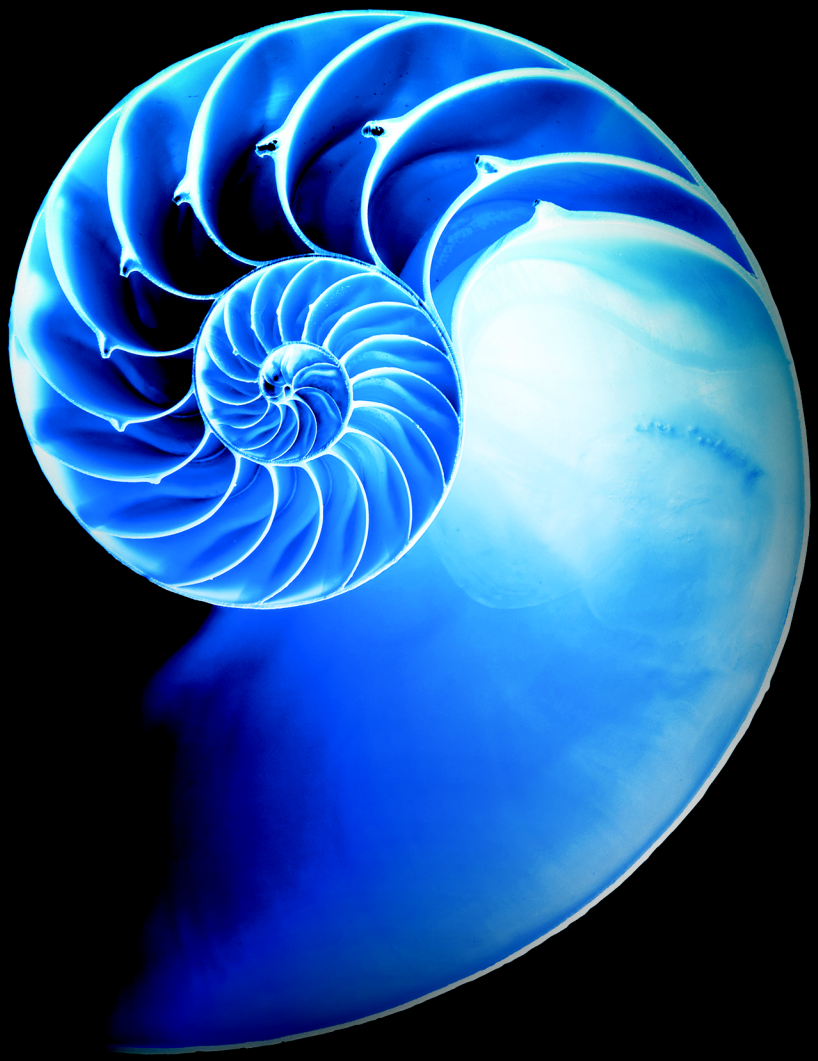
31

JavaServer™ Faces Web Apps: Part 2



Objectives

In this chapter you'll learn:

- To access databases from JSF applications.
- The basic principles and advantages of Ajax technology.
- To use Ajax in a JSF web app.

31.1 Introduction

This chapter continues our discussion of JSF web application development with two additional examples. In the first, we present a simple address book app that retrieves data from and inserts data into a Java DB database. The app allows users to view the existing contacts in the address book and to add new contacts. In the second example, we add so-called *Ajax* capabilities to the `Validation` example from Section 30.7. As you'll learn, Ajax improves application performance and responsiveness. This chapter's examples, like those in Chapter 30, were developed in NetBeans, but similar capabilities are available in other IDEs.

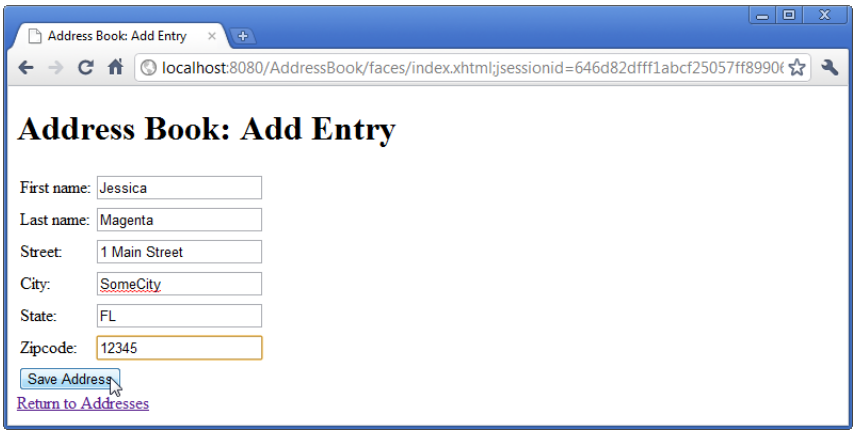
31.2 Accessing Databases in Web Apps

Many web apps access databases to store and retrieve persistent data. In this section, we build an address book web app that uses a Java DB database display contacts from the address book on a web page and to store contacts in the address book. Figure 31.1 shows sample interactions with the `AddressBook` app.

a) Table of addresses displayed when the `AddressBook` app is first requested

Fig. 31.1 | Sample outputs from the `AddressBook` app. (Part 1 of 2.)

b) Form for adding an entry



c) Table of addresses updated with the new entry added in Part (b)

Fig. 31.1 | Sample outputs from the `AddressBook` app. (Part 2 of 2.)

If the app's database already contains addresses, the initial request to the app displays those addresses as shown in Fig. 31.1(a)—we populated the database with the sample addresses shown. Clicking **Add Entry** displays the `addentry.xhtml` page for adding an address to the database (Fig. 31.1(b)). Clicking **Save Address** validates the form's fields. If the fields are valid, the JSF app adds the address to the database and returns to the `index.xhtml` page to show the updated list of addresses (Fig. 31.1(c)). This example also introduces the `h:dataTable` element for displaying data in tabular format.

The next several sections explain how to build the `AddressBook` application. First, we set up the database (Section 31.2.1). Next, we present class `AddressBean` (Section 31.2.2), which enables the app's Facelets pages to interact with the database. Finally, we present the `index.xthml` (Section 31.2.3) and `addentry.xhtml` (Section 31.2.4) Facelets pages.

31.2.1 Setting Up the Database

You'll now create the `addressbook` database and populate it with sample data.

Open NetBeans and Ensure that Java DB and GlassFish Are Running

Before you can create the data source in NetBeans, the IDE must be open and the Java DB and GlassFish servers must be running. Perform the following steps:

1. Open the NetBeans IDE.
2. On the **Services** tab, expand the **Databases** node then right click **Java DB**. If **Java DB** is not already running the **Start Server** option will be enabled. In this case, **Select Start** server to launch the Java DB server.
3. On the **Services** tab, expand the **Servers** node then right click **GlassFish Server 4.1** (or the version that was installed with NetBeans). If **GlassFish Server 4.1** is not already running the **Start** option will be enabled. In this case, select **Start** to launch GlassFish.

