception ex )
    {
        result.addItem( "Exception getting matching names." );
    } // end catch

} // end method nameAutoComplete_complete

Fig. 27.13 | Page bean that suggests names in the **AutoComplete Text Field.** (Part 3 of 3.)
Method nameAutoComplete_complete (lines 630–670) is invoked after every keystroke in the AutoComplete Text Field to update the list of suggestions based on the text the user has typed so far. The method receives a string (prefix) containing the text the user has entered and a CompletionResult object (result) that is used to display suggestions to the user. The method loops through the rows of the addressesDataProvider, retrieves the name from each row, checks whether the name begins with the letters typed so far and, if so, adds the name to result. Line 635 sets the cursor to the first row in the data provider. Line 637 determines whether there are more rows in the data provider. If so, lines 640–644 retrieve the last name and first name from the current row and create a String in the format last name, first name. Line 648 compares the lowercase versions of name and prefix to determine whether the name starts with the characters typed so far. If so, the name is a match and line 650 adds it to result.

Recall that the data provider wraps a CachedRowSet object that contains a SQL query which returns the rows in the database sorted by last name, then first name. This allows us to stop iterating through the data provider once we reach a row whose name comes alphabetically after the text entered by the user—names in the rows beyond this will all be alphabetically greater and thus are not potential matches. If the name does not match the text entered so far, line 656 tests whether the current name is alphabetically greater than the prefix. If so, line 658 terminates the loop.

Performance Tip 27.1

When using database columns to provide suggestions in an AutoComplete Text Field, sorting the columns eliminates the need to check every row in the database for potential matches. This significantly improves performance when dealing with a large database.

If the name is neither a match nor alphabetically greater than prefix, then line 663 moves the cursor to the next row in the data provider. If there is another row, the loop iterates again, checking whether the name in the next row matches the prefix and should be added to results.

Lines 666–669 catch any exceptions generated while searching the database. Line 668 adds text to the suggestion box indicating the error to the user.

27.5 Google Maps Map Viewer Component

We now complete the AddressBook application by adding functionality to the Look Up Button. When the user clicks this Button, the name in the AutoComplete Text Field is used to search the AddressBook database. We also add a Map Viewer Ajax-enabled JSF component to the page to display a map of the area for the address. A Map Viewer uses the Google Maps API web service to find and display maps. (The details of web services are covered in Chapter 28.) In this example, using the Google Maps API is analogous to making ordinary method calls on a Map Viewer object and its supporting bean in the page bean file. When a contact is found, we display a map of the neighborhood with a Map Marker that points to the location and indicates the contact’s name and address.

27.5.1 Obtaining a Google Maps API Key

To use the Map Viewer component, you must have an account with Google. Visit the site https://www.google.com/accounts/ManageAccount to register for a free account if you
do not have one. Once you have logged in to your account, you must obtain a key to use the Google Maps API from www.google.com/apis/maps. The key you receive will be specific to this web application and will limit the number of maps the application can display per day. When you sign up for the key, you will be asked to enter the URL for the application that will be using the Google Maps API. If you are deploying the application only on Java Studio Creator 2’s built-in Sun Application Server 8 test server, enter http://localhost:29080/ as the URL.

After you accept Google’s terms and conditions, you’ll be redirected to a page containing your new Google Maps API key. Save this key in a text file in a convenient location for future reference.

27.5.2 Adding a Map Viewer Component to a Page

Now that you have a key to use the Google Maps API, you are ready to complete the AddressBook application. With AddressBook.jsp open in Design mode, add a Map Viewer component named mapViewer below the nameAutoComplete. In the Properties window, set the Map Viewer’s key property to the key you obtained for accessing the Google Maps API. Set the rendered property to false so that the map will not be displayed when the user has not yet searched for an address. Set the zoomLevel property to 1 (In) so the user can see the street names on the map.

Drop a Map Marker (named mapMarker) from the AJAX Support Beans section of the Palette anywhere on the page. This component (which is not visible in Design view) marks the contact’s location on the map. You must bind the marker to the map so that the marker will display on the map. To do so, right click the Map Viewer in Design mode component and choose Property Bindings… to display the Property Bindings dialog. Select info from the Select bindable property column of the dialog, then select mapMarker from the Select binding target column. Click Apply, then Close.

