26.4 *(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.

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.

—Eugéne Delacroix

Ajax-Enabled JavaServer™ Faces Web Applications

OBJECTIVES

In this chapter you will learn:

- To use data providers to access databases from web applications built in Netbeans.
- To include Ajax-enabled JSF components in a Netbeans web application project.
- To configure virtual forms that enable subsets of a form's input components to be submitted to the server.

Outline

27.1 Introduction
27.2 Accessing Databases in Web Applications
27.2.1 Building a Web Application That Displays Data from a Database
27.2.2 Modifying the Page Bean File for the AddressBook Application
27.3 Ajax-Enabled JSF Components
27.4 AutoComplete Text Field and Virtual Forms
27.4.1 Configuring Virtual Forms
27.4.2 JSP File with Virtual Forms and an AutoComplete Text Field
27.4.3 Providing Suggestions for an AutoComplete Text Field
27.5 Google Maps Map Viewer Component
27.5.1 Obtaining a Google Maps API Key
27.5.2 Adding a Map Viewer Component to a Page
27.5.3 JSP File with a Map Viewer Component
27.5.4 Page Bean That Displays a Map in the Map Viewer Component
27.6 Web Resources
Summary | Terminology | Self-Review Exercises | Exercises

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. [*Note:* This chapter assumes that you know Java. To learn more about Java, check out *Java How to Program, Seventh Edition,* or visit our Java Resource Centers at `www.deitel.com/ResourceCenters.html`.]

We present a single address book application developed in three stages to illustrate these concepts. The application is backed by a Java DB 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 **AutoComplete 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 **Map Viewer** component that is powered by Google Maps (`maps.google.com`).

As in Chapter 26, this chapter's examples were developed in Netbeans. 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. These Ajax-enabled components use the Dojo Toolkit (which we introduced in Chapter 15) on the client side.

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/iw3htp4/). 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

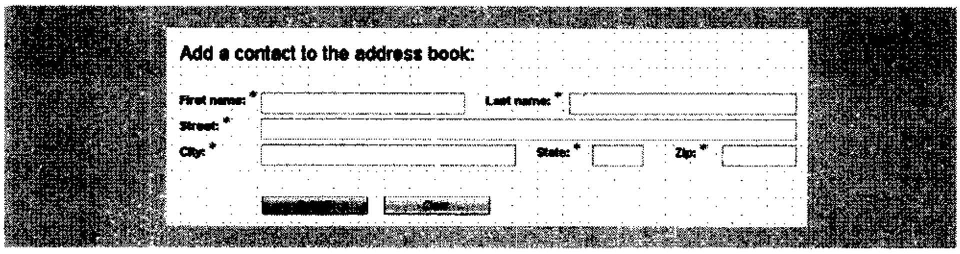
We now explain how to build the AddressBook application's GUI and set up a data binding that allows the **Table** component to display information from the database. We present the generated JSP file later in the section, and we discuss the related page bean file in Section 27.2.2. To build the AddressBook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

Step 1: Creating the Project

In Netbeans, create a **Visual 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 need 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.



Add a contact to the address book:

First name: * Last name: *

Street: *

City: * State: * Zip: *

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

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 `title` property to `Contacts`. We show how to configure the **Table** to interact with the `AddressBook` database shortly.

Step 4: Creating a Java DB Database

This example uses a database called `AddressBook` to store the address information. To create this database, perform the following steps:

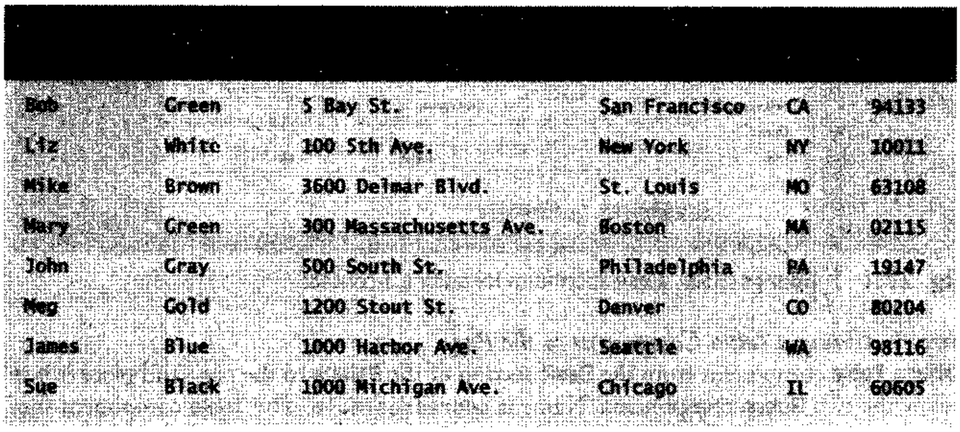
1. Select **Tools > Java DB Database > Create Java DB Database....**
2. Enter the name of the database to create (`AddressBook`), a username (`iw3htp4`) and a password (`iw3htp4`), then click **OK** to create the database.

In the Netbeans **Runtime** tab (to the right of the **Projects** and **Files** tabs), the preceding steps create a new entry in the **Databases** node showing the URL of the database (`jdbc:derby://localhost:1527/AddressBook`). This URL indicates that the database resides on the local machine and accepts connections on port 1527.

Step 5: Adding a Table and Data to the AddressBook *Database*

You can use the **Runtime** tab to create tables and to execute SQL statements that populate the database with data:

1. Click the **Runtime** tab and expand the **Databases** node.
2. Netbeans must be connected to the database to execute SQL statements. If Netbeans is already connected, proceed to *Step 3*. If Netbeans is not connected to the database, the icon ▣ appears next to the database's URL (`jdbc:derby://localhost:1527/AddressBook`). In this case, right click the icon and click **Connect....** Once connected, the icon changes to ■.
3. Expand the node for the `AddressBook` database, right click the **Tables** node and select **Execute Command...** to open a **SQL Command** editor in Netbeans. We provided the file `AddressBook.sql` in this chapter's examples folder. Open that file in a text editor, copy the SQL statements and paste them into the **SQL Command** editor in Netbeans. Then, highlight all the SQL commands, right click inside the **SQL Command** editor and select **Run Selection**. This will create the Addresses table with the sample data shown in Fig. 27.2. You may need to refresh the **Tables** node of the **Runtime** tab to see the new table.



Bob	Green	5 Bay St.	San Francisco	CA	94133
Liz	White	100 5th Ave.	New York	NY	10011
Mike	Brown	3600 Delmar Blvd.	St. Louis	MO	63108
Mary	Green	300 Massachusetts Ave.	Boston	MA	02115
John	Gray	500 South St.	Philadelphia	PA	19147
Meg	Gold	1200 Stout St.	Denver	CO	80204
James	Blue	1000 Harbor Ave.	Seattle	WA	98116
Sue	Black	1000 Michigan Ave.	Chicago	IL	60605

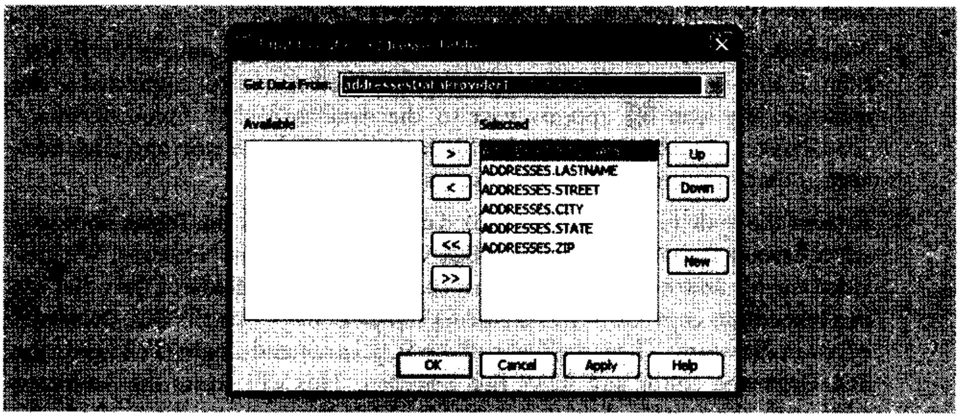
Fig. 27.2 | Addresses table data.

Step 6: 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.

To select specific columns to display, right click the **Table** component and select **Bind to Data** to display the **Bind to Data** dialog containing the list of the columns in the Addresses database table (Fig. 27.3). The items under the **Selected** heading will be displayed in the **Table**. To remove a column, select it and click the **<** button. We'd like to display all the columns in this example, so you should simply click **OK** to exit the dialog.

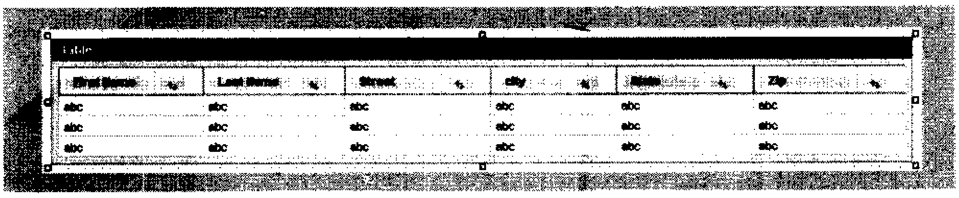
By default, the **Table** uses the database table's column names in all uppercase letters as headings. To change these headings, select a column and edit its headerText property in the **Properties** window. To select a column, click the column's name in the **Design** mode. We also changed the id property 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.



Get Data From: addressesDataProvider1

Selected:
ADDRESSES.LASTNAME
ADDRESSES.STREET
ADDRESSES.CITY
ADDRESSES.STATE
ADDRESSES.ZIP

Fig. 27.3 | Dialog for binding to the Addresses table.



Table

First Name	Last Name	Street	City	State	Zip
abc	abc	abc	abc	abc	abc
abc	abc	abc	abc	abc	abc
abc	abc	abc	abc	abc	abc

Fig. 27.4 | `Table` component after binding it to a database table and editing its column names for display purposes.

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. This adds buttons to the bottom of the **Table** for moving forward and backward between groups of contacts. You may use the **Table Layout** dialog's **Options** tab to select the number of rows to display at a time. To view this tab, right click the **Table**, select **Table Layout...**, then click the **Options** tab. For this example, we set the **Page Size** property to `5`.

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.

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'll see in the **Properties** window that its `CachedRowSet` property is set to `addressesRowSet`, an object (in the session bean) that implements interface `CachedRowSet`.

