

JavaServer™ Faces Web Apps: Part 2

31

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.



Outline

31_2 Chapter 31 JavaServer™ Faces Web Apps: Part 2

- | | |
|---|--|
| 31.1 Introduction
31.2 Accessing Databases in Web Apps | 31.3 Ajax
31.4 Adding Ajax Functionality to the Validation App
31.5 Wrap-Up |
|---|--|

[Summary](#) | [Self-Review Exercise](#) | [Answers to Self-Review Exercise](#) | [Exercises](#)

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

First Name	Last Name	Street	City	State	Zip code
Sue	Black	1000 Michigan Ave.	Chicago	IL	60605
James	Blue	1000 Harbor Ave.	Seattle	WA	98116
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
Bob	Green	5 Bay St.	San Francisco	CA	94133
Mary	Green	300 Massachusetts Ave.	Boston	MA	02115
Liz	White	100 5th Ave.	New York	NY	10011

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

31.2 Accessing Databases in Web Apps

31_3

b) Form for adding an entry

Address Book: Add Entry

First name: Jessica
 Last name: Magenta
 Street: 1 Main Street
 City: SomeCity
 State: FL
 Zipcode: 12345

[Save Address](#) [Return to Addresses](#)

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

Address Book

First Name	Last Name	Street	City	State	Zip code
Sue	Black	1000 Michigan Ave.	Chicago	IL	60605
James	Blue	1000 Harbor Ave.	Seattle	WA	98116
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
Bob	Green	5 Bay St.	San Francisco	CA	94133
Mary	Green	300 Massachusetts Ave.	Boston	MA	02115
Jessica	Magenta	1 Main Street	SomeCity	FL	12345
Liz	White	100 5th Ave.	New York	NY	10011

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.

31_4 Chapter 31 JavaServer™ Faces Web Apps: Part 2

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.xhtml` (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 

31.2 Accessing Databases in Web Apps 31_5

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 I of 5.)

31.6 Chapter 31 JavaServer™ Faces Web Apps: Part 2

```

16 // define the data source
17 @DataSourceDefinition(
18     name = "java:global/jdbc/addressbook",
19     className = "org.apache.derby.jdbc.ClientDataSource",
20     url = "jdbc:derby://localhost:1527/addressbook",
21     databaseName = "addressbook",
22     user = "APP",
23     password = "APP")
24
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.)

31.2 Accessing Databases in Web Apps **31_7**

