Official Guide

SAMPLE CHAPTER

Liferay IN ACTION

Richard Sezov, Jr.

FOREWORD BY Brian Kim

MANNING

Liferay in Action

by Rich Sezov, Jr.

Chapter 4

Copyright 2012 Manning Publications

brief contents

PART 1 WORKING WITH LIFERAY AND PORTLETS 1

1 ■ The Liferay difference 3
2 ■ Getting started with the Liferay development platform 30

PART 2 WRITING APPLICATIONS ON LIFERAY'S PLATFORM........ 63

3 ■ A data-driven portlet made easy 65
4 ■ MVC the Liferay way 91
5 ■ Designing your site with themes and layout templates 128
6 ■ Making your site social 154
7 ■ Enabling user collaboration 176

PART 3 CUSTOMIZING LIFERAY.. 209

8 ■ Hooks 211
9 ■ Extending Liferay effectively 241
10 ■ A tour of Liferay APIs 263

4 MVC the Liferay way

This chapter covers

- The MVC design pattern
- Liferay's implementation of MVC: `MVCPortlet`
- Control panel portlets
- Liferay JSP patterns and objects, including `ThemeDisplay`
- Tag libraries such as the search container and AlloyUI Taglibs
- Internationalization in Liferay
- Liferay permissions

I hope you didn't have a visceral reaction to the first three letters in the chapter title. If you did, I certainly can understand why. MVC (which stands for Model-View-Controller) is probably one of the most overused buzzwords (if you can call an abbreviation a buzzword) you'll see. Framework after framework has been released, all claiming to implement MVC in one way or another. At the time of this writing, the Wikipedia article on MVC lists a total of 17 MVC frameworks for Java alone. They seem to keep multiplying like some kind of virus.

With that in mind, what in the world is Liferay thinking by having its *own* MVC framework? The answer to this question becomes apparent when you see the framework. Many of the MVC frameworks that are available can be heavy, with somewhat of a learning curve. They have configuration files that point to various parts of the application, and these files need to be kept in sync with the Java code they point to. Liferay's MVC doesn't have any of that. It's much simpler to use than the other frameworks: there's no configuration file like struts-config.xml or faces-config.xml to worry about. And it's a simple extension of the `GenericPortlet` class you've already seen. For that reason, I like to use it. But if you're still balking at the idea of another MVC framework, try thinking of it this way: you're getting into Liferay now, so you might as well become familiar with `MVCPortlet`, because if you need to look at any of Liferay's portlets, you'll see it anyway. And you may find that you like it once you start using it.

You'll continue working with the Product Registration portlet that you started in the last chapter. In that chapter, you used Service Builder to create the database persistence layer, or service layer, of your application. Now that the all foundational code is finished, you can concentrate on the layers of your application that interact with your users. These layers form the Model, the View, and the Controller.

4.1 Using Model-View-Controller

By using this pattern, you separate your concerns into various layers of the application, just as you did with Data Transfer Objects (DTOs) and Data Access Objects (DAOs) in your service layer. If you've been developing Java-based web applications for a while, you're probably familiar with some of the MVC frameworks that are available. The same concepts apply here, but as you'll see, they're implemented in a way that is a bit easier to use. Let's look at the various components of the MVC design pattern and see how they're implemented.

- *Model*—The model layer of the application holds the data of the application and contains any business rules for manipulating that data. Your `PRProduct` object with its fields containing values for the name and serial number is part of your model and was generated by Service Builder. Any logic that would change those values based on certain rules would also be part of the model layer.
- *View*—The view layer of the application contains all the logic for displaying the data to the user. Handling fields, check boxes, and other form elements, as well as hiding or showing data, are functions provided by the view layer. You'll create JSPs that will handle the view layer, and you'll see some tools that Liferay provides that makes this easy.
- *Controller*—The controller layer acts as a traffic director. It passes data back and forth to and from the model and view layers, providing a separation of concerns. The controller, for example, might be responsible for determining which action a user has clicked and directing processing to the proper function to update the model. Generally, the model and view speak only to the controller, with the exception that the view may use objects from the model for display purposes (such as iterating over a List to populate a table).

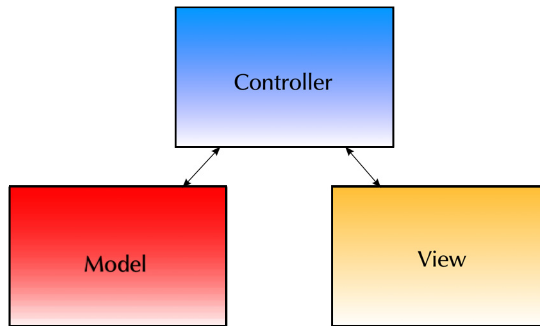


Figure 4.1 The MVC design pattern provides an easy way to structure your web application by separating concerns into easily digestible components.

These three layers are pictured in figure 4.1.

How would you implement this design pattern in a portlet? The model layer is generated for you by Service Builder from your table design. I stated earlier that your JSPs will be your view layer. That leaves the portlet class as your controller. As you'll see, this design will provide a good implementation of the MVC design pattern, allowing you to separate your concerns in a way that is maintainable and straightforward.

Liferay has subclassed `GenericPortlet` to provide functionality that makes it easy to create MVC applications using portlets. This subclass is called `MVCPortlet`, and it enhances the way portlets handle actions and page management. All of Liferay's portlet plugins are based on `MVCPortlet` instead of `GenericPortlet`, because it's a much better option.

What's different? Well, `MVCPortlet` is a lot easier to use. You won't have to worry about page management anymore; you get it for free. Liferay's `MVCPortlet` provides a simple means for page management. If you want to determine what JSP to display, all you need to do is point a render parameter called `jspPage` to the location of the JSP, and that's the page that will be displayed to the user. This functionality does several things for you:

- You don't have to worry about `doView()`, `doEdit()`, or any other `do` method.
- Your portlet class doesn't need to implement any portlet-specific APIs.
- Your portlet class is simple: all it contains is action methods.
- If you wish, you can use any piece of the standard portlet API that you want: `MVCPortlet` is subclassed from `GenericPortlet`, so feel free to override any functionality for your own purposes.

One of the other things this project will have is a portlet in Liferay's Control Panel as an alternative to Edit mode for administrative functions.

4.1.1 Edit mode? What Edit mode?

One of the things Liferay has learned with regard to portlets is that some things that are part of the portlet standard don't necessarily work well in real-world use. Portlet modes are one example. Sometimes the inventors of a thing envision a particular use for that thing, but when it gets in the hands of the general populace, uses are found

that the inventors never dreamed of. Consider the case of Lawn Chair Larry. Larry always dreamed of flying, but poor eyesight kept him from serving in the United States Air Force. Undaunted, he purchased 45 weather balloons from an Army-Navy surplus store, filled them with helium, and tied them to his lawn chair. Thinking he'd float leisurely to a height of 30 feet or so, he had some friends cut the cord that kept him tied down. Instead of floating slowly to 30 feet, he shot to a height of 16,000 feet, scared himself half to death, interfered with flight traffic into and out of LAX, knocked out the power to a Long Beach neighborhood for 20 minutes, and got himself arrested.[1] The point? I'm sure neither the inventor of the weather balloon nor the inventor of the lawn chair ever envisioned someone using them for this purpose.

Similarly (and I say that with tongue firmly planted in cheek), the inventors of the portlet envisioned a single use for portlets: a web desktop environment for large enterprises. The design doesn't provide enough flexibility for the wild, wild west of custom designed web sites whose designers don't want to be locked into the boxy, window-like interface that was the original design for a portlet. Minimize, maximize, and close buttons are for desktop operating systems, not the web. And portlet modes haven't been used very much. I have yet to see someone implement Help mode in a portlet. Why? Because there are much better ways to implement a help system in an application than by using Help mode.

Edit mode can be confusing from an end user point of view. It's not intuitive to click an icon in the portlet window title bar to enter another mode from which you can control various portlet settings. And of course, if you use Edit mode, you're locked into that boxy design for your web site, because you have to provide a title bar so the icon can reside there.

Liferay has the unique position of being used for internet-based web sites as often as it's used internally for large enterprises. For that reason, a slightly different (and I think better) paradigm is used: the Control Panel.

Instead of providing an Edit mode for your portlet where you can control settings, you can implement a separate portlet for those settings and embed it in Liferay's Control Panel. Doing so puts all settings and options in one place, rather than having them scattered around your web site. It also frees your site designers to come up with whatever they want, because they're not bound to the window paradigm espoused by the portlet standard. Rather than use Edit mode to control the settings for your Product Registration portlet, you'll create a separate portlet in the same application and embed it in the Control Panel.

There are benefits to this approach for the developer too. Rather than having one big portlet that does everything, you can implement your application as several smaller, more tightly focused portlets. This not only makes the code easier to follow but also gives your end users freedom to arrange the application on the page any way they like.