You may need to wait a few moments for the servers to begin executing.

Creating the Database

In web apps that receive many requests, it's inefficient to create separate database connections for each request. Instead, you use a **connection pool** to allow the server to manage a limited number of database connections and share them among requests. To create a connection pool for this app, perform the following steps:

1. On the **Services** tab, expand the **Databases** node, right click **Java DB** and select **Create Database...**. This opens the **Create Java DB Database** dialog.
2. Specify the following values:

- **Database Name:** `addressbook`
- **User Name:** `APP`
- **Password:** `APP`

3. Click **OK** to create the database.

You can specify any **User** name and **Password** you like and should change these as appropriate for real applications. The preceding steps create a new entry in the **Databases** node showing the database's URL (`jdbc:derby://localhost:1527/addressbook`). The database server that provides access to this database resides on the local machine and accepts connections on port 1527.

Populate the `addressbook` *Database with Sample Data*

You'll now populate the database with sample data using the `AddressBook.sql` SQL script that's provided with this chapter's examples. NetBeans must be connected to the database to execute SQL statements. If NetBeans is already connected to the database, the icon

is displayed next to the database's URL; otherwise, the icon is displayed. In this case, right click the icon and select **Connect...**.

To populate the database with sample data, perform the following steps:

1. Expand the `jdbc:derby://localhost:1527/addressbook` node, then expand the nested **APP** node.
2. Right click the **Tables** node and select **Execute Command...** to open a **SQL** editor tab in NetBeans. In a text editor, open the file `AddressBook.sql` from this chapter's examples folder, then copy the SQL statements and paste them into the **SQL** editor in NetBeans. Next, right click in the **SQL Command** editor and select **Run File**. This will create the `Addresses` table with the sample data in Fig. 31.1(a). [*Note:* The SQL script attempts to remove the database's `Addresses` table if it already exists. If it doesn't exist, you'll receive an error message in the NetBeans **Output** window, but the table will still be created properly.] Expand the **Tables** node to see the new table. You can view the table's data by right clicking **ADDRESSES** and selecting **View Data...**. Notice that we named the columns with all capital letters. We'll be using these names in Section 31.2.3.

31.2.2 Class AddressBean

[*Note:* To build this app from scratch, use the techniques you learned in Chapter 30 to create a JSF web application named `AddressBook` and add a second Facelets page named `addentry.xhtml` to the app.] Class `AddressBean` (Fig. 31.2) enables the `AddressBook` app to interact with the `addressbook` database. The class provides properties that represent the first name, last name, street, city, state and zip code for an entry in the database. These are used by the `addentry.xhtml` page when adding a new entry to the database. In addition, this class declares a `DataSource` (lines 39–40) for interacting with the database, method `getAddresses` (lines 115–148) for obtaining the list of addresses from the database and method `save` (lines 151–191) for saving a new address into the database. These methods use various JDBC techniques you learned in Chapter 24. [*Note:* It's also possible to implement this app's data storage using the JPA techniques from Chapter 29.]

```
1  // AddressBean.java
2  // Bean for interacting with the AddressBook database
3  package addressbook;
4
5  import java.io.Serializable;
6  import java.sql.Connection;
7  import java.sql.PreparedStatement;
8  import java.sql.ResultSet;
9  import java.sql.SQLException;
10 import javax.annotation.Resource;
11 import javax.annotation.sql.DataSourceDefinition;
12 import javax.inject.Named;
13 import javax.sql.DataSource;
14 import javax.sql.rowset.CachedRowSet;
15 import javax.sql.rowset.RowSetProvider;
```

Fig. 31.2 | `AddressBean` interacts with a database to store and retrieve addresses. (Part 1 of 5.)

```
16
17  // define the data source
18  @DataSourceDefinition(
19     name = "java:global/jdbc/addressbook",
20     className = "org.apache.derby.jdbc.ClientDataSource",
21     url = "jdbc:derby://localhost:1527/addressbook",
22     databaseName = "addressbook",
23     user = "APP",
24     password = "APP")
25
26  @Named("addressBean")
27  @javax.faces.view.ViewScoped
28  public class AddressBean implements Serializable
29  {
30     // instance variables that represent one address
31     private String firstName;
32     private String lastName;
33     private String street;
34     private String city;
35     private String state;
36     private String zipcode;
37
38     // allow the server to inject the DataSource
39     @Resource(lookup="java:global/jdbc/addressbook")
40     DataSource dataSource;
41
42     // get the first name
43     public String getFirstName()
44     {
45        return firstName;
46     }
47
48     // set the first name
49     public void setFirstName(String firstName)
50     {
51        this.firstName = firstName;
52     }
53
54     // get the last name
55     public String getLastName()
56     {
57        return lastName;
58     }
59
60     // set the last name
61     public void setLastName(String lastName)
62     {
63        this.lastName = lastName;
64     }
65
66     // get the street
67     public String getStreet()
68     {
```

Fig. 31.2 | `AddressBean` interacts with a database to store and retrieve addresses. (Part 2 of 5.)

```
69          return street;
70       }
71
72       // set the street
73       public void setStreet(String street)
74       {
75          this.street = street;
76       }
77
78       // get the city
79       public String getCity()
80       {
81          return city;
82       }
83
84       // set the city
85       public void setCity(String city)
86       {
87          this.city = city;
88       }
89
90       // get the state
91       public String getState()
92       {
93          return state;
94       }
95
96       // set the state
97       public void setState(String state)
98       {
99          this.state = state;
100      }
101
102      // get the zipcode
103      public String getZipcode()
104      {
105         return zipcode;
106      }
107
108      // set the zipcode
109      public void setZipcode(String zipcode)
110      {
111         this.zipcode = zipcode;
112      }
113
114      // return a ResultSet of entries
115      public ResultSet getAddresses() throws SQLException
116      {
117         // check whether dataSource was injected by the server
118         if (dataSource == null)
119         {
120            throw new SQLException("Unable to obtain DataSource");
121         }
```

