I do not fear computers. I fear the lack of them.
► Isaac Asimov

Chapter 1
Adding Form Input Controls

Fasten your seat belt, because the next few chapters will be fast-paced. We're going to crank up the webmail application by giving life to the pages that we laid out in the previous chapter, using all kinds of nifty Stripes features. We'll start with the three pages that deal with email messages: the *Message List*, *Message Details*, and *Message Compose* pages.

The *Message List* page shows the messages that are in a folder. Each message has a link which displays the text of the message in the *Message Details* page. The user can write and send emails in the *Message Compose* page.

There's a lot going on in these pages. In this chapter, we'll concentrate on using the different types of form input controls—checkboxes, radio buttons, and so on. In the next chapter, we'll discuss the other features that we need to finish implementing the three *Message* pages.

A few supporting classes—model classes, a DAO, and type converters—are involved here, but I don't want to spend too much time and space discussing the details because they don't involve anything we haven't seen before. So let's briefly discuss the essentials. The classes involved in supplying information about folders and messages are illustrated in Figure 1.1, on the following page.

- The Folder model class represents a folder and contains a List of Message objects, which contain the information concerning an individual message: who it's from, the subject, the message text, etc.
- The FolderDao interface defines methods to manage folders and the messages that they contain. MockFolderDao is an implementation

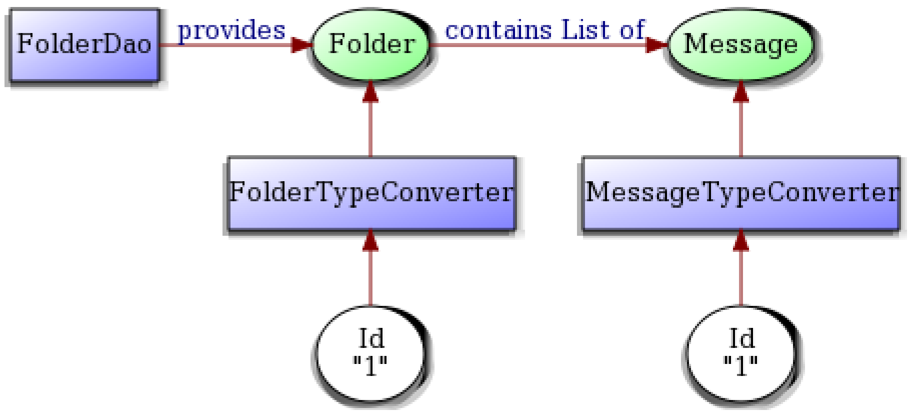


Figure 1.1: FOLDERS AND MESSAGES.

that contains a few fictitious messages so that we see something when trying out the example.

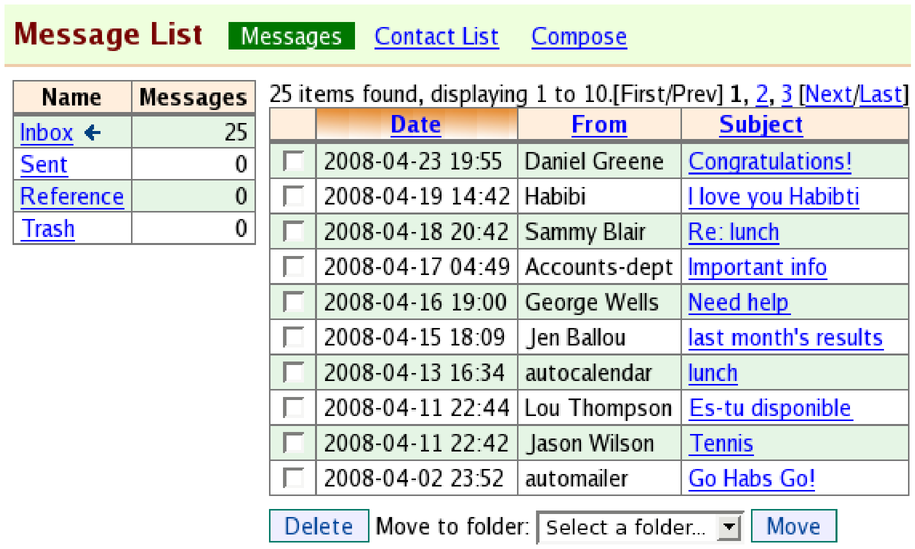
- FolderTypeConverter converts a String Id to a Folder, while MessageTypeConverter does the same for a Message. With these type converters, we can use Id parameters in forms and the Folder and Message property types in Action Beans. We discussed this technique in the (as yet) unwritten *sec.WorkingWithDataTypes:IdTypeConverter*.