Finally, drop a Geocoding Service Object (named geoCoder) from the AJAX Support Beans section of the Palette anywhere on the page. This object (which is not visible in Design view) converts street addresses into latitudes and longitudes that the Map Viewer component uses to display an appropriate map.

Adding a Data Provider to the Page

To complete this application, you need a second data provider to search the AddressBook database based on the first and last name entered in the AutoComplete Text Field. Open the Servers window and expand the AddressBook node and its Tables node to reveal the Addresses table. Right click the table’s node and select Add To Page to display the Add New Data Provider with RowSet dialog (Fig. 27.14). We want to create a new data source rather than reuse the existing one, because the query to search for contacts is different from the query to display all the contacts. Select the Create option for the SessionBean and enter the name addressesSearch for the data provider. Click OK to create the new data provider. In the Outline window, a new node named addressesSearchDataProvider has been added to the AddressBook node, and a node named addressesSearch has been added to the SessionBean node.

Double click the addressesSearch node to edit the SQL statement for this RowSet. Since we will use this row set to search the database for a given last and first name, we need to add search parameters to the SELECT statement the RowSet will execute. To do this,
enter the text "= ?" in the Criteria column of both the first-name and last-name rows in the SQL statement editor table. The number 1 should appear in the Order column for first name and 2 should appear for last name. Notice that the lines

```
WHERE JHTP7 ADDRESSES FIRSTNAME = ?
AND JHTP7 ADDRESSES LASTNAME = ?
```

have been added to the SQL statement. This indicates that the RowSet now executes a parameterized SQL statement. The parameters can be set programmatically, with the first name as the first parameter and the last name as the second.

### 27.5.3 JSP File with a Map Viewer Component

Figure 27.15 presents the JSP file for the completed address book application. It is nearly identical to the JSP for the previous two versions of this application. The new feature is the Map Viewer component (and its supporting components) used to display a map with the contact’s location. We discuss only the new elements of this file. [Note: This code will not run until you have specified your own Google Maps key in lines 227–229. You can paste your key into the Map Viewer component’s key property in the Properties window.]