[1] http://darwinawards.com/stupid/stupid1998-11.html

Figure 4.2 The Product Admin portlet allows portal administrators to add products that the Product Registration portlet displays in a selection box. This enables end users to choose from a list which product they wish to register, to enable their one-year warranty protection.

Your first task, therefore, will be to create the administrative portlet for the Control Panel, shown in figure 4.2.

Before you get started, let's take a quick look at how Liferay does MVC. You'll see that it's a welcome simplification of many of the concepts you've no doubt encountered before.

4.1.2 MVC according to Liferay

When you first create a portlet in the Plugins SDK, you've already seen that no portlet class is generated—yet if you deploy the portlet, you get the equivalent of Hello World functionality. First question: how does the portlet do that?

If you look at the portlet.xml that is generated with your project, you'll see that the portlet class is defined as `com.liferay.util.bridges.mvc.MVCPortlet`. This class is based on the `GenericPortlet` class you've already been working with, but it contains enhancements. `MVCPortlet` can do all page management for you. It does so by providing a default view, which is defined as an `init` parameter for each portlet mode. Because portlets default to View mode—and aren't required to implement any other mode—this portlet automatically directs to a JSP file called view.jsp, because

that's what's defined in the `init` parameter. You can see this if you look at the portlet.xml file:

```
<init-param>
   <name>view-jsp</name>
   <value>/view.jsp</value>
</init-param>
```

By generating the project, you have a portlet that works (although it doesn't do much), but it doesn't have a portlet class. To do anything interesting, you need to create a portlet class, as you've done in the previous chapters.

What you'll do now is extend `MVCPortlet` instead of `GenericPortlet` so you can take advantage of its page-management features and thus make your portlet smaller and easier to work with. When you're finished, you'll find that your portlet contains nothing but action methods. All page management is controlled through the mechanisms that `MVCPortlet` gives you.

An alternative pattern

Some developers like to use the default implementation of `MVCPortlet` and implement their applications completely with JSPs. Although this obviously goes against the MVC design pattern, it's a way of doing development that will be more familiar to those who cut their teeth on scripting languages for the web, such as PHP. None of the examples in this book follow this pattern, but you'll be able to glean enough information from the examples to do it this way if you wish.

The first thing you need to do is configure some deployment descriptors to reflect the two portlets you'll have.

4.2 Configuring the portlet project

To make your life easier, you'll set things up in a way that lets you reuse some configuration going forward. You haven't had to worry much about this yet, because you've only concentrated on the persistence layer so far. But when you first created the project, the Plugins SDK generated a project containing a single portlet with the same name as the project, which you called `product-registration`. This is a good guess on the Plugins SDK's part, but it isn't exactly what you want. Because you'll be having a Product Registration portlet, you'll keep that configuration, and you'll add a new portlet for the Product Admin portlet. You can do this by modifying the deployment descriptors for your project and adding a section for the other portlet you'll need. The nice thing about this is that you can have two portlets (or more) in one project, which enables you to share common code easily.

Because you're sharing common code in Java, you may as well do the same for your JSP files. I'll also show you a pattern Liferay uses in JSP files to share common imports and objects on the page.

4.2.1 Defining portlets in your deployment descriptors

Open the portlet.xml file, and add the following portlet configuration below the portlet that is already there.

Listing 4.1 Adding the Product Admin portlet

```
<portlet>
  <portlet-name>product-admin</portlet-name>
  <display-name>Product Administration</display-name>
  <portlet-class>
    com.inkwell.internet.productregistration.registration.
    ➥ portlet.ProductAdminPortlet
  </portlet-class>
  <init-param>
    <name>view-jsp</name>
    <value>/admin/view.jsp</value>          <-- (1) Moves JSP to admin folder
  </init-param>
  <init-param>
    <name>add-process-action-success-action</name>   <-- (2) Disables Liferay status messages
    <value>false</value>
  </init-param>
  <expiration-cache>0</expiration-cache>
  <supports>
    <mime-type>text/html</mime-type>
  </supports>
  <resource-bundle>content.Language</resource-bundle>   <-- (3) Create multilingual portlet
  <portlet-info>
    <title>Product Administration</title>
    <short-title>Product Administration</short-title>
    <keywords>Product Administration</keywords>
  </portlet-info>
  <security-role-ref>
    <role-name>administrator</role-name>
  </security-role-ref>
  <security-role-ref>
    <role-name>guest</role-name>
  </security-role-ref>
  <security-role-ref>
    <role-name>power-user</role-name>
  </security-role-ref>
  <security-role-ref>
    <role-name>user</role-name>
  </security-role-ref>
</portlet>
```

docroot/WEB-INF/portlet.xml

When you've added this portlet, go ahead and create the admin folder under the docroot folder of the project, because (1) tells `MVCPortlet` that your JSP for View mode is in this folder. If you've used Liferay before, you've probably noticed that when you save anything in the built-in portlets, a green status message is displayed, saying that it was saved successfully. `MVCPortlet` by default does this automatically after every portlet action. Because you don't want it to do this every time, you can use the initialization

parameter ❷ to turn off that functionality. You'll also be creating a multilingual portlet, so you need a language bundle ❸.

The next file you need to edit is liferay-portlet.xml. As you'll remember, this file is the Liferay-specific deployment descriptor. You need some custom settings for the administration portlet, as follows.

Listing 4.2 Configuring the admin portlet for the Control Panel

```
<portlet>
  <portlet-name>product-admin</portlet-name>
  <icon>/icon.png</icon>
  <control-panel-entry-category>content</control-panel-entry-category>
  <control-panel-entry-weight>1.5</control-panel-entry-weight>
  <header-portlet-css>/css/product-admin.css</header-portlet-css>
  <header-portlet-javascript>/js/test.js</header-portlet-javascript>
</portlet>
```

docroot/WEB-INF/liferay-portlet.xml

To put your portlet in the Control Panel, you need only tell Liferay you want to do that via its deployment descriptor. You add the portlet to the Content area of the control panel and specify a weight that determines where it appears in the list.

The Control Panel is divided into four areas, each of which has a particular purpose:

- *Personal*—Used for administration items that the logged-in user needs to access, such as My Account.
- *Content*—Used to administer any type of content in the portal. In this chapter you'll define Inkwell's products as content, so you'll add the portlet to this section.
- *Portal*—Contains administrative tools that affect the portal globally.
- *Server*—Contains administrative tools that affect the entire Liferay installation.

Your configuration places the portlet in the *content* section of the Control panel.

The Control Panel looks a bit different from the rest of the portal, but now you know it's populated entirely with portlets! You already have the basic knowledge for adding anything you want to the Control Panel.

Defining the weight this way makes sense when you understand that the weights for the default items range from 1.0 to 11.0. By making the weight for your portlet 1.5, you're making sure it appears second in the list.

Last, you have one final configuration file: liferay-display.xml. This is how you configure the way the portlets appear in the Add > More menu:

```
<?xml version="1.0"?>
<!DOCTYPE display PUBLIC "-//Liferay//DTD Display 6.0.0//EN"
➥ "http://www.liferay.com/dtd/liferay-display_6_0_0.dtd">

<display>
  <category name="Inkwell Internet">
    <portlet id="product-registration" />
  </category>
```

```
  <category name="category.hidden">
    <portlet id="product-admin" />
  </category>

</display>
```

As you can see, you're creating a category to hold Inkwell's portlets destined for its internet site. You also have another category, `category.hidden`. Any portlets in this category are prevented from appearing in the Add > More menu, so you put the administrative portlet there. Because it appears in the Control Panel, you don't want users to also be able to place it on pages.

You've now finished configuring your portlet project and can move on to implementing it.

4.2.2 Having one location for JSP dependencies

If you've ever looked into Liferay's source code, you've probably seen that it doesn't follow a pattern that many organizations have tried to get to over the years. That pattern is the artificial separation between site designers and site programmers. The vision for this is simple: site designers understand tags and styling; programmers understand code. Therefore, JSPs should be as free of code as possible so as not to confuse the site designers. Many products have been architected to support this idea.

Liferay goes in a different direction and mixes code with tags. I know some people will have a visceral reaction to this. My goal isn't necessarily to evangelize one way over the other (that is up to individual developers, who should do things the way they're most comfortable doing them) but instead to let you know before you get into the code that I'm well aware that I'm not doing things the way many others do them. My goal is to show you the Liferay way of doing things, because that will enable you to understand Liferay's code better—and you may decide you like it. If you don't, you're always free to use another framework such as Struts or ICEfaces for your portlets. Although I was a bit puzzled by Liferay's code at first, I eventually came on board with Liferay's philosophy. If you think about your experience, rather than what all the articles will tell you, this division between designers and programmers doesn't exist in most cases. And if it does, the designers don't understand *any* code, including markup and CSS. They tend to deliver graphics files that front-end developers chop up and turn into pages.