Step 7: Modifying **`addressesRowSet`***'s SQL Statement*

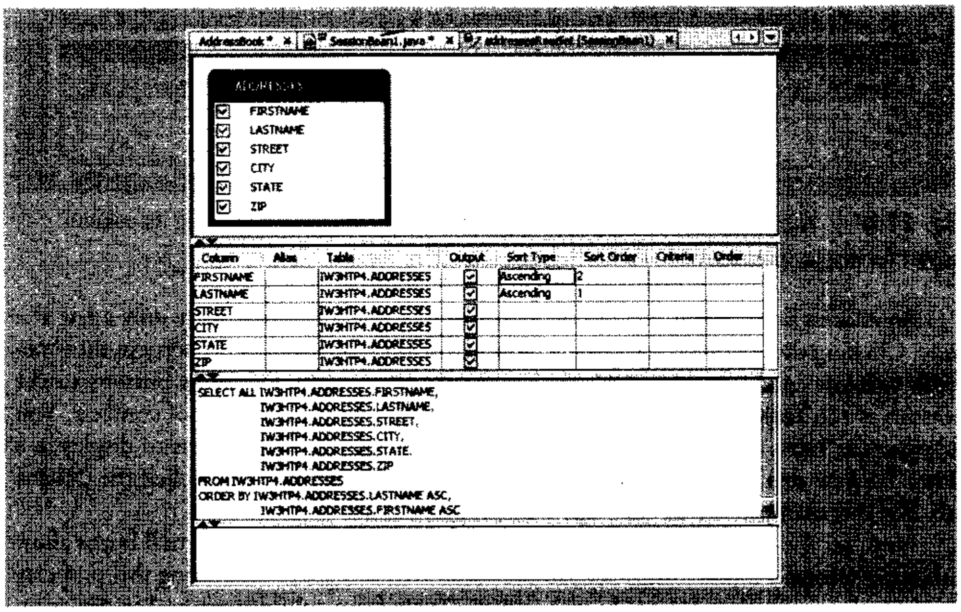
The `CachedRowSet` object wrapped by our `addressesDataProvider` is configured by default to execute a SQL query that selects all the data in the `Addresses` table of the `AddressBook` database. You can edit this SQL query by expanding the `SessionBean` node in the **Outline** window and double clicking 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 IW3HTP4.ADDRESSES.LASTNAME ASC,
   IW3HTP4.ADDRESSES.FIRSTNAME ASC
```

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

Step 8: 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` (except the ID column) and have length restrictions. For this reason, you should



Column	Alias	Table	Output	Sort Type	Sort Order	Criteria	Order
FIRSTNAME		IW3HTP4.ADDRESSES	☑	Ascending	2		
LASTNAME		IW3HTP4.ADDRESSES	☑	Ascending	1		
STREET		IW3HTP4.ADDRESSES	☑				
CITY		IW3HTP4.ADDRESSES	☑				
STATE		IW3HTP4.ADDRESSES	☑				
ZIP		IW3HTP4.ADDRESSES	☑				

```
SELECT ALL IW3HTP4.ADDRESSES.FIRSTNAME,
           IW3HTP4.ADDRESSES.LASTNAME,
           IW3HTP4.ADDRESSES.STREET,
           IW3HTP4.ADDRESSES.CITY,
           IW3HTP4.ADDRESSES.STATE,
           IW3HTP4.ADDRESSES.ZIP
FROM IW3HTP4.ADDRESSES
ORDER BY IW3HTP4.ADDRESSES.LASTNAME ASC,
           IW3HTP4.ADDRESSES.FIRSTNAME ASC
```

Fig. 27.5 | Editing addressesRowSet's SQL statement.

either add a **Length Validator** 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 30, 30, 150, 30, 2 and 5 characters, respectively.

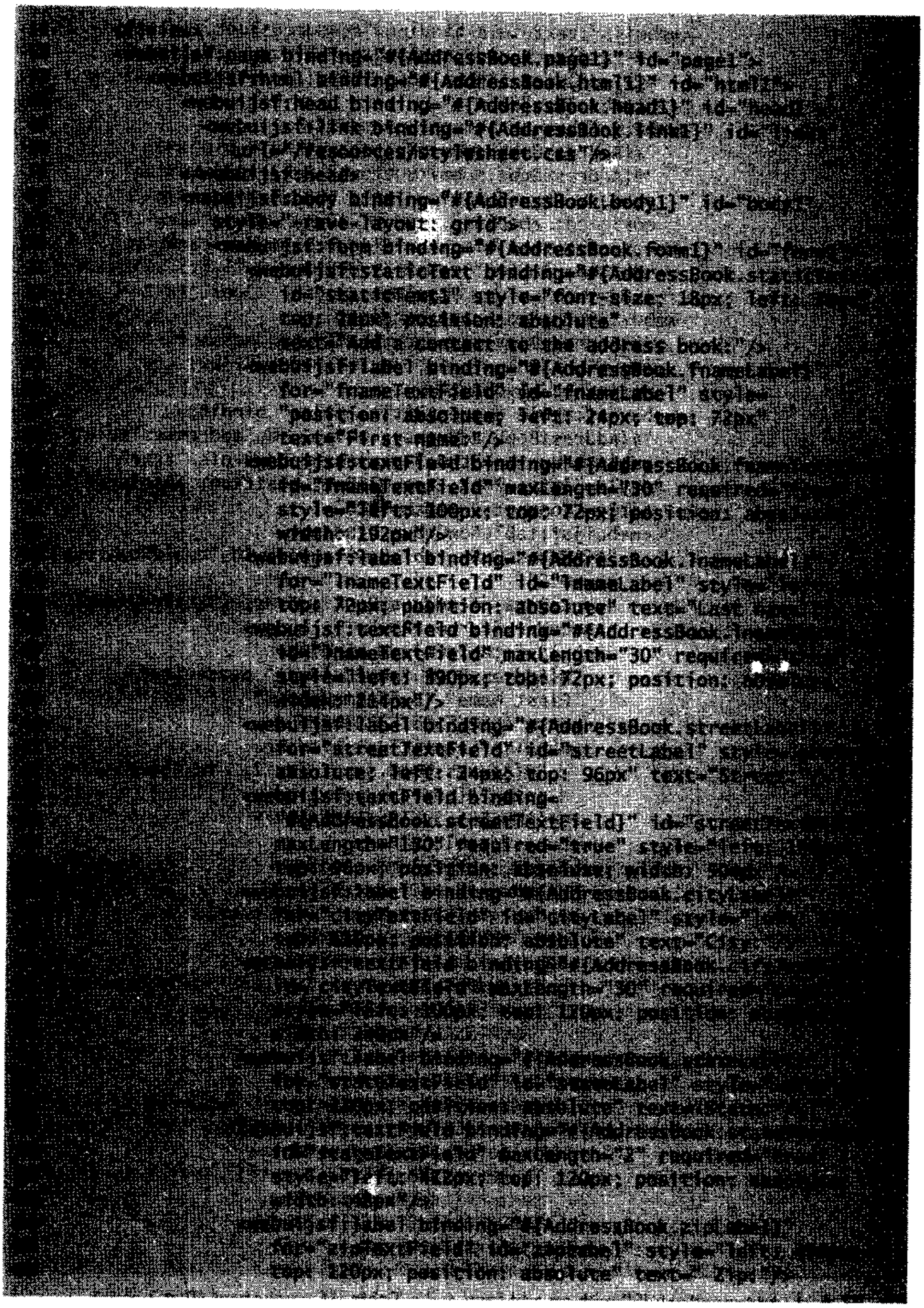
Finally, drag a **Message Group** component onto your page to the right of the **Text Fields**. A **Message Group** component displays system messages. We use this component to display an error message when an attempt to add a contact to the database fails. Set the **Message Group**'s showGlobalOnly 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 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:h="http://java.sun.com/jsf/html"
   xmlns:jsp="http://java.sun.com/JSP/Page"
   xmlns:webuijsf="http://www.sun.com/webui/webuijsf">
   <jsp:directive.page contentType="text/html;charset=UTF-8"
      pageEncoding="UTF-8"/>
```

Fig. 27.6 | AddressBook JSP with an add form and a **Table** JSF component (Part 1 of 5.)



```
[illegible]
   <webuijsf:page binding="#{AddressBook.page1}" id="page1">
      <webuijsf:html binding="#{AddressBook.html1}" id="html1">
         <webuijsf:head binding="#{AddressBook.head1}" id="head1">
            <webuijsf:link binding="#{AddressBook.link1}" id="link1"
               url="/resources/stylesheet.css"/>
         </webuijsf:head>
         <webuijsf:body binding="#{AddressBook.body1}" id="body1"
            style="-rave-layout: grid">
            <webuijsf:form binding="#{AddressBook.form1}" id="form1">
               <webuijsf:staticText binding="#{AddressBook.staticText1}"
                  id="staticText1" style="font-size: 18px; left: [illegible]
                  top: [illegible]; position: absolute"
                  text="Add a contact to the address book:"/>
               <webuijsf:label binding="#{AddressBook.fnameLabel}"
                  for="fnameTextField" id="fnameLabel" style=
                  "position: absolute; left: 24px; top: 72px"
                  text="First name:"/>
               <webuijsf:textField binding="#{AddressBook.fnameTextField}"
                  id="fnameTextField" maxLength="30" required="true"
                  style="left: 100px; top: 72px; position: absolute;
                  width: 192px"/>
               <webuijsf:label binding="#{AddressBook.lnameLabel}"
                  for="lnameTextField" id="lnameLabel" style="[illegible]
                  top: 72px; position: absolute" text="Last name:"/>
               <webuijsf:textField binding="#{AddressBook.lnameTextField}"
                  id="lnameTextField" maxLength="30" required="true"
                  style="left: 390px; top: 72px; position: absolute;
                  width: 214px"/>
               <webuijsf:label binding="#{AddressBook.streetLabel}"
                  for="streetTextField" id="streetLabel" style="position:
                  absolute; left: 24px; top: 96px" text="Street:"/>
               <webuijsf:textField binding=
                  "#{AddressBook.streetTextField}" id="streetTextField"
                  maxLength="150" required="true" style="left: 100px;
                  top: 96px; position: absolute; width: [illegible]
               <webuijsf:label binding="#{AddressBook.cityLabel}"
                  for="cityTextField" id="cityLabel" style="[illegible]
                  top: 120px; position: absolute" text="City:"/>
               <webuijsf:textField binding="#{AddressBook.cityTextField}"
                  id="cityTextField" maxLength="30" required="true"
                  style="left: 100px; top: 120px; position: absolute;
                  width: [illegible]"/>
               <webuijsf:label binding="#{AddressBook.stateLabel}"
                  for="stateTextField" id="stateLabel" style="[illegible]
                  top: 120px; position: absolute" text="State:"/>
               <webuijsf:textField binding="#{AddressBook.stateTextField}"
                  id="stateTextField" maxLength="2" required="true"
                  style="left: 412px; top: 120px; position: absolute;
                  width: [illegible]"/>
               <webuijsf:label binding="#{AddressBook.zipLabel}"
                  for="zipTextField" id="zipLabel" style="left: [illegible]
                  top: 120px; position: absolute" text="Zip:"/>
```