Fig. 31.2 | `AddressBean` interacts with a database to store and retrieve addresses. (Part 3 of 5.)

```
122
123        // obtain a connection from the connection pool
124        Connection connection = dataSource.getConnection();
125
126        // check whether connection was successful
127        if (connection == null)
128        {
129           throw new SQLException("Unable to connect to DataSource");
130        }
131
132        try
133        {
134           // create a PreparedSt atement to insert a new address book entry
135           PreparedStatement getAddresses = connection.prepareStatement(
136              "SELECT FIRSTNAME, LASTNAME, STREET, CITY, STATE, ZIP " +
137              "FROM ADDRESSES ORDER BY LASTNAME, FIRSTNAME");
138
139           CachedRowSet rowSet =
140              RowSetProvider.newFactory().createCachedRowSet();
141           rowSet.populate(getAddresses.executeQuery());
142           return rowSet;
143        }
144        finally
145        {
146           connection.close(); // return this connection to pool
147        }
148     }
149
150     // save a new address book entry
151     public String save() throws SQLException
152     {
153        // check whether dataSource was injected by the server
154        if (dataSource == null)
155        {
156           throw new SQLException("Unable to obtain DataSource");
157        }
158
159        // obtain a connection from the connection pool
160        Connection connection = dataSource.getConnection();
161
162        // check whether connection was successful
163        if (connection == null)
164        {
165           throw new SQLException("Unable to connect to DataSource");
166        }
167
168        try
169        {
170           // create a PreparedStatement to insert a new address book entry
171           PreparedStatement addEntry =
172              connection.prepareStatement("INSERT INTO ADDRESSES " +
173                 "(FIRSTNAME,LASTNAME,STREET,CITY,STATE,ZIP)" +
174                 " VALUES (?, ?, ?, ?, ?, ?)");
```

Fig. 31.2 | `AddressBean` interacts with a database to store and retrieve addresses. (Part 4 of 5.)

```
175
176             // specify the PreparedStatement's arguments
177             addEntry.setString(1, getFirstName());
178             addEntry.setString(2, getLastName());
179             addEntry.setString(3, getStreet());
180             addEntry.setString(4, getCity());
181             addEntry.setString(5, getState());
182             addEntry.setString(6, getZipcode());
183
184             addEntry.executeUpdate(); // insert the entry
185             return "index"; // go back to index.xhtml page
186          }
187          finally
188          {
189             connection.close(); // return this connection to pool
190          }
191       }
192    }
```

Fig. 31.2 | `AddressBean` interacts with a database to store and retrieve addresses. (Part 5 of 5.)

Defining a Data Source with the Annotation `@DataSourceDefinition`

To connect to the `addressbook` database from a web app, you must configure a **data source name** that will be used to locate the database. Lines 18–24

```
@DataSourceDefinition(
   name = "java:global/jdbc/addressbook",
   className = "org.apache.derby.jdbc.ClientDataSource",
   url = "jdbc:derby://localhost:1527/addressbook",
   databaseName = "addressbook",
   user = "APP",
   password = "APP")
```

use Java EE 7's **`@DataSourceDefinition` annotation** to create a data source name for the `addressbook` database. Here we specified the following attributes:

- `name`—The JNDI (Java Naming and Directory Interface) name we'll use to look up the data source. JNDI is a technology for locating application components (such as databases) in a distributed application (such as a multitier web application).
- `className`—The `DataSource` subclass. An object of this class will be used to interact with the database. A **`DataSource`** (package `javax.sql`) enables a web application to obtain a `Connection` to a database. `ClientDataSource` is one of several `DataSource` subclasses provided by Java DB. Apps that are expected to manage many connections at once would typically use `ClientConnectionPoolDataSource` or `ClientXADataSource`.
- `url`—The URL for connecting to the database. This is the database URL is specified in the NetBeans **Services** tab's **Databases > Java DB** node.
- `databaseName`—The database's name.
- `user`—The username for logging into the database.
- `password`—The password for logging into the database.[1]

Though we do not do so here, the `@DataSourceDefinition` annotation also can create the database, by specifiying the attribute

```
properties = {"createDatabase=create"}
```

The app could then create the database's table(s) programmatically. We manually created the database in advance so we could prepopulate it with sample address data.

Class `AddressBean`*'s Annotations—*`@Named` *and* `@javax.faces.view.ViewScoped`

In Chapter 30, we introduced the `@ManagedBean` annotation (from the package `javax.faces.bean`) to indicate that the JSF framework should create and manage the JavaBean object(s) used in the application. `@ManagedBean` is deprecated in Java EE 7 and Contexts and Dependency Injection (CDI) should be used instead. Switching to CDI simply requires changing from JSF's `@ManagedBean` annotation to CDI's **`@Named` annotation** (line 26):

```
@Named("addressBean")
```

As with `@ManagedBean`, if you do not specify a name in parentheses, the JavaBean object's variable name will be the JavaBean class's name with a lowercase first letter.

We also added the annotation

```
@javax.faces.view.ViewScoped
```

to indicate that CDI should manage this JavaBean's lifetime, based on the JSF view that first referenced the JavaBean. A `ViewScoped` JavaBean's class must be `Serializable` (as indicated in line 28).

Injecting the `DataSource` *into Class* `AddressBean`

Lines 39–40 use the **`@Resource`** annotation to inject a `DataSource` object into the `AddressBean`. The annotation's `lookup` attribute specifies the JNDI name for the data source we created in lines 18–24. The `@Resource` annotation enables the server (GlassFish in our case) to hide all the complex details of setting up a `DataSource` object that can interact with the `addressbook` database. The server creates a `DataSource` for you—an object of the type you specified in the `@DataSourceDefinition`—and assigns the `DataSource` object to the annotated variable declared at line 40. You can now trivially obtain a `Connection` for interacting with the database.

`AddressBean` *Method* `getAddresses`

Method `getAddresses` (lines 115–148) is called when the `index.xhtml` page is requested. The method returns a list of addresses for display in the page (Section 31.2.3). First, we check whether variable `dataSource` is `null` (lines 118–121), which would indicate that the server was unable to create the `DataSource` object. If the `DataSource` was created successfully, we use it to obtain a `Connection` to the database (line 124). Next, we check whether variable `connection` is `null` (lines 127–130), which would indicate that we were unable to connect. If the connection was successful, lines 135–142 get the set of addresses from the database and return them.

---

1. Future versions of Java EE might include password aliases to hide the database's password for additional security.

The `PreparedStatement` at lines 135–137 obtains all the addresses. Because database connections are a limited resources, you should use and close them quickly in your web apps. For this reason, we create a `CachedRowSet` and populate it with the `ResultSet` returned by the `PreparedStatement`'s `executeQuery` method (lines 139–141). We then return the `CachedRowSet` (a disconnected `RowSet`) for use in the `index.xhtml` page (line 142) and close the `connection` object (line 146) in the `finally` block.