Let's get to using form input controls, starting with checkboxes.

1.1 Checkboxes

The *Message List* page, shown in Figure 1.2, on the next page, allows the user to select a folder on the left and view the messages that are in that folder. The list of messages includes links to view the details of a message. There's also a column of checkboxes on the left, allowing the user to select messages. With the controls at the bottom, the user can delete the selected messages or move them to a different folder.

The MessageListActionBean class is the Action Bean associated to this page, with a default event handler that forwards to message_list.jsp. This JSP renders the page. To create the column of checkboxes in the message list, we'll use the <*s:checkbox*> tag:



Name	Messages
Inbox ←	25
Sent	0
Reference	0
Trash	0

25 items found, displaying 1 to 10.[First/Prev] 1, 2, 3 [Next/Last]

	Date	From	Subject
☐	2008-04-23 19:55	Daniel Greene	Congratulations!
☐	2008-04-19 14:42	Habibi	I love you Habibti
☐	2008-04-18 20:42	Sammy Blair	Re: lunch
☐	2008-04-17 04:49	Accounts-dept	Important info
☐	2008-04-16 19:00	George Wells	Need help
☐	2008-04-15 18:09	Jen Ballou	last month's results
☐	2008-04-13 16:34	autocalendar	lunch
☐	2008-04-11 22:44	Lou Thompson	Es-tu disponible
☐	2008-04-11 22:42	Jason Wilson	Tennis
☐	2008-04-02 23:52	automailer	Go Habs Go!

Figure 1.2: THE MESSAGE LIST PAGE.

Stripes form input tags

All Stripes form input tags use the name= attribute to bind the value of the input to an Action Bean property. Also note that these tags must always be nested within an <*s:form*> tag.

Using Stripes form input tags gives Stripes a chance to do several things for you, such as automatically populating the fields with existing data. This works both for updating the properties of an existing object and for redisplaying a form when validation errors have occurred. Other goodies include adding the maxlength= attribute to a text field when the corresponding property has a maxlength= validation, using model objects to generate options in a select box (as we'll see in this chapter), and looking up labels in resource bundles (explored in the (as yet) unwritten *chp.MultilingualApplications*.)

Finally, the Stripes input tags also support all HTML attributes. The attributes for which Stripes has no use are passed through without modification. Yes, this includes the class= attribute—you don't have to use styleClass=, cssClass=, or any other renamed attribute like some other frameworks require you to do.

email_19/web/WEB-INF/jsp/message_list.jsp

```
<s:form beanclass="stripesbook.action.MessageListActionBean">
  <d:table name="${folder.messages}" requestURI="" id="message"
    pagesize="10" defaultsort="2" defaultorder="descending">
    <d:column>
      <s:checkbox name="selectedMessages" value="${message.id}"/>
    </d:column>
    <!-- ... -->
  </d:table>
</s:form>
```

The checkboxes are bound to the selectedMessages property of MessageListActionBean. For every checkbox that the user checks, the value that's in the value= attribute will be sent as input—in this case, the Id of the corresponding message. We can use a List<Message> property to receive all the selected messages:[1]

email_19/src/stripesbook/action/MessageListActionBean.java

```
private List<Message> selectedMessages;
public List<Message> getSelectedMessages() {
    return selectedMessages;
}
public void setSelectedMessages(List<Message> selectedMessages) {
    this.selectedMessages = selectedMessages;
}
```