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!-- Fig. 27.15: AddressBook.jsp -->
<-- AddressBook JSP page with a Map Viewer component. -->
<jsp:root version="1.2" xmlns:bpe="http://java.sun.com/blueprints/ui/14"
xmlns:ui="http://www.sun.com/web/ui">
```

Fig. 27.15 | AddressBook JSP with a Map Viewer component. (Part 1 of 5.)
<jsp:directive.page contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"/>

<f:view>
  <ui:page binding="#{AddressBook.page1}" id="page1">
    <ui:html binding="#{AddressBook.html1}" id="html1">
      <ui:head binding="#{AddressBook.head1}" id="head1">
        <ui:link binding="#{AddressBook.link1}" id="link1" url="/resources/stylesheet.css"/>
      </ui:head>
      <ui:body binding="#{AddressBook.body1}" id="body1" style="-rave-layout: grid">
        <ui:form binding="#{AddressBook.form1}" id="form1" virtualFormsConfig="addForm | streetTextField fnameTextField lnameTextField | submitButton, searchForm | nameAutoComplete | lookUpButton">
          <ui:staticText binding="#{AddressBook.staticText1}" id="staticText1" style="font-size: 18px; left: 24px; top: 24px; position: absolute" text="Add a contact to the address book:">
            <ui:textField binding="#{AddressBook.fnameTextField}" id="fnameTextField" maxLength="20" required="true" style="left: 120px; top: 72px; position: absolute; width: 192px"/>
            <ui:textField binding="#{AddressBook.lnameTextField}" id="lnameTextField" maxLength="30" required="true" style="left: 432px; top: 72px; position: absolute; width: 240px"/>
            <ui:textField binding="#{AddressBook.streetTextField}" id="streetTextField" maxLength="100" required="true" style="left: 120px; top: 96px; position: absolute; width: 552px"/>
            <ui:textField binding="#{AddressBook.cityTextField}" id="cityTextField" maxLength="30" required="true" style="left: 120px; top: 120px; position: absolute"/>
            <ui:textField binding="#{AddressBook.stateTextField}" id="stateTextField" maxLength="2" required="true" style="left: 456px; top: 120px; position: absolute; width: 48px"/>
            <ui:textField binding="#{AddressBook.zipTextField}" id="zipTextField" maxLength="5" required="true" style="left: 576px; top: 120px; position: absolute; width: 96px"/>
          </ui:form>
        </ui:body>
      </ui:head>
    </ui:html>
  </ui:page>
</f:view>

Fig. 27.15 | AddressBook JSP with a Map Viewer component. (Part 2 of 5.)
27.5 Google Maps Map Viewer Component

Fig. 27.15 | AddressBook JSP with a Map Viewer component. (Part 3 of 5.)
Fig. 27.15 | AddressBook JSP with a Map Viewer component. (Part 4 of 5.)
Lines 242–247 define the `mapViewer` component that displays a map of the area surrounding the address. The component’s `center` attribute is bound to the page bean property `mapViewer_center`. This property is manipulated in the page bean file to center the map on the desired address.
The Look Up Button’s action attribute is now bound to method `lookUp_action` in the page bean (line 226). This action handler searches the AddressBook database for the name entered in the AutoComplete Text Field and displays the contact’s name and address on a map of the contact’s location. We discuss this method in Section 27.5.3.

### 27.5.4 Page Bean that Displays a Map in the Map Viewer Component

Figure 27.16 presents the page bean for the completed AddressBook application. Most of this file is identical to the page beans for the first two versions of this application. We discuss only the new action-handler method, `lookUpButton_action`.

Method `lookUpButton_action` (lines 646–704) is invoked when the user clicks the Look Up button in the lower form on the page. Lines 649–652 retrieve the name from the AutoComplete Text Field and split it into Strings for the first and last name. Lines 662–669 each obtain the `addressesSearchDataProvider`’s `CachedRowSet`, then use its method `setObject` to set the parameters for the query to the first and last name. The `setObject` method replaces a parameter in the SQL query with a specified string. Line 661 refreshes the data provider, which executes the wrapped `RowSet`’s query with the new parameters. The result set now contains only rows that match the first and last name from the AutoComplete Text Field. Lines 662–669 fetch the street address, city, state and zip code for this contact from the database. Note that in this example, we assume there are not multiple entries in the address book for the same first and last name, as we fetch only the address information for the first row in the data provider. Any additional rows that match the first and last name are ignored.

