

# Liferay

## IN ACTION

The Official Guide to Liferay Portal Development

Richard Sezov, Jr.

MEAP



MANNING





**MEAP Edition  
Manning Early Access Program**

Copyright 2010 Manning Publications

For more information on this and other Manning titles go to  
[www.manning.com](http://www.manning.com)

©Manning Publications Co. Please post comments or corrections to the Author Online forum:  
<http://www.manning-sandbox.com/forum.jspa?forumID=642>

Licensed to Pyramid-Consulting <jean-paul@pyco.be>

## ***Table of Contents***

*Preface*

*Acknowledgments*

*About this book*

*About the authors*

## ***Part 1 Getting started with Liferay***

1. Liferay is a different portal
2. *Appray*: Development at light Speed

## ***Part 2 Building custom applications***

3. A data-driven portlet made easy
4. MVC the Liferay way
5. Using themes and layout templates
6. Making your site social
7. Enabling team collaboration

## ***Part 3 Customizing and extending Liferay***

8. Hooks
9. Extending Liferay
10. Liferay architectural overview

## ***Appendixes***

- A. Liferay and IDEs
- B. Introduction to the Portlet API
- C. How to contribute to Liferay
- D. Portal design forms

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

# 1

## *Liferay is a different portal*

This chapter covers

- Understanding portals then and now
- Exploring what Liferay is and how to work with it
- Defining basic Portal concepts
- Using Liferay to design a portal

Everybody needs a web site these days. Whether you're building one for a company, for a service organization, or for personal reasons, you need one. And when trying to decide how to build it, you've probably found a dizzying array of choices running on a dizzying array of platforms. So how do you go about choosing which platform is best?

First of all, if Liferay Portal isn't on your list, you should put it at the top right away. Liferay Portal is a Java-based open source portal, containing an unprecedented number of features, which will help you to implement your site in as little time as possible. And once you have Liferay on your list, let me respectfully submit that your search can end with Liferay Portal, which is hands down the best platform upon which to build a web site.

I can hear your objection now: "Of course you'll say that—you work for Liferay!"

Ah, but I did not always work for Liferay. I was a Liferay user for some time before I wound up working for them. So yes, I took the red pill,<sup>1</sup> so to speak, but I've also experienced Liferay from the outside, and so I know what it's like to be doing that search for a platform. I can tell you from experience that you're going to find working with Liferay to be a pleasure, and you'll be happy to know that using the platform that Liferay offers you will free you from limitations. Using Liferay as a platform will speed up your development cycle and give you features that you likely wouldn't have had time or the inclination to build yourself. Most of the time, potential Liferay users focus on Liferay as a product—because it boasts such a huge range of features—but they don't stop to consider the rich development platform it offers. Liferay as a development platform encourages you to take advantage of everything

---

<sup>1</sup> From the 1999 film *The Matrix*.

Liferay has to offer. By the end of this chapter, you'll have a good understanding of what Liferay is all about and what it can do for your web site. And I have no doubt that you'll find many reasons to choose Liferay for your next development project.

Choosing Liferay is also safe: You're putting yourself in a group with some of the largest organizations (and the largest web sites) out there that have also chosen Liferay as the platform for their web sites. So if I can give you any advice, it would be to end your search with Liferay and begin learning how you can leverage the platform to build the site of your dreams.

This chapter will go over several important topics. I'll show why Liferay calls itself a "portal," what a portal used to be, and how Liferay pioneered getting past its early limitations—giving you the freedom to use the platform for what it was meant to be: a robust fast track to implementation. We'll then take a helicopter ride over Liferay's feature set to see what it can do at a high level. After this, we'll delve into how Liferay helps you structure a web site. You'll also get to see what Liferay looks like by default and how you can navigate around it. And finally, using all the information we've presented, I'll show you how you can begin to imagine how your site might be implemented using Liferay Portal.

But first, to get our bearings, let's start by exploring why Liferay calls itself a portal and what that term has come to mean in the industry historically.

## ***1.1 The Java portal promise: from disappointment to fulfillment***

Liferay calls itself a portal. What do you commonly think of when you hear the word portal? As a big fan of sci-fi and fantasy, I tend to think of a doorway to another dimension or time like the portal that Kirk and Spock went through, chasing after McCoy to stop him from doing whatever he did to change the timeline. I'll tell you right away: Liferay Portal isn't that elaborate (but you've likely already figured that out). So why do we call it a portal? Let's start with the so-called official definition of a portal.

### **Portal**

A portal is a web-based gateway that allows users to locate and create relevant content and use the applications they commonly need to be productive.

That comes from a bullet on a slide I've used to teach Liferay to prospective users. I might even have written that bullet, but I'm not sure. Generally, the reaction I get is a narrowing of the eyes, some pursed lips, and then heads begin nodding up and down. This tells me that people want me to think that what I've just said makes sense, but they're being kind and reserving judgment on my teaching abilities, because it actually made no sense at all.

The problem with definitions like that is that they try to say too much in one sentence. Liferay is many, many things, and you can't capture it all in one sentence. But just for fun, let's try it again.

### **Portal**

A portal is designed to be a single web-based environment from which all of a user's applications can run, and these applications are integrated together in a consistent and systematic way.

That one's a bit closer when viewed in the context of the web. When we talk about Liferay as a development platform, that's exactly what we mean. At its base, Liferay is a container for integrated

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

applications. Those applications are what make the difference between Liferay and competing products. You're free to use the applications you like, write your own, and disable the rest. And this is what sets Liferay apart. Figure 1.1 shows how you can easily mix and match your applications with Liferay's.

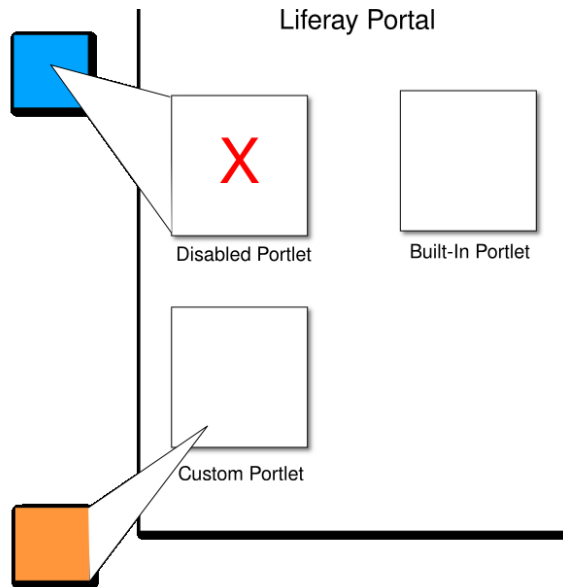


Figure 1.1 Liferay contains many built-in applications, called portlets. If there are some you'll never use, you can disable them. You can also write your own portlets and deploy them. These custom portlets are indistinguishable from portlets that ship with Liferay Portal.

As an analogy, think back to the eighties and early nineties. If you bought a computer and you needed to use it to write something, you also bought a word processor. If you then decided you wanted to calculate numbers with it, you bought a spreadsheet. And if you needed to store and retrieve data of some kind (perhaps for a mailing list), you bought a database. (Nobody created electronic slides back then; they used an overhead projector. And yes, I am dating myself.)

Most of the time, people would pick what was considered the best of whichever program they wanted. One vendor had the best word processor. Another had the best spreadsheet. A third had the best database. So if you had to perform all three functions, it was likely that you had three separate programs written by separate entities, but individually they were the best.

Pretty soon, people wanted to create graphs in their spreadsheets that they would insert into a word processing document that they would send to a mailing list stored in the database. The problem with that was that all of these programs were created by different vendors, and they didn't always work together all that well. Much effort on users' parts had to be spent on trying to get them to work together well.

You know the rest of the story. We wound up with office suites, consisting of programs written on the same platform that were designed to work together. Not only did this save us all some money

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

(because buying the separate programs cost a fortune), it also gave us a level of integration that had so far been unavailable.

The same thing is happening with software that can be used as an engine for running web sites. Liferay consists of that engine as well as the many applications that run on that engine. When you use this platform, your applications can have a level of integration with the rest of Liferay's applications that will make your users' experience seamless and smooth. Why? Because we believe the integrated experience is far better than the nonintegrated experience. This is the difference that makes Liferay stand above all the other portals out there. I have to say that because there are some who view the word portal with disdain, and sometimes this is with good reason. Let me explain further.

### **1.1.1 The Java portal disappointment**

When Java portals were first announced, they were hailed as the solution to many of the problems facing enterprises and other solution architects. The web had grown up. Instead of proprietary interfaces to everything, everybody had finally standardized on TCP/IP networking and open protocols such as HTTP, IMAP, SMTP, SOAP, and the like. Services, applications, and email operated on these open protocols, and products that once had relied on proprietary protocols had now opened up to the web. Those products that didn't (or whose vendors had delayed it) were relegated to the dust bin of history. And once we had all of these siloed services speaking the same language, we needed something to bring it all together for the end user.

Enter the Java portal. The release of the Java portal specification came with the promise of bringing all of these services together in a single unified "web desktop." Not only would it unify everything for a corporation's internal applications, it would also be the hub of all B2B (business to business), B2C (business to consumer), B2E (business to employee), and even G2P (government to public) communication. It would be the presentation layer for the brand-new service-oriented architecture that you finished (or were in the process of) implementing. It could also be a platform for new applications. And it could finally bring together your static web sites and your applications, which resided on separate application servers.

Do you think too much was promised? How's that old saying go? "If it seems too good to be true, it probably is?" Well, you're right. What happened? At least three issues emerged that prevented Java portals from achieving widespread acceptance.

For one, it was difficult to develop solutions using a portal. The initial Portal API turned out to be something like getting to the least common denominator. Instead of providing all the features developers would need to bring all this stuff together, it defined what seemed like the absolute minimum that all the vendors could standardize on and then left everything else up to the individual vendors. This meant that developers had to spend more time implementing features that should have been part of the platform in the first place. One example of this is that the initial standard did not include any way for portlets to communicate with each other.

Second, the portal servers themselves were too big and complex (not to mention hideously expensive), often taking days to get set up. And for the developer trying to get a development environment going, it was sometimes even worse. I can remember trying to work with one of the first portals (sorry, can't tell you which one it was) and finding it impossible to get a development environment properly configured on my laptop. At the time, I was a team lead and was trying to get this install process to a repeatable procedure for the rest of the developers on my team. My solution? I went

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

to a conference, grabbed one of the presenters after his talk, and made him help me install the development environment on my machine. When he heard of my plight, he understood completely and told me everybody was having this problem and that they had to make this process easier.

Third, other things were happening in the industry at the same time. The Web 2.0 concept was beginning to get popular, and the portlet specification had left no room whatsoever for enabling a rich, client-side experience for the end user. In order to compete, portal vendors started to implement their own proprietary extensions to the portlet specification. We all know what this leads to: vendor lock-in, which is precisely what defining a standard is supposed to avoid.

At the same time Java portals were getting a bad rap, sites like Facebook and MySpace came out and pretty much implemented what portals were meant to do all along. And as they got more and more popular, suddenly other sites like Amazon.com and other software like Jira began to implement the social collaboration features Facebook and MySpace had, along with their slick, AJAX-enabled user interfaces. What powered all of these new and improved web sites? What enabled them to implement such rich features for the end user so quickly? You guessed it: open source.

Open source solves a lot of the problems inherent in the old Java portal paradigm. Open source projects don't wait around for committees to decide on things; they tend to implement what the users want as fast as possible. There are no barriers to entry with open source; the development tools and the software are made available for free. Open source products also tend to be lighter weight; you don't need a large, dedicated server to start building your solution. Development goes faster, because developers don't have to learn the entire architecture to be effective. And you don't need a huge initial investment to get started using an open source solution. You can start small (free) and then grow your application and hardware as your needs grow. Facebook is the perfect example of this: Implemented using PHP (which is an open source web development platform), the site has grown organically as its user base has grown. This is what the market really wanted. And as you're about to see, Liferay Portal provides the same kind of open source platform that has allowed many organizations to do the same.

### **1.1.2 Liferay keeps the Java portal promises**

From the beginning, Liferay Portal has been an open source project. Its whole purpose for existence was to level the playing field so that smaller organizations such as nonprofits, small businesses, and open source projects could take advantage of its platform without having to incur huge expenditures for either software or hardware. So right out of the gate it was doing things differently. An open source project doesn't have the luxury of making it difficult for developers to work on the platform. Instead, developers need to find the platform to be easy to work with, or the project will have major hurdles to community gestation. And if an open source project can't foster the birth and growth of a vibrant community, it's dead. So right away, Liferay was (and continues to be) easy for developers to use, adapting to many different development styles, and not requiring any specific tools to be installed beyond what is already in any Java developer's toolset.