```
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        }
122    }
123
124    // return a single address entry
125    public Address getAddress()
126    {
127        return address;
128    }
129
130    // set a single address entry
131    public void setAddress(Address address)
132    {
133        this.address = address;
134    }
135
136    // return a list of addresses
137    public List<Address> getAddresses()
138    {
139        return addresses;
140    }
141
142    // set a list of addresses
143    public void setAddresses(List<Address> addresses)
144    {
145        this.addresses = addresses;
146    }
147
148    // return the number of addresses
149    public int getAddressCount()
150    {
151        return addresses.size();
152    }
153
154    // add an address to the list
155    public void addAddress(Address address)
156    {
157        addresses.add(address);
158    }
159
160    // remove an address from the list
161    public void removeAddress(Address address)
162    {
163        addresses.remove(address);
164    }
165
166    // clear the list of addresses
167    public void clearAddresses()
168    {
169        addresses.clear();
170    }
171
172    // return the street
173    public String getStreet()
174    {
175        return street;
176    }
177
178    // set the street
179    public void setStreet(String street)
180    {
181        this.street = street;
182    }
183
184    // return the city
185    public String getCity()
186    {
187        return city;
188    }
189
190    // set the city
191    public void setCity(String city)
192    {
193        this.city = city;
194    }
195
196    // return the state
197    public String getState()
198    {
199        return state;
200    }
201
202    // set the state
203    public void setState(String state)
204    {
205        this.state = state;
206    }
207
208    // return the zipcode
209    public String getZipcode()
210    {
211        return zipcode;
212    }
213
214    // set the zipcode
215    public void setZipcode(String zipcode)
216    {
217        this.zipcode = zipcode;
218    }
219
220    // return the address
221    public Address getAddress()
222    {
223        return address;
224    }
225
226    // set the address
227    public void setAddress(Address address)
228    {
229        this.address = address;
230    }
231
232    // return the list of addresses
233    public List<Address> getAddresses()
234    {
235        return addresses;
236    }
237
238    // set the list of addresses
239    public void setAddresses(List<Address> addresses)
240    {
241        this.addresses = addresses;
242    }
243
244    // return the count of addresses
245    public int getAddressCount()
246    {
247        return addresses.size();
248    }
249
250    // add an address to the list
251    public void addAddress(Address address)
252    {
253        addresses.add(address);
254    }
255
256    // remove an address from the list
257    public void removeAddress(Address address)
258    {
259        addresses.remove(address);
260    }
261
262    // clear the list of addresses
263    public void clearAddresses()
264    {
265        addresses.clear();
266    }
267
268    // return the street
269    public String getStreet()
270    {
271        return street;
272    }
273
274    // set the street
275    public void setStreet(String street)
276    {
277        this.street = street;
278    }
279
280    // return the city
281    public String getCity()
282    {
283        return city;
284    }
285
286    // set the city
287    public void setCity(String city)
288    {
289        this.city = city;
290    }
291
292    // return the state
293    public String getState()
294    {
295        return state;
296    }
297
298    // set the state
299    public void setState(String state)
300    {
301        this.state = state;
302    }
303
304    // return the zipcode
305    public String getZipcode()
306    {
307        return zipcode;
308    }
309
310    // set the zipcode
311    public void setZipcode(String zipcode)
312    {
313        this.zipcode = zipcode;
314    }
315
316    // return the address
317    public Address getAddress()
318    {
319        return address;
320    }
321
322    // set the address
323    public void setAddress(Address address)
324    {
325        this.address = address;
326    }
327
328    // return the list of addresses
329    public List<Address> getAddresses()
330    {
331        return addresses;
332    }
333
334    // set the list of addresses
335    public void setAddresses(List<Address> addresses)
336    {
337        this.addresses = addresses;
338    }
339
340    // return the count of addresses
341    public int getAddressCount()
342    {
343        return addresses.size();
344    }
345
346    // add an address to the list
347    public void addAddress(Address address)
348    {
349        addresses.add(address);
350    }
351
352    // remove an address from the list
353    public void removeAddress(Address address)
354    {
355        addresses.remove(address);
356    }
357
358    // clear the list of addresses
359    public void clearAddresses()
360    {
361        addresses.clear();
362    }
363
364    // return the street
365    public String getStreet()
366    {
367        return street;
368    }
369
370    // set the street
371    public void setStreet(String street)
372    {
373        this.street = street;
374    }
375
376    // return the city
377    public String getCity()
378    {
379        return city;
380    }
381
382    // set the city
383    public void setCity(String city)
384    {
385        this.city = city;
386    }
387
388    // return the state
389    public String getState()
390    {
391        return state;
392    }
393
394    // set the state
395    public void setState(String state)
396    {
397        this.state = state;
398    }
399
400    // return the zipcode
401    public String getZipcode()
402    {
403        return zipcode;
404    }
405
406    // set the zipcode
407    public void setZipcode(String zipcode)