```
// Fig. 27.16: AddressBook.java
// Page bean for adding a contact to the address books.
package addressbook;
import com.sun.data.provider.RowKey;
import com.sun.rave.web.ui.component.Body;
import com.sun.rave.web.ui.component.Form;
import com.sun.rave.web.ui.component.Head;
import com.sun.rave.web.ui.component.Html;
import com.sun.rave.web.ui.component.Link;
import com.sun.rave.web.ui.component.Page;
import javax.faces.FacesException;
import com.sun.rave.web.ui.component.StaticText;
import com.sun.rave.web.ui.component.TextField;
import com.sun.rave.web.ui.component.Label;
import com.sun.rave.web.ui.component.Button;
import com.sun.rave.web.ui.component.Table;
import com.sun.rave.web.ui.component.TableRowGroup;
import com.sun.rave.web.ui.component.TableColumn;
import com.sun.data.provider.impl.CachedRowSetDataProvider;
import com.sun.rave.web.ui.component.MessageGroup;
import com.sun.j2ee.blueprints.ui.autocomplete.AutoCompleteComponent;
import com.sun.j2ee.blueprints.ui.autocomplete.CompletionResult;
import javax.faces.context.FacesContext;
```

**Fig. 27.16** Page bean that gets a map to display in the MapViewer component. (Part 1 of 5.)
import com.sun.j2ee.blueprints.ui.mapviewer.MapComponent;
import com.sun.j2ee.blueprints.ui.mapviewer.MapPoint;
import com.sun.j2ee.blueprints.ui.geocoder.GeoCoder;
import com.sun.j2ee.blueprints.ui.geocoder.GeoPoint;
import com.sun.j2ee.blueprints.ui.mapviewer.MapMarker;

public class AddressBook extends AbstractPageBean
{
    private int __placeholder;

    private void _init() throws Exception
    {
        addressesDataProvider.setCachedRowSet((javax.sql.rowset.CachedRowSet)getValue("#{SessionBean1.addressesRowSet}") );
        addressesTable.setInternalVirtualForm(true);
        addressesSearchDataProvider.setCachedRowSet((javax.sql.rowset.CachedRowSet)getValue("#{SessionBean1.addressesSearch}") );
        mapViewer.setRendered(false);
    } // end method _init

    // Lines 48-544 of the autogenerated code were removed to save space.
    // The complete source code is provided in this example's folder.

    public void prerender()
    {
        addressesDataProvider.refresh();
    } // end method prerender

    public void destroy()
    {
        addressesSearchDataProvider.close();
        addressesDataProvider.close();
    } // end method destroy

    // action handler that adds a contact to the AddressBook database
    // when the user clicks submit
    public String submitButton_action()
    {
        if (addressesDataProvider.canAppendRow())
        {
            try
            {
                RowKey rk = addressesDataProvider.appendRow();
                addressesDataProvider.setCursorRow(rk);
                addressesDataProvider.setValue("ADDRESSES.FIRSTNAME", fnameTextField.getValue());
                addressesDataProvider.setValue("ADDRESSES.LASTNAME", lnameTextField.getValue());
                addressesDataProvider.setValue("ADDRESSES.STREET", streetTextField.getValue());
            }
        }
    }

Fig. 27.16 | Page bean that gets a map to display in the MapViewer component. (Part 2 of 5.)
Chapter 27  Web Applications: Part 2

addressesDataProvider.setValue( "ADDRESSES.CITY", cityTextField.getValue() );
addressesDataProvider.setValue( "ADDRESSES.STATE", stateTextField.getValue() );
addressesDataProvider.setValue( "ADDRESSES.ZIP", zipTextField.getValue() );
addressesDataProvider.commitChanges();

// reset text fields
lnameTextField.setValue( "" );
fnameTextField.setValue( "" );
streetTextField.setValue( "" );
cityTextField.setValue( "" );
stateTextField.setValue( "" );
zipTextField.setValue( "" );

} // end try

catch ( Exception ex )
{
error( "The address book was not updated. " + ex.getMessage() );
} // end catch

return null;

} // end method submitButton_action

public void nameAutoComplete_complete( FacesContext context, String prefix, CompletionResult result )
{
try
{
boolean hasNext = addressesDataProvider.cursorFirst();

while ( hasNext )
{

// get a name from the database
String name = (String) addressesDataProvider.getValue( "ADDRESSES.LASTNAME" ) + ", " +
(String) addressesDataProvider.getValue( "ADDRESSES.FIRSTNAME" );

// if the name in the database starts with the prefix, add it
// to the list of suggestions
if ( name.toLowerCase().startsWith( prefix.toLowerCase() ) )
{
result.addItem( name );
}
else
{

Fig. 27.16 | Page bean that gets a map to display in the MapViewer component. (Part 3 of 5.)
// terminate the loop if the rest of the names are
// alphabetically less than the prefix
if ( prefix.compareTo( name ) < 0 )
{
    break;
}
// end if
// move cursor to next row of database
hasNext = addressesDataProvider.cursorNext();
// end while
} // end try
catch ( Exception ex )
{
    result.addItem( "Exception getting matching names." );
} // end catch
} // end method nameAutoComplete_complete

// action handler for the lookUpButton that searches the address book
// database and displays the requested address on a corresponding map.
public String lookUpButton_action()
{
    // split text in autocomplete field into first and last name
    String name = String.valueOf( nameAutoComplete.getValue() );
    int splitIndex = name.indexOf( " ");
    String lname = name.substring( 0, splitIndex );