`AddressBean` Method `save`

Method `save` (lines 151–191) stores a new address in the database (Section 31.2.4). This occurs when the user submits the `addentry.xhtml` form—assuming the form's fields validate successfully. As in `getAddresses`, we ensure that the `DataSource` is not `null`, then obtain the `Connection` object and ensure that its not `null`. Lines 171–174 create a `PreparedStatement` for inserting a new record in the database. Lines 177–182 specify the values for each of the parameters in the `PreparedStatement`. Line 184 then executes the `PreparedStatement` to insert the new record. Line 185 returns the string `"index"`, which as you'll see in Section 31.2.4 causes the app to display the `index.xhtml` page again.

31.2.3 `index.xhtml` Facelets Page

`index.xhtml` (Fig. 31.3) is the default web page for the `AddressBook` app. When this page is requested, it obtains the list of addresses from the `AddressBean` and displays them in tabular format using an **`h:dataTable` element**. The user can click the **Add Entry** button (line 17) to view the `addentry.xhtml` page. Recall that the default action for an `h:commandButton` is to submit a form. In this case, we specify the button's **`action` attribute** with the value `"addentry"`. The JSF framework assumes this is a page in the app, appends `.xhtml` extension to the `action` attribute's value and returns the `addentry.xhtml` page to the client browser.

The `h:dataTable` Element

The `h:dataTable` element (lines 19–46) inserts tabular data into a page. We discuss only the attributes and nested elements that we use here. For more details on this element, its attributes and other JSF tag library elements, visit `bit.ly/JSF2TagLibraryReference`.

```
1  <?xml version='1.0' encoding='UTF-8' ?>
2
3  <!-- index.html -->
4  <!-- Displays an h:dataTable of the addresses in the address book -->
5  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
6     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
7  <html xmlns="http://www.w3.org/1999/xhtml"
8     xmlns:h="http://java.sun.com/jsf/html"
9     xmlns:f="http://java.sun.com/jsf/core">
10    <h:head>
11       <title>Address Book</title>
12       <h:outputStylesheet name="style.css" library="css"/>
13    </h:head>
```

Fig. 31.3 | Displays an `h:dataTable` of the addresses in the address book. (Part 1 of 2.)

```
14      <h:body>
15         <h1>Address Book</h1>
16         <h:form>
17            <p><h:commandButton value="Add Entry" action="addentry"/></p>
18         </h:form>
19         <h:dataTable value="#{addressBean.addresses}" var="address"
20            rowClasses="oddRows,evenRows" headerClass="header"
21            styleClass="table" cellpadding="5" cellspacing="0">
22            <h:column>
23               <f:facet name="header">First Name</f:facet>
24               #{address.FIRSTNAME}
25            </h:column>
26            <h:column>
27               <f:facet name="header">Last Name</f:facet>
28               #{address.LASTNAME}
29            </h:column>
30            <h:column>
31               <f:facet name="header">Street</f:facet>
32               #{address.STREET}
33            </h:column>
34            <h:column>
35               <f:facet name="header">City</f:facet>
36               #{address.CITY}
37            </h:column>
38            <h:column>
39               <f:facet name="header">State</f:facet>
40               #{address.STATE}
41            </h:column>
42            <h:column>
43               <f:facet name="header">Zip code</f:facet>
44               #{address.ZIP}
45            </h:column>
46         </h:dataTable>
47      </h:body>
48   </html>
```

Fig. 31.3 | Displays an `h:dataTable` of the addresses in the address book. (Part 2 of 2.)

The `h:dataTable` element's **`value` attribute** (line 19) specifies the collection of data you wish to display. In this case, we use `AddressBean`'s `addresses` property, which calls the `getAddresses` method (Fig. 31.2). The collection returned by this method is a `CachedRowSet`, which is a type of `ResultSet`.

The `h:dataTable` iterates over its `value` collection and, one at a time, assigns each element to the variable specified by the **`var` attribute**. This variable is used in the `h:dataTable`'s nested elements to access each element of the collection—each element in this case represents one row (i.e., address) in the `CachedRowSet`.

The **`rowClasses` attribute** (line 20) is a space-separated list of CSS style class names that are used to style the rows in the tabular output. These style classes are defined in the app's `styles.css` file in the `css` library (which is inserted into the document at line 12). You can open this file to view the various style class definitions. We specified two style classes— all the odd numbered rows will have the first style (`oddRows`) and all the even numbered rows the second style (`evenRows`). You can specify as many styles as you like—

they'll be applied in the order you list them one row at a time until all the styles have been applied, then the `h:DataTable` will automatically cycle through the styles again for the next set of rows. The **`columnClasses` attribute** works similarly for columns in the table.

The **`headerClass` attribute** (line 20) specifies the column header CSS style. Headers are defined with `f:facet` elements nested in `h:column` elements (discussed momentarily). The **`footerClass` attribute** works similarly for column footers in the table.

The **`styleClass` attribute** (line 21) specifies the CSS styles for the entire table. The **`cellpadding`** and **`cellspacing`** attributes (line 21) specify the number of pixels around each table cell's contents and the number of pixels between table cells, respectively.

The `h:column` Elements

Lines 22–45 define the table's columns with six nested **`h:column` elements**. We focus here on the one at lines 22–25. When the `CachedRowSet` is populated in the `AddressBean` class, it automatically uses the database's column names as property names for each row object in the `CachedRowSet`. Line 28 inserts into the column the `FIRSTNAME` property of the `CachedRowSet`'s current row. To display a column header above the column, you define an **`f:facet` element** (line 23) and set its `name` attribute to `"header"`. Similarly, to display a column footer, use an `f:facet` with its `name` attribute set to `"footer"`. The header is formatted with the CSS style specified in the `h:dataTable`'s `headerClass` attribute (line 20). The remaining `h:column` elements perform similar tasks for the current row's `LASTNAME`, `STREET`, `CITY`, `STATE` and `ZIP` properties.

31.2.4 `addentry.xhtml` Facelets Page

When the user clicks **Add Entry** in the `index.xhtml` page, `addentry.xhtml` (Fig. 31.4) is displayed. Each `h:inputText` in this page has its `required` attribute set to `"true"` and includes a `maxlength` attribute that restricts the user's input to the maximum length of the corresponding database field. When the user clicks **Save** (lines 48–49), the input element's values are validated and (if successful) assigned to the properties of the `addressBean` managed object. In addition, the button specifies as its `action` the EL expression

```
#{addressBean.save}
```

which invokes the `addressBean` object's `save` method to store the new address in the database. When you call a method with the `action` attribute, if the method returns a value (in this case, it returns the string `"index"`), that value is used to request the corresponding page from the app. If the method does not return a value, the current page is re-requested.

```
1  <?xml version='1.0' encoding='UTF-8' ?>
2
3  <!-- addentry.html -->
4  <!-- Form for adding an entry to an address book -->
5  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
6     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
7  <html xmlns="http://www.w3.org/1999/xhtml"
8     xmlns:h="http://java.sun.com/jsf/html">
```