408    {
409        this.zipcode = zipcode;
410    }
411
412    // return the address
413    public Address getAddress()
414    {
415        return address;
416    }
417
418    // set the address
419    public void setAddress(Address address)
420    {
421        this.address = address;
422    }
423
424    // return the list of addresses
425    public List<Address> getAddresses()
426    {
427        return addresses;
428    }
429
430    // set the list of addresses
431    public void setAddresses(List<Address> addresses)
432    {
433        this.addresses = addresses;
434    }
435
436    // return the count of addresses
437    public int getAddressCount()
438    {
439        return addresses.size();
440    }
441
442    // add an address to the list
443    public void addAddress(Address address)
444    {
445        addresses.add(address);
446    }
447
448    // remove an address from the list
449    public void removeAddress(Address address)
450    {
451        addresses.remove(address);
452    }
453
454    // clear the list of addresses
455    public void clearAddresses()
456    {
457        addresses.clear();
458    }
459
460    // return the street
461    public String getStreet()
462    {
463        return street;
464    }
465
466    // set the street
467    public void setStreet(String street)
468    {
469        this.street = street;
470    }
471
472    // return the city
473    public String getCity()
474    {
475        return city;
476    }
477
478    // set the city
479    public void setCity(String city)
480    {
481        this.city = city;
482    }
483
484    // return the state
485    public String getState()
486    {
487        return state;
488    }
489
490    // set the state
491    public void setState(String state)
492    {
493        this.state = state;
494    }
495
496    // return the zipcode
497    public String getZipcode()
498    {
499        return zipcode;
500    }
501
502    // set the zipcode
503    public void setZipcode(String zipcode)
504    {
505        this.zipcode = zipcode;
506    }
507
508    // return the address
509    public Address getAddress()
510    {
511        return address;
512    }
513
514    // set the address
515    public void setAddress(Address address)
516    {
517        this.address = address;
518    }
519
520    // return the list of addresses
521    public List<Address> getAddresses()
522    {
523        return addresses;
524    }
525
526    // set the list of addresses
527    public void setAddresses(List<Address> addresses)
528    {
529        this.addresses = addresses;
530    }
531
532    // return the count of addresses
533    public int getAddressCount()
534    {
535        return addresses.size();
536    }
537
538    // add an address to the list
539    public void addAddress(Address address)
540    {
541        addresses.add(address);
542    }
543
544    // remove an address from the list
545    public void removeAddress(Address address)
546    {
547        addresses.remove(address);
548    }
549
550    // clear the list of addresses
551    public void clearAddresses()
552    {
553        addresses.clear();
554    }
555
556    // return the street
557    public String getStreet()
558    {
559        return street;
560    }
561
562    // set the street
563    public void setStreet(String street)
564    {
565        this.street = street;
566    }
567
568    // return the city
569    public String getCity()
570    {
571        return city;
572    }
573
574    // set the city
575    public void setCity(String city)
576    {
577        this.city = city;
578    }
579
580    // return the state
581    public String getState()
582    {
583        return state;
584    }
585
586    // set the state
587    public void setState(String state)
588    {
589        this.state = state;
590    }
591
592    // return the zipcode
593    public String getZipcode()
594    {
595        return zipcode;
596    }
597
598    // set the zipcode
599    public void setZipcode(String zipcode)
600    {
601        this.zipcode = zipcode;
602    }
603
604    // return the address
605    public Address getAddress()
606    {
607        return address;
608    }
609
610    // set the address
611    public void setAddress(Address address)
612    {
613        this.address = address;
614    }
615
616    // return the list of addresses
617    public List<Address> getAddresses()
618    {
619        return addresses;
620    }
621
622    // set the list of addresses
623    public void setAddresses(List<Address> addresses)
624    {
625        this.addresses = addresses;
626    }
627
628    // return the count of addresses
629    public int getAddressCount()
630    {
631        return addresses.size();
632    }
633
634    // add an address to the list
635    public void addAddress(Address address)
636    {
637        addresses.add(address);
638    }
639
640    // remove an address from the list
641    public void removeAddress(Address address)
642    {
643        addresses.remove(address);
644    }
645
646    // clear the list of addresses
647    public void clearAddresses()
648    {
649        addresses.clear();
650    }
651
652    // return the street
653    public String getStreet()
654    {
655        return street;
656    }
657
658    // set the street
659    public void setStreet(String street)
660    {
661        this.street = street;
662    }
663
664    // return the city
665    public String getCity()
666    {
667        return city;
668    }
669
670    // set the city
671    public void setCity(String city)
672    {
673        this.city = city;
674    }
675
676    // return the state
677    public String getState()
678    {
679        return state;
680    }
681
682    // set the state
683    public void setState(String state)
684    {
685        this.state = state;
686    }
687
688    // return the zipcode
689    public String getZipcode()
690    {
691        return zipcode;
692    }
693
694    // set the zipcode
695    public void setZipcode(String zipcode)
696    {