    String fname = name.substring( splitIndex + 2 );
    try
    {
        // set the parameters for the addressesSelected query
        addressesSearchDataProvider.getCachedRowSet().setObject(1, fname);
        addressesSearchDataProvider.getCachedRowSet().setObject(2, lname);
        addressesSearchDataProvider.refresh();
        String street = (String) addressesSearchDataProvider.getValue("ADDRESSES.STREET");
        String city = (String) addressesSearchDataProvider.getValue("ADDRESSES.CITY");
        String state = (String) addressesSearchDataProvider.getValue("ADDRESSES.STATE");
        String zip = (String) addressesSearchDataProvider.getValue("ADDRESSES.ZIP");
        // format the address for Google Maps
        String googleAddress = street + " ", " + city + ", "+ state + " "+ zip;
        // get the geopoints for the address
        GeoPoint points[] = geoCoder.geoCode( googleAddress );

        // terminate the loop if the rest of the names are
        // alphabetically less than the prefix
    }
Chapter 27  Web Applications: Part 2

Lines 672–673 format the address as a String for use with the Google Maps API.

Line 219 calls the Geocoding Service Object’s geoCode method with the address as an argument. This method returns an array of GeoPoint objects representing locations that match the address parameter. GeoPoint objects provide the latitude and longitude of a given location. We supply a complete address with a street, city, state and zip code as an argument to geoCode, so the returned array will contain just one GeoPoint object. Line 679 determines whether the array of GeoPoint objects is null. If so, the address could not be found, and lines 681–683 display a message in the Message Group informing the user of the search error, hide the Map Viewer and return null to terminate the processing.

Lines 687–688 set the latitude and longitude of the Map Viewer’s center to the latitude and longitude of the GeoPoint that represents the selected address. Lines 691–694 set the Map Marker’s latitude and longitude, and set the text to display on the marker. Line 696 displays the recentered map containing the Map Marker that indicates the contact’s location.

Lines 698–701 catch any exceptions generated throughout the method body and display an error message in the Message Group. If the user has simply selected a name from the list of selections in the AutoComplete Text Field, there will be no errors in searching the database, as the name is guaranteed to be in the proper last name, first name format and included in the AddressBook database. We did not include any special error-handling code for cases in which the user types a name that cannot be found in the AddressBook or for improperly formatted names.
27.6 Wrap-Up

In this chapter, we presented a three-part case study on building a web application that interacts with a database and provides rich user interaction using Ajax-enabled JSF components. We first showed how to build an AddressBook application that allows a user to add addresses to the AddressBook and browse its contents. Through this example, you learned how to insert user input into a JavaDB database and how to display the contents of a database on a web page using a Table JSF component.

You learned how to download and import the Java BluePrints Ajax-enabled component library. We then extended the AddressBook application to include an AutoComplete TextField component. We showed how to use a database to display suggestions in the AutoComplete TextField. You also learned how to use virtual forms to submit subsets of a form’s input components to the server for processing.

Finally, we completed the third part of the AddressBook application by adding functionality to the search form. You learned how to use a Map Viewer, a Map Marker and a Geocoding Service Object from the Java BluePrints Ajax-enabled component library to display a Google map that shows a contact’s location.

In the next chapter, you’ll learn how to create and consume web services with Java. You’ll use the NetBeans 5.5 IDE to create web services and consume them from desktop applications, and you’ll use the Java Studio Creator IDE to consume a web service from a web application. If you would prefer to perform all these tasks in one IDE, you can download the NetBeans Visual Web Pack 5.5 (www.netbeans.org/products/visualweb/) for Netbeans 5.5.

27.7 Web Resources

www.deitel.com/ajax/Ajax_resourcercenter.html
Explore our Ajax Resource Center for links to Ajax articles, tutorials, applications, community websites, and more.
developers.sun.com/prodtech/javatools/jscreator/learning/tutorials/index.jsp
Provides dozens of tutorials on Java Studio Creator 2. Particularly useful for this chapter are the Access Databases and Work with Ajax Components sections.
developers.sun.com/prodtech/javadb/
Sun’s official site on JavaDB—overviews the technology and provides links to technical articles and a manual on using Apache Derby databases.
java.sun.com/reference/blueprints/
The Sun Developer Network reference site for the Java BluePrints.
blueprints.dev.java.net/
The java.net site for the Java BluePrints project.
blueprints.dev.java.net/ajaxcomponents.html
Information about the Ajax-enabled components provided by the Java BluePrints library.