This same philosophy translates to its size. Open source projects also don't have the luxury of being too big or taking up too many system resources; they may be running on new hardware or five-year-old hardware that was donated to a nonprofit that can't afford anything else. Liferay Portal is much smaller and simpler to configure than its competitors. Can you run Liferay on big hardware with a proprietary Java application server? Sure you can. Can you run it on a shared server with a small servlet container like Tomcat? Absolutely. Liferay Portal is provided as a standard .war file—only 75 MB in size—which can

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>



be installed on any application server, or as a “bundle,” preinstalled in your open source application server of choice. You don’t have to go through long installation routines and complex command-line incantations to get it working. If you use a bundle, installing Liferay is as easy as unzipping an archive and editing a text file to point it to your database.

And guess what? Instead of giving you by default an empty portlet container into which portlet applications can be installed, Liferay Portal comes with over 60 portlet applications included. These applications cover about all of the “standard” functionality you’re likely to need in a web site: content management, forums, wikis, blogs, and much more—leaving you to implement only the features specific to your site. And for developers, your setup time will be measured in minutes, not hours. You also don’t have to know everything about the architecture to be effective—it’s really easy to get started.

Open source software also has to be innovative in order to compete with its proprietary competition. Liferay Portal was the first portal to implement that slick, Web 2.0 interface, back in 2006. The first time I saw a portlet being dragged across the browser window and dropped into another spot on the page, I was blown away, because I was used to the old, proprietary solutions that hadn’t implemented that yet. Because Liferay Portal was open source, it could respond to market demands faster than the other guys, using the same standards they were using. You’ll continue to see that in Liferay Portal, because the open source paradigm works. What users demand gets implemented, without sacrificing adherence to standards.

As far as standards go, Liferay is also based on widely used, standard ways of doing things. It adheres to the JSR-286 portlet standard. In addition to that, though, it includes utilities such as Service Builder to automatically generate interfaces to databases (something not covered by the standard). Under the hood, Service Builder is just Spring and Hibernate—widely used by Java developers everywhere. So you get the benefit of using the platform to get your site done faster, while still taking advantage of standards that keep your code free.

Now that I’ve spent so much time extolling the virtues of this magical, mystical thing known as Liferay Portal, you’re probably anxious to see what this wonderful specimen I’ve described looks like.

## **1.2 Getting to know Liferay**

Liferay Portal is an open source project that uses the LGPL open source license. This is the GPL license you know and love with one important exception: Liferay can be “linked” to software that is not open source. As long as you use Liferay’s extension points for your custom code, you don’t have to release your code as open source if you don’t wish to. You can keep it, sell it, or do whatever you want with it; it’s yours. If, however, you make a change to Liferay itself by modifying Liferay’s source code and want to redistribute the product thereafter, then you need to contribute that change back to Liferay. So you get an important exception with the LGPL: You can still use Liferay as a base for your own product and either open source the result or sell it commercially if you wish. Or, if you want to change Liferay directly, you can contribute to the open source project. It’s entirely up to you. You can download the open source version of Liferay Portal for free from Liferay’s web site.

Alternatively, Liferay sells an Enterprise Edition of Liferay Portal. This is a commercially available version of the product that comes with support and a hot-patching system for bug fixes and performance improvements. There are web sites running on both versions of Liferay Portal, and both are perfectly appropriate for serving up your site.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

In this section, we'll take a quick tour of some of the things you can do quickly with Liferay to begin building a web site. We're going to play around with the interface a bit so you can get to know it a little better. Figure 1.2 shows the default Liferay Portal 6 user interface.

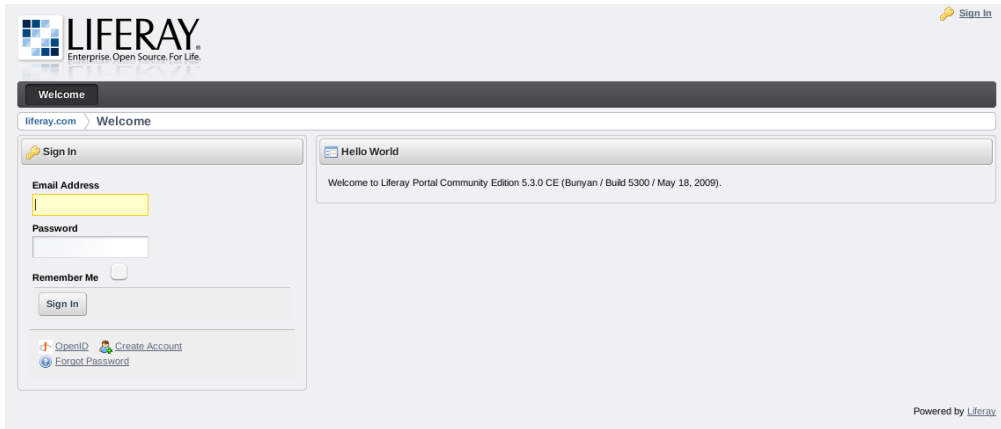


Figure 1.2 Liferay Portal 6, as it looks the first time you start it. It presents you with a basic interface at first, but as you'll see, you can easily jazz it up.

Okay, I agree; it doesn't look like much, does it? But there's an awful lot of power hidden in the humble interface that Liferay shows you by default. If you're ahead of the game and already have Liferay running, you can follow along. If not, just sit back and enjoy the ride: we'll go over how to get Liferay installed and running on your system in Chapter 2.

### 1.2.1 Liferay is an application aggregator

We've been saying that Liferay Portal is not just a product; it's a platform. This platform runs applications, and these applications are integrated together in ways that separate applications cannot be, by virtue of their shared platform.

What this means is that we can take that default Liferay page and load it up with integrated applications. Liferay makes doing something like that very easy. First, we have to log in as the default administrative user, whose user name is `test@liferay.com` and whose password is `test`. This will display the *Dockbar* (see figure 1.3) at the top of the page, which gives us access to several other functions.



Figure 1.3 Hovering the mouse over the Add menu in the Dockbar opens a drop-down menu. To see a full list of available applications, click More.

We'll come to all of the things you can do with the Dockbar in a moment. For now, however, all we want to look at is applications, which you can access from the *Add* menu. Commonly used applications appear directly in the menu, but if you want to see the whole list, click *More*. This will pop up a fully searchable, categorized view of all the applications that have been installed in your Liferay Portal by default. As an aside, by the time you are finished with this book, one of the things you'll be able to do is write your own applications, which can appear in this list.

We're going to fill this page with applications so you can see how Liferay aggregates them. You can browse the applications by opening the categories to which they're assigned. Or if you know the name of the application you're looking for, you can search for it by using the search bar at the top of the Applications window. Let's pick some cool applications to add to our page. Note that in a real-world web site, you'd likely never put all of these on one page—we're doing an experiment here to show the concept. To the left column, add Navigation, Activities, Dictionary, and Translator. To the right column, add Message Boards, Wiki, and Calendar. You can add an application to a specific column by dragging the application off the Applications window and dropping it into the appropriate column, as shown in Figure 1.4.

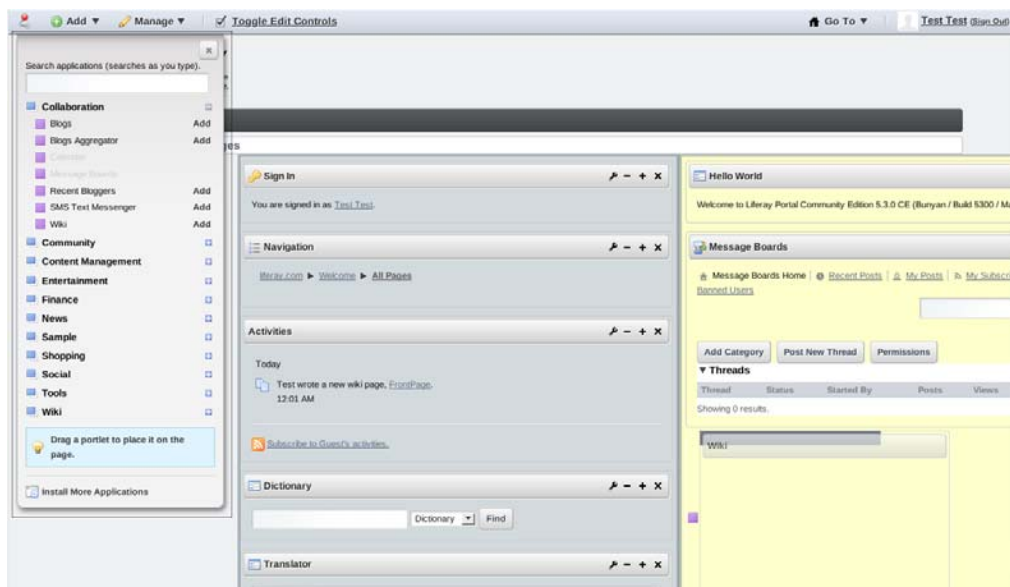


Figure 1.4 Most of the applications have been added. This screen shot was taken while dragging the Wiki application into the column on the right.

Now we have a single page with a whole bunch of applications on it. These applications can perform a lot of different functions.

The Message Boards application is a complete implementation of web-based forum software. If you're planning to have discussion forums on your web site, Liferay already has them built in. And the cool thing about it is you don't have to integrate anything. They already work with Liferay's user management and security features, as do all of Liferay's applications.

You've also added a Wiki application to the page. Again, this is a full-fledged wiki that you can use for whatever purpose suits you. As with the Message Boards application, the Wiki is integrated with Liferay's user management and security. But (and this is the cool part) the Wiki also is integrated with Liferay's Message Boards application, because it borrows functionality from that application to provide comment threads at the bottom of Wiki articles. Those threads will use your users' profile information (including pictures) in the threads to uniquely identify them in a consistent way throughout your site, which is yet another level of integration.

What about the Calendar? Again, it's totally integrated, complete with email notifications and more. And it's a full-fledged calendar application that supports export and import of calendar data from other applications.

The other applications are smaller, and I don't want to gloss over them, but you're probably getting the picture at this point. Let me point out one other thing, though, and that's the Activities application. Notice in figure 1.5 what it says.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>



Figure 1.5 Every application in Liferay that uses the Social API can capture activities unique to that application. The Activities portlet displays those activities. Did you really create a new wiki page?

When you added the Wiki application to your page, you created a top-level Wiki page, which is by default called FrontPage. Because the Wiki application uses Liferay's Social API to capture its unique activities, the Activities application can report on what you did (and even provides an RSS feed of activities). Liferay has an API that lets you tap into this social capability. This opens up all kinds of possibilities for your own applications, doesn't it? (We'll cover this API in detail in chapter 6.)

#### NOTE

Because Liferay is a portal, its applications are called *portlets*. I have been careful so far to refer to them only as applications, but for the remainder of this book, we'll use the terms *portlet* and *application* interchangeably.

Naturally, we'd never in the real world create a page such as this. Your users would throw conniption fits if they had to navigate such a thing. You might do better by your users if you put some of these applications on different pages. I just wanted to illustrate how integrated Liferay's applications are.

In addition to providing a development platform and a slew of applications out of the box, Liferay is also a powerful content management system (CMS).

### 1.2.2 Liferay is a content manager

If you have lots of web content that you wish to publish and you wish to publish that content using a workflow, or on a schedule, statically or dynamically, to staging or production, with templates or without, then you might want to check out Liferay's CMS.

You can access the web content functions from the same Applications window you've already seen; in fact, they're in their own category. But the quickest way to do it is to simply select Web Content Display, right from the *Add* menu. It's right in the menu for convenience—if you are building a content-rich web site, you'll use it a lot. Once it's added, you can drag it to whatever position on the page you want. Figure 1.6 shows this portlet added to the right column on the page.



Figure 1.6 The Web Content Display portlet is added, but it has no content (yet). We'll remedy that very quickly.

A Web Content Display portlet does what its name implies: it displays web content. So in order for it to do its job, you'll have to create some web content. You can do that very quickly by clicking the *Add Web Content* icon, which is the icon at the bottom-right. You are then brought to a form where you can add content. Figure 1.6 shows this page.

For now, don't worry about all the options on the right side of the screen (Structure, Template, Workflow, and so on.). For basic content management, all you have to do is start adding content. Give your piece of content a name and a description, and type some content into the editor. Notice in figure 1.7 that you can apply all sorts of formatting in the editor: fonts, tables, bullets, colors, and images.

