Extracted from:

Stripes

. . . and Java Web Development Is Fun Again

Joe Asks. . .

What If I Don't Like How Messages Are Displayed?

By default, information messages are displayed in a plain unordered list (<ul> and <li> tags). We'll see how to customize this format in Chapter 6, *Customizing Stripes Messages*, on page 123.

the contact list. Let's see about creating new contacts and updating existing contacts with the Contact Form page.

3.6 Creating Forms

Forms are a breeze to create in Stripes. There is a Stripes tag for every type of input field (text field, radio button, and so on) and for submit buttons. Using these tags instead of plain HTML gives you extra features such as repopulating the inputs, highlighting them when they are in error, and supporting localization.

When the user submits a form, Stripes binds the values in the form fields to the corresponding properties in the action bean and triggers the event handler associated with the submit button. You can have multiple submit buttons without having to do anything special to figure out which button the user clicked: each button triggers its own event handler on the action bean.

Input fields have to be associated to properties of an action bean, but you don't have to copy the properties of a model object to the action bean. Instead, you put the model object directly in the action bean and use *nested* properties.

For example, you can add a Contact property in ContactListActionBean and create a text field associated with the contact's first name with <s:text name="contact.firstName"/>. To set the value, Stripes calls getContact().setFirstName() on the action bean. You don't even have to worry about a NullPointerException. If getContact() returns **null**, Stripes creates a new Contact object for you. This saves you a great deal of code because you don't have to copy each model property in the action bean and transfer information back and forth. If your model objects use other

Figure 3.10: The contact form

model objects, that's no problem either—Stripes happily uses deeply nested properties, such as "contact.address.street.name". Let's put all this to work and build a form for contacts.

Creating a Blank Form

The <*s:form*> tag creates a form associated with the action bean indicated in its beanclass= attribute. Within the tag, we add input fields with tags such as <*s:text*>, <*s:radio*>, and every other type of input. These tags all have a name= attribute in which we put the name of the action bean property that receives the user's input. To complete the form, we add one or more submit buttons with the <*s:submit*> tag and the name= of the event handler associated with the button.

Have a look at the following code. This creates the form shown in Figure 3.10:

Download email_05/web/WEB-INF/jsp/contact_form.jsp

```
❶ <s:form beanclass="stripesbook.action.ContactFormActionBean">
    <table class="form">
      <tr>
        <td>Email:</td>
❷       <td><s:text name="contact.email"/></td>
      </tr>
      <tr>
        <td>First name:</td>
        <td><s:text name="contact.firstName"/></td>
      </tr>
```

```
      <tr>
        <td>Last name:</td>
        <td><s:text name="contact.lastName"/></td>
      </tr>
      <tr>
        <td>Phone number:</td>
        <td><s:text name="contact.phoneNumber"/></td>
      </tr>
      <tr>
        <td>Birth date:</td>
        <td><s:text name="contact.birthDate"/></td>
      </tr>
      <tr>
        <td> </td>
        <td>
❸         <s:submit name="save" value="Save"/>
          <s:submit name="cancel" value="Cancel"/>
        </td>
      </tr>
    </table>
  </s:form>
```

At ❶, we're creating a form associated with the ContactFormActionBean class, which we'll be writing shortly. Starting at ❷, the text input fields for the contact's information are created with the *<s:text>* tag and name= attributes for the properties of the Contact class. The submit buttons (❸) call either save() or cancel() on the action bean according to which one the user clicked. The value= attribute is the button's label.

Notice how there is a very clean and clear relationship between the JSP and the action bean. The action bean's class name is indicated in the form tag's beanclass= attribute, each input's name= corresponds to an action bean property, and each submit button's name= is an action bean's event handler.

Let's create the ContactFormActionBean to handle the form submission. We'll need the following:

- A default event handler that forwards to contact_form.jsp
- The save() and cancel() event handlers
- The contactId and contact properties
- The ContactDao to save the contact

Looking at those last two points, you'll realize that the ContactListActionBean class already has the contact properties and DAO. You probably don't like copying and pasting code any more than I do, so let's do a little refactoring.