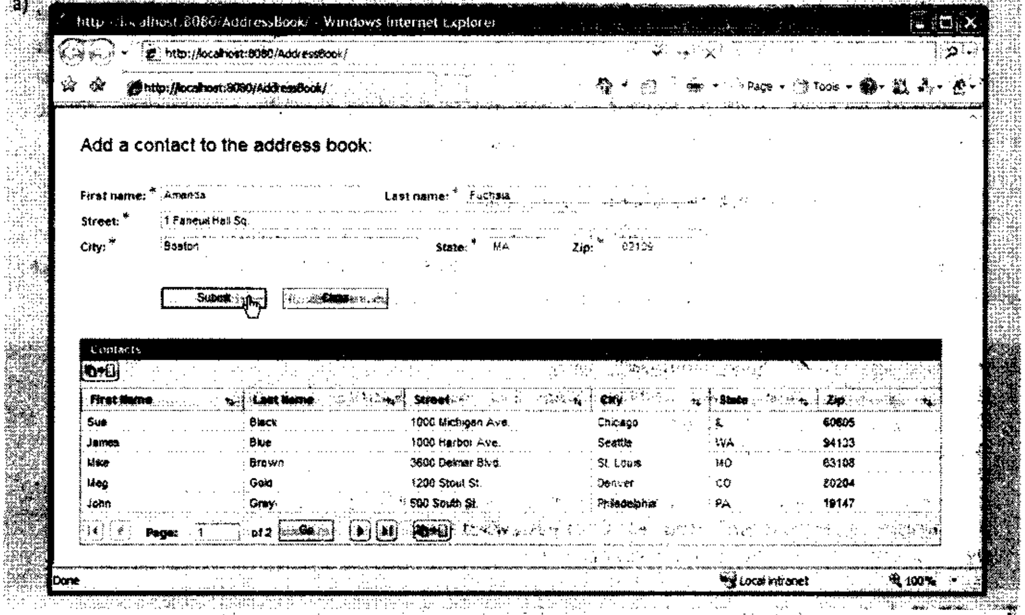
Fig. 27.6 | AddressBook JSP with an add form and a **Table** JSF component (Part 2 of 5.)

```
<webuijsf:textField binding="#{AddressBook.zipTextField}"
   id="zipTextField" maxLength="5" required="true"
   style="left: 534px; top: 128px; position: absolute;
   width: 70px"/>
<webuijsf:button actionExpression=
   "#{AddressBook.submitButton_action}" binding=
   "#{AddressBook.submitButton}" id="submitButton"
   primary="true" style="left: 100px; top: 160px;
   position: absolute; width: 100px" text="Submit"/>
<webuijsf:button binding="#{AddressBook.clearButton}"
   id="clearButton" reset="true" style="left: 248px; top:
   160px; position: absolute; width: 100px" text="Clear"/>
<webuijsf:messageGroup binding=
   "#{AddressBook.messageGroup1}" id="messageGroup1"
   showGlobalOnly="true" style="left: 624px; top: 72px;
   position: absolute"/>
<webuijsf:table augmentTitle="false" binding=
   "#{AddressBook.addressesTable}" id="addressesTable"
   paginateButton="true" paginationControls="true"
   style="left: 24px; top: 216px; position: absolute"
   title="Contacts" width="816">
   <webuijsf:tableRowGroup binding=
      "#{AddressBook.tableRowGroup1}" id="tableRowGroup1"
      rows="5"
      sourceData="#{AddressBook.addressesDataProvider}"
      sourceVar="currentRow">
      <webuijsf:tableColumn binding=
         "#{AddressBook.fnameColumn}" headerText=
         "First Name" id="fnameColumn"
         sort="ADDRESSES.FIRSTNAME">
         <webuijsf:staticText binding=
            "#{AddressBook.staticText2}" id="staticText2"
            text="#{currentRow.value[
            'ADDRESSES.FIRSTNAME']}"/>
      </webuijsf:tableColumn>
      <webuijsf:tableColumn binding=
         "#{AddressBook.lnameColumn}"
         headerText="Last Name" id="lnameColumn"
         sort="ADDRESSES.LASTNAME">
         <webuijsf:staticText binding=
            "#{AddressBook.staticText3}" id="staticText3"
            text="#{currentRow.value[
            'ADDRESSES.LASTNAME']}"/>
      </webuijsf:tableColumn>
      <webuijsf:tableColumn binding=
         "#{AddressBook.streetColumn}" headerText="Street"
         id="streetColumn" sort="ADDRESSES.STREET">
         <webuijsf:staticText binding=
            "#{AddressBook.staticText4}" id="staticText4"
            text="#{currentRow.value[
            'ADDRESSES.STREET']}"/>
      </webuijsf:tableColumn>
```

Fig. 27.6 | AddressBook JSP with an add form and a **Table** JSF component (Part 3 of 5.)

```
                     <webuijsf:tableColumn binding=
                        "#{AddressBook.cityColumn}" headerText="City"
                        id="cityColumn" sort="ADDRESSES.CITY">
                        <webuijsf:staticText binding=
                           "#{AddressBook.staticText5}" id="staticText5"
                           text="#{currentRow.value['ADDRESSES.CITY']}"/>
                     </webuijsf:tableColumn>
                     <webuijsf:tableColumn binding=
                        "#{AddressBook.stateColumn}" headerText="State"
                        id="stateColumn" sort="ADDRESSES.STATE">
                        <webuijsf:staticText binding=
                           "#{AddressBook.staticText6}" id="staticText6"
                           text="#{currentRow.value['ADDRESSES.STATE']}"/>
                     </webuijsf:tableColumn>
                     <webuijsf:tableColumn binding=
                        "#{AddressBook.zipColumn}" headerText="Zip"
                        id="zipColumn" sort="ADDRESSES.ZIP" width="106">
                        <webuijsf:staticText binding=
                           "#{AddressBook.staticText7}" id="staticText7"
                           text="#{currentRow.value['ADDRESSES.ZIP']}"/>
                     </webuijsf:tableColumn>
                  </webuijsf:tableRowGroup>
               </webuijsf:table>
            </webuijsf:form>
         </webuijsf:body>
      </webuijsf:html>
   </webuijsf:page>
   </f:view>
</jsp:root>
```

a)



http://localhost:8080/AddressBook/ - Windows Internet Explorer
http://localhost:8080/AddressBook/
http://localhost:8080/AddressBook/
Page
Tools
Done
Local intranet
100%

Add a contact to the address book:

First name: * Amanda Last name: * Fuchsia
Street: * 1 Faneuil Hall Sq.
City: * Boston State: * MA Zip: * 02109

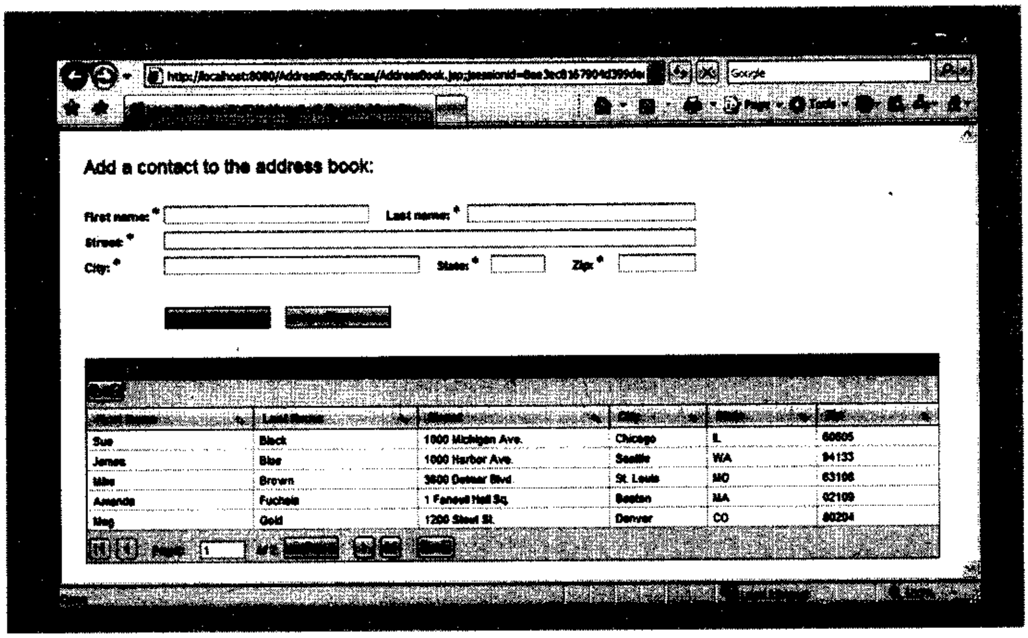
Submit Clear

Contacts

First Name	Last Name	Street	City	State	Zip
Sue	Black	1000 Michigan Ave.	Chicago	IL	60605
James	Blue	1000 Harbor Ave.	Seattle	WA	94133
Mike	Brown	3600 Delmar Blvd.	St. Louis	MO	63108
Meg	Gold	1200 Stout St.	Denver	CO	80204
John	Gray	500 South St.	Philadelphia	PA	19147

Page: 1 of 2 Go

Fig. 27.6 | AddressBook JSP with an add form and a **Table** JSF component (Part 4 of 5.)



Add a contact to the address book:

First name:* □ Last name:* □

Street:* □

City:* □ State:* □ Zip:* □

Sue	Black	1000 Michigan Ave.	Chicago	IL	60605
James	Blue	1000 Harbor Ave.	Seattle	WA	94133
Mike	Brown	3600 Delmar Blvd.	St. Louis	MO	63108
Amanda	Fuchsia	1 Faneuil Hall Sq.	Boston	MA	02109
Meg	Gold	1200 Stout St.	Denver	CO	80204

Fig. 27.6 | AddressBook JSP with an add form and a **Table** JSF component (Part 5 of 5.)

Lines 21–75 contain the JSF components for the form that gathers user input. Lines 80–138 define the **Table** element (`webuijsf:table`) that displays address information from the database. JSF **Tables** may have multiple groups of rows displaying different data. This **Table** has a single `webuijsf:tableRowGroup` with a start tag in lines 85–89. 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 `webuijsf:tableColumn` element (e.g., lines 90–98) contains a `webuijsf:staticText` element with its `text` attribute bound to a column in the data provider `currentRow`. These `webuijsf: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 Netbeans 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).

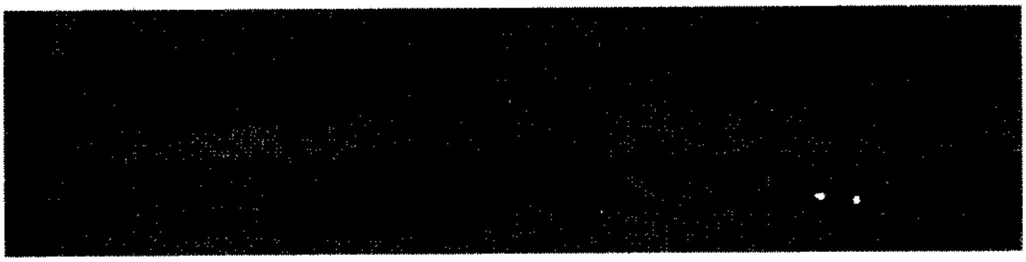


Fig. 27.7 | Session Bean that initializes the data source for the `AddressBook` database. (Part 1 of 2.)

```
private void _init() throws Exception
{
   addressesRowSet.setDataSourceName(
      "java:comp/env/jdbc/dataSource" );
   addressesRowSet.setCommand(
      "SELECT ALL IW3HTP4.ADDRESSES.FIRSTNAME, \n" +
      "IW3HTP4.ADDRESSES.LASTNAME, \n" +
      "IW3HTP4.ADDRESSES.STREET, \n" +
      "IW3HTP4.ADDRESSES.CITY, \n" +
      "IW3HTP4.ADDRESSES.STATE, \n" +
      "IW3HTP4.ADDRESSES.ZIP \n" +
      "FROM IW3HTP4.ADDRESSES\n" +
      "ORDER BY IW3HTP4.ADDRESSES.LASTNAME ASC, \n" +
      "IW3HTP4.ADDRESSES.FIRSTNAME ASC " );
   addressesRowSet.setTableName( "ADDRESSES" );
} // end method _init

// To save space, we omitted the code in lines 42-78. The complete
// source code is provided with this chapter's examples.
```

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

The _init method (lines 14–29) configures addressesRowSet to interact with the AddressBook database (lines 16–28). 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. Line 28 sets the RowSet's table name.