Fig. 31.4 | Form for adding an entry to an address book. (Part 1 of 2.)

```
 9     <h:head>
10        <title>Address Book: Add Entry</title>
11        <h:outputStylesheet name="style.css" library="css"/>
12     </h:head>
13     <h:body>
14        <h1>Address Book: Add Entry</h1>
15        <h:form>
16           <h:panelGrid columns="3">
17              <h:outputText value="First name:"/>
18              <h:inputText id="firstNameInputText" required="true"
19                 requiredMessage="Please enter first name"
20                 value="#{addressBean.firstName}" maxlength="30"/>
21              <h:message for="firstNameInputText" styleClass="error"/>
22              <h:outputText value="Last name:"/>
23              <h:inputText id="lastNameInputText" required="true"
24                 requiredMessage="Please enter last name"
25                 value="#{addressBean.lastName}" maxlength="30"/>
26              <h:message for="lastNameInputText" styleClass="error"/>
27              <h:outputText value="Street:"/>
28              <h:inputText id="streetInputText" required="true"
29                 requiredMessage="Please enter the street address"
30                 value="#{addressBean.street}" maxlength="150"/>
31              <h:message for="streetInputText" styleClass="error"/>
32              <h:outputText value="City:"/>
33              <h:inputText id="cityInputText" required="true"
34                 requiredMessage="Please enter the city"
35                 value="#{addressBean.city}" maxlength="30"/>
36              <h:message for="cityInputText" styleClass="error"/>
37              <h:outputText value="State:"/>
38              <h:inputText id="stateInputText" required="true"
39                 requiredMessage="Please enter state"
40                 value="#{addressBean.state}" maxlength="2"/>
41              <h:message for="stateInputText" styleClass="error"/>
42              <h:outputText value="Zipcode:"/>
43              <h:inputText id="zipcodeInputText" required="true"
44                 requiredMessage="Please enter zipcode"
45                 value="#{addressBean.zipcode}" maxlength="5"/>
46              <h:message for="zipcodeInputText" styleClass="error"/>
47           </h:panelGrid>
48           <h:commandButton value="Save Address"
49              action="#{addressBean.save}"/>
50        </h:form>
51        <h:outputLink value="index.xhtml">Return to Addresses</h:outputLink>
52     </h:body>
53  </html>
```

Fig. 31.4 | Form for adding an entry to an address book. (Part 2 of 2.)

31.3 Ajax

The term **Ajax**—short for **Asynchronous JavaScript and XML**—was coined by Jesse James Garrett of Adaptive Path, Inc., in 2005 to describe a range of technologies for developing highly responsive, dynamic web applications. Ajax applications include Google Maps, Yahoo's FlickR and many more. Ajax separates the *user interaction* portion of an ap-

plication from its *server interaction*, enabling both to proceed *in parallel*. This enables Ajax web-based applications to perform at speeds approaching those of desktop applications, reducing or even eliminating the performance advantage that desktop applications have traditionally had over web-based applications. This has huge ramifications for the desktop applications industry—the applications platform of choice is shifting from the desktop to the web. Many people believe that the web—especially in the context of abundant open-source software, inexpensive computers and exploding Internet bandwidth—will create the next major growth phase for Internet companies.

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

Ajax applications typically make use of client-side scripting technologies such as JavaScript to interact with page elements. They use the browser's `XMLHttpRequest` object to perform the asynchronous exchanges with the web server that make Ajax applications so responsive. This object can be used by most scripting languages to pass XML data from the client to the server and to process XML data sent from the server back to the client.

Using Ajax technologies in web applications can dramatically improve performance, but programming Ajax directly is complex and error prone. It requires page designers to know both scripting and markup languages. As you'll soon see, JSF makes adding Ajax capabilities to your web apps fairly simple.

Traditional Web Applications

Figure 31.5 presents the typical interactions between the client and the server in a traditional web application, such as one that uses a user registration form.

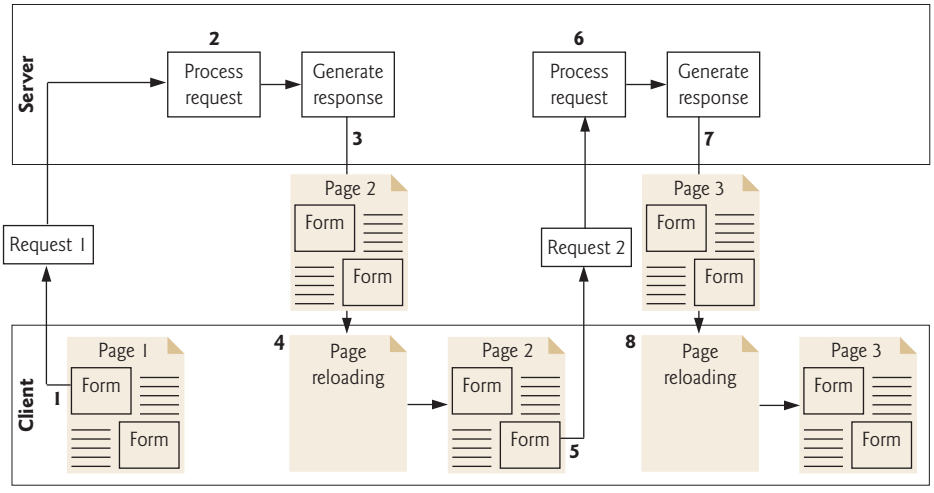


Fig. 31.5 | Classic web application reloading the page for every user interaction.

The user first fills in the form's fields, then *submits* the form (Fig. 31.5, *Step 1*). The browser generates a request to the server, which receives the request and processes it (*Step 2*). The server generates and sends a response containing the exact page that the browser will render (*Step 3*), which causes the browser to load the new page (*Step 4*) and temporarily makes the browser window blank. The client *waits* for the server to respond and *reloads the entire page* with the data from the response (*Step 4*). While such a **synchronous request** is being processed on the server, *the user cannot interact with the client web page*. If the user interacts with and submits another form, the process begins again (*Steps 5–8*).

This model was originally designed for a web of *hypertext documents*—what some people call the "brochure web." As the web evolved into a full-scale applications platform, the model shown in Fig. 31.5 yielded "choppy" application performance. Every full-page refresh required users to reestablish their understanding of the full-page contents. Users began to demand a model that would yield the responsiveness of desktop applications.

Ajax Web Applications

Ajax applications add a layer between the client and the server to manage communication between the two (Fig. 31.6).