When the user submits the form, selectedMessages will contain the messages that the user checked. It's very simple, then, to do something with these messages in an event handler. For example, with the Delete button associated to the delete() event handler, here's how you would delete the selected messages:

email_19/src/stripesbook/action/MessageListActionBean.java

```
public Resolution delete() {
    for (Message message : selectedMessages) {
        folderDao.deleteMessage(message);
    }
    return new RedirectResolution(getClass());
}
private FolderDao folderDao = MockFolderDao.getInstance();
```

Making sure that at least one message is selected is easy too. When no checkbox is checked, the corresponding property on the Action Bean

1. This is one of several ways to use checkboxes. I explain the different ways in the sidebar on page 6.

will be **null**. So all you have to do is make selectedMessages a required field:

email_19/src/stripesbook/action/MessageListActionBean.java

```
@Validate(required=true, on={"delete", "moveToFolder"})
private List<Message> selectedMessages;
```

Let's not forget to display the potential error message to the user! Adding an <*s:errors*> tag under the message table and an error message in the resource bundle does the trick:

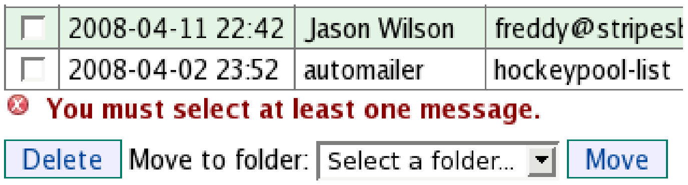
email_19/web/WEB-INF/jsp/message_list.jsp

```
<div><s:errors field="selectedMessages"/></div>
```

email_19/res/StripesResources.properties

```
selectedMessages.valueNotPresent=You must select at least one message.
```

Now, pressing the Delete button with no selected checkbox gives an error message as shown below.

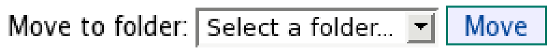


☐	2008-04-11 22:42	Jason Wilson	freddy@stripesb
☐	2008-04-02 23:52	automailer	hockeypool-list

You must select at least one message.

1.2 Select boxes

To move the selected messages to a different folder, the user must first select a folder. For that, we'll add a select box to the left of the Move button:



The <*s:select*> tag creates a select box that allows a single selection by default. Stripes offers helper tags that generate the options of the select box from a Collection, an enum, or a Map. In our case, we want the options to be the list of folders. We can obtain this list from the FoldersViewHelper class that we wrote on page ??. Here's the code to create the select box:

email_19/web/WEB-INF/jsp/common/message_action.jsp

```
<jsp:useBean class="stripesbook.view.FoldersViewHelper" id="folders"/>
  Move to folder:
  <s:select name="selectedFolder">
    <s:option value="">Select a folder...</s:option>
```

A bit more about how checkboxes work

In Stripes you can use checkboxes with different types of properties in the Action Bean: with an individual property, a Collection, or a Map.

- With an individual property:

  ```
  <s:checkbox name="property" value="value1"/>
  ```

  ```
  T property;
  ```

  If the checkbox is checked, property is set to value1 with the usual type conversion to the type T. If the checkbox is unchecked, property is set to its *default value*: null for Object, false for boolean, 0 for int, and so on.

- With a Collection property:

  ```
  <s:checkbox name="property" value="value1"/>
  <s:checkbox name="property" value="value2"/>
  ...
  <s:checkbox name="property" value="valueN"/>
  ```

  ```
  Collection<T> property;
  ```

  For each checked checkbox, the corresponding value (value1, value2, ...) is added to the property collection. Again, values are converted to the type T.

  If no checkbox is checked, property is set to null.

- With a Map property:

  ```
  <s:checkbox name="property.key1" value="value1"/>
  <s:checkbox name="property.key2" value="value2"/>
  ...
  <s:checkbox name="property.keyN" value="valueN"/>
  ```

  ```
  Map<K,V> property;
  ```

  If **at least one** checkbox is checked, the property map contains **every** key: key1, key2, ... , keyN. For each key, the value is the corresponding value1, value2, and so on if the checkbox is checked, or null if the checkbox is unchecked.

  Type conversion occurs both for converting keys to K and values to V.