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.

```
1   // Fig. 27.8: AddressBook.java
2   // Page bean for AddressBook.jsp.
3   package addressbook;
4
5   import com.sun.data.provider.RowKey;
6   import com.sun.data.provider.impl.CachedRowSetDataProvider;
7   import com.sun.rave.web.ui.appbase.AbstractPageBean;
8   import com.sun.webui.jsf.component.Body;
9   import com.sun.webui.jsf.component.Button;
10  import com.sun.webui.jsf.component.Form;
11  import com.sun.webui.jsf.component.Head;
12  import com.sun.webui.jsf.component.Html;
13  import com.sun.webui.jsf.component.Label;
14  import com.sun.webui.jsf.component.Link;
15  import com.sun.webui.jsf.component.MessageGroup;
16  import com.sun.webui.jsf.component.Page;
17  import com.sun.webui.jsf.component.StaticText;
18  import com.sun.webui.jsf.component.Table;
19  import com.sun.webui.jsf.component.TableColumn;
20  import com.sun.webui.jsf.component.TableRowGroup;
21  import com.sun.webui.jsf.component.TextField;
22  import com.sun.webui.jsf.model.DefaultTableDataProvider;
23  import javax.faces.FacesException;
24
25  public class AddressBook extends AbstractPageBean
26  {
27     private int __placeholder;
28
29     private void _init() throws Exception
30     {
31        addressesDataProvider.setCachedRowSet(
32           ( javax.sql.rowset.CachedRowSet ) getValue(
33              "#{SessionBean1.addressesRowSet}" ) );
34        addressesTable.setInternalVirtualForm( true );
35     }// end method _init
36
37     // To save space, we omitted the code in lines 37-505. The complete
38     // source code is provided with this chapter's examples.
39
506    public void prerender()
507    {
508       addressesDataProvider.refresh();
509    } // end method prerender
510
511    public void destroy()
512    {
513       addressesDatıProvider.close();
514    } // end method destroy
515
516    // To save space, we omitted the code in lines 516-530. The complete
517    // source code is provided with this chapter's examples.
518
```

Fig. 27.8 | Page bean for adding a contact to the address book. (Part 1 of 2.)

```
531      // action handler that adds a contact to the AddressBook database
532      // when the user clicks Submit
533      public String submitButton_action()
534      {
535         if ( addressesDataProvider.canAppendRow() )
536         {
537            try
538            {
539               RowKey rk = addressesDataProvider.appendRow();
540               addressesDataProvider.setCursorRow( rk );
541
542               addressesDataProvider.setValue( "ADDRESSES.FIRSTNAME",
543                  fnameTextField.getValue() );
544               addressesDataProvider.setValue( "ADDRESSES.LASTNAME",
545                  lnameTextField.getValue() );
546               addressesDataProvider.setValue( "ADDRESSES.STREET",
547                  streetTextField.getValue() );
548               addressesDataProvider.setValue( "ADDRESSES.CITY",
549                  cityTextField.getValue() );
550               addressesDataProvider.setValue( "ADDRESSES.STATE",
551                  stateTextField.getValue() );
552               addressesDataProvider.setValue( "ADDRESSES.ZIP",
553                  zipTextField.getValue() );
554               addressesDataProvider.commitChanges();
555
556               // reset text fields
557               lnameTextField.setValue( "" );
558               fnameTextField.setValue( "" );
559               streetTextField.setValue( "" );
560               cityTextField.setValue( "" );
561               stateTextField.setValue( "" );
562               zipTextField.setValue( "" );
563            } // end try
564            catch ( Exception ex )
565            {
566               error( "The address book was not updated. " +
567                  ex.getMessage() );
568            } // end catch
569         } // end if
570
571         return null;
572      } // end method submitButton_action
573   } // end class AddressBook
```

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

Lines 533–572 contain the event-handling code for the **Submit** button. Line 535 determines whether a new row can be appended to the data provider. If so, a new row is appended at line 539. Every row in a CachedRowSetDataProvider has its own key; method **appendRow** returns the key for the new row. Line 540 sets the data provider's cursor to the new row, so that any changes we make to the data provider affect that row. Lines 542–553 set each of the row's columns to the values entered by the user in the corresponding **Text Fields**. Line 554 stores the new contact by calling method `commitChanges` of class `CachedRowSetDataProvider` to insert the new row into the `AddressBook` database.

Lines 557–562 clear 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 resets them to the values they held the last time the form was submitted.

Lines 564–568 catch any exceptions that might occur while updating the `AddressBook` database. Lines 566–567 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 508 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 513) in the `destroy` method.

27.3 Ajax-Enabled JSF Components

The Java BluePrints Aja. .omponent 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 Netbeans. We demonstrate the **AutoComplete Text Field** and **Map Viewer** components in the next two sections.

AutoComplete Text Field	Makes Ajax requests to display a list of suggestions as the user types in the text field.
Buy Now Button	Initiates a transaction through the PayPal website.
Map Viewer	Uses the Google Maps API to display a map that pans, zooms, and can display markers for locations of interest.
Popup Calendar	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.
Progress Bar	Visually displays the progress of a long-running operation. Uses a programmer-supplied calculation to determine the progress percentage.
Rating	Provides a customizable five-star rating bar that can display messages as the user moves the mouse over the ratings.
Rich Textarea Editor	Provides an editable text area that allows the user to format text with fonts, colors, hyperlinks and backgrounds.
Select Value Text Field	Displays a list of suggestions in a drop-down list as the user types, similar to the **AutoComplete Text Field**.

Fig. 27.9 | Java BluePrints component library's Ajax-enabled components.

Downloading the Java BluePrints Ajax-Enabled Components

To use the Java BluePrints Ajax-enabled components in Netbeans, you must download and import them. The IDE provides a wizard for installing this group of components (Internet access is required). 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, locate and select **BluePrints AJAX Components** then click the **Add >** button to add them 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.

Importing the Java BluePrints Ajax-Enabled Components into the Netbeans Palette

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

To see the new components in the **Palette**, you must add the **BluePrints AJAX Components** library to your visual web application. To do so, make sure your application's node is expanded in the **Projects** tab. Right click the **Component Libraries** node and select **Add Component Library**. In the **Add Component Library** dialog box, select the **BluePrints AJAX Components library** and click **Add Component Library**. You should now see two new nodes in the **Palette**. The first, **BluePrints AJAX Components**, provides the eight components listed in Fig. 27.9. The second, **BluePrints 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 modifying the form in 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*

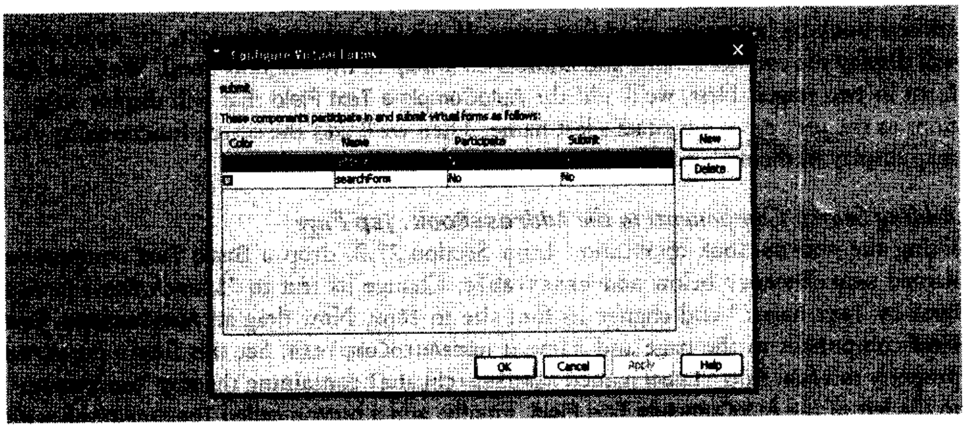
Using the `AddressBook` application from Section 27.2, drop a **Static Text** component named `searchHeader` below `addressesTable`. Change its text to `"Search the address book by last name:"` and change its font size to `18px`. Now drag an **AutoComplete Text Field** component to the page and name 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 Fields** 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** component and one or more **participant** components 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. Click **OK** to exit.

Repeat the process described above to create a second virtual form named `searchForm` for the lower form. Figure 27.10 shows the **Configure Virtual Forms** dialog after both virtual forms have been added. The **Look Up** `Button` should submit the `searchForm`, and `name` `AutoComplete` 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 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.



Configure Virtual Forms

submit

These components participate in and submit virtual forms as follows:

Color	Name	Participate	Submit
[illegible]	[illegible]	[illegible]	[illegible]
▨	searchForm	No	No

Fig. 27.10 | **Configure Virtual Forms** dialog.

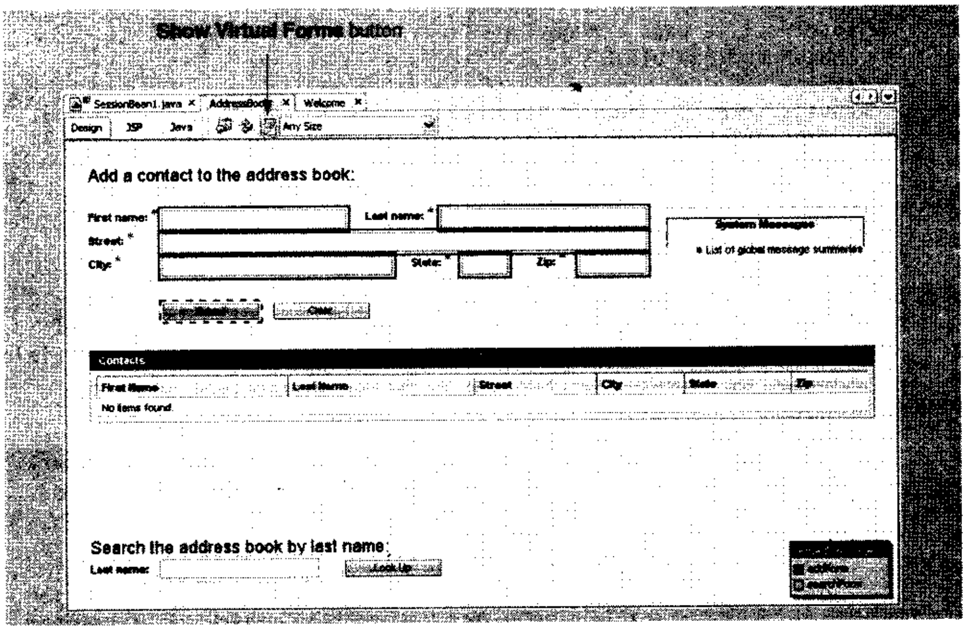


Fig. 27.11 | Virtual forms legend.