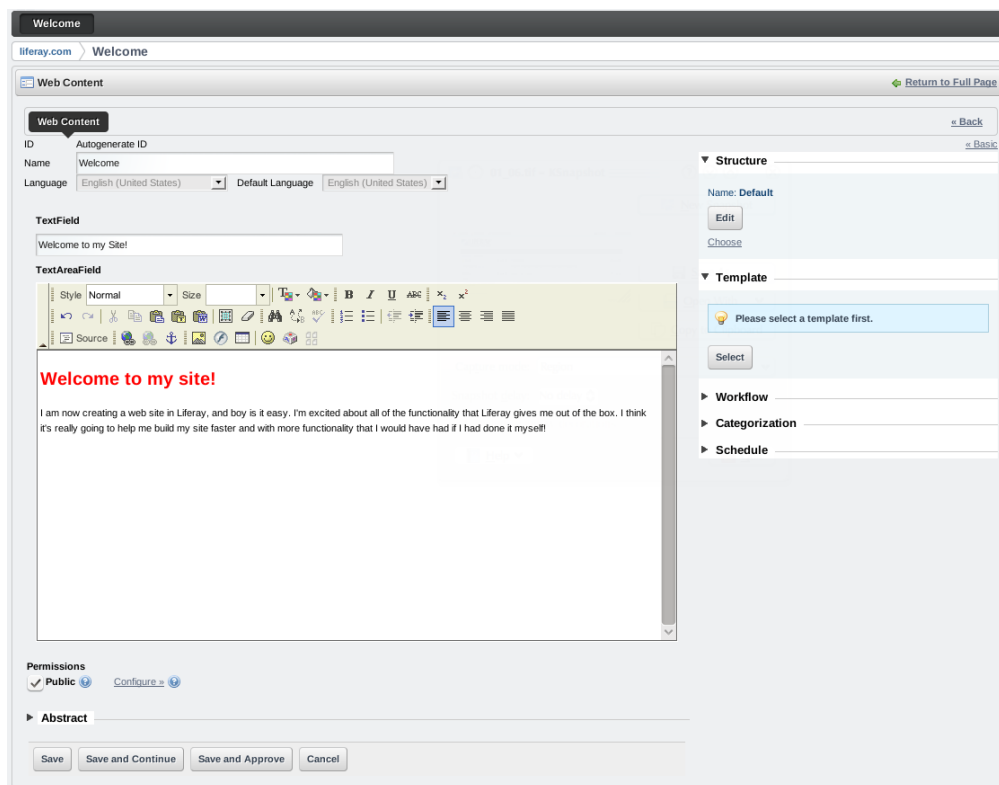


Figure 1.7 Entering content in Liferay's Content Management System.

Once you've finished adding your content, notice the buttons at the bottom of the page. Though there is a whole workflow process you can go through, you are logged in as the portal administrator. This means you can short-circuit the workflow process by clicking the button marked *Save and Approve*. So go ahead and do that. You'll be brought back to your original portal page, and the Web Content portlet will contain the content you just added.

We could go further with Liferay's Web Content Management system, but suffice it to say that it's sufficiently powerful for whatever content needs you may have. For example, you can create your own structures with custom fields for your content, as well as templates to go along with your structures to display your content in exactly the way you want it displayed. You can stage your content on a staging server and have it published on a schedule of your choosing. You can write powerful, scripted templates in XSL, Velocity, or Freemarker.

You've seen so far that Liferay can be a platform or a UI for your applications, and it can also manage your site's content. The last ingredient that you need Liferay provides in spades, and that's a way for your users to find and collaborate with each other.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

### 1.2.3 Liferay is a collaboration tool

Liferay Portal is ideal for setting up collaboration environments among workgroups. Whether you call these environments communities or virtual team rooms, Liferay can be used to help your team get their work done. It does this by providing applications which are geared specifically toward document sharing and communicating with one another.

One of the portlets you can add to a page in Liferay is the Document Library. This application provides a facility for sharing documents with your entire team. It keeps a complete version history of all of your documents and is integrated with Liferay's permissions system. This integration allows you to grant access to shared documents or prevent some of your users from accessing sensitive documents. And if your users need an easier way to access the documents than the web interface provides, the Document Library supports WebDAV, allowing documents to be uploaded and downloaded through their operating system's familiar interface. Figure 1.8 shows both of these interfaces.

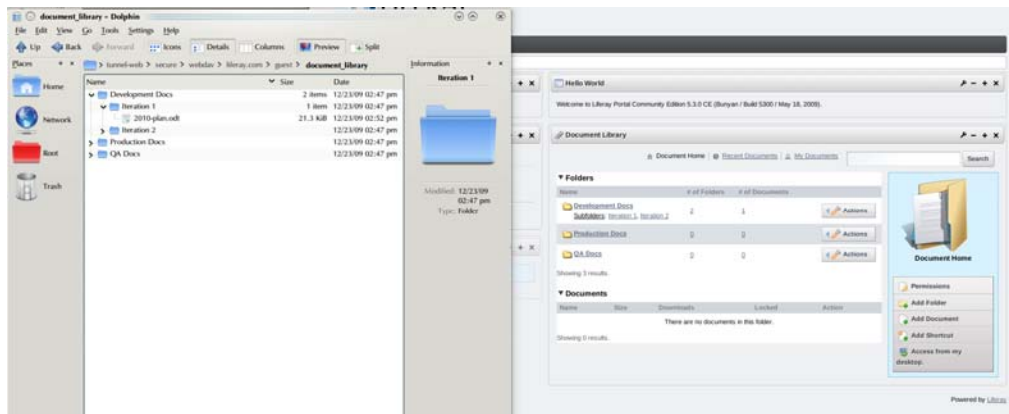


Figure 1.8 Accessing the same folders in the Document Library in the operating system via WebDAV or using the browser interface.

Documents are one thing, but what about communication? Liferay's portlets allow for communication right in context, so your users can keep all the relevant information in the right place. So the Document Library allows your users to create discussion threads right next to the documents they need to talk about. The Wiki does the same thing. And applications are provided for both chat and email, so that currently logged-on users can communicate in real time, no matter what their physical distance is from one another.

Need a group calendar? The Calendar portlet can be used for either individuals or for groups. Additionally, your users can all have their own individual blogs on their own pages which are then aggregated using the Blogs Aggregator to the community page. This enables you to display a "blog of blogs," allowing your team to stay updated on what everyone is doing. Combining this with Activities makes for a consistent, rolling list of what the team is up to.

All of the functionality I've mentioned so far is what is built in to Liferay (and there's more we haven't touched on). But Liferay is extensible too.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>



### 1.2.4 Liferay is anything you want it to be and any way you want it to look

Liferay offers a level of customization that is unparalleled, because you can modify *anything* in Liferay, from simple functionality changes all the way to making your own product out of it.

This book will systematically show you how to write your own portlets so that your applications can be added seamlessly to your Liferay-powered web site's pages in a way that is indistinguishable from the built-in portlets. You'll also learn how to customize Liferay's layout templates so that your page layouts can be what you want them to be. You'll also learn about hooks, which let you customize Liferay by substituting your own classes and JSPs in the place of Liferay's. And finally, you'll learn about Ext plugins, which let you override anything in Liferay with functionality of your own.

No discussion of customizing Liferay would be complete without covering themes. Using themes, you can transform Liferay's look and feel to make it look any way you want it to (see figure 1.9). In short, Liferay can be anything you want it to be, and it can look any way you want it to look. This gives you the power and flexibility you need to build your own custom site, with the functionality you need to get it done in a timely fashion.

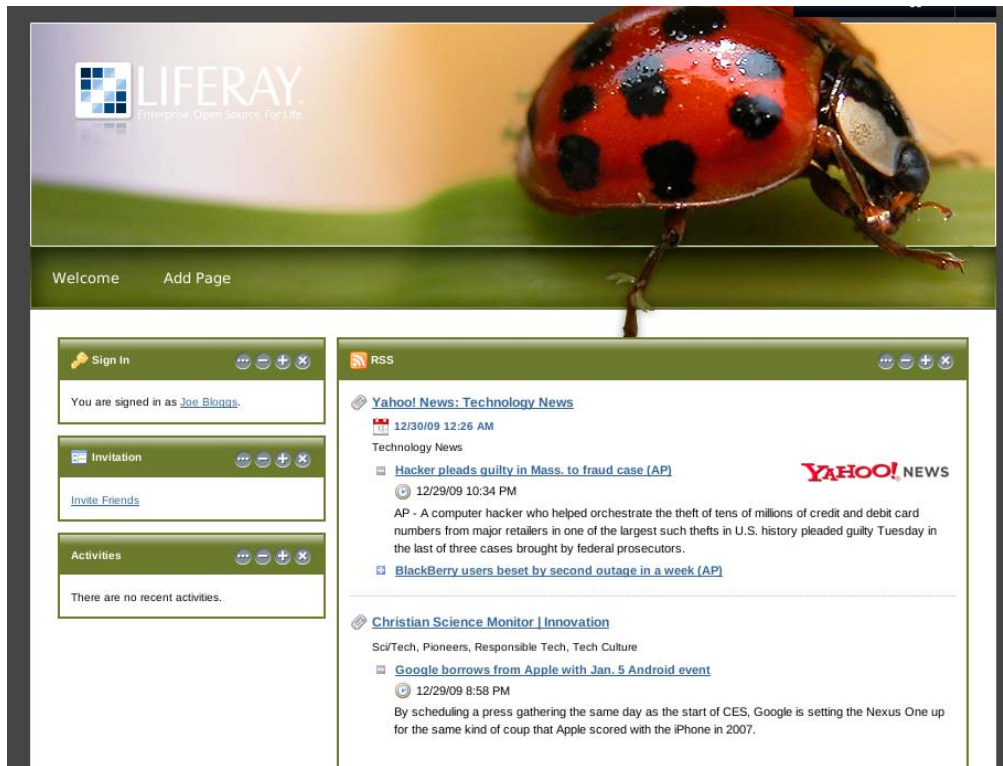


Figure 1.9 Liferay with a custom theme applied. This is just one of many themes in Liferay's community repository.

Liferay provides you the freedom to make your site look the way you want it to look, using skills you already have. Themes are nothing more than custom HTML and CSS applied to the page. So you'll have

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

the same ability to design your site as you would have if you were writing the whole thing from scratch—except for the fact that you'll have less work to do, because of Liferay's built-in functionality and rich development platform.

Liferay also comes connected to two repositories of ready-made plugins which extend Liferay's functionality. One of these is a Liferay-provided repository, and the other is a community repository. The screen shot in figure 1.8 above is an example of a theme provided by Liferay's community through the community repository. Liferay's repositories make it very easy to both distribute and to install new software that runs on Liferay's platform, as you can see in figure 1.10.

Plugin Installer

[Browse Repository](#)
[Upload File](#)
[Download File](#)
[Configuration](#)
[< Back](#)

Portlet Plugins

[Theme Plugins](#)
[Layout Template Plugins](#)
[Hook Plugins](#)
[Web Plugins](#)

Keywords

Tag
All

Repository
All

Install Status
Not Installed or Out of Date

Search

Showing 21 - 40 of 111 results.
Items per Page 20
Page 2 of 6
[First](#)
[Previous](#)
[Next](#)
[Last](#)

Portlet Plugin	Trusted	Tags	Installed Version	Available Version	Modified Date ▲
<a href="#">Add New User Wizard 5.2.2.1</a> ID: robisoft/03/5.2.2.1/war A plugin portlet using Spring Wizard	No	sample, spring, wizard	-	5.2.2.1	3/22/09 1:24 AM
<a href="#">WOL 5.2.2.1</a> ID: liferay/world-of-liferay/5.2.2.1/war This is the World of Liferay portlet that integrates social networking features.	Yes	wol	-	5.2.2.1	2/27/09 5:49 PM
<a href="#">Web Form 5.2.2.1</a> ID: liferay/web-form/5.2.2.1/war This is the Web Form Portlet.	Yes	web form	-	5.2.2.1	2/27/09 5:49 PM
<a href="#">Mail 5.2.2.1</a> ID: liferay/mail-portlet/5.2.2.1/war This is the Liferay Mail portlet.	Yes		-	5.2.2.1	2/27/09 5:46 PM
<a href="#">Google Maps 5.2.2.1</a> ID: liferay/google-maps-portlet/5.2.2.1/war This portlet allows easy integration with Google Maps.	Yes	google	-	5.2.2.1	2/27/09 5:45 PM

Figure 1.10 Browsing Liferay's plugin repository from within the control panel. Installing any plugin is a simple matter of clicking on the plugin and then clicking the *install* button that appears with the full description of the plugin.

As you can see, a lot of functionality is built in to Liferay Portal, and it is also extremely easy to add functionality to Liferay Portal. So you can rest assured that the software you create on Liferay's platform

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

will be easily installed by your users. Let's take a step back now so we can see what we have so far accomplished with just a few clicks.

### **1.2.5 So what has this little exercise accomplished?**

Hopefully you see the power that Liferay gives to you. In about ten minutes—and without any additional software—we've created a web site that contains web content, forums, a wiki, displays users' activities, shares documents, and has a custom look and feel. We didn't have to get separate applications to do all of that—instead, all that functionality (and more) is already included in Liferay. And because we didn't have to use separate applications to implement what we wanted, we didn't have to spend any time integrating those applications. Users get the experience of being able to sign into your site once and then navigate to the content they have access to, and you don't have to do *anything* to make that work.

Pretty awesome, isn't it?

Obviously, this only scratches the surface. You're going to need ways of organizing and granting permissions to all those users you're going to have. In order to do this, you'll need to understand the reinforcement beams, foundation blocks, and structures Liferay gives you to support that portal full of users.