developers.sun.com/prodtech/javatools/jscreator/reference/code/samplecomps/index.html
Demonstrates the eight Ajax-enabled components provided by the Java BluePrints library.
googIe.com/apis/maps
Google account holders can sign up here for a key to use the Google Maps API.
ajax.dev.java.net/
The Project jMaki Ajax framework for building your own Ajax-enabled components.
Section 27.2 Accessing Databases in Web Applications

Many web applications access databases to store and retrieve persistent data. In this section, we build a web application that uses a Java DB database to store contacts in the address book and display contacts from the address book on a web page.

The web page enables the user to enter new contacts in a form. This form consists of text fields for the contact’s first name, last name, street address, city, state and zip code. The form also has a submit button to send the data to the server and a clear button to reset the form’s fields. The application stores the address book information in a database named AddressBook, which has a single table named Addresses. (We provide this database in the examples directory for this chapter. You can download the examples from www.deitel.com/books/jhtp7.) This example also introduces the Table JSF component, which displays the addresses from the database in tabular format. We show how to configure the Table component shortly.

The Table component formats and displays data from database tables.

To use a database in a Java Studio Creator 2 web application, you must first start the IDE’s bundled database server, which includes drivers for many types of databases, including Java DB.

To start the server, click the Servers tab below the File menu, right click Bundled Database Server at the bottom of the Servers window and select Start Bundled Database Server.

To add a Java DB database to a project, right click the Data Sources node at the top of the Servers window and select Add Data Source…. In the Add Data Source dialog, enter the data source name and select Derby for the server type. Specify the user ID and password for the database. For the database URL, enter jdbc:derby://localhost:21527/YourDatabaseNameHere. This URL indicates that the database resides on the local machine and accepts connections on port 21527. Click the Select button to choose a table that will be used to validate the database. Click Select to close this dialog, then click Add to add the database as a data source for the project and close the dialog.

To configure a Table component to display database data, right click the Table and select Bind to Data to display the Table Layout dialog. Click the Add Data Provider… button to display the Add Data Provider dialog, which contains a list of the available data sources. Expand the database’s node, select a table and click Add. The Table Layout dialog displays a list of the columns in the database table. All of the items under the <Selected heading will be displayed in the Table. To remove a column from the Table, you can select it and click the < button.

By default, the Table’s column headings are the column names in the database table displayed in capital letters. You can change these headings by selecting a column and editing its headerText property in the Properties window. To select a column, expand the table’s node in the Outline window (while in Design mode), then select the appropriate column object.

Clicking the checkbox next to a Table’s paginationControls property in the Properties window configures a Table for automatic pagination. Five rows will be displayed at a time, and buttons for moving forward and backward between groups of five contacts will be added to the bottom of the Table.

Binding a Table to a data provider adds a new CachedRowSetDataProvider to the application’s node in the Outline window. A CachedRowSetDataProvider provides a scrollable RowSet that can be bound to a Table component to display the RowSet’s data.

The CachedRowSet object wrapped by our addressesDataProvider is configured by default to execute a SQL query that selects all the data in the database table. You can edit this SQL query by expanding the SessionBean node in the Outline window and double clicking the RowSet element to open the query editor window.
• Every row in a CachedRowSetDataProvider has its own key; method appendRow, which adds a new row to the CachedRowSet, returns the key for the new row.
• Method commitChanges of class CachedRowSetDataProvider applies any changes to the CachedRowSet to the database.
• CachedRowSetDataProvider method refresh re-executes the wrapped CachedRowSet’s SQL statement.

Section 27.3 Ajax-Enabled JSF Components
• The term Ajax—short for Asynchronous JavaScript and XML—was coined by Jesse James Garrett of Adaptive Path, Inc. in February 2005 to describe a range of technologies for developing highly responsive, dynamic web applications.
• Ajax separates the user interaction portion of an application from its server interaction, enabling both to proceed asynchronously in parallel. This enables Ajax web-based applications to perform at speeds approaching those of desktop applications.
• Ajax makes asynchronous calls to the server to exchange small amounts of data with each call.