With that said, because there will be some Java code in your JSPs, you'll need to manage the imported classes and tag libraries. A pattern that Liferay uses to make this easier is to throw all imports, tag library declarations, and variable initializations in one file called init.jsp. Every other JSP that is created imports init.jsp so it can take advantage of those declarations (see figure 4.3).

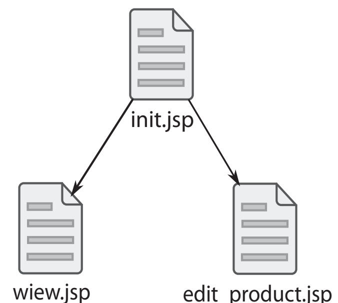


Figure 4.3 Every JSP file in the project includes init.jsp so that all initialization and configuration can be done in one easy-to-maintain file.

The completed init.jsp appears in listing 4.3. Obviously, if you were writing this portlet from scratch, you'd add imports and initialization code as needed. I have the advantage of being able to give you the completed file for the whole project, with everything you'll use already in it.

Listing 4.3 Putting initialization code in one place

```
<%@ taglib
➥ uri="http://java.sun.com/portlet_2_0" prefix="portlet" %>     Tag library declarations
<%@ taglib
➥ uri="http://java.sun.com/jstl/core_rt" prefix="c" %>
<%@ taglib
uri="http://liferay.com/tld/aui" prefix="aui" %>
➥ <%@ taglib
uri="http://liferay.com/tld/portlet" prefix="liferay-portlet" %>
<%@ taglib
➥ uri="http://liferay.com/tld/security"
➥ prefix="liferay-security" %>
<%@ taglib
➥ uri="http://liferay.com/tld/theme" prefix="liferay-theme" %>
<%@ taglib
➥ uri="http://liferay.com/tld/ui" prefix="liferay-ui" %>
<%@ taglib
➥ uri="http://liferay.com/tld/util" prefix="liferay-util" %>

<%@ page import="java.util.List" %>     Import statements
<%@ page import="java.util.Calendar" %>
<%@ page import="java.util.Collections" %>
<%@ page import="com.liferay.portal.kernel.util.HtmlUtil" %>
<%@ page import="com.liferay.portal.kernel.util.ParamUtil" %>
<%@ page import="com.liferay.portal.kernel.util.CalendarFactoryUtil" %>
<%@ page import="com.liferay.portal.kernel.dao.search.ResultRow" %>
<%@ page import="com.liferay.portal.kernel.dao.search.SearchEntry" %>

<%@ page import="com.liferay.portal.kernel.exception.SystemException" %>
<%@ page import="com.liferay.portal.kernel.util.WebKeys" %>
<%@ page import="com.liferay.portal.security.permission.ActionKeys" %>
<%@ page import="com.liferay.portal.kernel.util.ListUtil" %>
<%@ page
➥ import="com.liferay.portal.service.permission.PortalPermissionUtil" %>
<%@ page
➥ import="com.liferay.portal.service.permission.PortletPermissionUtil"
➥ %>
<%@ page
➥ import="com.inkwell.internet.productregistration.model.PRProduct" %>
<%@ page
➥ import="com.inkwell.internet.productregistration.model.PRRegistration"
➥ %>
<%@ page import="com.inkwell.internet.productregistration.model.PRUser" %>
<%@ page import=
➥ "com.inkwell.internet.productregistration.registration.portlet.
ActionUtil" %>
<%@ page import="com.inkwell.internet.productregistration.service.
➥ PRProductLocalServiceUtil" %>
```

```
<%@ page import="
➥ com.inkwell.internet.productregistration.service.
➥ PRRegistrationLocalServiceUtil" %>
<%@ page import="javax.portlet.PortletURL" %>

<portlet:defineObjects />                    ◁ Initializes portlet taglibs

<liferay-theme:defineObjects />              ◁ Initializes Liferay taglibs
```

docroot/init.jsp

As you can see, this JSP is fairly simple: it declares the tag libraries you'll be using in this portlet application, imports the classes you'll need in your scriptlets, and then initializes any tag libraries that need initializing. What makes this nice is that because you'll be using all this stuff in the rest of your JSPs, you don't have to bother reimporting classes or reinitializing tag libraries in every JSP. You can add this at the top of any JSP you create:

```
<%@include file="/init.jsp" %>
```

The last tag in this file is Liferay-specific. This tag, like the Portlet API's similarly used tag, makes available to the page several variables you're likely to need. These variables are shown in table 4.1.

Table 4.1 Liferay objects in JSPs

Object	Description
`account`	The user's `Account` object. This object maps to the Account table in the Liferay database.
`colorScheme`	An object representing the current color scheme in the theme that is being rendered by the portal.
`company`	The current `Company` object. This represents the portal instance on which the user is currently navigating.
`contact`	The user's `Contact` object. This object maps to the Contact table in the Liferay database.
`layout`	The page to which the user has currently navigated.
`layoutTypePortlet`	This object can be used to programmatically add or remove portlets from a page.
`locale`	The current user's locale, as defined by Java.
`permissionChecker`	An object that can determine–given a particular resource–whether the current user has a particular permission for that resource.
`plid`	A portal layout ID. This is a unique identifier for any page that exists in the portal, across all portal instances.
`portletDisplay`	An object that gives the programmer access to many attributes of the current portlet, including the portlet name, the portlet mode, the ID of the column on the layout in which it resides, and more.

Table 4.1 Liferay objects in JSPs *(continued)*

Object	Description
`realUser`	When an administrator is impersonating a user, this variable tracks the administrator's `User` object.
`scopeGroupId`	By default, contains the groupId for the community or organization in which this portlet resides. If the `scopeable` attribute is set to true, this may contain a unique scope identifier for custom scopes, such as the page scope that was introduced in Liferay Portal 5.2, if the portlet has been configured to use a custom scope.
`theme`	An object representing the current theme that is being rendered by the portal.
`themeDisplay`	A runtime object that contains many useful items, such as the logged-in user, the layout, logo information, paths, and much more.
`timeZone`	The current user's time zone, as defined by Java.
`user`	The `User` object representing the current user.

Because you've added this initialization tag to init.jsp, these objects are available on all your pages.

You have one place where all initialization stuff is maintained, and all your JSPs can benefit from it. Now you're ready to add functionality, and the first thing you'll create is a form that lets users add products.

4.3 Creating a form with AlloyUI taglibs

As you've seen, `MVCPortlet` uses an initialization parameter to forward processing to a JSP for the user to view. Your first step will be to make this JSP display what you want to show to the user. If you refer back to figure 4.2, you can see that you want to show the user a form that allows them to quickly add products. On that form, you'll display a table of products that have already been added and that users can view, edit, or delete.

To create the form, you'll use another Liferay tool: AlloyUI tag libraries. The form you're creating is small, so you won't necessarily see the benefits of using AlloyUI taglibs just yet; but suffice to say you'll be glad you're getting an introduction, because it will make your life a lot easier later in the chapter.

But first, what is AlloyUI?

4.3.1 Getting started with AlloyUI tag libraries

AlloyUI is an *interface metaframework*. Okay, that probably sounded like gobbledegook, but bear with me for a second. Web site front ends are created using a combination of three technologies: HTML, CSS, and JavaScript (see figure 4.4), right? These three technologies together form the user experience for any site. HTML provides the overall structure of the document served up by the site, including its content. CSS provides the visual layer: how the document is presented visually to the user. It depends on a well-defined structure from the HTML in order to do this. JavaScript provides the

Figure 4.4 AlloyUI unifies three components, HTML, CSS, and JavaScript, in one easy-to-use metaframework.

interactive elements of any web page. If something moves, changes, or can be dragged, dropped, resized, or removed, JavaScript is enabling that for the end user.

AlloyUI was designed because user interface developers tend to have to solve the same kinds of problems over and over. Rather than continuing that cycle, you can use AlloyUI to solve common problems across the spectrum of HTML, CSS, and JavaScript. It combines the best of existing solutions under one consistent API, which makes it easier to use than trolling the internet to find a cookbook recipe for a common problem you know someone has already solved. Been there, done that, don't want to do it anymore.

One important way AlloyUI does this is by generating the proper markup for you. To do this, it has two types of components: a tag library and a JavaScript API. This chapter is concerned with the tag libraries, which are incredibly helpful for building web forms. In chapter 5, we'll look at some of AlloyUI's JavaScript functions.

Let's start gently with AlloyUI. You have a simple form that hardly needs it, but you'll use AlloyUI to build the form because it's a best practice when coding on Liferay's platform. This form is shown in the following listing.