697        this.zipcode = zipcode;
698    }
699
700    // return the address
701    public Address getAddress()
702    {
703        return address;
704    }
705
706    // set the address
707    public void setAddress(Address address)
708    {
709        this.address = address;
710    }
711
712    // return the list of addresses
713    public List<Address> getAddresses()
714    {
715        return addresses;
716    }
717
718    // set the list of addresses
719    public void setAddresses(List<Address> addresses)
720    {
721        this.addresses = addresses;
722    }
723
724    // return the count of addresses
725    public int getAddressCount()
726    {
727        return addresses.size();
728    }
729
730    // add an address to the list
731    public void addAddress(Address address)
732    {
733        addresses.add(address);
734    }
735
736    // remove an address from the list
737    public void removeAddress(Address address)
738    {
739        addresses.remove(address);
740    }
741
742    // clear the list of addresses
743    public void clearAddresses()
744    {
745        addresses.clear();
746    }
747
748    // return the street
749    public String getStreet()
750    {
751        return street;
752    }
753
754    // set the street
755    public void setStreet(String street)
756    {
757        this.street = street;
758    }
759
760    // return the city
761    public String getCity()
762    {
763        return city;
764    }
765
766    // set the city
767    public void setCity(String city)
768    {
769        this.city = city;
770    }
771
772    // return the state
773    public String getState()
774    {
775        return state;
776    }
777
778    // set the state
779    public void setState(String state)
780    {
781        this.state = state;
782    }
783
784    // return the zipcode
785    public String getZipcode()
786    {
787        return zipcode;
788    }
789
790    // set the zipcode
791    public void setZipcode(String zipcode)
792    {
793        this.zipcode = zipcode;
794    }
795
796    // return the address
797    public Address getAddress()
798    {
799        return address;
800    }
801
802    // set the address
803    public void setAddress(Address address)
804    {
805        this.address = address;
806    }
807
808    // return the list of addresses
809    public List<Address> getAddresses()
810    {
811        return addresses;
812    }
813
814    // set the list of addresses
815    public void setAddresses(List<Address> addresses)
816    {
817        this.addresses = addresses;
818    }
819
820    // return the count of addresses
821    public int getAddressCount()
822    {
823        return addresses.size();
824    }
825
826    // add an address to the list
827    public void addAddress(Address address)
828    {
829        addresses.add(address);
830    }
831
832    // remove an address from the list
833    public void removeAddress(Address address)
834    {
835        addresses.remove(address);
836    }
837
838    // clear the list of addresses
839    public void clearAddresses()
840    {
841        addresses.clear();
842    }
843
844    // return the street
845    public String getStreet()
846    {
847        return street;
848    }
849
850    // set the street
851    public void setStreet(String street)
852    {
853        this.street = street;
854    }
855
856    // return the city
857    public String getCity()
858    {
859        return city;
860    }
861
862    // set the city
863    public void setCity(String city)
864    {
865        this.city = city;
866    }
867
868    // return the state
869    public String getState()
870    {
871        return state;
872    }
873
874    // set the state
875    public void setState(String state)
876    {
877        this.state = state;
878    }
879
880    // return the zipcode
881    public String getZipcode()
882    {
883        return zipcode;
884    }
885
886    // set the zipcode
887    public void setZipcode(String zipcode)
888    {
889        this.zipcode = zipcode;
890    }
891
892    // return the address
893    public Address getAddress()
894    {
895        return address;
896    }
897
898    // set the address
899    public void setAddress(Address address)
900    {
901        this.address = address;
902    }
903
904    // return the list of addresses
905    public List<Address> getAddresses()
906    {
907        return addresses;
908    }
909
910    // set the list of addresses
911    public void setAddresses(List<Address> addresses)
912    {
913        this.addresses = addresses;
914    }
915
916    // return the count of addresses
917    public int getAddressCount()
918    {
919        return addresses.size();
920    }
921
922    // add an address to the list
923    public void addAddress(Address address)
924    {
925        addresses.add(address);
926    }
927
928    // remove an address from the list
929    public void removeAddress(Address address)
930    {
931        addresses.remove(address);
932    }
933
934    // clear the list of addresses
935    public void clearAddresses()
936    {
937        addresses.clear();
938    }
939
940    // return the street
941    public String getStreet()
942    {
943        return street;
944    }
945
946    // set the street
947    public void setStreet(String street)
948    {
949        this.street = street;
950    }
951
952    // return the city
953    public String getCity()
954    {
955        return city;
956    }
957
958    // set the city
959    public void setCity(String city)
960    {
961        this.city = city;
962    }
963
964    // return the state
965    public String getState()
966    {
967        return state;
968    }
969
970    // set the state
971    public void setState(String state)
972    {
973        this.state = state;
974    }
975
976    // return the zipcode
977    public String getZipcode()
978    {
979        return zipcode;
980    }
981
982    // set the zipcode
983    public void setZipcode(String zipcode)
984    {
985        this.zipcode = zipcode;