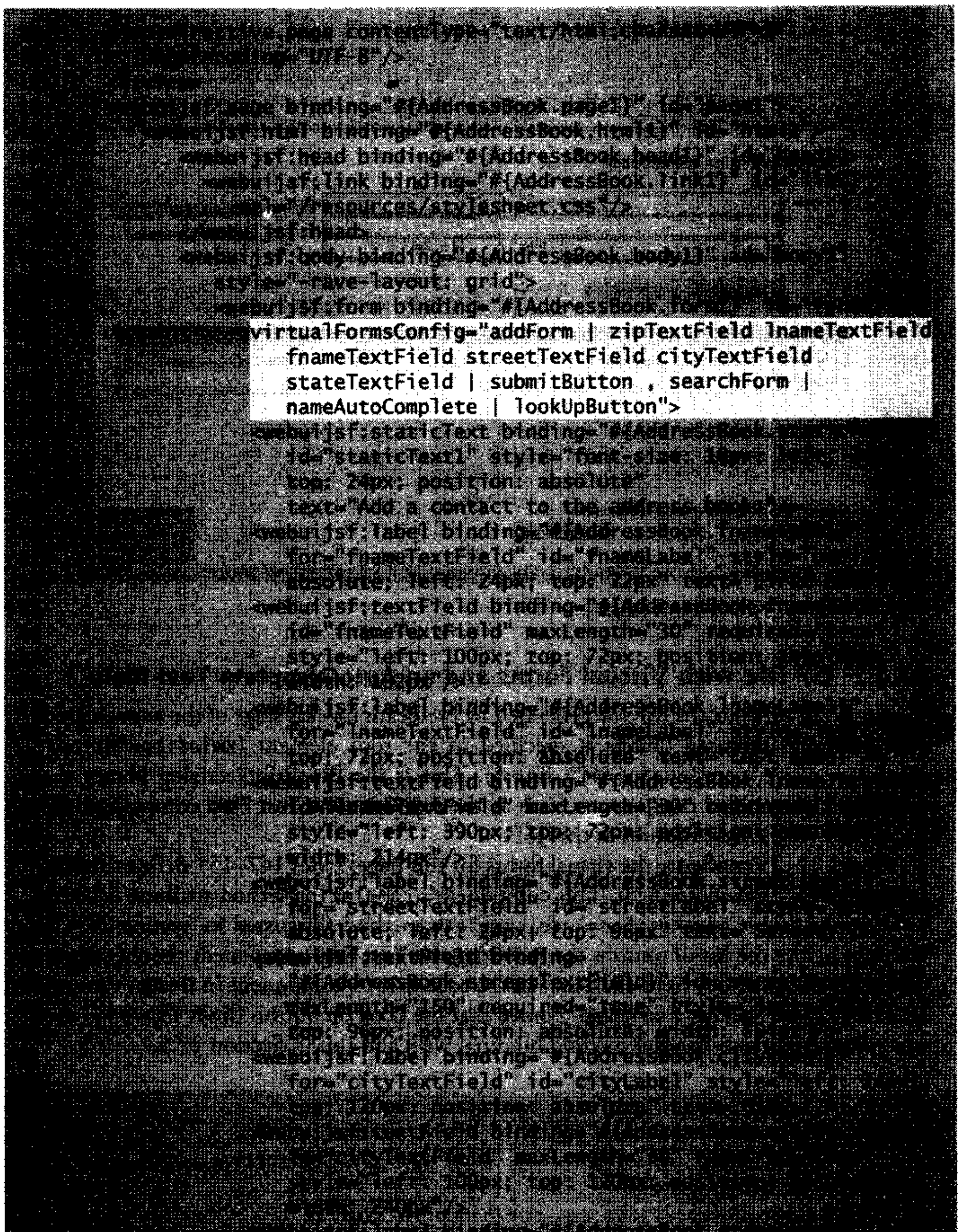
27.4.2 JSP File with Virtual Forms and an AutoComplete Text Field

Figure 27.12 presents the JSP file generated by Netbeans for this stage of the AddressBook application. A new tag library is specified in the root element (xmlns:bp="http://java.sun.com/blueprints/ui/14"; line 5). 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.

Lines 22–25 configure the virtual forms for this page. Lines 147–151 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 155–157) does not specify an action-handler method binding; we'll add this in Section 27.5.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!-- Fig. 27.12: AddressBook.jsp -->
3  <!-- AddressBook JSP with an add form and a Table JSF component -->
4  <jsp:root version="1.2"
5     xmlns:bp="http://java.sun.com/blueprints/ui/14"
6     xmlns:f="http://java.sun.com/jsf/core"
7     xmlns:h="http://java.sun.com/jsf/html"
8     xmlns:jsp="http://java.sun.com/JSP/Page"
9     xmlns:webuijsf="http://www.sun.com/webui/webuijsf">
```

Fig. 27.12 | AddressBook JSP with an **AutoComplete Text Field** component. (Part 1 of 5.)



```
virtualFormsConfig="addForm | zipTextField lnameTextField
   fnameTextField streetTextField cityTextField
   stateTextField | submitButton , searchForm |
   nameAutoComplete | lookUpButton">
```

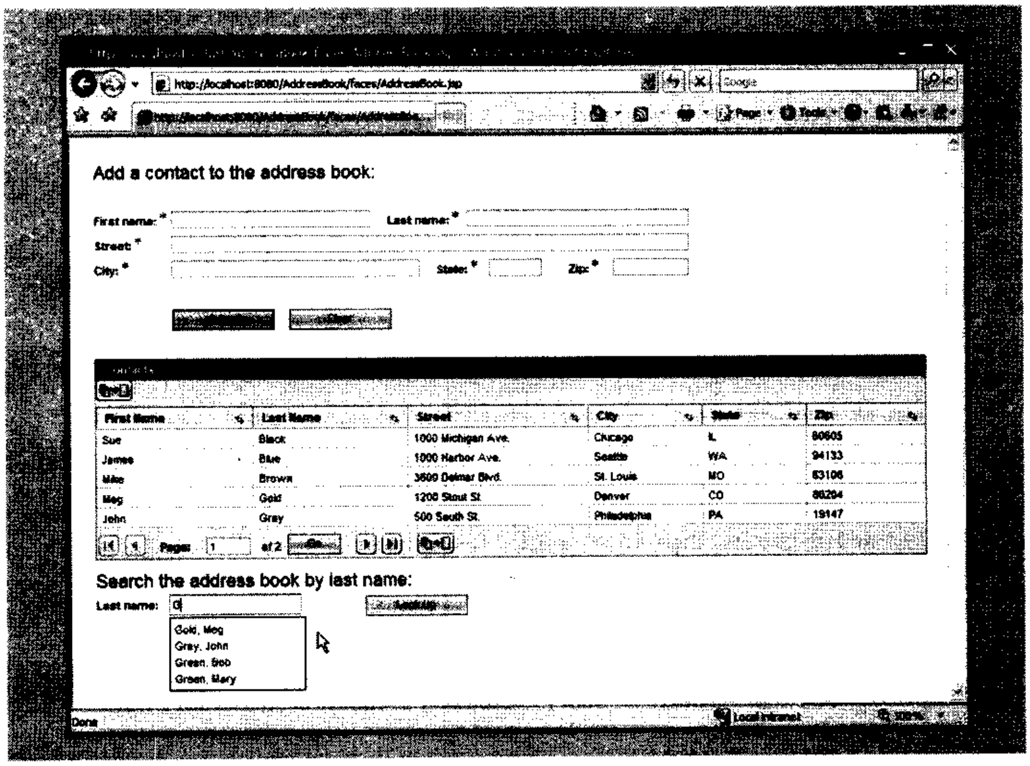
Fig. 27.12 | AddressBook JSP with an **AutoComplete Text Field** component. (Part 2 of 5.)

```
<webuijsf:textField binding="#{AddressBook.[illegible]
    [illegible]TextField" maxLength="2" [illegible]
    style="left: [illegible]px; top: 320px; position: [illegible]
    width: 40px"/>
<webuijsf:label binding="#{AddressBook.zip[illegible]
    for="zipTextField" id="zipLabel" style="[illegible]
    top: 320px; position: absolute" text="[illegible]
<webuijsf:textField binding="#{AddressBook.[illegible]
    id="zipTextField" maxLength="5" required="[illegible]
    style="left: [illegible]px; top: 320px; position: [illegible]
    width: 70px"/>
<webuijsf:button actionExpression=
    "#{AddressBook.[illegible]
    "#{AddressBook.[illegible]
    "true" style="left: [illegible]px; top: 160px; [illegible]
    absolute; width: 100px" text="Submit"/>
<webuijsf:button binding="#{AddressBook.[illegible]
    id="clearButton" reset="true" style="[illegible]
    160px; position: absolute; width: 100px" [illegible]
<webuijsf:messageGroup binding=
    "#{AddressBook.messageGroup1}" id="[illegible]
    showGlobalOnly="true" style="left: [illegible]
    position: absolute"/>
<webuijsf:table augmentTitle="false" binding=
    "#{AddressBook.addressesTable}" [illegible]
    [illegible] paginationControls="[illegible]
    style="left: 24px; top: 210px; position: [illegible]
    title="Contacts" width="816">
    <webuijsf:tableRowGroup binding=
        "#{AddressBook.tableRowGroup1}" [illegible]
        rows="5" [illegible]
        "#{AddressBook.[illegible]
        [illegible]
        <webuijsf:tableColumn binding=
            "#{AddressBook.tableColumn1}" [illegible]
            [illegible]
            sort="ADDRESSES.FIRSTNAME">
            <webuijsf:staticText binding=
                "#{AddressBook.staticText1}" [illegible]
                text="#{currentRow.value[
                'ADDRESSES.FIRSTNAME']}"/>
        </webuijsf:tableColumn>
        <webuijsf:tableColumn binding=
            "#{AddressBook.tableColumn2}" [illegible]
            [illegible]
            sort="ADDRESSES.LASTNAME">
            <webuijsf:staticText binding=
                "#{AddressBook.staticText2}" [illegible]
                text="#{currentRow.value[
                'ADDRESSES.LASTNAME']}"/>
        </webuijsf:tableColumn>
```

Fig. 27.12 | AddressBook JSP with an **AutoComplete Text Field** component. (Part 3 of 5.)