Listing 4.4 An AlloyUI form

```
<portlet:actionURL name="addProduct" var="addProductURL"/>

<aui:form action="<%= addProductURL.toString() %>" method="post">

  <aui:fieldset>

    <aui:input name="productName" size="45" />

    <aui:input name="productSerial" size="45" />

    <aui:button-row>

      <aui:button type="submit" />

    </aui:button-row>
```

```
  </aui:fieldset>

</aui:form>
```

docroot/admin/view.jsp

Because this is a portlet, you have to let Liferay create the URL for submitting the form, which you can then use in the `action` attribute of the form declaration. You're using AlloyUI tags here, which you declared in init.jsp. The form is simple, with just the two fields you need to add products. You should recognize these fields from the Service Builder configuration you used in the last chapter to generate database tables and Java code for accessing them.

Liferay's `MVCPortlet` class enhances the `GenericPortlet` class in one important way: naming a method with the same name as the action URL's name causes the portlet to execute that method if that URL is clicked. To enable your form, all you have to do is create a method in your portlet class called `addProduct`, and that method will be executed. Let's look at that in the next listing.

Listing 4.5 Adding products

```
public void addProduct(ActionRequest request, ActionResponse response)
    throws Exception {

    ThemeDisplay themeDisplay =
        (ThemeDisplay) request.getAttribute(WebKeys.THEME_DISPLAY);
    PRProduct product = ActionUtil.productFromRequest(request);   <-- Convenience method
    ArrayList<String> errors = new ArrayList<String>();

    if (ProdRegValidator.validateProduct(product, errors)) {   <-- Validate input
        PRProductLocalServiceUtil.addProduct(
          product,themeDisplay.getUserId());                    <-- Call service layer
        SessionMessages.add(request, "product-saved-successfully");

    }
    else {
       SessionErrors.add(request, "fields-required");

    }

}
```

docroot/WEB-INF/src/com/inkwell/internet/productregistration/portlet/ProductAdminPortlet.java

I don't know about you, but I like to keep my methods short and sweet. Rather than having tons of logic embedded in a method, I'll offload some of it to another class—particularly if I can use it again somewhere else. You can see two examples in this code. First, you call a method in `ActionUtil` that can retrieve a `PRProduct` object out of the form the user submits. Because it's likely you'll want to pull `PRProduct`s from the request over and over, this method is placed in a class that can be shared by both portlets. Next, you call a method in `ProdRegValidator`. This class contains code that validates the input from the user-submitted form. Again, because you'll likely

reuse these validation routines, they're in a class that can be used by both portlets. We'll go over the validator later, because that class uses Liferay's `Validator` class to perform the actual validations.

Next, I want to highlight the use of the service layer you created in the previous chapter. If the values on the form pass validation, you need to save them to the database. You can do this easily, because all of your services are available via static methods in a `-LocalServiceUtil` class. In this case, you're using `PRProductLocalServiceUtil`, because you're saving `PRProduct` entities.

You now have basic functionality for saving product records from a web form to a database. Let's take a closer look at some features that provide an underlying infrastructure for implementing this functionality.

4.3.2 Providing feedback and messages

You probably noticed that some vital information was missing from the form in listing 4.4. Generally, when you have a field on a form, you also have a label for that field, as shown in figure 4.5.

Product Name

Figure 4.5 A field and a label. If you had just a field, users would have no idea what to type. This is common sense, but the code in listing 4.4 doesn't reflect the fact that you have a label. Or does it?

But your code looks like this:

```
<aui:input name="productName" size="45" />
```

Where is the text that prints the words *Product Name* for the field label? Back when you were configuring the portlet.xml file, you included a line of code for a resource bundle. To refresh your memory, here's the line:

```
<resource-bundle>content.Language</resource-bundle>
```

The text comes from there. This file allows you not only to put all your form messages in one place, but also to support multiple languages for your portlet application.

Create a folder called content in your src folder, and create a file called Language.properties there. The file for this project has the following contents.

Listing 4.6 Supporting multiple languages with Language.properties

```
Product=Product
com.inkwell.internet.productregistration.model.PRProduct=Product
model.resource.com.inkwell.internet.productregistration.model.PRProduct=
➥ Product

product-saved-successfully=Product Saved Successfully
productDeleted=The product has been deleted successfully.
product-name=Product Name
product-serial=Serial Number Mask
productUpdated=The product was updated successfully.
```

Product form messages

```
there-are-no-products=There are no products yet to display.

error-deleting=There has been an error deleting this product.     <-- Product form errors
error-updating=There has been an error updating this product.
fields-required=Please fill out all fields
product-name-required=Product Name is required
serial-number-prefix-required=Please enter the serial number prefix

add-registration=Register a New Inkwell Product     <-- Registration form fields
address1=Street Address 1
address2=Street Address 2
catalog=Catalog
city=City
country=Country
birth-date=Date of Birth
date-purchased=Date Purchased
email-address=Email Address
first-name=First Name
friend-family-member=Friend / Family Member
gender=Gender
gift=Gift
home-shopping=Home Shopping
how-hear=How did you hear about this Inkwell product?
inkwell.com=inkwell.com
last-name=Last Name
magazine-article=Magazine Article
online-retailer=Online Retailer
other-web-site=Other Web Site
other=Other
phone-number=Phone Number
please-choose=Please Choose
postal-code=Zip
product-serial-number=Product Serial Number
product-type=Product Type
radio-advertisement=Radio Advertisement
registration-saved-successfully=Registration Saved Successfully
retail-store=Retail Store
state=State
thank-you-message=Thank you for registering your product with us!
trade-show=Trade Show
tv-advertisement=TV Advertisement
tv-news=TV News
tv-shopping-network=TV Shopping Network
where-purchase=Where did you purchase this Inkwell product?

                                                     Registration form errors
address-required=Address Required                   <--
birthdate-required=Birth Date Required
date-purchased-required=Please enter the date you purchased the product
email-required=Please enter your email address
enter-valid-date=Enter A Valid Date
error-saving-registration=Error Saving Registration
firstname-required=First Name Required
gender-required=Gender Required
howhear-required=Please tell us how you heard about our product
```

```
lastname-required=Last Name Required
missing-company-id=Missing Company ID
missing-group-id=Missing Group ID
phone-number-required=Phone Number Required
product-type-required=Please enter the type of product you purchased
serial-number-required=Serial Number Required
where-purchased-required=Please tell us where you purchased the product

display-registrations=Display Registrations
there-are-no-registrations=There are no registrations
```

docroot/WEB-INF/src/content/Language.properties

You use the one file for the messages both portlets will need. This is part of the power of AlloyUI form tags. If you have a tag that specifies a field in the format `fieldName`, the tag will search the resource bundle for a matching language property in the format `field-name` and use that property value as the label for the field.

There's more, of course. There are two lines of code in the `addProduct` method that I didn't mention before, but about which you may have been curious. If you're successful in adding a product to the database, you add a key to an object called `SessionMessages`. If you aren't successful, you add a key to an object called `SessionErrors`. These keys correspond to messages in the Language.properties file. The key you add to `SessionMessages` is `product-saved-successfully`, whose matching value is the English text *Product Saved Successfully*. Similarly, you add a key for an error message to `SessionErrors`. Liferay makes these objects available to any JSP it's serving, and it has tag libraries that can take advantage of the fact that these objects are available. You can make these messages appear to your users by adding the following lines above the form in view.jsp:

```
<liferay-ui:success key="productSaved"
➥ message="product-saved-successfully" />
<liferay-ui:success key="productDeleted" message="productDeleted" />
<liferay-ui:success key="productUpdated" message="productUpdated" />
<liferay-ui:error key="fields-required" message="fields-required" />
<liferay-ui:error key="error-deleting" message="error-deleting" />
<liferay-ui:error key="error-updating" message="error-updating" />
```

If any of these messages appear in `SessionMessages` or `SessionErrors`, the tags activate and the messages from the Language.properties file are displayed to the user. If the messages aren't in `SessionMessages` or `SessionErrors`, nothing appears on the page. If you've used Liferay for any amount of time, you've seen these messages. An example is shown in figure 4.6.

Figure 4.6 When you save a product, this message appears to show that everything's okay and the action the user took was successful. This message is taken out of the Language.properties file and displayed properly to the user.

And of course, there's even more. You can make these messages appear in the user's language.

4.3.3 Translating messages to multiple languages

You can provide alternate translations for your message keys. Normally, you do this by creating companion files to Language.properties that have two-letter language codes appended to them. For example, if you wanted to provide message keys in Spanish, you could create a file called Language_es.properties, and the application would automatically pick up the values in that file if the end user has `es` (the language code for Spanish, which is *Español* natively) as his or her default locale.

This approach is generally a lot of work, because you have to get someone to translate your messages and place those translations in a file for each language you want to support. But what if you could generate a translation automatically? Liferay lets you do just that.