986    }
987
988    // return the address
989    public Address getAddress()
990    {
991        return address;
992    }
993
994    // set the address
995    public void setAddress(Address address)
996    {
997        this.address = address;
998    }
999
1000   // return the list of addresses
1001  public List<Address> getAddresses()
1002  {
1003      return addresses;
1004  }
1005
1006  // set the list of addresses
1007  public void setAddresses(List<Address> addresses)
1008  {
1009      this.addresses = addresses;
1010  }
1011
1012  // return the count of addresses
1013  public int getAddressCount()
1014  {
1015      return addresses.size();
1016  }
1017
1018  // add an address to the list
1019  public void addAddress(Address address)
1020  {
1021      addresses.add(address);
1022  }
1023
1024  // remove an address from the list
1025  public void removeAddress(Address address)
1026  {
1027      addresses.remove(address);
1028  }
1029
1030  // clear the list of addresses
1031  public void clearAddresses()
1032  {
1033      addresses.clear();
1034  }
1035
1036  // return the street
1037  public String getStreet()
1038  {
1039      return street;
1040  }
1041
1042  // set the street
1043  public void setStreet(String street)
1044  {
1045      this.street = street;
1046  }
1047
1048  // return the city
1049  public String getCity()
1050  {
1051      return city;
1052  }
1053
1054  // set the city
1055  public void setCity(String city)
1056  {
1057      this.city = city;
1058  }
1059
1060  // return the state
1061  public String getState()
1062  {
1063      return state;
1064  }
1065
1066  // set the state
1067  public void setState(String state)
1068  {
1069      this.state = state;
1070  }
1071
1072  // return the zipcode
1073  public String getZipcode()
1074  {
1075      return zipcode;
1076  }
1077
1078  // set the zipcode
1079  public void setZipcode(String zipcode)
1080  {
1081      this.zipcode = zipcode;
1082  }
1083
1084  // return the address
1085  public Address getAddress()
1086  {
1087      return address;
1088  }
1089
1090  // set the address
1091  public void setAddress(Address address)
1092  {
1093      this.address = address;
1094  }
1095
1096  // return the list of addresses
1097  public List<Address> getAddresses()
1098  {
1099      return addresses;
1100  }
1101
1102  // set the list of addresses
1103  public void setAddresses(List<Address> addresses)
1104  {
1105      this.addresses = addresses;
1106  }
1107
1108  // return the count of addresses
1109  public int getAddressCount()
1110  {
1111      return addresses.size();
1112  }
1113
1114  // add an address to the list
1115  public void addAddress(Address address)
1116  {
1117      addresses.add(address);
1118  }
1119
1120  // remove an address from the list
1121  public void removeAddress(Address address)
1122  {
1123      addresses.remove(address);
1124  }
1125
1126  // clear the list of addresses
1127  public void clearAddresses()
1128  {
1129      addresses.clear();
1130  }
1131
1132  // return the street
1133  public String getStreet()
1134  {
1135      return street;
1136  }
1137
1138  // set the street
1139  public void setStreet(String street)
1140  {
1141      this.street = street;
1142  }
1143
1144  // return the city
1145  public String getCity()
1146  {
1147      return city;
1148  }
1149
1150  // set the city
1151  public void setCity(String city)
1152  {
1153      this.city = city;
1154  }
1155
1156  // return the state
1157  public String getState()
1158  {
1159      return state;
1160  }
1161
1162  // set the state
1163  public void setState(String state)
1164  {
1165      this.state = state;
1166  }
1167
1168  // return the zipcode
1169  public String getZipcode()
1170  {
1171      return zipcode;
1172  }
1173
1174  // set the zipcode
1175  public void setZipcode(String zipcode)
1176  {
1177      this.zipcode = zipcode;
1178  }
1179
1180  // return the address
1181  public Address getAddress()
1182  {
1183      return address;
1184  }
1185
1186  // set the address
1187  public void setAddress(Address address)
1188  {
1189      this.address = address;
1190  }
1191
1192  // return the list of addresses
1193  public List<Address> getAddresses()
1194  {
1195      return addresses;
1196  }
1197
1198  // set the list of addresses
1199  public void setAddresses(List<Address> addresses)
1200  {
1201      this.addresses = addresses;
1202  }
1203
1204  // return the count of addresses
1205  public int getAddressCount()
1206  {
1207      return addresses.size();
1208  }
1209
1210  // add an address to the list
1211  public void addAddress(Address address)
1212  {
1213      addresses.add(address);
1214  }
1215
1216  // remove an address from the list
1217  public void removeAddress(Address address)
1218  {
1219      addresses.remove(address);
1220  }
1221
1222  // clear the list of addresses
1223  public void clearAddresses()
1224  {
1225      addresses.clear();
1226  }
1227
1228  // return the street
1229  public String getStreet()
1230  {
1231      return street;
1232  }
1233
1234  // set the street
1235  public void setStreet(String street)
1236  {
1237      this.street = street;
1238  }
1239
1240  // return the city
1241  public String getCity()
1242  {
1243      return city;
1244  }
1245
1246  // set the city
1247  public void setCity(String city)
1248  {
1249      this.city = city;
1250  }
1251
1252  // return the state
1253  public String getState()
1254  {
1255      return state;
1256  }
1257
1258  // set the state
1259  public void setState(String state)
1260  {
1261      this.state = state;
1262  }
1263
1264  // return the zipcode
1265  public String getZipcode()
1266  {
1267      return zipcode;
1268  }
1269
1270  // set the zipcode
1271  public void setZipcode(String zipcode)
1272  {
1273      this.zipcode = zipcode;
1274  }
1275
1276  // return the address
1277  public Address getAddress()
1278  {
1279      return address;
1280  }
1281
1282  // set the address
1283  public void setAddress(Address address)
1284  {