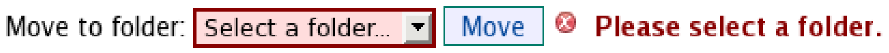
  If no checkbox is checked, property is set to null.

No matter how you use checkboxes, the value= attribute of the *<s:checkbox>* tag is optional. If omitted, the default value is true.

```
    <s:options-collection collection="${folders.folders}"
      value="id" label="name"/>
  </s:select>
  <s:submit name="moveToFolder" value="Move"/>
  <s:errors field="selectedFolder"/>
```

By using <*s:options-collection*> within <*s:select*>, you can easily generate options from the list of folders. Each option has a value and a label obtained by calling the value= and label= properties on each object of the collection. Here, getId() and getName() are called on each Folder object. The user sees the name of the folder as each option of the select box, and the folder's Id is set on the selectedFolder property of the Action Bean.

Also notice the first option, labeled *Select a folder...*. The <*s:option*> tag creates a single option that's added to the select box. But we don't want this option to be a valid selection—we're just using it so that the user sees the *Select a folder...* message in the select box. We can use value="" so that the option sends the empty string as a value. Since Stripes treats empty strings as **null**, the select box acts like a blank input field when the *Select a folder...* option is selected. By making selectedFolder a required field with @Validate(required=true, on="moveToFolder"), pressing Move without selecting a folder will show an error message:



Please select a folder.

This way of distinguishing the first option as a message to the user (and *not* a valid option) is much more elegant than putting a value of -1 or some other "magic number" for that option. Our code is not polluted with checks for this special value in the Action Bean and the manual creation of a required-field validation error. Using an empty string and making the field required with @Validate fits in very naturally into the Stripes validation mechanism.

Using multiple selection boxes

Select boxes can also allow the user to select more than one item at a time. For example, the *Message Compose* page, shown in Figure 1.3, on the following page, has a select box on the right side that contains the list of contacts. The user can easily add recipients by selecting the contacts and pressing one of the arrow buttons.

To allow multiple selection in the select box, add multiple=*"true"* to the <*s:select*> tag and, optionally, the size= attribute to indicate how many rows to show at a time:

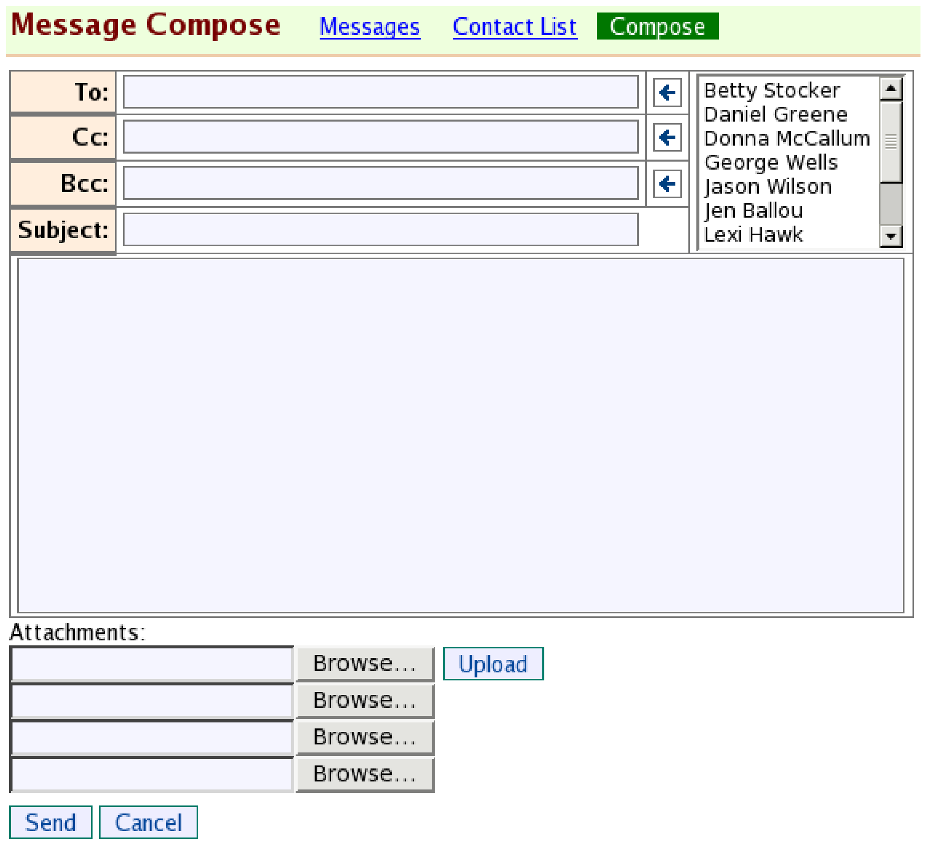


Figure 1.3: THE MESSAGE COMPOSE PAGE.

email_19/web/WEB-INF/jsp/message_compose.jsp

```
<jsp:useBean class="stripesbook.action.ContactListActionBean"
  id="contacts"/>
  <!-- ... -->
  <s:form beanclass="stripesbook.action.MessageComposeActionBean">
    <!-- ... -->
▶             <s:select name="contacts" multiple="true" size="7">
                <s:options-collection collection="${contacts.contacts}"
                  value="id" sort="firstName"/>
              </s:select>
    <!-- ... -->
  </s:form>
</s:layout-component>
```

Inside <*s:select*>, the <*s:options-collection*> tag generates options from the list of contacts obtained from ContactListActionBean. Notice that you can use the sort= attribute to indicate the property by which to sort the objects of the collection. We're sorting the contacts by their first name.

A multiple-selection box acts much like a series of checkboxes that are bound to a Collection property. In this case, the selected contacts are bound to the contacts property of the MessageComposeActionBean class:

email_19/src/stripesbook/action/MessageComposeActionBean.java

```
private List<Contact> contacts;
public List<Contact> getContacts() {
    return contacts;
}
public void setContacts(List<Contact> contacts) {
    this.contacts = contacts;
}
```

Stripes automatically puts the selected contacts in the list so that we can add them to the *To*, *Cc*, or *Bcc* field.

1.3 Image buttons and text areas

Continuing the *Message Compose* page, let's add the image buttons with arrows that are next to the contact list select box, and the text area where the user can compose the text of the message.

Using image buttons

The <*s:image*> tag creates an image button that invokes the Action Bean event handler indicated in the name= attribute. The src= attribute contains the path to the image:

email_19/web/WEB-INF/jsp/message_compose.jsp

```
  <c:set var="arrow" value="/images/arrow.png"/>
  <tr>
    <th>To:</th>
    <td><s:text name="message.to" size="60"/></td>
▶   <td><s:image name="addTo" src="${arrow}"/></td>
  </tr>
  <tr>
    <th>Cc:</th>
    <td><s:text name="message.cc" size="60"/></td>
▶   <td><s:image name="addCc" src="${arrow}"/></td>
  </tr>
  <tr>
    <th>Bcc:</th>
    <td><s:text name="message.bcc" size="60"/></td>
▶   <td><s:image name="addBcc" src="${arrow}"/></td>
```

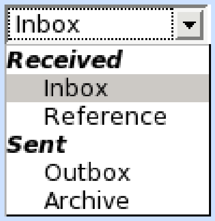
More select box features

Besides the *<s:options-collection>* tag, you can also use *<s:options-map>* and *<s:options-enumeration>* to generate options in a select box from a Map and an enum, respectively. With *<s:options-map>*, the collection of values is the map's set of Map.Entry objects, as obtained from entrySet(). For *<s:options-enumeration>*, specify the enumeration in the enum= attribute to generate options based on the values defined by the enumeration.

With all three *<s:options-xx>* tags, you can also use the group= attribute to generate *<optgroup/>* tags within a select box. This groups options together with a different label for each group. For example, say the Folder class had a type property. With two folders named Inbox and Reference having the *Received* type, and two folders named Outbox and Archive having the *Sent* type, you could use group="type" to group folders by their type:

```
<s:select name="selectedFolder">
  <s:options-collection collection="${actionBean.folders}"
    value="id" label="name" group="type"/>
</s:select>
```

This would generate a select box as shown below.