Open the build.xml file that you've been using to deploy your project. Notice that it's pretty small, because all of its functionality is derived from other Ant scripts stored in the Plugins SDK. Insert the following Ant task just below the `<import>` tag:

```
<target name="build-lang">
  <antcall target="build-lang-cmd">
    <param name="lang.dir" value="docroot/WEB-INF/src/content" />
    <param name="lang.file" value="Language" />
  </antcall>
</target>
```

Save the file, and then run the Ant target you just created, using the following command on all OSs:

```
ant build-lang
```

You'll see messages like the following, among others:

```
Translating en_it Product Name
Translating en_it Serial Number Mask
Translating en_ja Product Name
Translating en_ja Serial Number Mask
Translating en_ko Product Name
Translating en_ko Serial Number Mask
Translating en_pt Product Name
Translating en_pt Serial Number Mask
Translating en_es Product Name
Translating en_es Serial Number Mask
```

When the task completes, look at your content folder. It should now contain files for many languages.

> **NOTE** If you're using Liferay IDE or Liferay Developer Studio, you don't need to create this Ant task, because this functionality is built into the product. You can right-click the project (or the build file) and then select Liferay > Build Languages to run this task.

What just happened? The Plugins SDK contains Ant targets that use an online service to translate all the keys in your file to multiple languages. By providing the Ant task you created, you give those Ant targets the parameters they need to do their work: namely, the folder where your Language.properties file exists and the name of the file. Then, you were able to use the service to generate the language files you now see in that folder.

NOTE You may run into a problem with the service blocking your IP because you've called it too much, especially if you have a lot of keys. Don't worry; the block only lasts for about five minutes. If you run the script again, it will pick up where it left off, and eventually, you'll get everything translated.

Because this is an automated translation service, sometimes the translations it provides aren't optimal. But they certainly serve as a good starting point and are better than nothing. If you're targeting your application for a specific audience in a specific language, you should send the generated translation file to someone who can go over the translation and make sure it's correct before you release it. If you don't have a resource who can do that, at least you have a basic translation file you can use. If you want to provide a translation yourself, you can provide to your translator the file with the .native extension appended to it. Liferay generates this file by default as well; it overrides the automatic translations. Have your translator put his or her translations in this file, and then copy it back to the content folder; those translations will then be used in place of the generated ones.

Another nice thing about the way languages work in Liferay is that the language files from the portal are inherited by the portlets. This means you don't have to redefine common actions (such as `Save` and `Cancel`) that are already used by the portal: you can use them in your portlets as is. Your portlets inherit the translations of these common labels that are provided by Liferay.

If you have Liferay's source, you can find Liferay's Language.properties file in the Liferay source code in portal-impl/src/content.

There's one more thing about this form that we haven't gone over: field validation. Yes, Liferay has tools for that too.

4.3.4 Validating user-submitted forms

Liferay includes a utility that can perform field validation. Rather than writing something yourself or using a framework, you can use Liferay's utility to easily validate the data your users enter. Often this utility is much easier to use than what comes with a particular framework—especially if you're learning the framework for the first time. It can be used for any portlet, whether you're using a framework or not.

Liferay's utility is the `com.liferay.portalkernel.util.Validator` class. You create a separate `Validator` class for your portlet that contains validation logic for all the objects you want to persist to the database. You've already seen this class in use in the

addProduct method. The following listing contains the method you call to validate the product coming from the form.

Listing 4.7 Validating fields from a form

```
public static boolean validateProduct(PRProduct product, List errors) {

    boolean valid = true;

    if (Validator.isNull(product.getProductName())) {
        errors.add("product-name-required");
        valid = false;
    }

    if (Validator.isNull(product.getSerialNumber())) {
        errors.add("serial-number-prefix-required");
        valid = false;
    }

    if (Validator.isNull(product.getCompanyId())) {
        errors.add("missing-company-id");
        valid = false;
    }

    if (Validator.isNull(product.getGroupId())) {
        errors.add("missing-group-id");
        valid = false;
    }

    return valid;
}
```

docroot/WEB-INF/src/com/inkwell/internet/productregistration/portlet/ProdRegValidator.java

As you can see, this is pretty basic. Obviously, you could implement better validation here, because all you're checking for is a value. Liferay's Validator class has many useful methods, such as isPhoneNumber(), isEmailAddress(), and so on. If any field fails validation, you set the boolean to false, and you also add a key from your Language.properties file to the List object called errors. This is the SessionErrors object you saw earlier, and because you've put the liferay-ui tags that display these messages on your form, any messages coming from the Validator class are displayed to the user.

If you were to submit your form without filling out any of the fields, it would look like figure 4.7.

The presence of messages in SessionErrors triggers the first message; the second message is the one you want to display to the user. You can have an individual message for every field, as you'll see when you get to the more complicated form for registering a product. Your page isn't complete yet. You still need a way to display the data you're entering so you can view and edit it. For that, you'll use another helpful utility from Liferay: the search container.

You have entered invalid data. Please try again.

Please fill out all fields

Add A Product

Product Name

Serial Number Mask

Save

Figure 4.7 Error messages look different from regular messages, and should immediately "pop" to the user.

4.3.5 Displaying data with the search container

Search Container is a class that works in conjunction with Liferay's UI tag libraries to provide a user interface wrapper around lists of objects. Whenever you create a data-driven application like this one, naturally you'll be working with lists of objects. In this case, you're working with lists of `PRProduct`s. You could conceivably create a way to manually iterate through `PRProducts` in a table and then create another table for `PRRegistration`s, and so on. But Liferay solves this problem through the search container. It wraps your list of objects (the type of object doesn't matter) and automatically provides features such as pagination (for very large lists) and table formatting (using the tag libraries that go with it). Because of this, it contains attributes such as column headers, rows, and cursor positions.

Search Container is powerful but easy to use. Let's see how to display and edit data using this component.

4.3.6 Using the search container to present your data

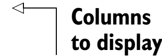
Let's jump right into the code so you can see how this works. Place the following code directly under the form in view.jsp.

Listing 4.8 Using the search container to show data

```
<liferay-ui:search-container
    emptyResultsMessage="there-are-no-products"

    delta="5">

  <liferay-ui:search-container-results>
  <%
  List<PRProduct> tempResults = ActionUtil.getProducts(renderRequest);

  results = ListUtil.subList(
      tempResults, searchContainer.getStart(),
```

Display works from these values

```
searchContainer.getEnd());
  total = tempResults.size();

  pageContext.setAttribute("results", results);
  pageContext.setAttribute("total", total);
  %>
  </liferay-ui:search-container-results>

  <liferay-ui:search-container-row
    className="com.inkwell.internet.productregistration.model.PRProduct"
    keyProperty="productId"
    modelVar="product">

    <liferay-ui:search-container-column-text
        name="product-name"
        property="productName"
 />
    <liferay-ui:search-container-column-text
        name="product-serial"
        property="serialNumber"
 />
    <liferay-ui:search-container-column-jsp
        path="/admin/admin_actions.jsp"
        align="right"
 />

  </liferay-ui:search-container-row>

  <liferay-ui:search-iterator />

</liferay-ui:search-container>
```



docroot/admin/view.jsp

The first thing you see is the initialization of the search container. You set an empty results message from `Language.properties` and set the delta for pagination to a pretty low number, because you're not envisioning many products in this search container. You get the results from the `ActionUtil` class, which calls the service layer you generated in chapter 3 to retrieve the products you need from the database. When you've calculated the total and the delta, you set these attributes in the `pageContext` so the search container can iterate over them.

After you get to the rows to be displayed, all you need to tell the search container is the name of the bean it's displaying, the property of the primary key, and the name of the variable to represent your model. From there, you can list the columns by their names (from Language.properties) and the property from the model bean.

Note the last column: it contains another JSP that defines the actions for each row. This JSP defines the Actions button that appears on every row of the search container table, allowing users to perform certain actions on `PRProduct`s. Figure 4.8 shows this button.

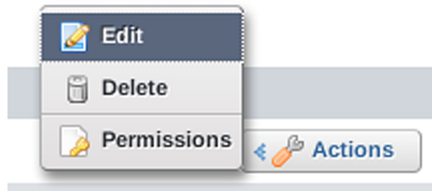


Figure 4.8 At the end of every row, an Actions button appears. When users click this button, they can perform three actions on each record: edit it, delete it, or set permissions on it.

The Actions button is implemented in another JSP, `admin_actions.jsp`.