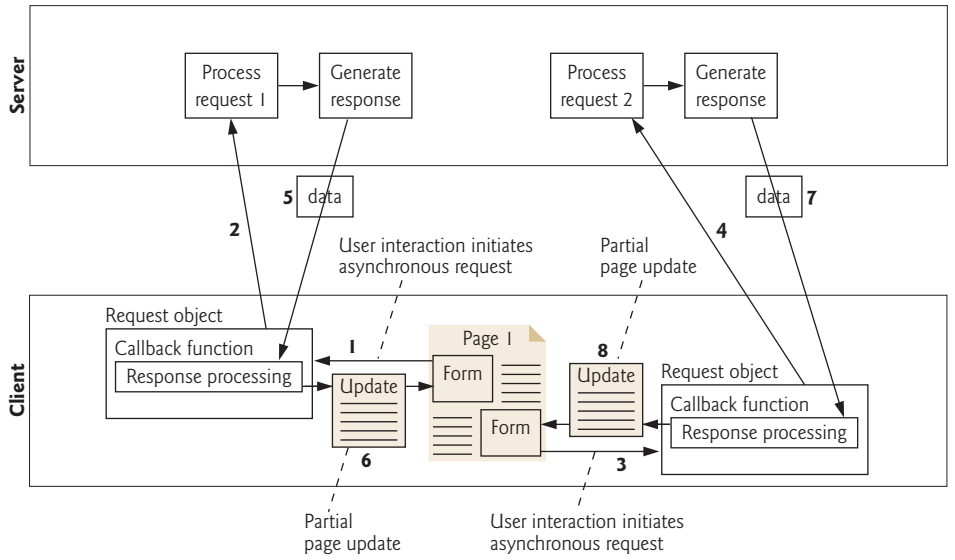


Fig. 31.6 | Ajax-enabled web application interacting with the server asynchronously.

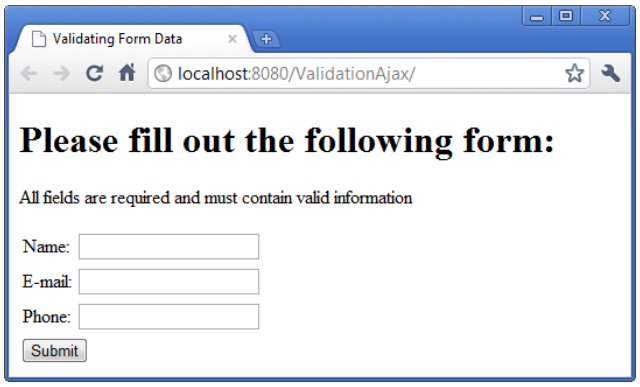
When the user interacts with the page, the client creates an `XMLHttpRequest` object to manage a request (*Step 1*). This object sends the request to the server (*Step 2*) and awaits the response. The requests are asynchronous, so the user can continue interacting with the application on the client side while the server processes the earlier request concurrently. Other user interactions could result in additional requests to the server (*Steps 3* and *4*). Once the server responds to the original request (*Step 5*), the `XMLHttpRequest` object that issued the request calls a client-side function to process the data returned by the server. This function—known as a **callback function**—uses **partial page updates** (*Step 6*) to dis-

play the data in the existing web page *without reloading the entire page*. At the same time, the server may be responding to the second request (*Step 7*) and the client side may be starting to do another partial page update (*Step 8*). The callback function updates only a designated part of the page. Such partial page updates help make web applications more responsive, making them feel more like desktop applications. The web application does not load a new page while the user interacts with it.

31.4 Adding Ajax Functionality to the Validation App

The example in this section adds Ajax capabilities to the `Validation` app that we presented in Section 30.7. Figure 31.7 shows the sample outputs from the `ValidationAjax` version of the app that we'll build momentarily. Part (a) shows the initial form that's displayed when this app first executes. Parts (b) and (c) show validation errors that are displayed when the user submits an empty form and invalid data, respectively. Part (d) shows the page after the form is submitted successfully.

a) Submitting the form before entering any information



b) Error messages displayed after submitting the empty form

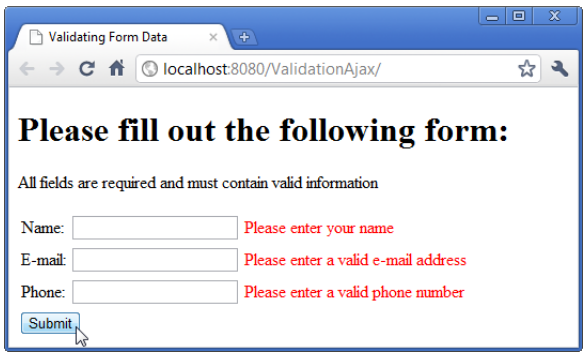
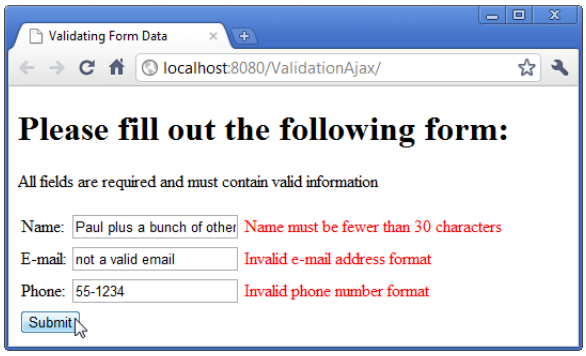


Fig. 31.7 | JSP that demonstrates validation of user input. (Part 1 of 2.)

c) Error messages displayed after submitting invalid information



d) Successfully submitted form

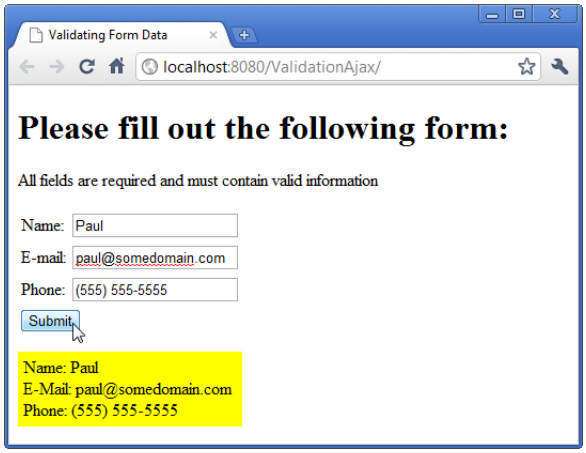


Fig. 31.7 | JSP that demonstrates validation of user input. (Part 2 of 2.)