```
                  <webuijsf:tableColumn binding=
                     "#{AddressBook.streetColumn}" headerText="Street"
                     id="streetColumn" sort="ADDRESSES.STREET">
                     <webuijsf:staticText binding=
                        "#{AddressBook.staticText4}" id="staticText4"
                        text="#{currentRow.value[
                        'ADDRESSES.STREET']}"/>
                  </webuijsf:tableColumn>
                  <webuijsf:tableColumn binding=
                     "#{AddressBook.cityColumn}" headerText="City"
                     id="cityColumn" sort="ADDRESSES.CITY">
                     <webuijsf:staticText binding=
                        "#{AddressBook.staticText5}" id="staticText5"
                        text="#{currentRow.value['ADDRESSES.CITY']}"/>
                  </webuijsf:tableColumn>
                  <webuijsf:tableColumn binding=
                     "#{AddressBook.stateColumn}" headerText="State"
                     id="stateColumn" sort="ADDRESSES.STATE">
                     <webuijsf:staticText binding=
                        "#{AddressBook.staticText6}" id="staticText6"
                        text="#{currentRow.value['ADDRESSES.STATE']}"/>
                  </webuijsf:tableColumn>
                  <webuijsf:tableColumn binding=
                     "#{AddressBook.zipColumn}" headerText="Zip"
                     id="zipColumn" sort="ADDRESSES.ZIP" width="106">
                     <webuijsf:staticText binding=
                        "#{AddressBook.staticText7}" id="staticText7"
                        text="#{currentRow.value['ADDRESSES.ZIP']}"/>
                  </webuijsf:tableColumn>
               </webuijsf:tableRowGroup>
            </webuijsf:table>
            <webuijsf:staticText binding="#{AddressBook.searchHeader}"
               id="searchHeader" style="font-size: 18px; left: 24px;
               top: 420px; position: absolute"
               text="Search the address book by last name:"/>
            <bp:autoComplete binding="#{AddressBook.nameAutoComplete}"
               completionMethod=
               "#{AddressBook.nameAutoComplete_complete}"
               id="nameAutoComplete"
               style="left: 96px; top: 444px; position: absolute"/>
            <webuijsf:label binding="#{AddressBook.label1}"
               for="nameAutoComplete" id="label1" style="left: 24px;
               top: 447px; position: absolute" text="Last name:"/>
            <webuijsf:button binding="#{AddressBook.lookUpButton}"
               id="lookUpButton" style="left: 288px; top: 446px;
               position: absolute; width: 100px" text="Look Up"/>
         </webuijsf:form>
      </webuijsf:body>
   </webuijsf:html>
  </webuijsf:page>
 </f:view>
</jsp:root>
```

Fig. 27.12 | AddressBook JSP with an **AutoComplete Text Field** component. (Part 4 of 5.)



Add a contact to the address book:

First name: * Last name: *
Street: *
City: * State: * Zip: *

Search the address book by last name:

Last name: G

First Name	Last Name	Street	City	State	Zip
Sue	Black	1000 Michigan Ave.	Chicago	IL	60605
James	Blue	1000 Harbor Ave.	Seattle	WA	94133
Mike	Brown	3600 Delmar Blvd.	St. Louis	MO	63106
Meg	Gold	1200 Stout St.	Denver	CO	80204
John	Gray	500 South St.	Philadelphia	PA	19147

Gold, Meg
Gray, John
Green, Bob
Green, Mary

Fig. 27.12 | AddressBook JSP with an **AutoComplete Text Field** component. (Part 5 of 5.)

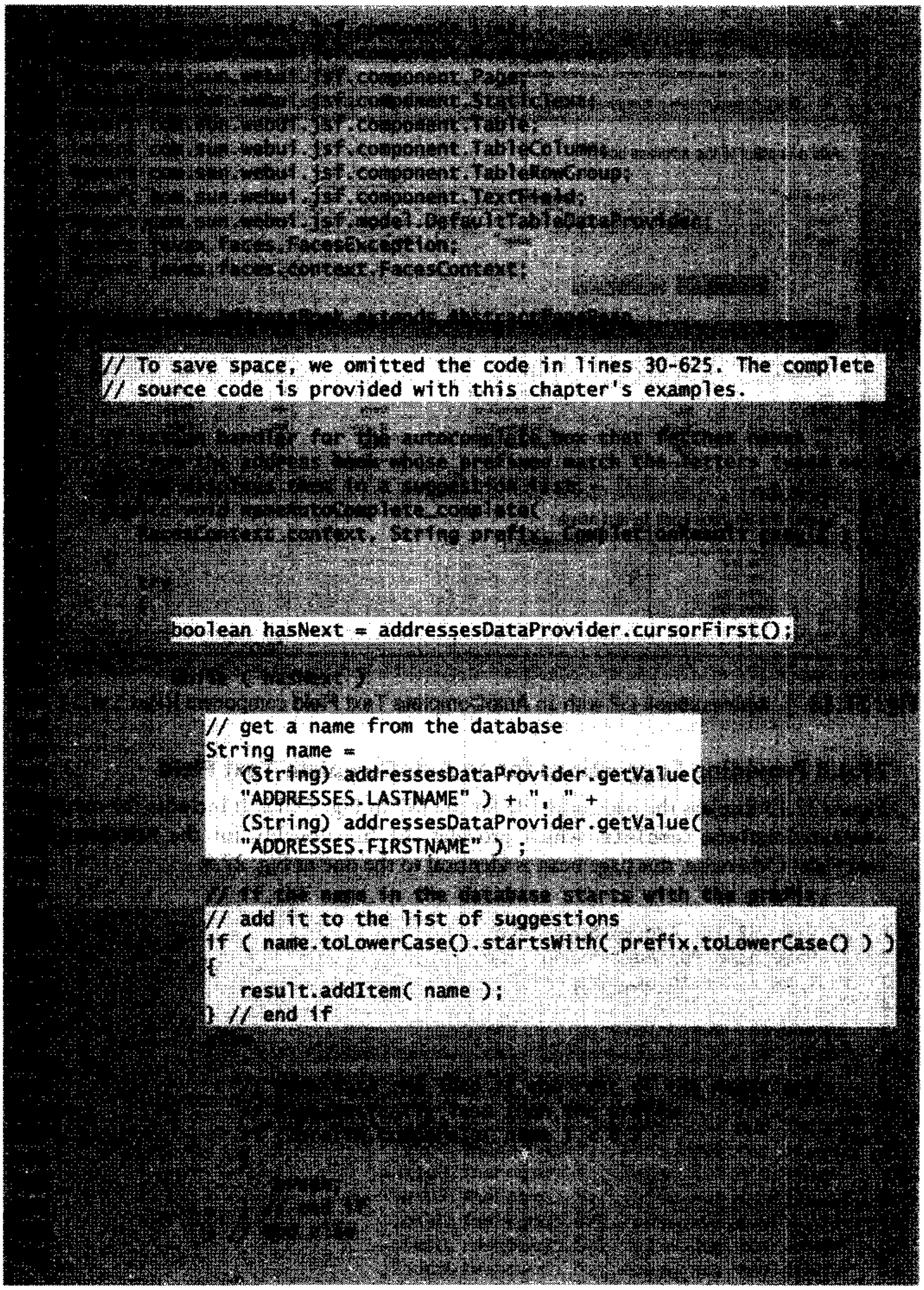
27.4.3 Providing Suggestions for an AutoComplete Text Field

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.

```
// Fig. 27.13: AddressBook.java
// Page bean for AddressBook.jsp.
package addressbook;

import com.sun.data.provider.RowKey;
[illegible]
[illegible]
[illegible]
[illegible]
[illegible]
[illegible]
[illegible]
[illegible]
[illegible]
[illegible]
[illegible]
```

Fig. 27.13 | Page bean that suggests names in the **AutoComplete Text Field**. (Part 1 of 3.)



```
[illegible]
[illegible]
       com.sun.webui.jsf.component.Page;
       com.sun.webui.jsf.component.StaticText;
       com.sun.webui.jsf.component.Table;
       com.sun.webui.jsf.component.TableColumn;
       com.sun.webui.jsf.component.TableRowGroup;
       com.sun.webui.jsf.component.TextField;
       com.sun.webui.jsf.model.DefaultTableDataProvider;
        javax.faces.FacesException;
        javax.faces.context.FacesContext;

                 [illegible] extends AbstractPageBean
[illegible]
   // To save space, we omitted the code in lines 30-625. The complete
   // source code is provided with this chapter's examples.

   // 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 )
         {
            // 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 );
            } // end if
[illegible]
```

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

```
661             // move cursor to next row of database
662             hasNext = addressesDataProvider.cursorNext();
663          } // end while
664       } // end try
665       catch ( Exception ex )
666       {
667          result.addItem( "Exception getting matching names." );
668       } // end catch
669    } // end method nameAutoComplete_complete
670 } // end class AddressBook
```

Fig. 27.13 | Page bean that suggests names in the **AutoComplete Text Field**. (Part 3 of 3.)

Method nameAutoComplete_complete (lines 629–669) 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 634 sets the cursor to the first row in the data provider. Line 636 determines whether there are more rows in the data provider. If so, lines 639–643 retrieve the last name and first name from the current row and create a String in the format *last name, first name*. Line 647 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 649 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 655 tests whether the current name is alphabetically greater than the prefix. If so, line 657 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 662 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 665–668 catch any exceptions generated while searching the database. Line 667 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 Sun Java System Application Server, enter `http://localhost:8080/` 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 **BluePrints 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 the **Outline** tab 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 **BluePrints 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**. 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. On the **Runtime** tab, expand the **Databases** node, the **AddressBook** database's node and its **Tables** node to reveal the **Addresses** table. Drag the **Addresses** table onto the page to create the new data provider. Select the new data provider in the **Navigator** tab and change its `id` to `addressesSearchDataProvider`. In the **Outline** tab, a new node named `addressesRowSet1` has been added to the `SessionBean1` node. Change the `id` of `addressesRowSet1` to `addressesSearchRowSet`.

Double click the `addressesSearchRowSet` 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 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 27.14 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 165–166. You can paste your key into the **Map Viewer** component's `key` property in the **Properties** window.]

Lines 162–168 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.