Here's one more tip. If you have a collection of objects from which you want to generate options in a select box, but want to display labels in different formats without polluting your model class with formatting code, you can always implement a Formatter with the different format types and patterns that you need. Then, you can generate the options with the plain *<s:option>* tag and use *<s:format>* to format the label in different ways. For example:

```
<s:select name="...">
  <c:forEach items="${someCollection}" var="item">
    <s:option value="${item.id}">
      <s:format value="${item}" formatType="..." formatPattern="..."/>
    </s:option>
  </c:forEach>
</s:select>
```

```
</tr>
```

This generates <input type="image" ...> tags. So what does the <*s:image*> tag do for you? It adds the application context path in front of the image path, and gives you the ability to look up images and alternate text in localized resource bundles as we'll see in the (as yet) unwritten *sec.MultilingualApplications:ImageButtons*.

When the user presses an arrow button, the selected contacts are added to the addresses that are in the field next to the button. But the user may have also entered other addresses directly in the text field. Combining the input of the text field with the input from the select box is somewhat tricky; we'll discuss this in the next chapter. Let's continue working with form input controls.

Adding the text area for the message text is a one-liner with the <*s:textarea*> tag:

email_19/web/WEB-INF/jsp/message_compose.jsp

```
<s:textarea name="message.text" cols="87" rows="12"/>
```

Again, using the Stripes equivalent instead of the plain HTML tag has the benefit of automatically repopulating the value—we don't want the user to lose the text if the form is submitted and a validation error occurs.

1.4 Using cross-page controls

Figure 1.4, on page 13 shows the *Message Details* page, which appears when the user clicks on a message subject from the *Message List* page. There's nothing spectacular about the *Message Details* page, but what's interesting is that the controls at the bottom are the same as in the *Message List* page; the only difference is that deleting or moving to a folder applies to the currently displayed message rather than a series of messages checked off in the *Message List* page. Let's see how we can define the controls in *one* place and reuse them in these two different contexts.

We'll start by putting the controls in a separate JSP (message_action.jsp) under the common directory since it's being used in more than one place:

email_19/web/WEB-INF/jsp/common/message_action.jsp

```
<jsp:useBean class="stripesbook.view.FoldersViewHelper" id="folders"/>
```

Tim Says. . .

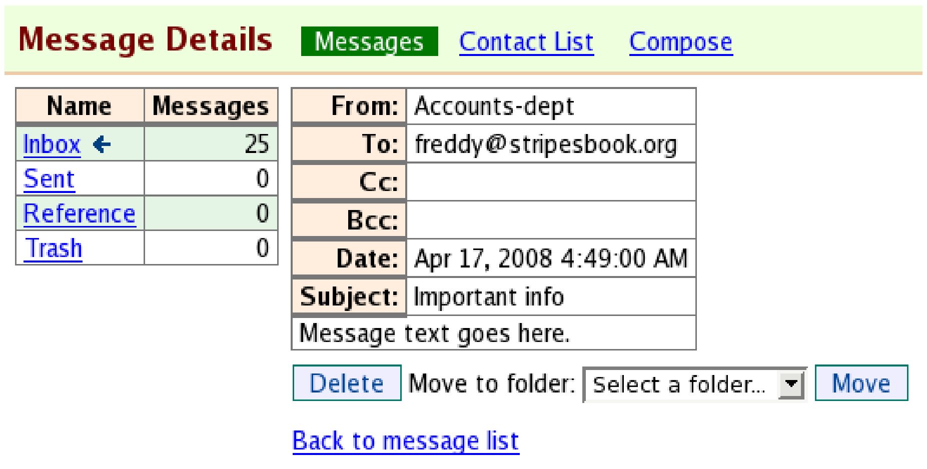
Input tags mimic the HTML tags as closely as possible

The Stripes input tags try very hard to mimic their HTML counterparts as closely as possible. This means that, in general, if you already know how to use HTML input, select, textarea, form tags and so on, that you should feel right at home with the Stripes tags. Even the class attribute for specifying CSS classes is the same.