## **1.3 How Liferay structures a portal**

Every portal is different in the way users, security, and pages are handled. Because these aspects of a portal are not covered by the JSR-286 standard, every portal vendor has implemented these concepts differently. So if you are going to start developing on Liferay's platform, you'll need to understand how a Liferay portal is configured and organized. Don't worry: it's not all that complicated, though it may look that way at first. Once you start using the system, you'll get the hang of it very quickly.

In this section, we'll see how you can collect users into various categories and what those categories can do for you. We'll also see how Liferay makes it easy to create web pages in your site and how content is placed on them.

### **1.3.1 The high-level view**

At its most basic level, a Liferay server consists of one or more portals. Portals have users, and these users can be categorized into various collections. Some of these collections can also have web pages that compose a portion of your site.

You can define many portals per portal server, and each portal has its own set of users and user collections. Figure 1.11 displays this graphically.

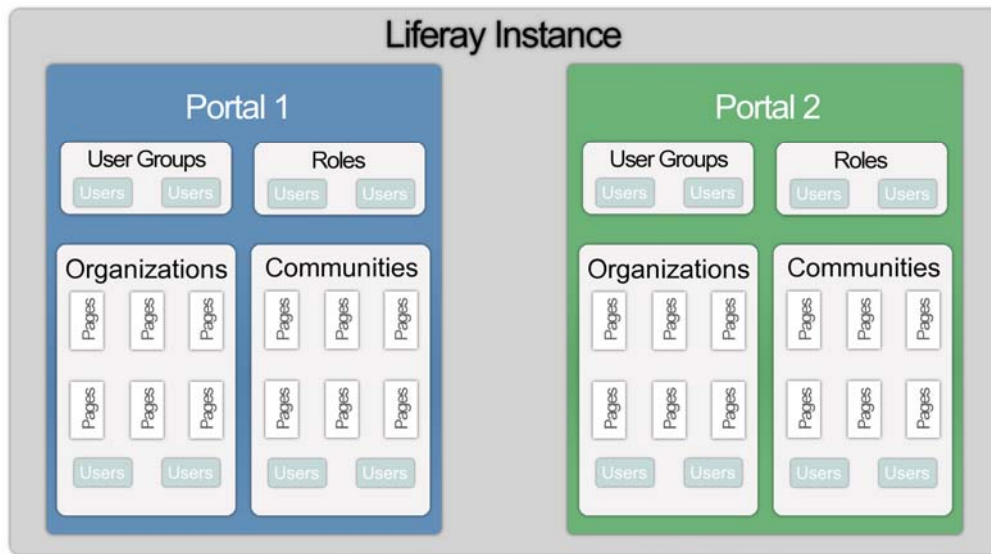


Figure 1.11 A single Liferay Portal installation can host many different portals, all with separate users and content.

As shown in figure 11, each portal has users, and those users themselves can be organized into several different types of collections: Roles, Organizations, Communities, User Groups, or any combination of those collections within that portal. Table 1.1 lists the collection types Liferay offers.

Table 1.1 Liferay Collection Types

Collection Type	Description
Role	Collects users by their function. Permissions in the portal can be attached to roles.
Organization	Collects users by their position in a hierarchy. Organizations can be nested in a tree structure. You would use organizations to represent things like a company's organizational chart.
Community	Collects users who have a common interest. They're single entities and can't be grouped hierarchically. By default, users can join and leave communities whenever they want, though you can administratively change it so that users are assigned to communities (or invited) by community administrators.
User Group	Collects users on an ad hoc basis. Defined by portal administrators

- Roles are inherently linked to permissions. You'd use a role to collect users who have the same permissions. A good example of this would be a Wiki Administrator role. This role would contain users who have permission to administer wikis.
- Organizations are hierarchical collections of users. Users can be members of one or many of them, up and down the hierarchy. Membership in organizations gives users access to the pages

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

of that organization. If you picture a hierarchical structure that represents a company called Inkwel, a user might belong to Inkwel, Sales Department, Mid-Atlantic Region. This would not only denote that employee's position in the company, but it would also give that employee access to the content he or she needs to do his or her job.

- Communities are ad hoc collections of users. Users can join and leave communities, and membership in communities gives them access to the pages in the communities of which they're members. You might have a community called Photography. Users of your site could join this community to share pictures.
- User Groups are defined by portal administrators. They can be used to collect users for purposes that tend to cut across the portal. For example, you might want to grant some users the ability to create a blog on your site. You would then create a User Group called Bloggers and create a page template for them that contains Liferay's Blog portlet. Regardless of these users' membership in other collections (as part of a hierarchy of organizations or as having joined several communities), User Groups provide a separate way of granting specific access to functions that don't depend on membership in other collections or on specific portal permissions.

That's the high-level view of a Liferay portal structure. While this describes a very powerful system for building your web site, it's only the basics. So let's move on to the next level.

### **1.3.2 Adding content to a collection with pages**

Three types of collections can have not only users, but also pages. Pages are, of course, clickable, navigable web pages. Organizations and Communities can have any number of pages defined within them. Pages are organized into *Layouts*, and there are two types of Layouts: Public and Private. So each Organization or Community can have public pages, allowing them to configure a public web site which can be used by members and non-members of the Organization or Community. And they can also have private pages, which are only accessible by the members of the Organization or Community. So you can begin to see how you can build out your site and separate functionality out by whoever is accessing the site.

User Groups don't have pages *per sé*, but rather can have *Page Templates*. These are configured by portal administrators, and become useful for users' personal communities. By default, each user gets a personal community, which itself has public and private layouts. This is a personal web site which the end user can configure (or which can be fairly static—or not exist at all, depending on how you have set up the portal). Portal Administrators can create Page Templates for User Groups. These Page Templates can be populated with the portlets that administrators want users to have. When users are then placed into the User Group, any Page Templates are copied into those users' personal communities. So if, for example, you want certain users to have a Blog, you might create a Blog page with the Blogs portlet on it in a User Group called Bloggers. Any user you add to this user group would have this page copied automatically to his or her personal community, and he or she can begin blogging immediately.

If you haven't already figured it out, a Roles collection has no pages because roles are used solely to define permissions. For example, you could define a role which has permission to view certain pages. This is how roles work together with organizations, communities, and user groups.

Liferay Portal also has the idea of *scope*, the topic of our next discussion.

### 1.3.3 Configuring a portlet's scope

Scope allows some of the concepts mentioned above to be refined. One user collection that is refined by Scope is Roles. As stated above, Roles are the only collection to which permissions can be attached. So you can create a Role called Wiki Administrator. This role would have permissions to the Wiki portlet, allowing users in this role to create new wikis and add, edit, delete, and move pages. This role can be created under one of two scopes:

- Portal Role
- Community/Organization Role

If you created this role as a Portal Role, then any members of this role would have the defined permissions across the whole portal, in any community or organization. So this would allow users in this role to administer Wikis in whatever communities or organizations they have access to. You can, however, define the role in another way, by scoping it only by community or organization. If the role were defined this way, then users would have the role's permission in only the community or organization in which that role was defined. So scope is very important when it comes to how permissions are defined.

Scope also comes into play with regard to certain of Liferay's built-in portlets. If you go back up to the Dockbar and click *Add > More*, you'll see that the portlets are marked with different icons. These icons tell you something about the portlets with which they are associated. But before we go over what they mean, let's take a look at some portal terminology first.

Sometimes I feel like Dr. Seuss when beginning to discuss this topic:

*If a portlet in a portal on a page in an org,  
Has a data-set saved as its own data-store,  
And the data would be different for other users' chores,  
We call that a non-instanceable portlet!*

*And...*

*When a portlet in a portal saves its data on the disk,  
And the user hits the data based on membership in this,  
If the portlet is configured to have its own instances,  
We call that an instanceable portlet!*

What I mean by this all has to do with scope.

#### NON-INSTANCEABLE PORTLETS

Let's stick with the Wiki example. If you place a Wiki portlet on a page, based on what I've described above, where is that page? Yes, you are correct: it's in a Community or Organization. That Wiki now belongs to that community or organization.

I cannot place another Wiki portlet on the same page, because that portlet is what Liferay calls *non-instanceable*. In other words, another instance of that portlet cannot be added to the community or organization: it is *scoped* just for the membership of that community or organization.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

I can place another Wiki portlet on a different page in that community or organization, but that Wiki portlet will simply display the same data as the first one. In other words, it will still be the same portlet instance (see figure 1.12).

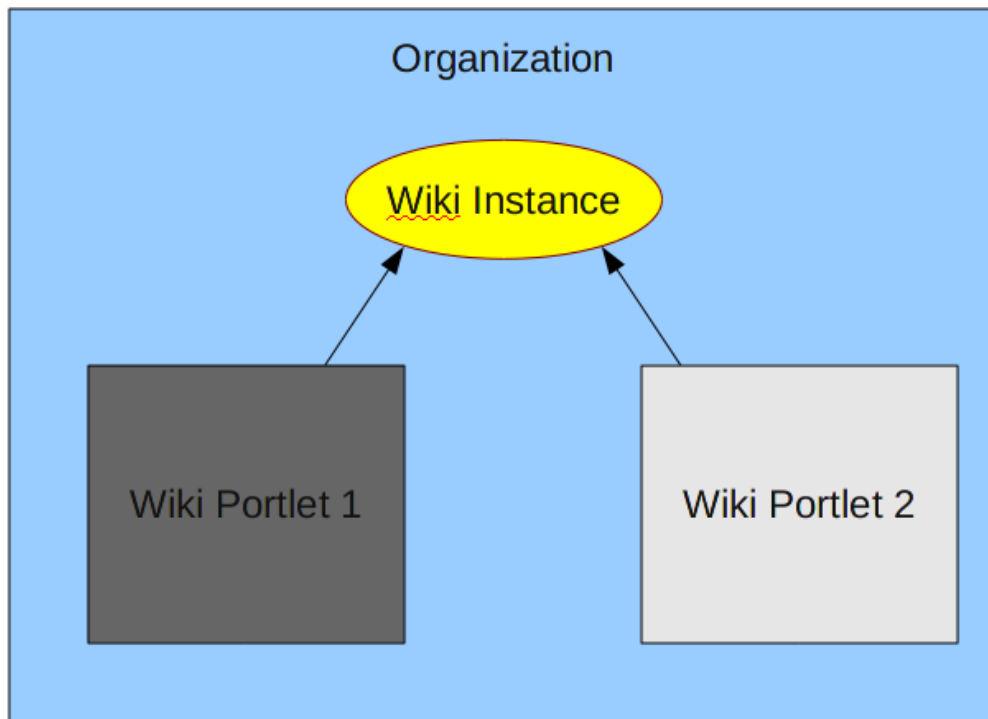


Figure 1.12 A non-instanceable portlet has its data scoped by the community or organization to which it belongs. No matter how many times you add it to a page with the community or organization, it will point to the same data.

So for non-instanceable portlets, you get one instance of that portlet per community or organization.

This may seem like a limitation, but it is actually a powerful benefit. You can have lots of Wikis on your site, and they can all be kept completely separate from the others. For example, say you are building a web site for Do It Yourself-ers. Your audience likes to build stuff, but the “stuff” they want to build differs wildly. So your site has communities for many different topics, including topics on home renovation all the way to hobby-like topics, like building model rockets or platforms for model railroads. To serve the needs of these users, you might want to give them a Wiki so that they can add helpful tips and articles based on their experiences. But the model rocket group and a home improvement plumbing group are not going to have very much in common (or maybe they will, depending on the size of the rocket—but that's not what we're focusing on right now). So you can give them separate Wikis in their own communities very easily with Liferay. And, of course, because of Liferay's powerful way of collecting users into Communities, all of your users will be members of your site (i.e., the portal), but not

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

necessarily of the same Communities. So they will have the freedom to navigate to the content that is most appropriate for them.

#### INSTANCEABLE PORTLETS

Other portlets in Liferay are *instanceable*. This means that as many of them can be placed on the same pages in any community or organization as you would like, and they all have their own sets of data. For example, the RSS Portlet is designed to show RSS feeds. You can add as many RSS portlets as you want to any page and configure each portlet to display different feeds, because this portlet is instanceable. The Web Content Display portlet is the same way: you can place as many Web Content Display portlets on a page as you wish, and each portlet can display a different piece of web content. When picking portlets from Liferay's Add > More window, the interface shows which portlets are instanceable and which are non-instanceable, as show in figure 1.13.