Listing 4.9 Adding actions to the search container

```
<%@include file="/init.jsp" %>

<%
ResultRow row =
 (ResultRow) request.getAttribute(
    WebKeys.SEARCH_CONTAINER_RESULT_ROW);
PRProduct myProduct = (PRProduct) row.getObject();      <-- Gets object from row
long groupId = themeDisplay.getLayout().getGroupId();
String name = PRProduct.class.getName();
String primKey = String.valueOf(myProduct.getPrimaryKey());
%>

<liferay-ui:icon-menu>

  <c:if test="<%= permissionChecker.hasPermission(groupId, name, primKey,
 ActionKeys.UPDATE) %>">
    <portlet:actionURL name="editProduct" var="editURL">
      <portlet:param name="resourcePrimKey" value="<%= primKey %>" />
    </portlet:actionURL>

    <liferay-ui:icon image="edit" message="Edit" url="
<%= editURL.toString() %>" />
  </c:if>

  <c:if test="<%= permissionChecker.hasPermission(groupId, name, primKey,
➥ ActionKeys.DELETE) %>">
    <portlet:actionURL name="deleteProduct" var="deleteURL">
      <portlet:param name="resourcePrimKey" value="<%= primKey %>" />
    </portlet:actionURL>

    <liferay-ui:icon-delete url="<%= deleteURL.toString() %>" />   <-- Delete is different
  </c:if>

  <c:if test="<%= permissionChecker.hasPermission(groupId, name, primKey,
➥ ActionKeys.PERMISSIONS) %>">
    <liferay-security:permissionsURL
        modelResource="<%= PRProduct.class.getName() %>"
        modelResourceDescription="<%= myProduct.getProductName() %>"
        resourcePrimKey="<%= primKey %>"
        var="permissionsURL"
 />
```

```
    <liferay-ui:icon image="permissions"
    ➥ url="<%= permissionsURL.toString() %>" />
  </c:if>
</liferay-ui:icon-menu>
```

docroot/admin/admin_actions.jsp

As the search container loops through records, certain attributes of that container are available in the request, such as the current row in the loop. For each row, you can retrieve the `PRProduct` object and pull information about it that you need in order to implement the Actions button functionality.

For example, the database code operates from primary key values. The view layer therefore needs to provide a primary key to the action method so that the `PRProduct` to be operated on can be found in the database. You use the list of `PRProduct` objects that are wrapped in the search container to retrieve these keys, and then you define the key that is found as a parameter on each action URL you generate. In this file you're defining the action URLs that you'll use to call functionality for specific `PRProduct` objects. These URLs are then used in Liferay tags that assemble the Actions button with its links that fly out when the button is clicked.

Note that you haven't used any JavaScript or CSS to generate this; it's automatically generated by the tag library. This makes the tag library convenient to use, and your code looks clean. Additionally, the delete action uses a slightly different tag:

```
<liferay-ui:icon-delete url="<%= deleteURL.toString() %>" />
```

Because you never want to delete something without asking the user, "Are you sure?" the `icon-delete` tag automatically generates a JavaScript-based dialog that pops up and asks that question, allowing users to cancel if they clicked it by mistake.

Pretty cool, eh?

Now that you've got things working in the front end, you need to make sure those actions have functionality behind them.

4.3.7 Editing and deleting data

By now, you're probably getting the idea that the amount of code you have to write in Liferay's `MVCPortlet` for these kinds of applications is drastically reduced compared to what you might be used to. When a user clicks the Edit button in the search container, the `editProduct` portlet action runs. Because the action URL contained in a parameter the primary key of the record you want to edit, you can use that to retrieve the entity for editing:

```
public void editProduct(ActionRequest request, ActionResponse response)
    throws Exception {

    long productKey = ParamUtil.getLong(request, "resourcePrimKey");

    if (Validator.isNotNull productKey)) {
        PRProduct product =
            PRProductLocalServiceUtil.getPRProduct(productKey);
```

```
        request.setAttribute("product", product);
        response.setRenderParameter("jspPage", editProductJSP);

    }

}
```

Of course, you validate the key to make sure it has a value before you pass it on in your processing logic. Then you grab the `PRProduct` from the database and put it in the request so you can display it on a form for editing. This form will be almost the same form used for adding products; in fact, it could be the same form if you wanted it to be. But I'd like to point out one more thing before you get to the form: the last line of the previous code. Because you've defined the location of the edit form in the portlet's instance variable `editProductJSP`, you can point the page-management render parameter, `jspPage`, to the page where you want the user directed when this action completes. The following listing shows that form.

Listing 4.10 Editing a product

```
<%@include file="/init.jsp" %>

<%
PRProduct product = (PRProduct) request.getAttribute("product");
%>

<portlet:renderURL var="cancelURL">
  <portlet:param name="jspPage" value="/admin/view.jsp" />
</portlet:renderURL>                                          <- Cancel button is render URL

<portlet:actionURL name="updateProduct" var="updateProductURL" />   <- Submit button is action URL

<h2>Edit A Product</h2>

<aui:form
   name="fm" action="<%= updateProductURL.toString() %>"
   method="post">

  <aui:fieldset>

    <aui:input
        name="resourcePrimKey"
        value="<%= product.getProductId() %>"
        type="hidden" />

    <aui:input
        name="productName"
        value="<%= product.getProductName() %>"
        size="45" />
    <aui:input
        name="productSerial"
        value="<%= product.getSerialNumber() %>"
        size="45" />

    <aui:button-row>
```

```
        <aui:button type="submit"/>
        <aui:button
            type="cancel"
            value="Cancel"
            onClick="<%= cancelURL.toString () %>"
        />

    </aui:button-row>

  </aui:fieldset>

</aui:form>
```

docroot/admin/edit_product.jsp

As you can see, you use the `jspPage` functionality again in the form to let the Cancel button know where to send the users if they decide editing this product isn't what they want to do. The action URL points to another action that updates the product. Can you see how much easier it is to manage page flow with `MVCPortlet`? The only place you have to look for where the users are directed next is the natural place: the view layer.

Because the `update` method is so similar to the `add` method, there's no point in outlining it here (although, of course, you can see it by downloading the source code for this chapter). The same goes for the `delete` method. Let's instead get to something more interesting. You've seen it sprinkled throughout the earlier code, but we haven't touched on it yet: permissions.

4.3.8 Protecting data with Liferay permissions

Liferay Portal has a robust permissions system that allows you to implement just about any security model you can think of. This system has been designed so that developers who write portlets to be deployed on Liferay can make the same use of the permissions system as the portlets that ship with Liferay. This means you can implement security all the way down to the object level of your code.

You can now implement security in a portlet to enable administrators to set permissions so that only the users they want can add products. Doing so is surprisingly easy, particularly if you're already familiar with conditionally displaying HTML fragments to users. You'll implement that part of it using JavaServer Pages Standard Tag Library (JSTL) for this example.

To configure your portlet to use Liferay permissions, you don't have to write any code. The only code you'll ever write with regard to permissions is simple `if` statements to check permissions. You enable permissions in a portlet by creating two files: portlet.properties and a permissions XML file.

4.3.9 Pointing to the permissions configuration

In the source folder of your project, create portlet.properties with the following content:

```
resource.actions.configs=resource-actions/default.xml
```

Save the file. This properties file is automatically read by Liferay and contains directives that configure the portlet or override settings from portal.properties.

The previous directive tells Liferay that the configuration file for the permissions system is in a folder called resource-actions and the file name is default.xml.

4.3.10 Configuring Liferay permissions

Create the resource-actions folder in your src folder, and create the default.xml file in this folder. The following listing shows the file's contents.

Listing 4.11 Defining Liferay permissions

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

<resource-action-mapping>

  <portlet-resource>                                        ❶ Portlet resource actions
    <portlet-name>product-admin</portlet-name>
    <permissions>
       <supports>
         <action-key>ADD_PRODUCT</action-key>
         <action-key>VIEW</action-key>
       </supports>
       <community-defaults>
         <action-key>VIEW</action-key>
       </community-defaults>
       <guest-defaults>
         <action-key>VIEW</action-key>
       </guest-defaults>

       <guest-unsupported>
         <action-key>ADD_PRODUCT</action-key>
       </guest-unsupported>

    </permissions>
  </portlet-resource>

  <model-resource>                                          ❷ Model resource actions
    <model-name>
      com.inkwell.internet.productregistration.model.PRProduct
    </model-name>
    <portlet-ref>
      <portlet-name>product-admin</portlet-name>
    </portlet-ref>
    <permissions>

      <supports>
        <action-key>DELETE</action-key>
        <action-key>PERMISSIONS</action-key>
        <action-key>UPDATE</action-key>
        <action-key>VIEW</action-key>
      </supports>
      <community-defaults>
        <action-key>VIEW</action-key>
```

```
        </community-defaults>
        <guest-defaults>
          <action-key>VIEW</action-key>
        </guest-defaults>
        <guest-unsupported>
          <action-key>UPDATE</action-key>
        </guest-unsupported>
      </permissions>
    </model-resource>

    <model-resource>
      <model-name>
        com.inkwell.internet.productregistration.model.PRUser
      </model-name>
      <portlet-ref>
        <portlet-name>product-registration</portlet-name>
      </portlet-ref>
      <permissions>
        <supports>
          <action-key>DELETE</action-key>
          <action-key>PERMISSIONS</action-key>
          <action-key>UPDATE</action-key>
          <action-key>VIEW</action-key>
        </supports>
        <community-defaults>
          <action-key>VIEW</action-key>
        </community-defaults>
        <guest-defaults>
          <action-key>VIEW</action-key>
        </guest-defaults>
        <guest-unsupported>
          <action-key>UPDATE</action-key>
        </guest-unsupported>
      </permissions>
    </model-resource>