1285      this.address = address;
1286  }
1287
1288  // return the list of addresses
1289  public List<Address> getAddresses()
1290  {
1291      return addresses;
1292  }
1293
1294  // set the list of addresses
1295  public void setAddresses(List<Address> addresses)
1296  {
1297      this.addresses = addresses;
1298  }
1299
1300  // return the count of addresses
1301  public int getAddressCount()
1302  {
1303      return addresses.size();
1304  }
1305
1306  // add an address to the list
1307  public void addAddress(Address address)
1308  {
1309      addresses.add(address);
1310  }
1311
1312  // remove an address from the list
1313  public void removeAddress(Address address)
1314  {
1315      addresses.remove(address);
1316  }
1317
1318  // clear the list of addresses
1319  public void clearAddresses()
1320  {
1321      addresses.clear();
1322  }
1323
1324  // return the street
1325  public String getStreet()
1326  {
1327      return street;
1328  }
1329
1330  // set the street
1331  public void setStreet(String street)
1332  {
1333      this.street = street;
1334  }
1335
1336  // return the city
1337  public String getCity()
1338  {
1339      return city;
1340  }
1341
1342  // set the city
1343  public void setCity(String city)
1344  {
1345      this.city = city;
1346  }
1347
1348  // return the state
1349  public String getState()
1350  {
1351      return state;
1352  }
1353
1354  // set the state
1355  public void setState(String state)
1356  {
1357      this.state = state;
1358  }
1359
1360  // return the zipcode
1361  public String getZipcode()
1362  {
1363      return zipcode;
1364  }
1365
1366  // set the zipcode
1367  public void setZipcode(String zipcode)
1368  {
1369      this.zipcode = zipcode;
1370  }
1371
1372  // return the address
1373  public Address getAddress()
1374  {
1375      return address;
1376  }
1377
1378  // set the address
1379  public void setAddress(Address address)
1380  {
1381      this.address = address;
1382  }
1383
1384  // return the list of addresses
1385  public List<Address> getAddresses()
1386  {
1387      return addresses;
1388  }
1389
1390  // set the list of addresses
1391  public void setAddresses(List<Address> addresses)
1392  {
1393      this.addresses = addresses;
1394  }
1395
1396  // return the count of addresses
1397  public int getAddressCount()
1398  {
1399      return addresses.size();
1400  }
1401
1402  // add an address to the list
1403  public void addAddress(Address address)
1404  {
1405      addresses.add(address);
1406  }
1407
1408  // remove an address from the list
1409  public void removeAddress(Address address)
1410  {
1411      addresses.remove(address);
1412  }
1413
1414  // clear the list of addresses
1415  public void clearAddresses()
1416  {
1417      addresses.clear();
1418  }
1419
1420  // return the street
1421  public String getStreet()
1422  {
1423      return street;
1424  }
1425
1426  // set the street
1427  public void setStreet(String street)
1428  {
1429      this.street = street;
1430  }
1431
1432  // return the city
1433  public String getCity()
1434  {
1435      return city;
1436  }
1437
1438  // set the city
1439  public void setCity(String city)
1440  {
1441      this.city = city;
1442  }
1443
1444  // return the state
1445  public String getState()
1446  {
1447      return state;
1448  }
1449
1450  // set the state
1451  public void setState(String state)
1452  {
1453      this.state = state;
1454  }
1455
1456  // return the zipcode
1457  public String getZipcode()
1458  {
1459      return zipcode;
1460  }
1461
1462  // set the zipcode
1463  public void setZipcode(String zipcode)
1464  {
1465      this.zipcode = zipcode;
1466  }
1467
1468  // return the address
1469  public Address getAddress()
1470  {
1471      return address;
1472  }
1473
1474  // set the address
1475  public void setAddress(Address address)
1476  {
1477      this.address = address;
1478  }
1479
1480  // return the list of addresses
1481  public List<Address> getAddresses()
1482  {
1483      return addresses;
1484  }
1485
1486  // set the list of addresses
1487  public void setAddresses(List<Address> addresses)
1488  {
1489      this.addresses = addresses;
1490  }
1491
1492  // return the count of addresses
1493  public int getAddressCount()
1494  {
1495      return addresses.size();
1496  }
1497
1498  // add an address to the list
1499  public void addAddress(Address address)
1500  {
1501      addresses.add(address);
1502  }
1503
1504  // remove an address from the list
1505  public void removeAddress(Address address)
1506  {
1507      addresses.remove(address);
1508  }
1509
1510  // clear the list of addresses
1511  public void clearAddresses()
1512  {
1513      addresses.clear();
1514  }
1515
1516  // return the street
1517  public String getStreet()
1518  {
1519      return street;
1520  }
1521
1522  // set the street
1523  public void setStreet(String street)
1524  {
1525      this.street = street;
1526  }
1527
1528  // return the city
1529  public String getCity()
1530  {
1531      return city;
1532  }
1533
1534  // set the city
1535  public void setCity(String city)
1536  {
1537      this.city = city;
1538  }
1539
1540  // return the state
1541  public String getState()
1542  {
1543      return state;
1544  }
1545
1546  // set the state
1547  public void setState(String state)
1548  {
1549      this.state = state;
1550  }
1551
1552  // return the zipcode
1553  public String getZipcode()
1554  {
1555      return zipcode;
1556  }
1557
1558  // set the zipcode
1559  public void setZipcode(String zipcode)
1560  {
1561      this.zipcode = zipcode;
1562  }
1563
1564  // return the address
1565  public Address getAddress()
1566  {
1567      return address;
1568  }
1569
1570  // set the address
1571  public void setAddress(Address address)
1572  {
1573      this.address = address;
1574  }
1575
1576  // return the list of addresses
1577  public List<Address> getAddresses()
1578  {
1579      return addresses;
1580  }
15
```

31.8 Chapter 31 JavaServer™ Faces Web Apps: Part 2