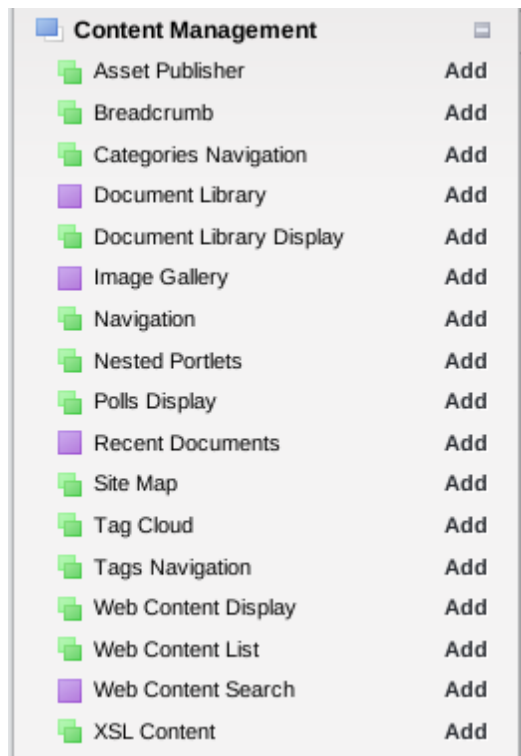


Figure 1.13 Instanceable and non-instanceable portlets in Liferay's Add window. Liferay's UI clearly shows you which portlets can be added to the same page and yet have different data (instanceable) and which cannot (non-instanceable).

You can tell which portlet is which in the user interface by looking at the icons in the Add > More window. If there's a green icon with two windows, the portlet is instanceable. If there's a purple icon with one window, the portlet is non-instanceable.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>



## PAGE SCOPES

Sometimes, however, Liferay's default scopes need to be enhanced with more flexibility. So I'm going to backtrack a bit on what I said above. If you *really* need to have two non-instanceable portlets with different data sets in your community, you can do that. It's just not available by default (and wasn't available at all in older versions of the product). This has to be configured on a per-portlet basis.

Still using the Wiki portlet as an example, if you click the configuration icon in the portlet window (which looks like a wrench in the default theme), a menu will pop open. Click *Configuration* in this menu and all the configuration options for this portlet will be displayed. One of the tabs in this window is called *Scope* (see figure 1.14).



Figure 1.14 Changing the scope in the Wiki portlet.

Here, you can change the scope from the default to the current page. This lets you turn a non-instanceable portlet into a portlet whose instance is tied to the page instead of the community or organization. What this means is that you can add another page to this community or organization and place *another* Wiki portlet on that page. Once that's done, you can set that portlet to have either community/organization scope or page scope. And so on. You won't be able to add multiple Wikis to the same page, but this lets you have multiple non-instanceable portlets per community or organization, provided the portlet supports page scopes.

I don't want to delve too much into these concepts at this stage. You'll be taking advantage of scope soon enough in your code. For now, just let it all sink in, and let's turn to something more concrete: how to navigate around Liferay.

### 1.3.4 Getting around in Liferay

Liferay's user interface has a philosophy behind it: get out of the way of the user. For that reason, it hides a lot of power behind what looks like a very simple interface. One of the main UI elements is the Dockbar.

You have already been introduced to some of the functionality of the Dockbar, so let's see what other functions it provides. Figure 1.15 shows the Dockbar in full.

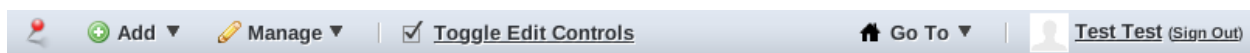


Figure 1.15 Liferay's Dockbar, which appears at the top of every page when a user is logged in.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

### PIN ICON

At the far left is a pin icon, which does what you would expect it to do: It pins the Dockbar to the screen so that no matter how far down you scroll, it stays at the top of the screen. This can be helpful if you're working with long pages and need to use the Dockbar's functionality to add portlets to the bottom of the screen. This is a toggle switch, so you can unpin the Dockbar by clicking the icon again.

Next in the Dockbar is the Add menu.

### ADD MENU

You've already seen most of the functionality of this menu for adding applications to the page. It can add pages too. If you click Add > Page, a new page will be added next to the page you're on, and a field will appear, allowing you to name the page. There's a much more powerful page administration screen, but this function allows you to quickly add pages to your web site as you're working on it.

The next item in the Dockbar is the Manage menu.

### MANAGE MENU

Use the Manage menu to manage pages, page layouts, and more. This is where you get access to the interface, which lets you group your pages in the order you wish as well as nest them into subpage levels. You can also apply themes to whole layouts or to single pages. The Manage Pages screen is shown in figure 1.16

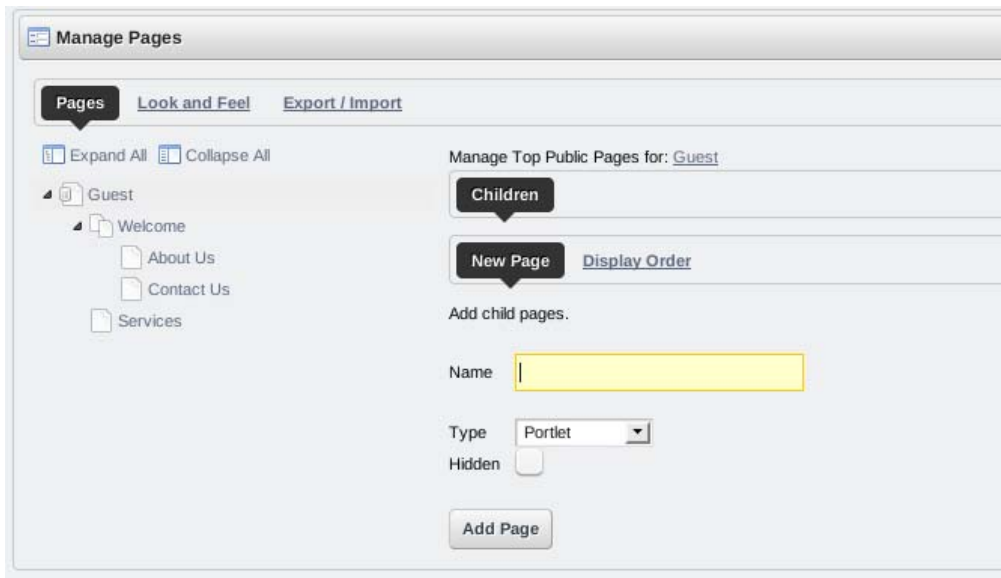


Figure 1.16 The Manage Pages screen allows you to nest your pages, change the display order by dragging and dropping them, change themes, and more.

Perhaps the most important item in the Manage menu, however, is the Control Panel.

Liferay's Control Panel is the central location where just about everything can be administered. The control panel is very easy to navigate. On the left side is a list of headings with functions underneath them. The headings are in alphabetical order, but the functions are in a logical order. Figure 1.17 shows

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

the Control Panel, which purposefully uses a different theme from the default pages, so you can instantly tell where you are.

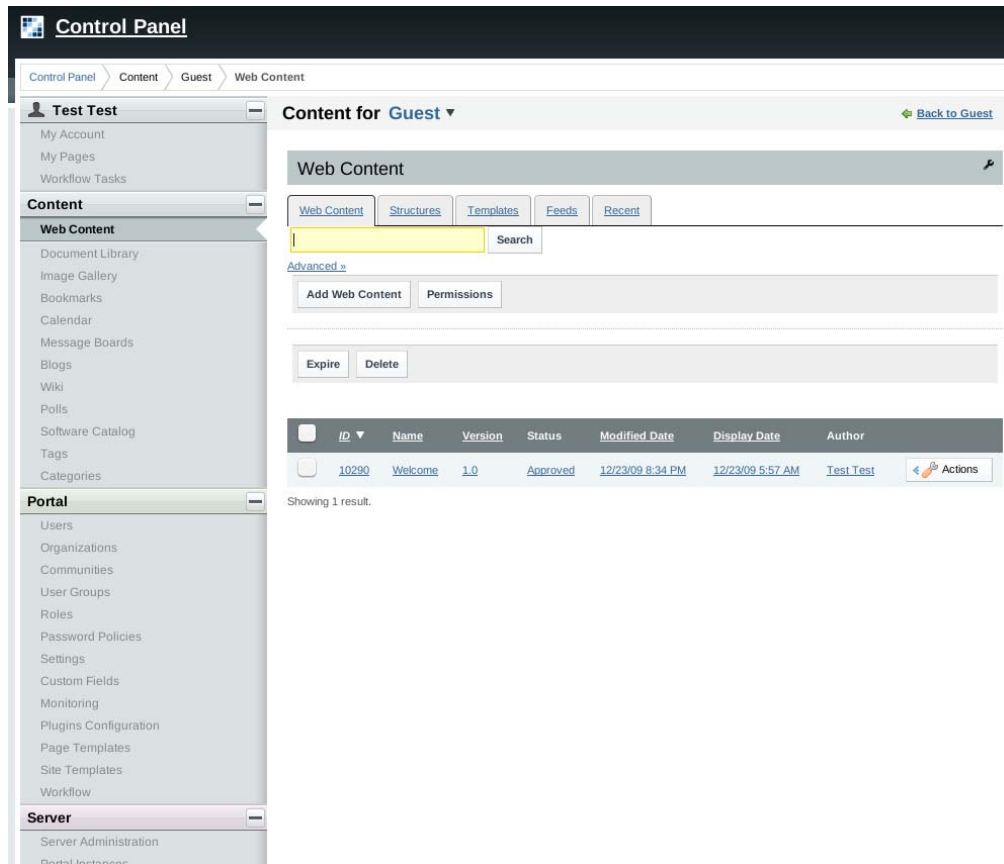


Figure 1.17 Liferay's Control Panel, with Web Content selected. Notice that the content is scoped by the Guest community, but this can be changed, depending on the user's privileges.

- **User Name**—The first heading is named for the logged-in user (Test Test in figure 14) and is used to manage the user's personal space. Here, you can change your account information and manage your own personal pages.
- **Content**—The Content section contains links to all of Liferay's content management functions. You can maintain web content, documents, images, bookmarks, and a calendar; administer a message board; configure a wiki; and more. These links are scoped for the particular community from which you navigated to the Control Panel, but this can be changed using the select box at the top of the screen. Figure 14 shows the Control Panel with Web Content displayed.
- **Portal**—The Portal section allows portal administrators to set up and maintain the portal. This is where you can add and edit users, organizations, communities, and roles as well as configure the

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

settings of the portal.

- **Server**—The **Server** section contains administrative functions for configuring portal instances, plug-ins, and more.

### TOGGLE EDIT CONTROLS

The next function in the Dockbar is not a menu; it's a toggle for the edit controls on the portlets. As an administrator, you get to see some icons in the title bars of the portlets on a page. These correspond roughly to the icons you might see in your operating system. There's an icon for closing a portlet, minimizing it, and for the configuration menu which you have already seen (we used this to change the scope of the Wiki portlet). If you are composing a page and would like to see something that more closely resembles what your users will see, you can use the Toggle Edit Controls link to turn off these controls.

Toward the end of the Dockbar is the Go To menu (shown in figure 1.18).

### GO TO MENU

Use the Go To menu to navigate to the various Community and Organization pages to which you have access. Each page name appears, along with its public and private layouts, if they have them.

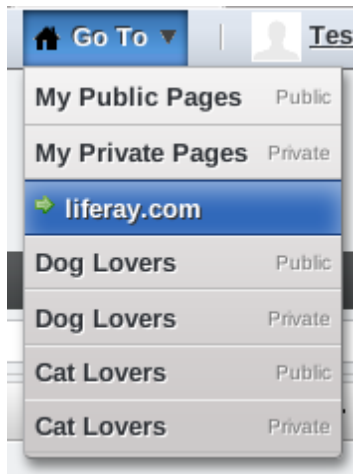


Figure 1.18 The **Go To** menu displaying public and private layouts for three communities: the default Guest community, Dog Lovers, and Cat Lovers. Notice that the default community only has public pages, so only one link appears.

The final link in the Dockbar will take you to your user account information in the Control Panel.

### USER ACCOUNT

The User Account menu item opens a page where you can change your name and email address, upload a profile picture, and maintain all information about you. You can also sign out of the portal from here.

As you can see, Liferay packs a lot of power in a deceptively simple user interface. The intent of this small tour was to give you an idea of where you can go in Liferay and how to get there as you begin to build your site.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

Even though we've now touched on several of the constructs that provide you with the building blocks you'll use to build a web site in Liferay, it's sometimes difficult to begin imagining how your site could be built using these building blocks, because many of the concepts are new and unique to Liferay. So let's spend a little time figuring out how you can imagine your site running in Liferay Portal.

## 1.4 *Imagining your site in Liferay*

Every successful web site does something unique, or does something in a way that is better than anyone else has done it before. While Liferay has tons of functionality out of the box, much of that functionality is a default implementation of features that are under the hood. What do I mean by that? Let me answer by giving you some examples.

Liferay portal has a collaboration API which contains features allowing users to post discussions, rate items, or tag content. This API has been used to provide everything from the Message Boards portlet to tagging wiki articles, to rating shopping cart items. This book will introduce you to these APIs, so that you can consider what kinds of applications *you* can build with these powerful features.

That, of course, is not the only API we'll cover. You'll also see Liferay's Social API, which gives you the ability to make your applications—indeed, even your entire web site—social. Your users will be able to connect with each other and share content and activities, and even share content and applications on other social networks. Again, the question remains: what will *you* do when given the power to build such applications?