    <model-resource>
      <model-name>
        com.inkwell.internet.productregistration.model.PRRegistration
      </model-name>
      <portlet-ref>
        <portlet-name>product-registration</portlet-name>
      </portlet-ref>
      <permissions>
        <supports>
          <action-key>DELETE</action-key>
          <action-key>PERMISSIONS</action-key>
          <action-key>UPDATE</action-key>
          <action-key>VIEW</action-key>
        </supports>
        <community-defaults>
          <action-key>VIEW</action-key>
        </community-defaults>
        <guest-defaults>
          <action-key>VIEW</action-key>
```

```
      </guest-defaults>
      <guest-unsupported>
        <action-key>UPDATE</action-key>
      </guest-unsupported>
    </permissions>
  </model-resource>
</resource-action-mapping>
```

docroot/WEB-INF/src/resource-actions/default.xml

Notice that there are two sections to this file: a section defining a *portlet resource* and a section defining a *model resource*.

DEFINING THE PORTLET RESOURCE

The portlet resource section ❶ defines all the actions the portlet supports. Two of the defaults are `configuration` and `view`. The `configuration` action is the standard Liferay configuration screen, to which users can navigate by clicking the Configuration button in the window title of a portlet. You don't have it here because your portlet is in the Control Panel and doesn't need a configuration screen.

The other default action you've added is `view`. This action lets users view the portlet. Without this action, no one would be able to view the portlet, which wouldn't make any sense.

The third action isn't a default action, but one you've defined as functionality for this portlet: `add product`. By defining this action, you're saying that you want to choose which users can add products and which users can't.

These actions are portlet actions, because they belong to the portlet itself. The portlet is the authority for whether `PRProduct`s (and `PRRegistration`s) can be created, edited, or viewed. Therefore the `add product` action is a portlet action. If you wanted to set permissions for this action in Liferay, you could do so by creating a role and then defining permissions for it.

DEFINING THE MODEL RESOURCE

The resources in ❷ are the objects you've created that are stored in the database. You've been providing the functionality for creating, editing, and deleting `PRProduct`s. If you want to wrap Liferay's permissions system around the objects that have been created, you need to define those permissions in this file as model resource permissions.

To define the permissions for a resource, specify the fully qualified class name for the object and then define what actions may be performed on instances of that object. `Delete`, `update`, and `view` are actions you may want to grant permission to do, and those functions have already been created. You're adding a new action—`permissions`—which is a user's ability to grant or take away permissions to do other actions. The nice thing about this is that *no coding has to be done to enable this.* You need to provide a link to Liferay's permissions system—which will be covered next—and this functionality will be added to your portlet. Figure 4.9 shows what happens when that link is in place: your entities can be permissioned in exactly the same way as any other entity in Liferay.

Figure 4.9 Any model resource you define can have permissions configured for it in Liferay. Here, you've created a role called Product Administrators. You can grant that role permission to delete, to define permissions, to update, or to view `PRProducts`.

In both the portlet resources and model resources sections of the file, you can define default permissions for guests or the community/organization in which the portlet is located. For portlet resources, you've given everyone the View permission by default and have made sure that for guests, the Add Product permission is unsupported. For model resources, you've given everyone permission to view `PRProduct` records (otherwise, users who aren't registered wouldn't be able to register their products online, because they wouldn't be able to pick them from a list) and have prevented guests from being able to update a product.

All these permissions can be overridden by administrators through the permissions system. These are the permissions that will be set by default when the portlet is added to a page.

Believe it or not, we've now covered all of the APIs that Liferay gives you for easily creating data-driven portlets. But you're not through yet: you'll probably need to do some more advanced things that I haven't shown you yet in the Product Registration portlet. Let's look at the registration form to hit those areas.

4.4 Generating different field types with AlloyUI taglibs

If you recall, the registration form has more than just text fields on it. Users are asked to pick from lists and fill out a couple of date fields. Anyone who has written data-driven applications knows that giving users a text field for these kinds of things leads

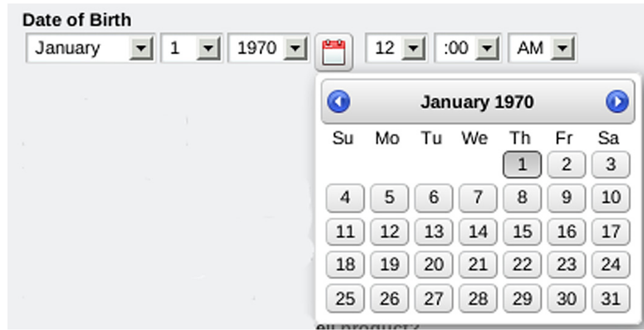


Figure 4.10 AlloyUI tag libraries make it easy to generate date pickers. No JavaScript required.

to inconsistent data. Dates can be entered in any number of ways, and users will likely never type things exactly the way you want them to. This means you need to provide ways to enter this data that both preserve the integrity of the data and are easy.

In this section, you'll create a date-picker control and a select box.

4.4.1 Generating date pickers

With AlloyUI taglibs, you can generate a date field that looks like figure 4.10.

You can generate this great date picker automatically by crafting the field as shown next.

Listing 4.12 Date picker without JavaScript

```
<liferay-ui:error
    key="birthdate-required"
    message="date-of-birth-required" />

<%
Calendar dob = CalendarFactoryUtil.getCalendar();
dob.setTime(regUser.getBirthDate());
%>                                              ← Initialize Calendar object

<aui:input
    name="birthDate"
    model="<%= PRUser.class %>"
    bean="<%= regUser %>"
    value="<%= dob %>" />                       ← Tag contains model, bean, value
```

docroot/registration/view_add_registration.jsp

Obviously, the date picker can't be created without JavaScript, so it may be an exaggeration to say it was. But *you* don't have to write any of it. By using the tag, the same JavaScript that Liferay wrote to provide date pickers for its own portlets is used to provide your date picker.

There's one thing missing, though. Notice that you're working with a date field, and the date in figure 4.10 is from 1970. This is so it's clear that users should pick a date in the *past*, not a recent date.

> **Why 1970?**
>
> Studies have shown that most people don't have the same perception of time that occurred before their birth that they have for time afterward. It's not as real to them. According to UNIX, time started on Thursday, January 1, 1970. Although it can represent time before that, that time isn't as real, because it has to be represented as a negative number. Why? Because UNIX (and most operating systems) represents time as "the number of seconds *since* midnight on January 1, 1970." This system of time works well, but it has some problems; for example, the Y2K38 bug will happen on Tuesday, January 19, 2038 when the integer that's used to represent time rolls over to a negative value.
>
> Why did I pick 1970 for the example? Convenience. It's somewhere in the middle of the time range when people who are using this application were born. Plus you get geek cred for using January 1, 1970.

To make the tag display just a date, you have to give it *model hints*. This is the developer's equivalent of pulling Liferay aside and whispering in its ear, "Hey, I want you to display the field *this* way." You do so using a file that was generated when Service Builder created Java classes to manipulate your data. Open the file portlet-model-hints.xml, which you'll find in the META-INF folder of your src folder. Scroll down in the file until you find the field for birth date, and replace it with the following code:

```
<field name="birthDate" type="Date">
  <hint name="year-range-delta">70</hint>
  <hint name="year-range-future">false</hint>
  <hint name="show-time">false</hint>
</field>
```

This tells the tag that for this field, you want a date range of 70 years, and you want to disable the user's ability to select dates that are in the future. And because it's a birth date, you don't want to show the time attributes on the field. This makes the data entry appropriate for a birth date field.

Later in the form, you have a Date Purchased field. For this field, all you need to do is turn off showing the time.

The form uses another field type we haven't covered: a select box.

4.4.2 Selecting data with AlloyUI taglibs

When users select the product for which they're registering, you populate that selection field with values from the database (see the Product Type field in figure 4.11). This is a simple matter of using the AlloyUI tag for a select field along with some Java code that retrieves the data, as shown in the following listing.

Listing 4.13 Filling an AlloyUI option tag

```
<%
List<PRProduct> products =
    PRProductLocalServiceUtil.getAllProducts(themeDisplay.getScopeGroupId());
```

```
%>

<aui:select name="productType">
  <aui:option value="-1">
    <liferay-ui:message key="please-choose" />
  </aui:option>

  <%
  for (PRProduct product : products) {
  %>
  <aui:option
 value="<%= product.getProductId() %>">
    <%= product.getProductName() %>
  </aui:option>
  <%
  }
  %>