• Ajax allows only the necessary portions of the page to reload, saving time and resources.
• Ajax applications are marked up in XHTML and CSS as any other web page and make use of client-side scripting technologies such as JavaScript to interact with page elements.
• The XMLHttpRequestObject enables the asynchronous exchanges with the web server that make Ajax applications so responsive. This object can be used by most scripting languages to pass XML data from the client to the server and to process XML data sent from the server to the client.
• Ajax libraries make it possible to reap Ajax’s benefits in web applications without the labor of writing “raw” Ajax.
• The Java BluePrints Ajax component library provides Ajax-enabled JSF components.
• To use the Java BluePrints Ajax-enabled components in Java Studio Creator 2, you must download and import them. Choose Tools > Update Center to display the Update Center Wizard dialog. Click Next > to search for available updates. In the Available Updates and New Modules area of the dialog, select BluePrints AJAX Components and click the right arrow (>) button to add it to the list of items you’d like to install. Click Next > and follow the prompts to accept the terms of use and download the components. When the download completes, click Next > then click Finish. Click OK to restart the IDE.
• Next, you must import the components into the Palette. Select Tools > Component Library Manager, then click Import... Click Browse... in the Import Component Library dialog that appears. Select the ui.complib file and click Open. Click OK to import both the BluePrints AJAX Components and the BluePrints AJAX SupportBeans.

Section 27.4 AutoComplete Text Field and Virtual Forms
• The AutoComplete Text Field provides a list of suggestions from a data source (such as a database or web service) as the user types.
• Virtual forms are used when you would like a button to submit a subset of the page’s input fields to the server.
• Virtual forms enable you to display multiple forms on the same page. They allow you to specify a submitter and one or more participants for each form. When the virtual form’s submitter component is clicked, only the values of its participant components will be submitted to the server.
• To add virtual forms to a page, right click the submitter component on the form and choose Configure Virtual Forms... from the pop-up menu to display the Configure Virtual Forms dialog. Click New to add a virtual form, then click in the Name column and specify the new form’s name. Dou-
ble click the Submit column and change the option to Yes to indicate that this button should be used to submit the virtual form. Click OK to exit the dialog. Next, select all the input components that will participate in the virtual form. Right click one of the selected components and choose Configure Virtual Forms. In the Participate column of the appropriate virtual form, change the option to Yes to indicate that the values in these components should be submitted to the server when the form is submitted.

- To see the virtual forms in the Design mode, click the Show Virtual Forms button at the top of the Visual Designer panel to display a legend of the virtual forms on the page.
- An AutoComplete Text Field component’s completionMethod attribute is bound to a page bean’s complete event handler. To create this method, right click the AutoComplete Text Field component in Design view and select Edit Event Handler > complete.
- The complete event handler is invoked after every keystroke in an AutoComplete Text Field to update the list of suggestions based on the text the user has typed so far. The method receives a string containing the text the user has entered and a CompletionResult object that is used to display suggestions to the user.

Section 27.5 Google Maps Map Viewer Component

- A Map Viewer Ajax-enabled JSF component uses the Google Maps API web service to find and display maps. A Map Marker points to a location on a map.
- To use the Map Viewer component, you must have an account with Google. Register for a free account at https://www.google.com/accounts/ManageAccount. You must obtain a key to use the Google Maps API from www.google.com/apis/maps. The key you receive will be specific to your web application and will limit the number of maps the application can display per day. When you sign up for the key, you will be asked to enter the URL for the application that will be using the Google Maps API. If you are deploying the application only on Java Studio Creator 2’s built-in test server, enter the URL http://localhost:29080/
- To use a Map Viewer, set its key property to the Google Maps API key you obtained.
- A Map Marker (from the AJAX Support Beans section of the Palette) marks a location on a map. You must bind the marker to the map so that the marker will display on the map. To do so, right click the Map Viewer in Design mode component and choose Property Bindings... to display the Property Bindings dialog. Select info from the Select bindable property column of the dialog, then select the Map Marker from the Select binding target column. Click Apply, then Close.