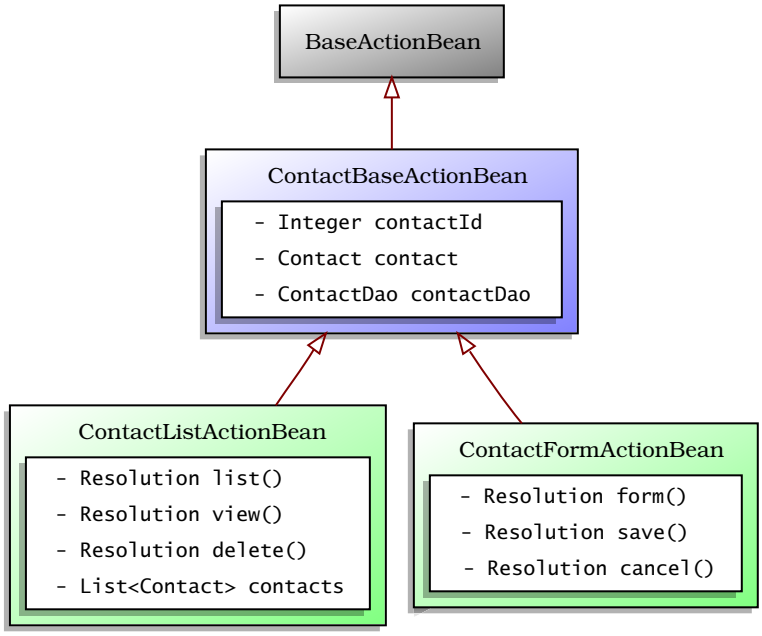

Figure 3.11: Action bean class diagram

Check out Figure 3.11. We'll create the ContactBaseActionBean class and put the common code in there. Then, ContactListActionBean and ContactFormActionBean can inherit from it.

Here is the ContactBaseActionBean class:

Download email_05/src/stripesbook/action/ContactBaseActionBean.java

```
package stripesbook.action;
public abstract class ContactBaseActionBean extends BaseActionBean {
    private ContactDao contactDao = MockContactDao.getInstance();
    protected ContactDao getContactDao() {
        return contactDao;
    }
    private Integer contactId;
    public Integer getContactId() {
        return contactId;
    }
    public void setContactId(Integer id) {
        contactId = id;
    }
    private Contact contact;
```

```
    public Contact getContact() {
        if (contactId != null) {
            return contactDao.read(contactId);
        }
        return contact;
    }
    public void setContact(Contact contact) {
        this.contact = contact;
    }
}
```

The code in the ContactFormActionBean class is now lean and mean:

Download email_05/src/stripesbook/action/ContactFormActionBean.java

```
package stripesbook.action;
public class ContactFormActionBean extends ContactBaseActionBean {
    private static final String FORM="/WEB-INF/jsp/contact_form.jsp";

    @DefaultHandler
❶   public Resolution form() {
        return new ForwardResolution(FORM);
    }
❷   public Resolution save() {
        Contact contact = getContact();
        getContactDao().save(contact);
        getContext().getMessages().add(
            new SimpleMessage("{0} has been saved.", contact)
        );
        return new RedirectResolution(ContactListActionBean.class);
    }
❸   public Resolution cancel() {
        getContext().getMessages().add(
            new SimpleMessage("Action cancelled.")
        );
        return new RedirectResolution(ContactListActionBean.class);
    }
}
```

The default event handler at ❶ forwards to contact_form.jsp. When the user clicks the Save button, save() is called (❷) and uses the DAO to save the contact. It then adds an information message to the list and redirects to ContactListActionBean, which displays the messages and the table of contacts. The event handler for the Cancel button (❸) just adds an information message and redirects to the contact list without saving the contact.

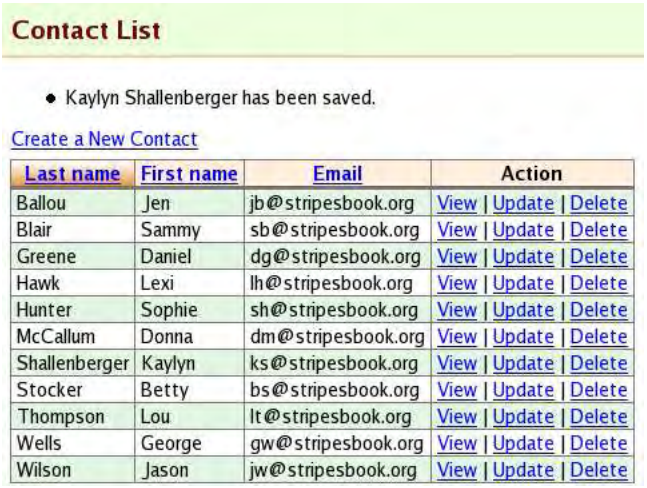

Figure 3.12: After creating a contact

To send the user from the contact list to the form, add a Create a New Contact link in contact_list.jsp:

Download email_05/web/WEB-INF/jsp/contact_list.jsp

```
<s:link beanclass="stripesbook.action.ContactFormActionBean">
  Create a New Contact
</s:link>
```

The result of using the form to create a new contact fictitiously named Kaylyn Shallenberger is shown in Figure 3.12.