The point of all of this is that when you're finished reading this book, you'll have the ability to make Liferay sing to *your* tune. And because there's so much power in the Liferay platform, you'll get a head start on building your site because the functionality you need is already built into the platform—all you need to do is implement it.

We'll go from the ground up in familiarizing you with Liferay development throughout the course of this book. For now, let's use the information we already have to begin imagining your site from within the Liferay constructs described in the preceding sections. Then later, as you see the full power of Liferay's development platform, you'll see how easy it is to use Liferay as the foundation of your web site, and you can plan how to integrate the features of your applications with the power of the platform.

Portal design is best done by breaking up your site into small chunks and then designing each chunk individually. That way, you don't get overwhelmed by the largeness of your task, and before you know it, breaking it up into smaller chunks has enabled you to design the whole site!

In this section, we'll walk through a design process that is based on a set of forms that I've used with success to design many portals.

### **TIP**

For your convenience, the portal design forms can be found in Appendix B. Tear them out or duplicate and then fill them out as you work through this section.

We'll break out the design process into three main portal chunks:

- User Groupings
- Organizations and Communities
- Content

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

### **1.4.1 Asking the right questions**

The first thing we want to do is figure out how we can get all of your ideas divided up into neat, organized chunks that can then be focused in on in more detail. Ask yourself the following questions:

- Will users be given freedom to sign up on the site?
- Will your user groupings be ad-hoc, static, or both? (If your user groupings will be ad-hoc, you know you'll be creating communities for your users to join and leave.)
- Will some regular users have access to things others won't? (If so, you know you'll be using Roles.)
- Will you be delegating administrative tasks to some users? (If so, you may have Community or Organization Administrators.)

Once you've answered these questions, go ahead and brainstorm the groupings or collections of users you may have.

### **1.4.2 Defining and categorizing collections**

Don't worry about trying to define them as User Groups, Communities, Organizations, or Roles. Start figuring out some groupings. Some examples are anonymous visitors (potential customers), customers, community members, and specific groupings based on your web site. For example, if you're building a web site for do-it-yourselfers, you might come up with categories such as carpentry, plumbing, model rocketry, or even old computers.

At this point, you should have a good list of your groupings. Now combine that list with what you answered to the previous questions. Will any of the groups require pages? If so, you know which ones are Communities or Organizations. Are the groupings associated in any way? If so, how? You're now beginning to identify a possible organization hierarchy.

Are there some groupings that cut across the entire portal (such as a bloggers group)? If so, that's a likely candidate for a User Group, and you can begin thinking about whether these users should have page templates defined for them. Or it may be a good candidate for a Community, if the grouping should have its own set of pages. Once you've categorized your collections of users by Organizations, Communities, User Groups, and Roles, you can begin designing your content.

### **1.4.3 Designing content**

Pages can be part of Organizations or Communities. By default, each can have public pages, which everyone can see and private pages for authenticated members of that Organization or Community. Take each Organization and Community you've identified, and determine the page hierarchy that will exist for each one. This may even help you to further define your Roles and User Groups.

When you are finished with this process, you should have a nice, high-level design for your web site. You may have something very simple, like Liferay's default: one community called Guest for everyone to use. Or you may have something more complex. The point is, it's a start. From here, we can delve into the custom applications you need to write to make your site unique, as well as the customizations to Liferay that you need to make to satisfy all of your requirements. That's what the rest of the book is all about.

## 1.5 Summary

Liferay Portal is an ideal choice for building your web site. Using the unique constructs that the platform gives you, you can design a site that can handle any situation you can throw at it. Liferay Portal also offers you an unbeatable platform for building your web applications, as well as a ton of applications that are already implemented, in order to help jump start the creation of your site.

In addition, Liferay Portal frees you from the limitations of the old Java portal standard. As an open source project, it enables you to be as lightweight or as heavyweight as you want to be. And because it provides a multitude of tools and utilities for increasing developer productivity, you'll be able to get your site done faster.

Liferay gives you a powerful paradigm for organizing your users and getting them access to the content they want to see. You can use Communities, Organizations, Roles, and User Groups to make sure that the right content gets to the right people and that restricted content is protected so that only the proper users can view it.

Because Liferay is so easy to use, you can create complex web sites quickly. Because all of the common applications you need to run a web site are already included, it's a simple matter to pick the applications you need and drop them onto your pages. Because no further work is needed to integrate these applications, your time is freed up to focus on the applications you need to build that are unique to your web site.

As we move further into this book, you'll learn how to customize Liferay to make it look the way you want it to look, act the way you want it to act, and host the applications that you design and write. This is going to be an interesting journey for us, and I'm sure you'll find it as rewarding as I have. I hope you'll come along and take the red pill with me—it's going to be an exciting ride.

In the next chapter, we'll get Liferay Portal 6 installed, unpack and configure the Plugins SDK, and dive into creating our first portlet application.

# 2

## *Appray: development at the speed of light*

This chapter covers

- Installing a Liferay bundle and setting up a database
- Generating plugin projects
- Writing your first portlet

Liferay provides you with an extremely powerful development platform which allows you to do everything from providing your own portlet applications to customizing the platform's core functionality. You probably have all kinds of ideas of what you want to do with your web site: what your users' first experience should be, how they will interact, and even mundane things like what the registration process will be like. You have the full ability to define these features any way you want to with Liferay, but you need to understand where and how this is done before you start implementing your site. So I want to give you some direction as to where to start and how to proceed. But first things first: we most definitely need to get Liferay installed before we start developing anything on it. Then we need to get the Plugins SDK installed so we can start generating projects. So let's get to it!

### **2.1 Installing Liferay stuff**

Liferay Portal is extremely easy to install, no matter which operating system you use. I don't want to get into operating system wars here, though I do, of course, have a preference (doesn't everybody?) for what I use every day. So for the purpose of this book, I will be operating system agnostic, as Liferay is, which benefits everybody.

The first thing you need to do before trying to install Liferay Portal is to make sure that you have the Java SDK installed. This is the *JDK*, or the Java Development Kit, not the *JRE*, which is the Java Runtime Environment. Why do you need the JDK? Because you'll be doing *development*, silly.

You need to do more than just install the JDK. On most—if not all—operating systems, the JDK install process does not set up one of the most important things for you: the `JAVA_HOME` environment

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>



variable. So you'll have to do this yourself. I would tell you how, but I'm being operating system agnostic.

Okay, I guess that wasn't very helpful. In order to explain this, then, I will have to define some rules for operating system agnosticity which are designed to keep everyone happy. This means that not everybody will be happy, of course, but I'm hoping you'll accept the trade of happiness for fairness.

- **Rule 1:** If something is done the same way in all operating systems, I will explain it only once.
- **Rule 2:** File paths will be denoted by forward slashes (/), because more operating systems use that than anything else.
- **Rule 3:** Operating systems will be presented in alphabetical order. This means that (L)inux, (M)ac, and (U)nix all beat (W)indows. This may make Windows users unhappy, but for fairness, see the next rule.
- **Rule 4:** Operating systems of the same family will be presented together, unless there is some difference between them that requires explanation (there usually isn't). For this reason, I will generally lump Linux, Mac, and Unix together (notice that they themselves are alphabetized?). Sorry guys, you get to go first, but I'm giving you a new name: LUM. Why LUM? Well, I only had one vowel to work with, and LUM sounds better than MUL, UML stands for Unified Modeling Language, and LMU is unpronounceable.

Okay; so first install the Java Development Kit. This should be installed by default on your Mac. On Linux, it will be available in your distribution's package manager. And on Unix and Windows, you will need to download it from Oracle.

Next you need to set that pesky `JAVA_HOME` environment variable.

This is fairly easy for LUM users: first you need to know where your JDK is installed, and then you edit a hidden file in your home directory called `.profile` and set the variable to that location. Here's a sample of what that might look like:

```
JAVA_HOME=/usr/lib/jvm/java-6-sun
export JAVA_HOME
```

If you are using a non-default shell such as `cs`h or `ts`h, you won't need that second line, and you should precede the first with `setenv`. Of course, if you are using `cs`h or `ts`h, you probably already know that.

On Windows, you need to navigate through the Control Panel to set environment variables. Windows 7 / Vista users need to navigate to Control Panel > System > Advanced Settings > Advanced tab > Environment Variables (think they hid it well enough?). Windows XP users can navigate to Control Panel > System > Advanced tab > Environment Variables. Once you get here, the dialog box looks the same (figure 2.1).

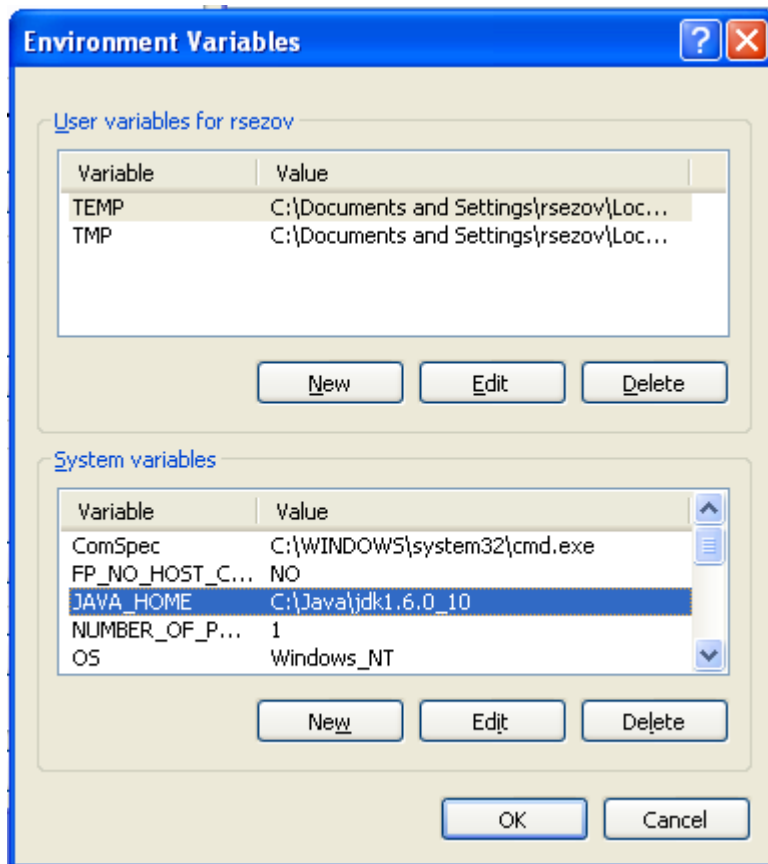


Figure 2.1 Use this dialog box to set environment variables on Windows operating systems.

Click the *New* button under System Variables and create a `JAVA_HOME` variable which points to the location where you installed the JDK. Then click *OK*.

You've now got a Java Development Kit installed. You can now begin all kinds of development using Java, but of course we're going to focus on Liferay here. Because of that, the next thing we need to do is get Liferay installed.

### 2.1.1 Installing a Liferay bundle

Once you have your Java Development Kit all set up, you've made it to the easy part: installing Liferay. This is done in two steps: unzip the archive and edit a text file—easy, right? Liferay Portal can be installed on a wide variety of application servers and also comes bundled with a number of open source application servers. You can choose among Glassfish, JBoss, Jetty, JOnAS, Resin, or Tomcat, and you should certainly use whichever bundle is right for your environment or organization. If, however, you don't know which one to choose, I recommend using the Liferay-Tomcat 6.0 bundle, as Tomcat is small, fast, and takes up less resources than most other containers. Any supported container is fine, however,

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

so you can use the container that is best for your organization. We will be using the Tomcat bundle for this book.

First, let me recommend a way of keeping all your code organized. I always create a folder which we'll denote in this book as [Code Home]. Call this folder whatever you want to call it, and stick it wherever you want to stick it. Windows users, I recommend that you put this folder in the root of your drive. Why? Because Windows' file system, NTFS, has a limitation on the total number of characters that can make up a path. Though you can nest folders as deeply as you want, the entire path can be comprised of no more than 256 characters. This means that if you put your workspaces in a folder like `c:\Documents and Settings\Administrator\Java`, you have already wasted 44 characters out of your total of 256. Because of Java's package naming conventions (consider Liferay's package `com.liferay.portal.security.permission.comparator`, which is already inside a folder structure of `portal-impl/src`, as an example), you can very quickly create a path that is too deep for Windows to handle if you don't put your source code somewhere near the root of the drive.

Now that you have [Code Home] created, create a folder inside that called *bundles*. Download the latest Liferay-Tomcat bundle and unzip it to [Code Home]/bundles. You can start Tomcat by navigating to the `[Code Home]/bundles/[bundle home]/tomcat-[version]/bin` folder and running the startup command.