- A Geocoding Service Object (from the AJAX Support Beans section of the Palette) converts street addresses into latitudes and longitudes that the Map Viewer component uses to display an appropriate map.
- The Map Viewer’s center attribute is bound to the page bean property mapViewer_center. This property is manipulated in the page bean file to center the map on the desired address.
- The Geocoding Service Object’s geoCode method receives an address as an argument and returns an array of GeoPoint objects representing locations that match the address parameter. GeoPoint objects provide the latitude and longitude of a given location.

Terminology

<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ajax (Asynchronous JavaScript and XML)</td>
<td>binding a JSF Table to a database table</td>
</tr>
<tr>
<td>Ajax-enabled component libraries</td>
<td>bundled database server</td>
</tr>
<tr>
<td>Ajax-enabled JSF components</td>
<td>Button JSF component</td>
</tr>
<tr>
<td>Apache Derby</td>
<td>Buy Now Button JSF component</td>
</tr>
<tr>
<td>AutoComplete Text Field JSF component</td>
<td>CachedRowSet interface</td>
</tr>
</tbody>
</table>
Self-Review Exercises

27.1 State whether each of the following is true or false. If false, explain why.
   a) The Table JSF component allows you to lay out other components and text in tabular format.
   b) Virtual forms allow multiple forms, each with its own Submit button, to be displayed on the same web page.
   c) A CachedRowSetDataProvider is stored in the SessionBean and executes SQL queries to provide Table components with data to display.
   d) The XMLHttpRequestObject provides access to a page’s request object.
   e) The complete event handler for an AutoComplete Text Field is called after every key-stroke in the text field to provide a list of suggestions based on what has already been typed.
   f) A data provider automatically re-executes its SQL command to provide updated database information at every page refresh.
   g) To recenter a Map Viewer component, you must set the longitude and latitude of the map’s center.

27.2 Fill in the blanks in each of the following statements.
   a) Ajax is an acronym for ________.
   b) Method ________ of class ________ updates a database to reflect any changes made in the database’s data provider.
   c) A(n) ________ is a supporting component used to translate addresses into latitudes and longitudes for display in a Map Viewer component.
   d) A virtual form specifies that certain JSF components are ________ whose data will be submitted when the submitter component is clicked.
   e) Ajax components for Java Studio Creator 2 such as the AutoComplete Text Field and Map Viewer are provided by the ________.

Answers to Self-Review Exercises

27.1 a) False. Table components are used to display data from databases. b) True. c) False. The CachedRowSetDataProvider is a property of the page bean. It wraps a CachedRowSet, which is stored
Chapter 27  Web Applications: Part 2

in the SessionBean and executes SQL queries. d) False. The XMLHttpRequestObject is an object that allows asynchronous exchanges with a web server. e) True. f) False. You must call method refresh on the data provider to re-execute the SQL command. g) True.

27.2  a) Asynchronous JavaScript and XML. b) commitChanges, CachedRowSetDataProvider. c) Geocoding Service Object. d) participants. e) Java BluePrints Ajax component library.

Exercises

27.3  (Guestbook Application) Create a JSF web page that allows users to sign and view a guestbook. Use the Guestbook database (provided in the examples directory for this chapter) to store guestbook entries. The Guestbook database has a single table, Messages, which has four columns: date, name, email and message. The database already contains a few sample entries. On the web page, provide Text Fields for the user’s name and email address and a Text Area for the message. Add a Submit Button and a Table component and configure the Table to display guestbook entries. Use the Submit Button’s action-handler method to insert a new row containing the user’s input and today’s date into the Guestbook database.

27.4  (AddressBook Application Modification) Modify the AddressBook application so that users enter searches in the AutoComplete Text Field in the format first name last name. You will need to add a new data provider (or modify the existing one) to sort the rows in the AddressBook database by first name, then last name.

27.5  (Map Search Application) Create a JSF web page that allows users to obtain a map of any address. Recall that a search for a location using the Google Maps API returns an array of GeoPoint objects. Search for locations a user enters in a Text Field and display a map of the first location in the resulting GeoPoint array. To handle multiple search results, display all results in a ListBox component. You can obtain a string representation of each result by invoking method toString on a GeoPoint object. Add a Button that allows users to select a result from the ListBox and displays a map for that result with a Map Marker showing the location on the map. Finally, use a Message Group to display messages regarding search errors. In case of an error, and when the page loads for the first time, recenter the map on a default location of your choosing.