As you can see, the app has the same functionality as the version in Section 30.7; however, you'll notice a couple of changes in how the app works. First, the URL displayed in the web browser always reads `localhost:8080/ValidationAjax/`, whereas the URL in the Section 30.7 changes after the form is submitted the first time. Also, in the non-Ajax version of the app, the page refreshes each time you press the **Submit** button. In the Ajax version, only the parts of the page that need updating actually change.

index.xhtml

The changes required to add Ajax functionality to this app are minimal. All of the changes are in the `index.xhtml` file (Fig. 31.8) and are highlighted. The `ValidationBean` class is identical to the version in Section 30.7, so we don't show it here.

```
<?xml version='1.0' encoding='UTF-8' ?>

<!-- index.xhtml -->
<!-- Validating user input -->
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
   xmlns:h="http://java.sun.com/jsf/html"
   xmlns:f="http://java.sun.com/jsf/core">
   <h:head>
      <title>Validating Form Data</title>
      <h:outputStylesheet name="style.css" library="css"/>
   </h:head>
   <h:body>
      <h:form>
         <h1>Please fill out the following form:</h1>
         <p>All fields are required and must contain valid information</p>
         <h:panelGrid columns="3">
            <h:outputText value="Name:"/>
            <h:inputText id="nameInputText" required="true"
               requiredMessage="Please enter your name"
               value="#{validationBean.name}"
               validatorMessage="Name must be fewer than 30 characters">
               <f:validateLength maximum="30" />
            </h:inputText>
            <h:message id="nameMessage" for="nameInputText"
               styleClass="error"/>
            <h:outputText value="E-mail:"/>
            <h:inputText id="emailInputText" required="true"
               requiredMessage="Please enter a valid e-mail address"
               value="#{validationBean.email}"
               validatorMessage="Invalid e-mail address format">
               <f:validateRegex pattern=
                  "\w+([-+.']\w+)*@\w+([-.]\w+)*\.\w+([-.]\w+)*" />
            </h:inputText>
            <h:message id="emailMessage" for="emailInputText"
               styleClass="error"/>
            <h:outputText value="Phone:"/>
            <h:inputText id="phoneInputText" required="true"
               requiredMessage="Please enter a valid phone number"
               value="#{validationBean.phone}"
               validatorMessage="Invalid phone number format">
               <f:validateRegex pattern=
                  "((\(\d{3}\) ?)|(\d{3}-))?\d{3}-\d{4}" />
            </h:inputText>
            <h:message id="phoneMessage" for="phoneInputText"
               styleClass="error"/>
         </h:panelGrid>
         <h:commandButton value="Submit">
            <f:ajax execute="nameInputText emailInputText phoneInputText"
               render=
               "nameMessage emailMessage phoneMessage resultOutputText"/>
         </h:commandButton>
```

Fig. 31.8 | Ajax enabling the Validation app. (Part 1 of 2.)

```
54            <h:outputText id="resultOutputText" escape="false"
55               value="#{validationBean.response}"/>
56         </h:form>
57      </h:body>
58   </html>
```

Fig. 31.8 | Ajax enabling the Validation app. (Part 2 of 2.)

Adding `id` *Attributes to Elements*

The Facelets elements that will be submitted as part of an Ajax request and the Facelets elements that will participate in the partial page updates must have `id` attributes. The `h:inputText` elements in the original `Validation` example already had `id` attributes. These elements will be submitted to the server as part of an Ajax request. We'd like the `h:Message` elements that show validation errors and the `h:outputText` element that displays the result to be updated with partial page updates. For this reason, we've added `id` attributes to these elements.

`f:ajax` Element

The other key change to this page is at lines 49–53 where the `h:commandButton` now contains an **`f:ajax` element**, which intercepts the form submission when the user clicks the button and makes an Ajax request instead. The `f:ajax` element's **`execute` attribute** specifies a space-separated list of element `id`s—the values of these elements are submitted as part of the Ajax request. The `f:ajax` element's **`render` attribute** specifies a space-separated list of element `id`s for the elements that should be updated via partial page updates.

31.5 Wrap-Up

In this chapter, we built an `AddressBook` application that allowed a user to add and view contacts. You learned how to insert user input into a Java DB database and how to display the contents of a database on a web page using an `h:dataTable` JSF element. We also demonstrated how to add Ajax capabilities to JSF web apps by enhancing the `Validation` app from Section 30.7. In Chapter 32, you'll use NetBeans to create web services and consume them from desktop and web applications.

Summary

Section 30.2.2 Class `AddressBean`

- To connect to the addressbook database from a web app, you must configure a data source name that will be used to locate the database.
- Java EE 7's `@DataSourceDefinition` annotation creates a data source name, specifying its JNDI (Java Naming and Directory Interface) name that is used to look up the data source.
- JNDI is a technology for locating application components (such as databases) in a distributed application (such as a multitier web application).
- A `DataSource` (package `javax.sql`) enables a web application to obtain a `Connection` to a database.
- `ClientDataSource` is one of several `DataSource` subclasses provided by Java DB. Apps that are expected to manage many connections at once would typically use `ClientConnectionPoolDataSource` or `ClientXADataSource`.

- `@ManagedBean` is deprecated in Java EE 7 and Contexts and Dependency Injection (CDI) should be used instead. Switching to CDI simply requires changing from JSF's `@ManagedBean` annotation to CDI's `@Named` annotation (line 26):
- As with `@ManagedBean`, if you do not specify a name in parentheses, the JavaBean object's variable name will be the JavaBean class's name with a lowercase first letter.
- The annotation `@javax.faces.view.ViewScoped` indicates that CDI should manage a JavaBean's lifetime, based on the JSF view that first referenced the JavaBean. A `ViewScoped` JavaBean's class must be `Serializable`.
- The annotation `@Resource` (p. 10) can be used to inject a `DataSource` object into a managed bean. The annotation's `lookup` attribute specifies the JNDI name of a data source.
- The `@Resource` annotation enables the server to hide all the complex details hide all the complex details of setting up a `DataSource` object that can interact with a database. The server creates a `DataSource` for you and assigns the `DataSource` object to the annotated variable. You can then trivially obtain a `Connection` for interacting with the database.
- Database connections are limited resources, so you should use and close them quickly in your web apps. You can use a `CachedRowSet` to store the results of a query for use later.