There's only one more thing we need to do: add the Update links in the Action column.

Updating Information with a Prepopulated Form

Clicking the Update link should open the contact form prepopulated with the selected contact's information, as in Figure 3.13, on the next page. First, create the link with the selected contact's ID as a parameter:

Download email_05/web/WEB-INF/jsp/contact_list.jsp

```
<s:link beanclass="stripesbook.action.ContactFormActionBean">
  <s:param name="contactId" value="${contact.id}"/>
  Update
</s:link>
```

Figure 3.13: Prepopulated form

Remember that the getContact() method in ContactBaseActionBean already retrieves the selected contact if the contact ID parameter was provided:

Download email_05/src/stripesbook/action/ContactBaseActionBean.java

```
public Contact getContact() {
    if (contactId != null) {
        return contactDao.read(contactId);
    }
    return contact;
}
```

The nice thing with the Stripes input tags is that they also *read* from the property in the name= attribute. So by making the selected contact available through getContact(), the inputs prepopulate themselves with the contact information such as *"contact.firstName"*, *"contact.lastName"*, and so on.

Just like that, we're almost there. To get the form to work for updating an existing contact, we need to resubmit the contact ID parameter that was sent with the Update link.

A hidden input does the trick:

Download email_05/web/WEB-INF/jsp/contact_form.jsp

```
    <s:form beanclass="stripesbook.action.ContactFormActionBean">
      <div><s:hidden name="contact.id"/></div>
      <table class="form">
```

The input obtains its value just like the other inputs and becomes a parameter when the form is submitted. It took very little code to add the

How Tags and Attributes Invoke Action Beans

We've used several tags and attributes to invoke methods on action beans. Here's a summary of what we've seen so far:

Tag and Attribute	Invocation on Action Bean
<s:link beanclass="*pkg.Name*">	pkg.Name's default event handler
<s:link event="*eventName*">	public Resolution eventName()
<s:link href="*URL*">	Action bean bound to *URL*
<s:param name="*property*">	setProperty(value)
<s:form beanclass="*pkg.Name*">	pkg.Name's default event handler
<s:form action="*URL*">	Action bean bound to *URL*
<s:hidden name="*property*">	setProperty(value)
<s:text name="*property*">	setProperty(value)
<s:submit name="*eventName*">	public Resolution eventName()

Update link and get inputs that autopopulate themselves, and before we know it, the contact form is complete.

3.7 Use a Forward or a Redirect?

After creating, updating, or deleting a contact, we're returning a RedirectResolution to ContactListActionBean instead of a ForwardResolution to contact_list.jsp. Why? Let's discuss the difference between the two resolutions and how to decide which one to use.

The Redirect-After-Side-Effect Pattern

The first thing to notice is the create, update, and delete operations all have *side effects*—they change the state of the data on the server.

Suppose that we returned a ForwardResolution to a contact_list.jsp after the user has deleted a contact. Looking at Figure 3.14, on the following page, we see that the last request is "delete this contact." The problem is that if the user clicks the browser's Reload button, the "delete this contact" request will be sent *again*, causing an error because the contact has already been deleted.

In general, it is a bad idea to use a forward after any request that should not be resubmitted by hitting Reload. Imagine a request that makes a purchase with the user's credit card. You wouldn't want to repeatedly charge the credit card!

The Pragmatic Bookshelf

The Pragmatic Bookshelf features books written by developers for developers. The titles continue the well-known Pragmatic Programmer style and continue to garner awards and rave reviews. As development gets more and more difficult, the Pragmatic Programmers will be there with more titles and products to help you stay on top of your game.

Visit Us Online

Stripes...and Java Web Development Is Fun Again's Home Page
http://pragprog.com/titles/fdstr
Source code from this book, errata, and other resources. Come give us feedback, too!

Register for Updates
http://pragprog.com/updates
Be notified when updates and new books become available.

Join the Community
http://pragprog.com/community
Read our weblogs, join our online discussions, participate in our mailing list, interact with our wiki, and benefit from the experience of other Pragmatic Programmers.

New and Noteworthy
http://pragprog.com/news
Check out the latest pragmatic developments in the news.

Buy the Book

If you liked this PDF, perhaps you'd like to have a paper copy of the book. It's available for purchase at our store: pragprog.com/titles/fdstr.

Contact Us

Phone Orders:	1-800-699-PROG (+1 919 847 3884)
Online Orders:	www.pragprog.com/catalog
Customer Service:	orders@pragprog.com
Non-English Versions:	translations@pragprog.com
Pragmatic Teaching:	academic@pragprog.com
Author Proposals:	proposals@pragprog.com