```

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

```

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

31.2 Accessing Databases in Web Apps 31_9

```

175      // specify the PreparedStatement's arguments
176      addEntry.setString(1, getFirstName());
177      addEntry.setString(2, getLastName());
178      addEntry.setString(3, getStreet());
179      addEntry.setString(4, getCity());
180      addEntry.setString(5, getState());
181      addEntry.setString(6, getZipcode());
182
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



31_10 Chapter 31 JavaServer™ Faces Web Apps: Part 2

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

31.2 Accessing Databases in Web Apps 31_11

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.)

31_12 Chapter 31 JavaServer™ Faces Web Apps: Part 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—

31.2 Accessing Databases in Web Apps 31_13

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.)

31_14 Chapter 31 JavaServer™ Faces Web Apps: Part 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.

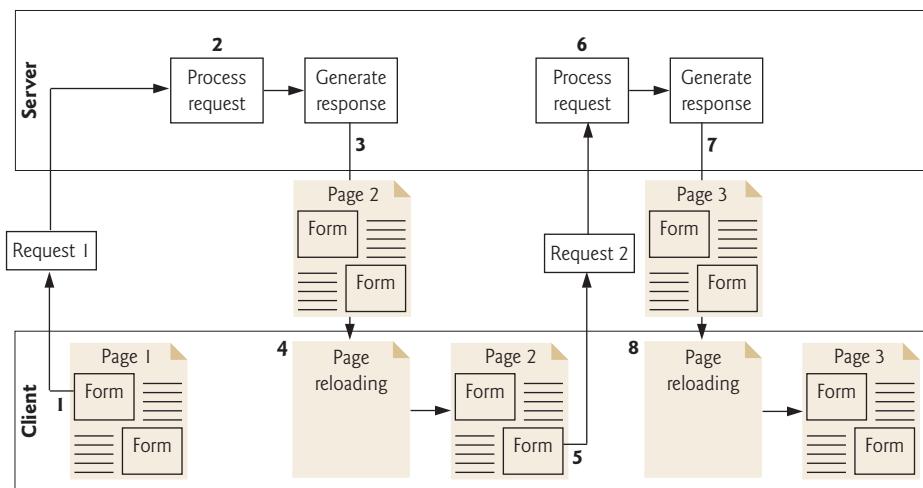


Fig. 31.5 | Classic web application reloading the page for every user interaction.

31_16 Chapter 31 JavaServer™ Faces Web Apps: Part 2

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).

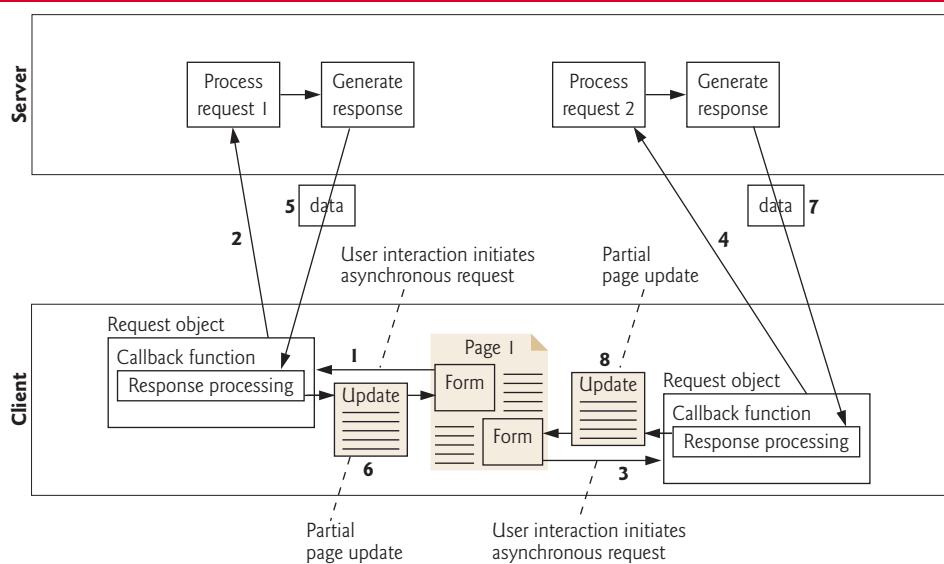


Fig. 31.6 | Ajax-enabled web application interacting with the server asynchronously.