Section 30.2.3 `index.xhtml` Facelets Page

- You can use an `h:dataTable` element (p. 11) to display a collection of objects, such as the rows in a `CachedRowSet`, in tabular format.
- If you specify an `h:commandButton`'s `action` attribute (p. 11) with a value that is the name of a web page (without the filename extension), the JSF framework assumes this is a page in the app, appends `.xhtml` extension to the `action` attribute's value and returns the page to the client browser.
- The `h:dataTable` element's `value` attribute (p. 12) specifies the collection of data you wish to display. The `h:dataTable` iterates over its `value` collection and, one at a time, assigns each element to the variable specified by the `var` attribute (p. 12). This variable is used in the `h:dataTable`'s nested elements to access each element of the collection.
- The `h:dataTable` `rowClasses` attribute (p. 12) is a space-separated list of CSS style class names that are used to style the rows in the tabular output. You can specify as many styles as you like—they'll be applied in the order you list them one row at a time until all the styles have been applied, then the `h:DataTable` will automatically cycle through the styles again for the next set of rows. The `columnClasses` attribute works similarly for columns in the table.
- The `headerClass` attribute (p. 13) specifies the column header CSS style. The `footerClass` attribute (p. 13) works similarly for column footers in the table.
- The `styleClass` attribute (p. 13) specifies the CSS styles for the entire table. The `cellpadding` and `cellspacing` attributes (p. 13) specify the number of pixels around each table cell's contents and the number of pixels between table cells, respectively.
- An `h:column` element (p. 13) defines a column in an `h:dataTable`.
- To display a column header above a column, define an `f:facet` element (p. 13) and set its `name` attribute to `"header"`. Similarly, to display a column footer, use an `f:facet` with its `name` attribute set to `"footer"`.

Section 30.2.4 `addentry.xhtml` Facelets Page

- You can call a managed bean's methods in EL expressions.

- When you call a managed bean method with the `action` attribute, if the method returns a value, that value is used to request the corresponding page from the app. If the method does not return a value, the current page is re-requested.

Section 31.3 Ajax

- The term Ajax—short for Asynchronous JavaScript and XML—was coined by Jesse James Garrett of Adaptive Path, Inc., in February 2005 to describe a range of technologies for developing highly responsive, dynamic web applications.
- Ajax separates the user interaction portion of an application from its server interaction, enabling both to proceed asynchronously in parallel. This enables Ajax web-based applications to perform at speeds approaching those of desktop applications.
- Ajax makes asynchronous calls to the server to exchange small amounts of data with each call. Where normally the entire page would be submitted and reloaded with every user interaction on a web page, Ajax reloads only the necessary portions of the page, saving time and resources.
- Ajax applications typically make use of client-side scripting technologies such as JavaScript to interact with page elements. They use the browser's `XMLHttpRequest` object to perform the asynchronous exchanges with the web server that make Ajax applications so responsive.
- In a traditional web application, the user fills in a form's fields, then submits the form. The browser generates a request to the server, which receives the request and processes it. The server generates and sends a response containing the exact page that the browser will render. The browser loads the new page, temporarily making the browser window blank. The client waits for the server to respond and reloads the entire page with the data from the response. While such a synchronous request is being processed on the server, the user cannot interact with the web page. This model yields "choppy" application performance.
- In an Ajax application, when the user interacts with the page, the client creates an `XMLHttpRequest` object to manage a request. This object sends the request to the server and awaits the response. The requests are asynchronous, so the user can interact with the application on the client side while the server processes the earlier request concurrently. Other user interactions could result in additional requests to the server. Once the server responds to the original request, the `XMLHttpRequest` object that issued the request calls a client-side function to process the data returned by the server. This callback function uses partial page updates to display the data in the existing web page without reloading the entire page. At the same time, the server may be responding to the second request and the client side may be starting to do another partial page update.
- Partial page updates help make web applications more responsive, making them feel more like desktop applications.

Section 31.4 Adding Ajax Functionality to the Validation App

- The Facelets elements that will be submitted as part of an Ajax request and the Facelets elements that will participate in the partial page updates must have `id` attributes.
- When you nest an `f:ajax` element (p. 20) in an `h:commandButton` element, the `f:ajax` element intercepts the form submission and makes an Ajax request instead.
- The `f:ajax` element's `execute` attribute (p. 20) specifies a space-separated list of element `id`s—the values of these elements are submitted as part of the Ajax request.
- The `f:ajax` element's `render` attribute (p. 20) specifies a space-separated list of element `id`s for the elements that should be updated via partial page updates.

Self-Review Exercise

31.1 Fill in the blanks in each of the following statements.

a) Ajax is an acronym for _____.
b) A(n) _____ allows the server to manage a limited number of database connections and share them among requests.
c) is a technology for locating application components (such as databases) in a distributed application.
d) A(n) _____ enables a web application to obtain a `Connection` to a database.
e) The annotation _____ can be used to inject a `DataSource` object into a managed bean.
f) A(n) _____ element displays a collection of objects in tabular format.
g) An `h:commandButton`'s _____ attribute can specify the name of another page in the web app that should be returned to the client.
h) To specify headers or footers for the columns in `h:dataTables`, use _____ elements nested with their `name` attributes set to _____ and _____, respectively.
i) _____ separates the user interaction portion of an application from its server interaction, enabling both to proceed asynchronously in parallel.
j) _____ help make web applications more responsive, making them feel more like desktop applications.
k) The `f:ajax` element's _____ attribute specifies a space-separated list of element `id`s—the values of these elements are submitted as part of the Ajax request.
l) The `f:ajax` element's _____ attribute specifies a space-separated list of element `id`s for the elements that should be updated via partial page updates.

Answers to Self-Review Exercise

31.1 a) Asynchronous JavaScript and XML. b) connection pool. c) JNDI (Java Naming and Directory Interface). d) `DataSource`. e) `@Resource`. f) `h:dataTable`. g) `action`. h) `f:facet`, `"header"`, `"footer"`. i) Ajax. j) partial page updates. k) `execute`. l) `render`.

Exercises

31.2 ***(`Guestbook` Application)*** Create a JSF web app that allows users to sign and view a guestbook. Use the `Guestbook` database to store guestbook entries. [*Note:* A SQL script to create the `Guestbook` database is provided in the examples directory for this chapter.] 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. Using the `AddressBook` app in Section 31.2 as your guide, create two Facelets pages and a managed bean. The `index.xhtml` page should show the Guestbook entries in tabular format and should provide a button to add an entry to the Guestbook. When the user clicks this button, display an `addentry.xhtml` page. Provide `h:inputText` elements for the user's name and email address, an `h:inputTextarea` for the message and a **Sign Guestbook** button to submit the form. When the form is submitted, you should store in the `Guestbook` database a new entry containing the user's input and the date of the entry.

31.3 ***(`AddressBook` Application Modification: Ajax)*** Combine the two Facelets pages of the `AddressBook` application (Section 31.2) into a single page. Use Ajax capabilities to submit the new address book entry and to perform a partial page update that rerenders `h:dataTable` with the updated list of addresses.

31.4 ***(`AddressBook` Application Modification)*** Modify your solution to Exercise 31.3 to add a search capability that allows the user to search by last name. When the user presses the **Search** button, use Ajax to submit the search key and perform a partial page update that displays only the matching addresses in the `h:dataTable`.