On LUM, this is  
`./startup.sh`

On Windows, this is  
`startup.bat`

Liferay Portal will start and your browser will automatically launch so that you can view your portal. By default, open source versions of Liferay Portal ship with a sample welcome page and web site already included, for a fictional company called 7cogs, which you can see in figure 2.2. This is a great way for you to click around and see how an actual implementation may work.



Figure 2.2 This is the sample web site that comes included with a Liferay bundle. It showcases many of the things that were described in Chapter 1, giving you a nice example of all of those elements (content management, forums, wiki, and more) all working together to implement a single web site.

You could leave Liferay configured this way, but this is not the most optimal configuration, as your portal is using an embedded database called HSQL to store all of its data. It's far better to use a real database. And you'll also want to make sure you start with a clean database, not one that already has a sample web site in it. So next we're going to take a page from the administration side of things and get your development server connected to a standalone, clean database.

### 2.1.2 A crash course in Liferay server administration

If you want to set up Liferay as a real server, you'll be far better served by checking out Liferay's documentation. Because we're only concerned with having a good development environment, we're only going to concentrate on connecting your Liferay bundle to a standalone database (hence, the "crash course" instead of the "full course"). This is generally for stability reasons: HSQL is great for demos and stuff like that, but if you're going to start doing development, it's far better to use a standalone database. This configuration more closely mirrors a production configuration, and it provides your data with a bit more stability, as the database is not running in the same process as Liferay. You may also

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

want to use the data querying tools that your database vendor provides with your database. It also performs better, because a database server is designed to have multiple connections to it at the same time. And finally, rather than storing the data inside the bundle, the data is stored with your database server, keeping it separate from your Liferay installation.

### SETTING UP A DATABASE

Here's the heart of the crash course. What follows is the same exact procedure you would use to set up Liferay as a real server. The only difference is that you'll be doing all of this locally on your machine, while in production you would likely have your database and your Liferay installation on separate machines.

As I had a recommendation for which bundle to use, similarly I recommend that you use MySQL for this purpose, as it is small, free, and very fast. It's also very easily obtained: on Linux, it's available in your package manager. If you're on Mac or Windows, it is easily downloaded and installed.

Again, if you use a different database, there is no reason not to use that database as long as you have the resources to run it. Liferay supports all of the widely-used databases in the industry today.

To install MySQL and its utilities, you will need four components: *MySQL Server*, *MySQL Query Browser*, and *MySQL Administrator*. The first component is the server itself, which on Windows will get installed as a service. The second component is a database browsing and querying tool, and the third is an administration utility that enables the end user to create databases and user IDs graphically. If you are running Windows or Mac, download these three components from MySQL's web site (<http://www.mysql.com>).

Once you have a running MySQL server, you will want to do two things: set the root password and create your database. By default, MySQL does not have an administrative (root) password set. You should definitely set one. To do this, drop to a command line and issue the following command:

```
mysqladmin -u root password NEWPASSWORD
```

Instead of NEWPASSWORD, you would, of course, type the password that you want (maybe 1337h4x0r). Next, you can start the MySQL command line utility via the following command:

```
mysql -u root -p
```

Once you launch it, it will display some messages and then a MySQL prompt:

```
Welcome to the MySQL monitor.  Commands end with ; or \g.
```

```
Your MySQL connection id is 119
```

```
Server version: 5.1.37-1ubuntu5 (Ubuntu)
```

```
Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.
```

```
mysql>
```

At the command prompt, type the following command:

```
create database lportal character set utf8;
```

MySQL should return the following message:

```
Query OK, 1 row affected (0.12 sec)
```

You will be back at the MySQL prompt. You can type *quit* and press enter, and you will return to your operating system's command prompt.

Note that on some Linux distributions MySQL is configured so that it will not listen on the network for connections. This is done for security reasons, but it prevents Java from being able to connect to MySQL via the JDBC driver. To fix this, search for your *my.cnf* file (it is probably in */etc* or */etc/sysconfig*). There are two ways in which this may be disabled. If you find a directive called *skip-networking*, comment it by putting a hash mark (#) in front of it. If you find a directive called *bind-address* and it is

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

configured to bind only to localhost (127.0.0.1), comment it out by putting a hash mark (#) in front of it. Save the file and then restart MySQL.

Liferay defines the folder it resides in as *Liferay Home*. The home folder is very important to the operation of Liferay. This folder is the top level folder which you extracted from the .zip file. Liferay creates certain resources that it needs in this folder. It contains some folders (*data*, *deploy*, and *license* if you're using Liferay Enterprise Edition), and there is a configuration file called *portal-ext.properties*, which you can place here to change some of the Liferay configuration. We will be working with this file to connect Liferay to your database.

Before you connect Liferay to your database, however, you'll need to remove the sample website if you're using Liferay Community Edition so that you can start with a fresh, clean installation like the one you saw in Chapter 1. This is extremely easy to do: all you have to do is undeploy them. If you're using Liferay Enterprise Edition, you can skip this step.

### REMOVING THE SAMPLE WEB SITE

To undeploy an application in Tomcat, all you need to do is navigate to the folder where the applications are stored and delete the folder that contains the application. In a Liferay-Tomcat bundle, Tomcat is located in [*Liferay Home*]/*tomcat-[version number]*. Inside of this folder is a folder called *webapps*, which is where Tomcat stores the applications that are installed. Go into this folder and you will see a list of folders containing Liferay and various plugins.

The one you want to delete is called **sevencogs-hook**. Remove this folder by either deleting it or moving it to another location on your system. That's all you need to do to prevent the sample web site from being created when Liferay first starts. Make sure that you do this any time you're setting up Liferay for development or as a real server, as you always want to start with a clean database. And speaking of databases, now that we've removed the sample website, we're ready to connect Liferay to our MySQL database.

### CONNECTING LIFERAY TO THE SQL DATABASE

To point your Liferay bundle to your database, create a file called *portal-ext.properties* in your Liferay Home folder. This file overrides default properties that come with Liferay. You are going to override the default configuration which points Liferay to the embedded HSQL database.

To connect your installation of Liferay Portal to your database, add the appropriate template for your particular database to your newly created *portal-ext.properties* file. The template for MySQL is provided as an example below.

```
#
# MySQL
#
jdbc.default.driverClassName=com.mysql.jdbc.Driver
jdbc.default.url=jdbc:mysql://localhost/lportal?useUnicode=true&characterEncoding=UTF
-8&useFastDateParsing=false
jdbc.default.username=
jdbc.default.password=
```

You would provide the user name and password to the database as values for the *username* and *password* directives. If you're using a different database, you'll find templates for the other databases in Liferay's documentation.

For a production machine, you would also generally connect Liferay to a mail server so that it can send mail notifications. Since on a developer machine it is likely that there is no mail server running, this step is not necessary.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

Save the file. You can now start your application server.

If you're JDBC-savvy, you'll notice one thing is missing from the instructions above: the JDBC driver for MySQL. Liferay includes it already for convenience, so you don't have to worry about downloading it and making it available to your Liferay bundle. You'll find it in the Tomcat bundle in [Tomcat Home]/lib/ext/mysql.jar. If you are using a different database, you'll need to copy your database's JDBC driver to this folder before starting Liferay.

Now that your Liferay bundle is set up, we can move on to installing the Plugins SDK.

### 2.1.3 Setting up the Plugins SDK

Building portlet and theme projects in the Plugins SDK requires that you have a utility from Apache called Ant installed on your machine. This utility is a scriptable build tool, and is what the Plugins SDK uses to provide its IDE-agnosticity. If you already have this installed or if it's built-in to the IDE you will be using (it is on every IDE I can think of), you can skip the next section.

#### INSTALL ANT

Download the latest version of Ant from <http://ant.apache.org>. Uncompress the archive into an appropriate folder of your choosing.

Next, you'll have to set two more environment variables like you did above. The first one is called ANT\_HOME, and it needs to point to the folder to which you installed Ant. The second one is called ANT\_OPTS, and contains the proper memory settings for building projects. After you do this, there's one more step, because you'll likely want to be able to run this from the command line—especially if you're the command-prompt-plus-text-editor type of developer. This step adds the Ant binary to your PATH, so that you can call it from anywhere on the command line. You can use the ANT\_HOME environment variable as a quick way to add the binaries for Ant to your PATH.

You can do this on LUM by modifying your .profile file as follows (assuming you installed Ant in /java):

```
ANT_HOME=/java/apache-ant-1.7.1
ANT_OPTS="-Xms256M -Xmx512M"
PATH=$PATH:$ANT_HOME/bin
export ANT_HOME ANT_OPTS PATH
```

Log out and log back in to make these settings take effect.

You can do this on Windows by going back to that Environment Variables dialog we used before in section 2.x. Under System Variables, select *New*. Make the Variable Name ANT\_HOME and the Variable Value the path to which you installed Ant (e.g., c:\java\apache-ant-1.8.1, and click *OK*.

Select *New* again. Make the Variable Name ANT\_OPTS and the Variable Value "-Xms256M -Xmx512M" and click *OK*.

Scroll down until you find the PATH environment variable. Select it and select *Edit*. Add %ANT\_HOME%\bin to the end or beginning of the Path. Select *OK*, and then select *OK* again.

On any operating system, open a command prompt and type *ant* and press Enter. If Ant attempts to run and you get a "build file not found" error, you have correctly installed Ant. If not, check your environment variable settings and make sure they are pointing to the directory to which you unzipped Ant.

#### INSTALLING THE PLUGINS SDK

Now you're ready to install the Plugins SDK. This is even easier than installing Liferay.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

Go to your [Code Home] folder. You should already have a folder in here called *bundles*, where you installed Liferay. Next, download or otherwise obtain the Plugins SDK from Liferay. Unzip the file to [Code Home]/plugins. That's it! It's installed.

The Plugins SDK contains many subfolders. The two you'll be working in mostly are the *portlets* and the *themes* folders. It is here that you will place your portlet and theme plugin projects.

### 2.1.4 Configuring the Plugins SDK

You will notice that the Plugins SDK contains a file called *build.properties*. Open this file in a text editor. At the top of the file is a message, "DO NOT EDIT THIS FILE." I just wanted you to see that (pictured in figure 2.3). This file contains the settings for where you have Liferay installed and where your deployment folder is going to be, but you don't want to customize this file. Instead, create a new file in the same folder called *build.[username].properties*, where *[username]* is your user ID on your machine. This is the ID you log in with every day. For example, if your user name is cooldude, you would create a file called *build.cooldude.properties*.

If you're using the default setup I've described so far, you won't have to do very much to set up the Plugins SDK. Create your *build.[username].properties* file and put in the following line:

```
app.server.dir=[full path to Liferay bundle install]
```

In the place of [full path to Liferay bundle install], put the path to Tomcat in your Liferay bundle. For example, a Liferay bundle extracts into a folder of its own. Inside that folder is a folder for Tomcat. Use the full path to Tomcat for this directive: this is the *server* directory. This tells the scripts in the Plugins SDK where everything else is relative to your Liferay install, because all paths (such as the deploy path) are defined relative to the server directory. Save and close the file; you're ready to begin creating projects.



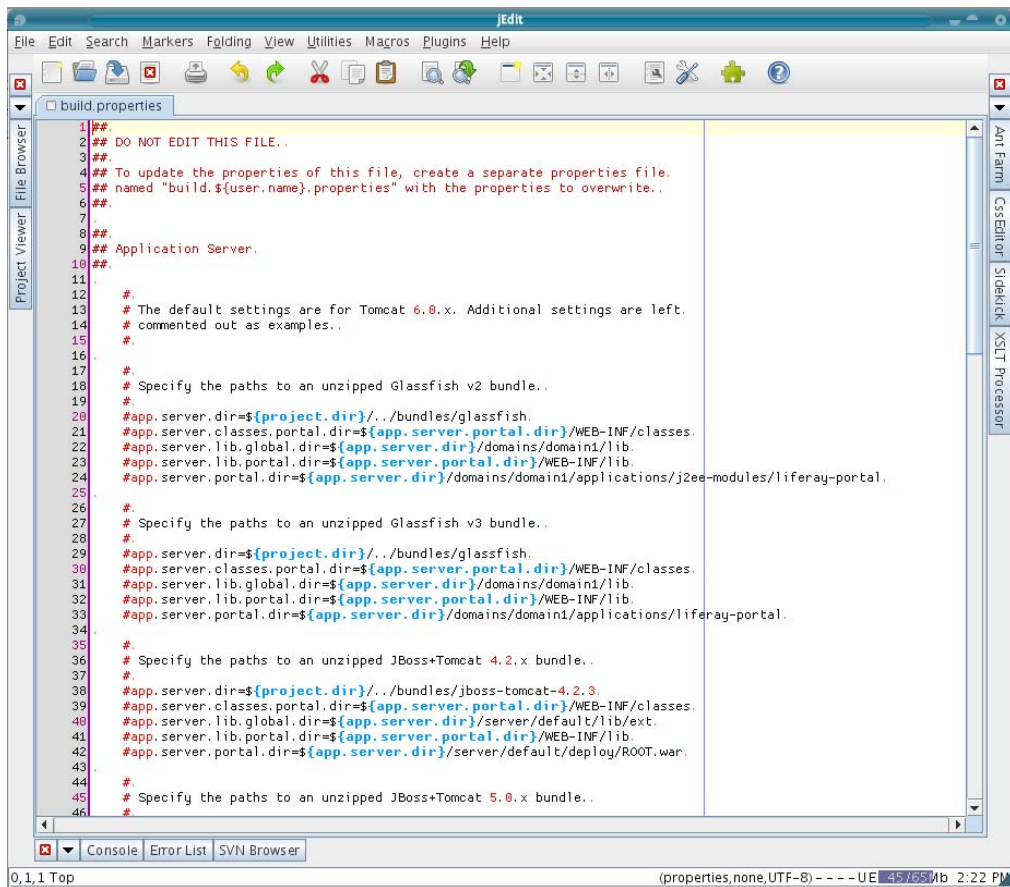


Figure 2.3 Do not edit the *build.properties* file that comes with the Plugins SDK. Override it with your own instead.