```
[illegible]
```

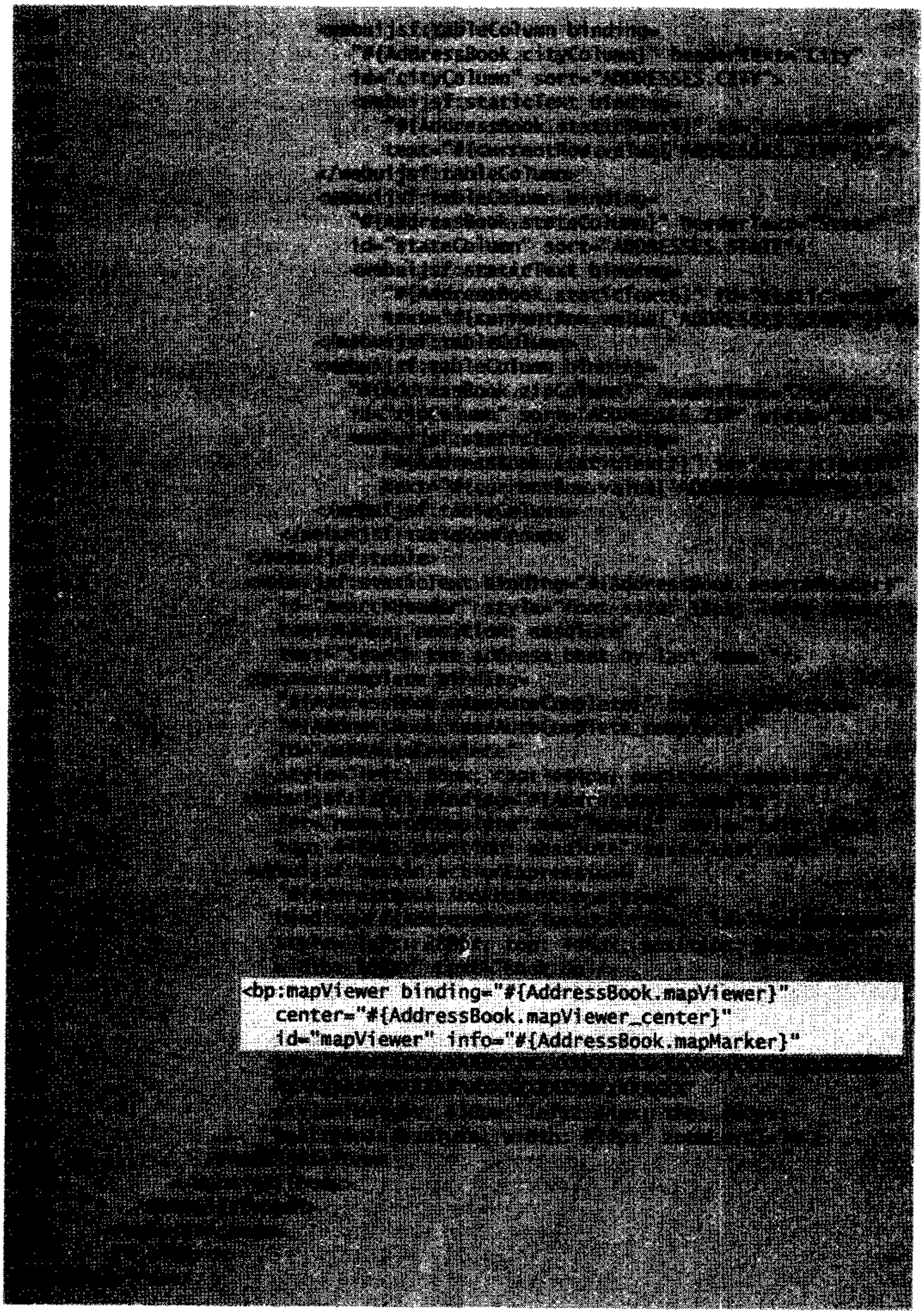
Fig. 27.14 | `AddressBook` JSP with a **Map Viewer** component. (Part 1 of 5.)

```
         <webuijsf:head binding="#{AddressBook.head1}" id="head1">
            <webuijsf:link binding="#{AddressBook.link1}" id="link1"
               url="/resources/stylesheet.css"/>
         </webuijsf:head>
         <webuijsf:body binding="#{AddressBook.body1}" id="body1"
            style="-rave-layout: grid">
            <webuijsf:form binding="#{AddressBook.form1}" id="form1"
               virtualFormsConfig="addForm | zipTextField lnameTextField
               fnameTextField streetTextField cityTextField stateTextField
               | submitButton , searchForm | nameAutoComplete |
               lookUpButton">
               <webuijsf:staticText binding=
                  "#{AddressBook.staticText1}" id="staticText1" style=
                  "font-size: 18px; left: 24px; top: 24px; position:
                  absolute" text="Add a contact to the address book:"/>
               <webuijsf:label binding="#{AddressBook.fnameLabel}"
                  for="fnameTextField" id="fnameLabel" style="position:
                  absolute; left: 24px; top: 72px" text="First name:"/>
               <webuijsf:textField binding="#{AddressBook.fnameTextField}"
                  id="fnameTextField" maxLength="30" required="true"
                  style="left: 100px; top: 72px; position: absolute;
                  width: 192px"/>
               <webuijsf:label binding="#{AddressBook.lnameLabel}"
                  for="lnameTextField" id="lnameLabel" style="left: 312px;
                  top: 72px; position: absolute" text="Last name:"/>
               <webuijsf:textField binding="#{AddressBook.lnameTextField}"
                  id="lnameTextField" maxLength="30" required="true"
                  style="left: 390px; top: 72px; position: absolute;
                  width: 214px"/>
               <webuijsf:label binding="#{AddressBook.streetLabel}"
                  for="streetTextField" id="streetLabel" style="position:
                  absolute; left: 24px; top: 96px" text="Street:"/>
               <webuijsf:textField binding=
                  "#{AddressBook.streetTextField}" id="streetTextField"
                  maxLength="150" required="true" style="left: 100px;
                  top: 96px; position: absolute; width: 504px"/>
               <webuijsf:label binding="#{AddressBook.cityLabel}"
                  for="cityTextField" id="cityLabel" style="left: 24px;
                  top: 120px; position: absolute" text="City:"/>
               <webuijsf:textField binding="#{AddressBook.cityTextField}"
                  id="cityTextField" maxLength="30" required="true"
                  style="left: 100px; top: 120px; position: absolute;
                  width: 240px"/>
               <webuijsf:label binding="#{AddressBook.stateLabel}"
                  for="stateTextField" id="stateLabel"
                  style="left: 360px; top: 120px; position: absolute"
                  text="State:"/>
               <webuijsf:textField binding="#{AddressBook.stateTextField}"
                  id="stateTextField" maxLength="2" required="true"
                  style="left: 412px; top: 120px; position: absolute;
                  width: 48px"/>
               <webuijsf:label binding="#{AddressBook.zipLabel}"
                  for="zipTextField" id="zipLabel" style="left: 490px;
```

Fig. 27.14 | AddressBook JSP with a **Map Viewer** component. (Part 2 of 5.)

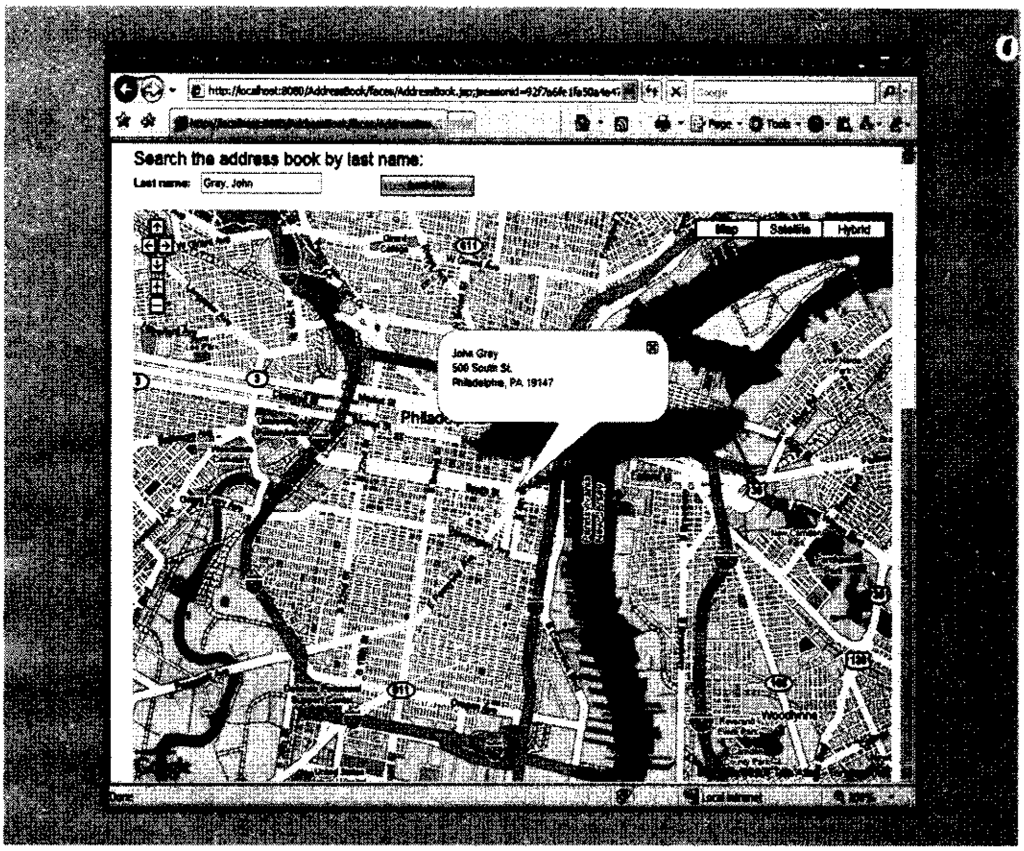
```
  top: 120px; position: absolute" text="Zip:"/>
<webuijsf:textField binding="#{AddressBook.zipTextField}"
  id="zipTextField" maxLength="5" required="true"
  style="left: 534px; top: 120px; position: absolute;
  width: 70px"/>
<webuijsf:button actionExpression=
  "#{AddressBook.submitButton_action}" binding=
  "#{AddressBook.submitButton}" id="submitButton"
  primary="true" style="left: 100px; top: 160px; position:
  absolute; width: 100px" text="Submit"/>
<webuijsf:button binding="#{AddressBook.clearButton}"
  id="clearButton" reset="true" style="left: [illegible]
  160px; position: absolute; width: 100px" [illegible]
<webuijsf:messageGroup binding=
  "#{AddressBook.messageGroup1}" id="messageGroup1"
  showGlobalOnly="true" style="left: [illegible]
  position: absolute"/>
<webuijsf:table augmentTitle="false" binding=
  "#{AddressBook.addressesTable}" id="addressesTable"
  paginateButton="true" paginationControls="true"
  style="left: 24px; top: [illegible]
  title="Contacts" width="816">
  <webuijsf:tableRowGroup binding=
    "#{AddressBook.tableRowGroup1}" id="tableRowGroup1"
    rows="5" sourceData=
    "#{AddressBook.addressesDataProvider}"
    sourceVar="currentRow">
    <webuijsf:tableColumn binding=
      "#{AddressBook.fnameColumn}"
      headerText="First Name" id="fnameColumn"
      sort="ADDRESSES.FIRSTNAME">
      <webuijsf:staticText binding=
        "#{AddressBook.staticText[illegible]
        text="#{currentRow.value[
        'ADDRESSES.FIRSTNAME']}"/>
    </webuijsf:tableColumn>
    <webuijsf:tableColumn binding=
      "#{AddressBook.lnameColumn}"
      headerText="Last Name" id="lnameColumn"
      sort="ADDRESSES.LASTNAME">
      <webuijsf:staticText binding=
        "#{AddressBook.staticText[illegible]
        text="#{currentRow.value[
        'ADDRESSES.LASTNAME']}"/>
    </webuijsf:tableColumn>
    <webuijsf:tableColumn binding=
      "#{AddressBook.streetColumn}" headerText="Street"
      id="streetColumn" sort="ADDRESSES.STREET">
      <webuijsf:staticText binding=
        "#{AddressBook.staticText[illegible]
        text="#{currentRow.value[
        'ADDRESSES.STREET']}"/>
    </webuijsf:tableColumn>
```

Fig. 27.14 | AddressBook JSP with a **Map Viewer** component. (Part 3 of 5.)