There are of course, some deviations though. First is one you may have already noticed—where in HTML you would write <input type="X">, in Stripes you write <s:X>, for example you would write <s:radio> instead of <input type="radio">. The main reason for doing this is to make your life easier—each of the different input types has different required and permitted attributes. Making them separate tags allows the set of fields to be validated at compile time, and checked in most popular IDEs. Keeping them as one tag would have led to unhelpful code completion and more runtime errors.

In addition, several Stripes tags add attributes to the list supported by their HTML equivalents. This is done to allow you to activate additional functionality offered by Stripes. For example the beanclass attribute on the Stripes form tag allows you to specify the ActionBean class to target instead of having to specify the URL, and the format* attributes on the input tags allow you to specify how values should be formatted when written to the page.

Lastly, as Freddy discusses, there are several "helper" tags that have no equivalent in HTML that help produce things like lists of options from collections and enumerations.


Message Details

Name	Messages
Inbox ←	25
Sent	0
Reference	0
Trash	0

From:	Accounts-dept
To:	freddy@stripesbook.org
Cc:	
Bcc:	
Date:	Apr 17, 2008 4:49:00 AM
Subject:	Important info

Message text goes here.

Figure 1.4: THE MESSAGE DETAILS PAGE.

```
<div id="action">
  <s:submit name="delete" value="Delete"/>
  Move to folder:
  <s:select name="selectedFolder">
    <s:option value="">Select a folder...</s:option>
    <s:options-collection collection="${folders.folders}"
      value="id" label="name"/>
  </s:select>
  <s:submit name="moveToFolder" value="Move"/>
  <s:errors field="selectedFolder"/>
</div>
```

This code will be *included* in message_list.jsp and message_details.jsp with the <%@include%> directive. Unlike <jsp:include/>, <%@include%> does not execute a request to the target; rather, it pulls the source code into the JSP at the location of the directive:

email_19/web/WEB-INF/jsp/message_list.jsp

```
<s:form beanclass="stripesbook.action.MessageListActionBean">
  <d:table ...>
    <!-- ... -->
    <d:column>
      <s:checkbox name="selectedMessages" value="${message.id}"/>
    </d:column>
    <!-- ... -->
  </d:table>
  <c:if test="${not empty folder.messages}">
    <div><s:errors field="selectedMessages"/></div>
```

```
►     <%@include file="/WEB-INF/jsp/common/message_action.jsp"%>
    </c:if>
  </s:form>
```

email_19/web/WEB-INF/jsp/message_details.jsp

```
<s:form beanclass="stripesbook.action.MessageListActionBean">
► <%@include file="/WEB-INF/jsp/common/message_action.jsp"%>
  <div>
    <s:hidden name="selectedMessages"
      value="${actionBean.message.id}"/>
  </div>
</s:form>
```

Notice that in both cases, the included code becomes nested within a <*s:form*> tag. Indeed, the code from message_action.jsp is not valid on its own, because it contains form input controls with no parent <*s:form*> tag.

For the *Message List* page, the controls are included only if the folder is not empty; it doesn't make sense to have controls for deleting or moving messages when no messages are being displayed in the page!

For the *Message Details* page, the user does not have to select messages because the controls apply to the currently displayed message. But we *do* need a parameter to indicate this message; the <*s:hidden*> tag takes care of that.

The great thing about doing this, besides reusing the message_action.jsp code in two different contexts, is that we don't even need to change anything in MessageListActionBean. Both forms submit the selectedMessages parameter to a property of type List<Message>. The message details page happens to submit only one value. The event handlers for deleting and moving messages work the same either way: they iterate over the list of selected messages, as you can see in the complete source for MessageListActionBean:

email_19/src/stripesbook/action/MessageListActionBean.java

```
package stripesbook.action;
public class MessageListActionBean extends BaseActionBean {
    private static final String LIST="/WEB-INF/jsp/message_list.jsp";

    @DefaultHandler
    public Resolution list() {
        return new ForwardResolution(LIST);
    }
►   public Resolution delete() {
        for (Message message : selectedMessages) {
            folderDao.deleteMessage(message);
        }
        return new RedirectResolution(getClass());
    }
►   public Resolution moveToFolder() {
        for (Message message : selectedMessages) {
            folderDao.addMessage(message, selectedFolder);
        }
        return new RedirectResolution(getClass());
    }
    @Validate(required=true, on={"delete", "moveToFolder"})
    private List<Message> selectedMessages;
    public List<Message> getSelectedMessages() {
        return selectedMessages;
    }
    public void setSelectedMessages(List<Message> selectedMessages) {
        this.selectedMessages = selectedMessages;
    }
    @Validate(required=true, on="moveToFolder")
    private Folder selectedFolder;
    public Folder getSelectedFolder() {
        return selectedFolder;
    }
    public void setSelectedFolder(Folder selectedFolder) {
        this.selectedFolder = selectedFolder;
    }
    private FolderDao folderDao = MockFolderDao.getInstance();
}
```

1.5 Radio buttons

To wrap up our discussion of form input controls, let's use radio buttons to add a feature to the *Contact Form* page: entering the contact's gender, as shown in Figure 1.5, on the following page.

We'll need a Gender property in the Contact class:

email_19/src/stripesbook/model/Gender.java

```
package stripesbook.model;
public enum Gender {
    Female,
    Male
}
```

email_19/src/stripesbook/model/Contact.java

```
package stripesbook.model;
public class Contact extends ModelBase {
    /* ... */
    private Gender gender;
    /* ... */
```

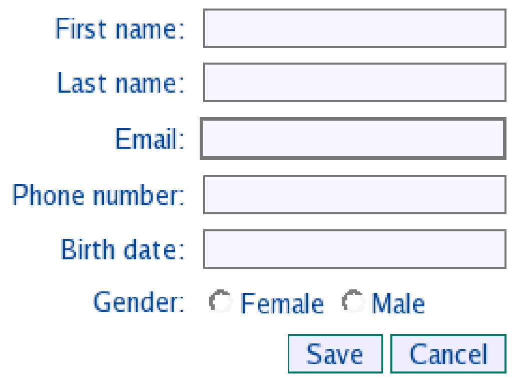


Figure 1.5: USING RADIO BUTTONS FOR THE CONTACT'S GENDER.

```
    public Gender getGender() {
        return gender;
    }
    public void setGender(Gender gender) {
        this.gender = gender;
    }
}
```

The value that the user enters for the gender must be either Female or Male, case sensitive. It's much easier and less error-prone for the user to choose a radio button than have to type in those values in a text field. Also, radio buttons allow only one selection, making them appropriate for the gender property.

The <*s:radio*> tag creates a radio button. Its name= attribute serves an additional purpose besides containing the name of the Action Bean property: it also *groups* together buttons that have the same name. Only one radio button from a group can be selected at a time.

Since we added the gender property to the Contact class, the name= of each radio button will be contact.gender. We just need the ContactFormActionBean to supply the possible values for the gender:

email_19/src/stripesbook/action/ContactFormActionBean.java

```
public Gender[] getGenders() {
    return Gender.values();
}
```

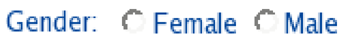
This makes it easy to create the radio buttons for the gender in contact_form.jsp:

email_19/web/WEB-INF/jsp/contact_form.jsp

```
<c:forEach var="gender" items="${actionBean.genders}">
  <s:radio name="contact.gender" value="${gender}"/>${gender}
</c:forEach>
```

In the value= attribute is the actual value to submit as input to the Action Bean property if the radio button is selected. This is the same as if the user had typed in that value in a text field. The value can be different from the label that is shown next to the radio button; in fact, notice that the label is not part of the *<s:radio>* tag at all. You can display the label wherever you want.

The radio buttons for the gender now appear in the contact form as shown below.



What's Next?

We learned about the different types of form input controls and how they work with Stripes tags and Action Bean properties. Along the way, we got quite a lot done in the webmail application. In the next chapter, we'll finish implementing the features of the *Message List*, *Message Details* and *Message Compose* pages.

Index

First page of blurb