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-

31.4 Adding Ajax Functionality to the Validation App 31_17

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

The screenshot shows a web browser window with a title bar "Validating Form Data". The address bar shows "localhost:8080/ValidationAjax/". The main content area has a heading "Please fill out the following form:". Below it is a message "All fields are required and must contain valid information". There are three input fields: "Name:", "E-mail:", and "Phone:". Below the fields is a "Submit" button.

b) Error messages displayed after submitting the empty form

The screenshot shows the same web browser window as part (a). After submission, red error messages appear next to each input field: "Name:" with "Please enter your name", "E-mail:" with "Please enter a valid e-mail address", and "Phone:" with "Please enter a valid phone number". The "Submit" button is highlighted with a cursor.

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

31_18 Chapter 31 JavaServer™ Faces Web Apps: Part 2

c) Error messages displayed after submitting invalid information

The screenshot shows a browser window titled "Validating Form Data" with the URL "localhost:8080/ValidationAjax/". The page contains the following text:

Please fill out the following form:

All fields are required and must contain valid information

Name: Name must be fewer than 30 characters
 E-mail: Invalid e-mail address format
 Phone: Invalid phone number format

d) Successfully submitted form

The screenshot shows a browser window titled "Validating Form Data" with the URL "localhost:8080/ValidationAjax/". The page contains the following text:

Please fill out the following form:

All fields are required and must contain valid information

Name:
 E-mail:
 Phone:

**Name: Paul
E-Mail: paul@somedomain.com
Phone: (555) 555-5555**

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.

31.4 Adding Ajax Functionality to the Validation App

31_19

```

1  <?xml version='1.0' encoding='UTF-8' ?>
2
3  <!-- index.xhtml -->
4  <!-- Validating user input -->
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>Validating Form Data</title>
12   <h:outputStylesheet name="style.css" library="css"/>
13 </h:head>
14 <h:body>
15   <h:form>
16     <h1>Please fill out the following form:</h1>
17     <p>All fields are required and must contain valid information</p>
18     <h:panelGrid columns="3">
19       <h:outputText value="Name:"/>
20       <h:inputText id="nameInputText" required="true"
21         requiredMessage="Please enter your name"
22         value="#{validationBean.name}"
23         validatorMessage="Name must be fewer than 30 characters">
24           <f:validateLength maximum="30" />
25         </h:inputText>
26         <h:message id="nameMessage" for="nameInputText"
27           styleClass="error"/>
28       <h:outputText value="E-mail:"/>
29       <h:inputText id="emailInputText" required="true"
30         requiredMessage="Please enter a valid e-mail address"
31         value="#{validationBean.email}"
32         validatorMessage="Invalid e-mail address format">
33           <f:validateRegex pattern=
34             "\w+([-.\']\w+)*@\w+([.-]\w+)*\.\w+([.-]\w+)*" />
35         </h:inputText>
36         <h:message id="emailMessage" for="emailInputText"
37           styleClass="error"/>
38       <h:outputText value="Phone:"/>
39       <h:inputText id="phoneInputText" required="true"
40         requiredMessage="Please enter a valid phone number"
41         value="#{validationBean.phone}"
42         validatorMessage="Invalid phone number format">
43           <f:validateRegex pattern=
44             "((\(\d{3}\)\s)?|\s(\d{3}-))?\d{3}-\d{4}" />
45         </h:inputText>
46         <h:message id="phoneMessage" for="phoneInputText"
47           styleClass="error"/>
48     </h:panelGrid>
49     <h:commandButton value="Submit">
50       <f:ajax execute="nameInputText emailInputText phoneInputText"
51         render=
52           "nameMessage emailMessage phoneMessage resultOutputText"/>
53     </h:commandButton>

```

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

31_20 Chapter 31 JavaServer™ Faces Web Apps: Part 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 *ids*—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 *ids* 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 mult-tier 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 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.



31_22 Chapter 31 JavaServer™ Faces Web Apps: Part 2

- 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 `ids`—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 `ids` for the elements that should be updated via partial page updates.

Self-Review Exercise**31_23**

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 `ids`—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 `ids` 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`.



31_24 Chapter 31 JavaServer™ Faces Web Apps: Part 2