```
[illegible]
<bp:mapViewer binding="#{AddressBook.mapViewer}"
   center="#{AddressBook.mapViewer_center}"
   id="mapViewer" info="#{AddressBook.mapMarker}"
[illegible]
```

Fig. 27.14 | AddressBook JSP with a **Map Viewer** component. (Part 4 of 5.)



http://localhost:8080/AddressBook/faces/AddressBook.jsp;jsessionid=92f7a6fe1fa50a4e47

Search the address book by last name:

Last name: Gray, John

John Gray
500 South St.
Philadelphia, PA 19147

Fig. 27.14 | AddressBook JSP with a **Map Viewer** component. (Part 5 of 5.)

The **Look Up** Button's action attribute is now bound to method lookUpButton_action in the page bean (lines 157–158). 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 27.5.4.

27.5.4 Page Bean That Displays a Map in the Map Viewer Component

Figure 27.15 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.

```
[illegible]
```

Fig. 27.15 | Page bean that gets a map to display in the **Map Viewer** component. (Part 1 of 3.)

```
import com.sun.j2ee.blueprints.ui.autocomplete.AutoCompleteComponent;
import com.sun.j2ee.blueprints.ui.autocomplete.CompletionResult;
import com.sun.j2ee.blueprints.ui.geocoder.GeoCoder;
import com.sun.j2ee.blueprints.ui.geocoder.GeoPoint;
import com.sun.j2ee.blueprints.ui.mapviewer.MapComponent;
import com.sun.j2ee.blueprints.ui.mapviewer.MapMarker;
import com.sun.j2ee.blueprints.ui.mapviewer.MapPoint;
import com.sun.rave.web.ui.appbase.AbstractPageBean;
import com.sun.webui.jsf.component.Body;
import com.sun.webui.jsf.component.Button;
import com.sun.webui.jsf.component.Form;
import com.sun.webui.jsf.component.Head;
import com.sun.webui.jsf.component.Html;
import com.sun.webui.jsf.component.Label;
import com.sun.webui.jsf.component.Link;
import com.sun.webui.jsf.component.MessageGroup;
import com.sun.webui.jsf.component.Page;
import com.sun.webui.jsf.component.StaticText;
import com.sun.webui.jsf.component.Table;
import com.sun.webui.jsf.component.TableColumn;
import com.sun.webui.jsf.component.TableRowGroup;
import com.sun.webui.jsf.component.TextField;
import javax.faces.FacesException;
import javax.faces.context.FacesContext;

public class AddressBook extends AbstractPageBean
{
   [illegible]

   private void _init() throws Exception
   {
      addressesDataProvider.setCachedRowSet(
         ( javax.sql.rowset.CachedRowSet ) getValue(
            "#{SessionBean1.addressesRowSet}" ) );
      addressesTable.setInternalVirtualForm( true );
      mapViewer.setRendered( false );
      addressesSearchDataProvider.setCachedRowSet(
         ( javax.sql.rowset.CachedRowSet ) getValue(
          "#{SessionBean1.addressesSearchRowSet}" ) );
   } // end method _init
   [illegible]
   // To save space, we omitted the code in lines 48-741. The complete
   // source code is provided with this chapter's examples.
   [illegible]
   // action handler for the lookUpButton that searches the address book
   // database and displays the requested address on a corresponding map.
   [illegible]
```

Fig. 27.15 | Page bean that gets a map to display in the **Map Viewer** component. (Part 2 of 3.)

```java

      try
      {
         // set the parameters for the addressesSearchDataProvider
         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 );

         // if Google Maps cannot find the address
         if ( points == null )
         {
            error( "Map for " + googleAddress + " could not be found" );
            mapViewer.setRendered( false ); // hide map
            return null;
         } // end if

         // center the map for the given address
         mapViewer_center.setLatitude( points[0].getLatitude() );
         mapViewer_center.setLongitude( points[0].getLongitude() );

         // create a marker for the address and set its display text
         mapMarker.setLatitude( points[0].getLatitude() );
         mapMarker.setLongitude( points[0].getLongitude() );
         mapMarker.setMarkup(  fname + " " + lname + "<br/>" + street +
            "<br/>" + city + ", " + state + " " + zip );

         mapViewer.setRendered( true ); // show map
      } // end try
      catch ( Exception e )
      {
         error( "Error processing search. " + e.getMessage() );
      } // end catch

      return null;
   } // end method lookUpButton_action
} // end class AddressBook
```

Fig. 27.15 | Page bean that gets a map to display in the **Map Viewer** component. (Part 3 of 3.)

Method `lookUpButton_action` (lines 744–802) is invoked when the user clicks the **Look Up** button in the lower form on the page. Lines 747–750 retrieve the name from the **AutoComplete Text Field** and split it into `Strings` for the first and last name. Lines 755–758 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 759 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 760–767 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.

Lines 770–771 format the address as a `String` for use with the Google Maps API. Line 774 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 777 determines whether the array of `GeoPoint` objects is `null`. If so, the address could not be found, and lines 779–781 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 785–786 set the latitude and longitude of the **Map Viewer**'s center to those of the `GeoPoint` that represents the selected address. Lines 789–792 set the **Map Marker**'s latitude and longitude, and set the text to display on the marker. Line 794 displays the recentered map containing the **Map Marker** that indicates the contact's location.

Lines 796–799 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 Web Resources

Our Java Resource Centers focus on the enormous amount of free Java content available online. We currently provide six Java-related Resource Centers:

```
www.deitel.com/java/
www.deitel.com/JavaCertification/
www.deitel.com/JavaDesignPatterns/
www.deitel.com/JavaEE5/
www.deitel.com/JavaFX/
www.deitel.com/JavaSE6Mustang/
```

You can view our complete list of Resource Centers at

```
www.deitel.com/ResourceCenters.html
```

Summary

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 **Table** component formats and displays data from database tables.
- Change the **Table**'s `title` property to specify the text displayed at the top of the **Table**.
- To create a database, select **Tools > Java DB Database > Create Java DB Database**.... Next, enter the name of the database to create a username and a password, then click **OK** to create the database.
- You can use the **Runtime** tab (to the right of the **Projects** and **Files** tabs) to create tables and to execute SQL statements that populate the database with data. To do so, click the **Runtime** tab and expand the **Databases** node.
- Netbeans must be connected to the database to execute SQL statements. If it is not, the icon appears next to the database's URL. In this case, right click the icon and click **Connect**.... Once connected, the icon changes to .
- To add a table to the database using SQL, expand the database's node, right click the **Tables** node and select **Execute Command**... to open a **SQL Command** editor in Netbeans. Paste the SQL code into the **SQL Command** editor in Netbeans. Then, highlight all the SQL commands, right click inside the **SQL Command** editor and select **Run Selection**.
- To configure a **Table** component to display a table's data, simply drag the database table from the **Servers** tab and drop it on the **Table** component to create the binding.
- To select specific columns to display, right click the **Table** component and select **Bind to Data** to display the **Bind to Data** dialog containing the list of the columns in the database table. The items under the **Selected** heading will be displayed in the **Table**. To remove a column, select it and click the < button.
- 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, click the column's name in the **Design** mode.
- Clicking the checkbox next to the table's `paginationControls` property in the **Properties** window configures this **Table** for automatic pagination. This adds buttons to the bottom of the **Table** for moving forward and backward between groups of contacts. You may use the **Table Layout** dialog's **Options** tab to select the number of rows to display at a time. To view this tab, right click the **Table**, select **Table Layout**..., then click the **Options** tab.
- Virtual forms allow subsets of a form's input components to be submitted to the server. Setting the `internalVirtualForm` property prevents the pagination control buttons on the **Table** from submitting other form components every time the user wishes to view the next group of records from the database.
- A `CachedRowSetDataProvider` provides a scrollable `RowSet` that can be bound to a **Table** component to display the `RowSet`'s data.
- 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.

Section 27.3 Ajax-Enabled JSF Components

- The Java BluePrints Ajax component library provides Ajax-enabled JSF components.
- To use the Java BluePrints Ajax-enabled components in Netbeans, you must download and import them. The IDE provides a wizard for installing this group of components (Internet access is required). 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, locate and select **BluePrints AJAX Components**, then click the **Add >** button to add them 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.
- You must import the components into the **Palette**. Select **Tools > Component Library Manager**, then click **Import...**. Click **Browse...** in the **Component Library Manager** dialog that appears. Select the `ui.complib` file and click **Open**. Click **OK** to import both the **BluePrints AJAX Components** and the **BluePrints AJAX Support Beans**. Close the **Component Library Manager** to return to the IDE.
- To see the new components in the **Palette**, you must add the **BluePrints AJAX Components** library to your visual web application. To do so, make sure your application's node is expanded in the **Projects** tab. Right click the **Component Libraries** node and select **Add Component Library**. In the **Add Component Library** dialog box, select the **BluePrints AJAX Components library** and click **Add Component Library**.

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. Double 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.
- To use a **Map Viewer**, set its key property to the Google Maps API key you obtained.
- A **Map Marker** (from the **BluePrints 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 **BluePrints 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

Ajax
Ajax-enabled JSF components
AutoComplete Text Field JSF component
binding a JSF **Table** to a database table
bundled database server
Button JSF component
Buy Now Button JSF component
`CachedRowSet` interface
`CachedRowSetDataProvider` class
`commitChanges` method of class`CachedRowSetDataProvider`
data provider
event-processing life cycle
Geocoding Service Object
`geoCode` method of a **Geocoding Service Object**
Google Maps
Google Maps API
Java BluePrints
Java BluePrints Ajax component library
Java DB
JavaServer Faces (JSF)
JSF element
Map Marker JSF component
Map Viewer JSF component
Message Group JSF component
participant component in a virtual form
Popup Calendar JSF component
`primary` property of a JSF **Button**
Progress Bar JSF component
Rating JSF component
`refresh` method of class`CachedRowSetDataProvider`
`reset` property of a JSF **Button**
Rich Textarea Editor JSF component
Select Value Text Field JSF component
submitter component in a virtual form
Table JSF component
virtual form
`webuijsf:staticText` JSF element
`webuijsf:table` JSF element
`webuijsf:tableRowGroup` JSF element

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 submitter component and participant components, 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 `complete` event handler for an **AutoComplete Text Field** is called after every keystroke in the text field to provide a list of suggestions based on what has already been typed.

e) A data provider automatically re-executes its SQL command to provide updated database information at every page refresh.

f) 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) Method ________ of class ________ updates a database to reflect any changes made in the database's data provider.

b) A(n) ________ is a supporting component used to translate addresses into latitudes and longitudes for display in a **Map Viewer** component.

c) A virtual form specifies that certain JSF components are ________ whose data will be submitted when the submitter component is clicked.

d) Ajax components for JSF such as the **AutoComplete Text Field** and **Map Viewer** are provided by the ________.

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 e-mail 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 *(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.