</aui:select>
```

docroot/registration/view_add_registration.jsp

You get the primary key as the value of the select, and you get the name of the product for display to the user. This populates the selection box, and AlloyUI takes care of the formatting. As you can see, AlloyUI tags do a lot in terms of providing both functionality and a consistent look and feel for your forms. You don't have to mess with CSS or JavaScript by default, and if you do want a custom look and feel for your fields and buttons, you can do so in one place by styling them in a theme (see chapter 5).

The completed form contains multiple field types, all generated using AlloyUI tag libraries (see figure 4.11).

I can't say this enough: use the AlloyUI tag libraries for forms. They're an incredible help and time saver, and they can make your forms a lot nicer with little effort.

Now that you've written the entire portlet, let's look at the portlet class as a whole to see the benefits of using Liferay's `MVCPortlet`.

First Name
Test
Last Name
Test
Address 1
Address 2
City
State
Zip
Country
Email Address
test@liferay.com
Phone Number
Date of Birth
January 1 1970 12 :00 AM
Gender
Male
Date Purchased
April 21 2010 3 :21 PM
How did you hear about this Inkwell product?
Please Choose
Where did you purchase this Inkwell product?
Please Choose
Product Type
Please Choose
Product Serial Number
Save Cancel

Figure 4.11 No HTML, CSS, or JavaScript was used to develop this form. Every element was created by AlloyUI tag libraries, which generate standard HTML and widgets to assist the user in filling out the form.

4.5 Using Liferay's MVC makes your portlets simpler

The great thing about Liferay's MVC framework is how it makes portlet classes simpler. You don't have to write page-management or page-flow code; `MVCPortlet` handles that for you. To demonstrate, the following listing shows the entire Product Registration portlet class.

Listing 4.14 Product Registration portlet class

```
public void addRegistration(
    ActionRequest request, ActionResponse response) {      <- (1) User clicks Register button

    ThemeDisplay themeDisplay = (ThemeDisplay) request.getAttribute(
        WebKeys.THEME_DISPLAY);

    PRRegistration registration = new PRRegistrationImpl();
    PRUser prUser = new PRUserImpl();

    if (themeDisplay.isSignedIn()) {
        User user = themeDisplay.getUser();
        List<Address> addresses = Collections.EMPTY_LIST;
        Address homeAddr = null;

        try {
            addresses =
                AddressLocalServiceUtil.getAddresses(
                    user.getCompanyId(),
                    User.class.getName(),
                    user.getUserId());
        }
        catch (SystemException ex) {

        }

        if (addresses.size() > 0) {
            homeAddr = addresses.get(0);
        }

        prUser.setFirstName(user.getFirstName());
        prUser.setLastName(user.getLastName());
        prUser.setEmail(user.getEmailAddress());

        try {
            prUser.setBirthDate(user.getBirthday());
            boolean male = user.getMale();
            if (male) {
                prUser.setGender("male");
            }
            else {
                prUser.setGender("female");
            }
            prUser.setMale(male);
        }
        catch (PortalException e) {
            prUser.setBirthDate(new Date());
        }
```

```
        catch (SystemException e) {
            prUser.setMale(true);
        }

        if (homeAddr != null) {
            prUser.setAddress1(homeAddr.getStreet1());
            prUser.setAddress2(homeAddr.getStreet2());
            prUser.setCity(homeAddr.getCity());
            prUser.setPostalCode(homeAddr.getZip());
            prUser.setCountry(homeAddr.getCountry().toString());
        }

        registration.setDatePurchased(new Date());
    }
    else {


        registration.setDatePurchased(new Date());

        Calendar dob = CalendarFactoryUtil.getCalendar();
        dob.set(Calendar.YEAR, 1970);
        prUser.setBirthDate(dob.getTime());
        prUser.setGender("");

    }


    request.setAttribute("regUser", prUser);
    request.setAttribute("registration", registration);
    response.setRenderParameter("jspPage", viewAddRegistrationJSP);
}

public void registerProduct(
        ActionRequest request, ActionResponse response)
throws Exception {
    PRUser regUser = ActionUtil.prUserFromRequest(request);
    PRRegistration registration =
        ActionUtil.prRegistrationFromRequest(request);
    ArrayList<String> errors = new ArrayList<String>();
    ThemeDisplay themeDisplay =
        (ThemeDisplay)request.getAttribute(WebKeys.THEME_DISPLAY);

    long userId = themeDisplay.getUserId();

    User liferayUser = UserLocalServiceUtil.getUser(userId);

    boolean userValid = ProdRegValidator.validateUser(regUser, errors);
    boolean regValid =
        ProdRegValidator.validateRegistration(registration, errors);

    if (userValid && regValid) {

        PRUser user = null;

        if (liferayUser.isDefaultUser()) {
            userId = 0;
            user = PRUserLocalServiceUtil.addPRUser(regUser, userId);
```

2 **User clicks Submit button**

```
        }
        else {

            user =
                PRUserLocalServiceUtil.getPRUser(
                    themeDisplay.getScopeGroupId(), userId);

            if (user == null) {
                regUser.setUserId(userId);
                user =
                    PRUserLocalServiceUtil.addPRUser(
regUser, userId);

            }

        }

        registration.setPrUserId(user.getPrUserId());

        PRRegistrationLocalServiceUtil.addRegistration(registration);
        SessionMessages.add(request,
"registration-saved-successfully");
        response.setRenderParameter("jspPage", viewThankYouJSP);
    }
    else {
        for (String error : errors) {
            SessionErrors.add(request, error);
        }
        SessionErrors.add(request, "error-saving-registration");
        response.setRenderParameter("jspPage", viewAddRegistrationJSP);
        request.setAttribute("regUser", regUser);
        request.setAttribute("registration", registration);
    }

}
```

docroot/WEB-INF/src/com/inkwell/internet/productregistration/portlet/ProductRegistrationPortlet.java

Note that to save space (as I'll do for the rest of the book), I've removed ancillary elements like import statements, Javadoc, comments, and package declarations. The important point is this: the entire portlet class has been reduced to only two methods, and both of them are the result of actions the user performs. The first is when users click the Register button to display the form ❶, and the second is when users click the Submit button to submit the registration ❷. The Product Administration portlet is the same: it consists only of action methods that users trigger by clicking something. The full source code for this project is available as a downloadable companion to this book; if you want to check it out in more detail, please do so.

4.6 Summary

Liferay's MVC framework speeds up the development of portlets. Portlet classes are reduced to only action methods, because the framework handles the page management logic through a simple render parameter called `jspPage`. Because Liferay's MVC

portlet is extended from the Portlet API's class, you can still use the full API of the portlet standard.

In addition to the MVC portlet itself, Liferay offers a wide range of utilities to assist the developer. The `Validator` class assists you in performing field validation for forms. AlloyUI tag libraries help you create good-looking, dynamic forms complete with CSS styling and autogenerated JavaScript widgets. Data tables are easy with the search container. Liferay also provides a way to automatically translate language bundles so they can support multiple languages easily. And finally, Liferay permissions are easy to integrate into any application by configuring those permissions in an XML file, which Liferay then reads in order to integrate its full permissions UI with your application.

We've covered a lot in this chapter, but I hope you're still with me and that you can see some of the power of Liferay's development platform. Next, we'll turn to another plugin type: themes. These let you completely control the way Liferay looks.

JAVA WEB DEVELOPMENT

The Official Guide to Liferay Portal Development

Liferay IN ACTION

Richard Sezov, Jr.

Liferay in Action is the official guide to building Liferay portal applications using Java and JavaScript. If you've never used Liferay before, don't worry. This book starts with the basics: setting up your development environment and creating a working portal. Then, it builds on that foundation to help you discover social features, tagging, ratings, and more. You'll also explore the Portlet 2.0 API, and learn to create custom themes and reusable templates.

Experienced developers will learn how to use new Liferay APIs to build social and collaborative sites, use the message bus and workflow, implement indexing and search, and more. This book was developed in close collaboration with Liferay engineers, so it answers the right questions, and answers them in depth.

What's Inside

- Complete coverage of Liferay Portal 6
- Covers both the commercial and open source versions
- Custom portlet development using the Portlet 2.0 spec
- Liferay's social network API
- Add functionality with hooks and Ext plugins

No experience with Liferay or the Portlets API is required, but basic knowledge of Java and web technology is assumed.

Rich Sezov is Liferay's Knowledge Manager and is the author of the *Liferay Portal Administrator's Guide*. He leads Liferay's documentation and training materials team.

For access to the book's forum and a free ebook for owners of this book, go to manning.com/LiferayinAction

"Flat-out the best guide for Liferay 6.0 and the upcoming 6.1 release."
—From the Foreword by Brian Kim, Liferay COO

"Excellent in-depth treatise on the most popular CMS on the planet."
—Sumit Pal, LeapFrogRx Inc.

"Harness the full power of this juggernaut technology."
—Tariq Ahmed Amcom Technology

"A great companion to Liferay's Admin Guide."
—John J. Ryan III Princigration LLC

"Expertly written, thorough coverage."
—John S. Griffin, Coauthor of *Hibernate Search in Action*

ISBN 13: 978-1-935182-82-5
ISBN 10: 1-935182-82-X

MANNING $44.99 / Can $47.99 [INCLUDING eBOOK]