If you have used a bundle other than the Tomcat bundle, you will need to customize the properties relating to your Liferay bundle. Simply copy the section for your application server of choice out of the *build.properties* file and paste that section into your *build.[username].properties* file, and then comment the section back in.

If you are not using a bundle, but have installed Liferay manually on a proprietary application server, you will have to customize the listed in table 2.x.

Table 2.x [caption here]

Property	Purpose
<b>app.server.dir</b>	This is the folder into which you have installed your development version of Liferay. It is a best practice to put this in a folder called <i>bundles</i> next to the <i>plugins</i> folder. This is the default configuration.
<b>app.server.classes.portal.dir</b>	This folder defines where the WEB-INF/classes folder in the Liferay installation can be found.
<b>app.server.lib.global.dir</b>	This folder is where .jars that should be on the global class path can be copied.
<b>app.server.lib.portal.dir</b>	This folder defines where the WEB-INF/lib folder in the Liferay installation can be found.
<b>app.server.portal.dir</b>	This folder defines where the Liferay installation can be found.  If you use Eclipse as your development environment, you may also find it convenient to modify the following property:
<b>java.compiler</b>	The default value for this is the standard compiler that comes with Java, <i>modern</i> . You can also use the Eclipse compiler, ECJ. ECJ is an alternate Java compiler with fast performance that allows you also to replace code in memory at run time (called <i>hot code replace</i> ). If you set this option to use ECJ, the ant script will install it for you by copying <i>ecj.jar</i> to your Ant folder. This may or may not work with the version of Ant that runs from within various integrated development environments.

After you save the file, you should have a directory structure that looks like this:

```
[Code Home]/bundles/[Liferay Bundle]
[Code Home]/plugins
```

You are now ready to start using the plugins SDK. Let's dive right in and create a portlet plugin.

#### Note:

If you've been using Liferay's platform for a while, you might be considering using the Ext plugin for most, if not all, development. If I may give you a piece of advice for right now: don't. With Liferay 6 (and to some extent, 5.2), the Ext plugin is recommended only in a few of the rarest scenarios. For a more thorough discussion of this topic, see Chapter 9.

## 2.2 Using the Plugins SDK to develop a portlet plugin

By now, you're probably saying something like, "Plugins, plugins plugins! All you can talk about is plugins! I have one simple question: what the heck is a plugin?!?" Hopefully, that's all you're saying, and you're not following it up with some variant of "\$%&@!" Without any further ado, here's your answer:

Plugins are .war files.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

Yes, that's all they are. If you have ever done a Java web application, you already know about 75% of what you need to know to write a portlet, theme, layout template, or hook. Plugins (portlets, themes, layout templates, hooks, and web modules) are how one adds functionality to Liferay, and they have several benefits over using the Ext plugin:

- Plugins can be composed of multiple smaller portlet and theme projects. This reduces the complexity of individual projects, allowing developers to more easily divide up project functionality.
- Plugins are completely separate from the Liferay core. Portlet plugins written to the Java standard are deployable on any portlet container.
- Plugins can be hot deployed (i.e., deployed while the server is running) and are available immediately. This prevents any server downtime for deployments.

Enough theory. Let's get down to setting up and using the Plugins SDK. In this section, you'll see how to use the Plugins SDK with just a text editor and Ant. If you're an IDE user, please see Appendix A, where you'll find how to set up your projects with Liferay IDE / Studio, Eclipse, or NetBeans.

The Plugins SDK is both a project generator and a location where your projects are stored. You'll create projects in the folders where they belong, and those folders will contain one or multiple projects of that type. You can then open the projects in an IDE or in a text editor in order to work on them.

If you check individual projects into a source code repository, you will want to check them out into a fully configured Plugins SDK, because the ant scripts in the projects depend on ant script within the Plugins SDK for their functionality. Let's see how we would create new projects.

### 2.2.1 Creating a portlet plugin: Hello World

Creating portlet plugins with the Plugins SDK is really easy. As noted before, there is a *portlets* folder inside the plugins SDK folder. This is where your portlet projects will reside. To create a new portlet, first decide what its name is going to be. You need both a project name (without spaces) and a display name (which can have spaces). When you have decided on your portlet's name, you are ready to create the project. On LUM, from the *portlets* folder, enter the following command:

```
./create.sh <project name> "<portlet title>"
```

So as a first exercise, let's create the classic example, which, of course, is Hello World. To create a portlet with a project folder of *hello-world* and a portlet title of *Hello World* on LUM, type:

```
./create.sh hello-world "Hello World"
```

On Windows, you would type:

```
create.bat hello-world "Hello World"
```

You should get a BUILD SUCCESSFUL message from Ant, and there will now be a new folder inside the *portlets* folder in your Plugins SDK. This folder is your new portlet project. This is where you will be implementing your own functionality. At this point, if you wish, you can check this project or your whole Plugins SDK into a source code repository in order to share your project with others. And of course, if you're the command-line-plus-text-editor type of developer, you can get to work right away.

Alternatively, you can open your newly created portlet project in your IDE of choice and work with it there. If you do this, you may need to make sure the project references some .jar files from your Liferay installation, or you may get compile errors. Since the ant scripts in the Plugins SDK do this for you automatically, you don't get these errors when working with the Plugins SDK. We will go over how to do this specifically for Eclipse and NetBeans later, but this information will apply to all IDEs.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

Regardless of you've used to add functionality to the plugin, when you've implemented some functionality, you'll want to deploy your plugin in order to test it.

## 2.2.2 Deploying the Hello World plugin

You can actually deploy this portlet right now if you wanted to, because it's already a Hello World portlet (how easy was that?). To deploy the portlet, go into its directory at the command prompt and type the following command (on all operating systems):

```
ant deploy
```

The portlet will be compiled and deployed to your running Liferay server (you do have Liferay Portal running, right?). If you are watching the logs, you'll see status messages which look like this:

```
04:07:41,558 INFO [AutoDeployDir:176] Processing hello-world-portlet-6.0.0.1.war
04:07:41,560 INFO [PortletAutoDeployListener:81] Copying portlets for
/home/me/code/bundles/deploy/hello-world-portlet-6.0.0.1.war
...
04:07:43,263 INFO [PortletAutoDeployListener:91] Portlets for
/home/me/code/bundles/deploy/hello-world-portlet-6.0.0.1.war copied successfully.
Deployment will start in a few seconds.
04:07:44,827 INFO [PortletHotDeployListener:250] Registering portlets for hello-
world-portlet
04:07:46,729 INFO [PluginPackageUtil:1391] Finished checking for available updates
in 5092 ms
04:07:48,839 INFO [PortletHotDeployListener:376] 1 portlet for hello-world-portlet
is available for use
```

When you see the "available for use" message, your plugin is deployed and can be used immediately. This applies for all plugin types, except for the Ext plugin.

To add the plugin to a page, log in to Liferay Portal using the administrative credentials (user name: *test@liferay.com*; password: *test*). Go up to the Dockbar and select Add > More. Your generated portlet project by default is placed in the *Samples* category (we'll see how to change this later). Open this category and you should see your portlet displayed :

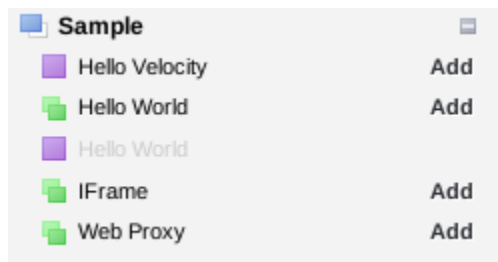


Figure 2.4 Your Hello World portlet is deployed. Notice that there are two Hello World portlets displayed, with different icons. The grayed out one is already on the page we're browsing, and is unavailable because it's non-instanceable.

By default, portlets are generated as instanceable portlets, so you'll see your portlet appear with the green icon. Drag it out onto the page. It should look like:

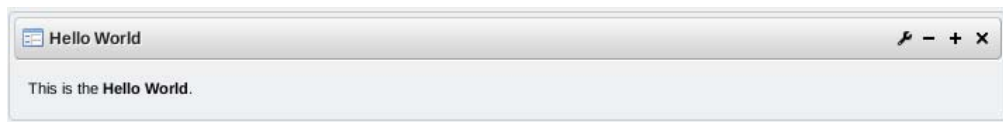


Figure 2.5 The Hello World portlet. This will definitely not win any awards, but it was certainly easy to create.

Obviously, it is very easy to create a Hello World portlet: we didn't even have to write any code. So let's make our portlet actually do something.

### 2.3 Making Hello World into Hello You

Hello World examples are all over the Internet, and they are used to introduce topics all the time. And maybe I'm just a rebel at heart, but I can't stand them. In my experience, Hello world examples are not generally all that helpful, as they don't tend to actually introduce anything useful. So we'll try to add a little functionality using the Portlet API for our first example. We'll call this the "Hello You" portlet. This portlet will display a standard "Hello" message when it is first rendered in View Mode.



Figure 2.6 The Hello You portlet will allow you to use the portlet's Edit Mode to enter your name, and will then use that name when displaying its "hello" message.

The functionality we're going to add is in edit mode: we'll allow a user to enter his or her name. This name will be stored using the Portlet API as a portlet preference, similar to the way a Weather portlet developer would store a zip code in order to display the proper weather map. We will then allow the portlet to display the user's name in the "Hello" message when that user logs in.



Figure 2.7 Once your name is entered, your name will be saved as a Portlet Preference, and will thereafter always be displayed when you log into the portal. Each preference is linked to the user who is logged in, so if somebody else logs in and sets a different name, that name will be displayed for that user. If you log in again, your name will be displayed.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

You get all this for free just by being on a portal platform.

This example will introduce to you several features of the Portlet API:

- Portlet Modes
- Portlet Actions and how they are processed
- Portlet Preferences

This will hopefully be a better foundation for building portlets than a simple “Hello World” portlet would provide. There's one additional thing I should mention: as I'm sure you've noticed, I've been throwing around some terms which relate to the Portlet API. If you need a short introduction to the Portlet API (upon which the Liferay development platform is based), you'll find one in Appendix B. This should give you the basic foundation you'll need for this first example and the rest of the chapters, which will build specifically on Liferay as a platform. If you are interested in exploring the Portlet API at a deeper level, I highly recommend Ashish Sarin's book, *Portlets in Action*.

For now, let's get to the portlet.

### 2.3.1 Anatomy of a portlet project

A portlet project is made up at a minimum of three components:

- Java Source
- Configuration files
- Client-side files (\*.jsp, \*.css, \*.js, graphics, etc.)

These files are stored in a standard directory structure which looks like figure 2.x. The example is a fully deployable portlet which can be deployed to your configured Liferay server by running the *deploy* ant task. In fact, you already did this in the previous chapter.

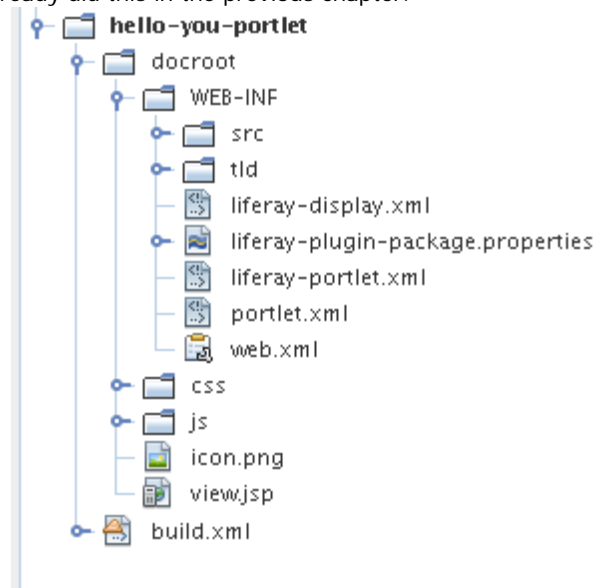


Figure 2.8 This is the folder structure of the Hello You portlet (or any portlet project, for that matter). As you can see, it is very simple to follow and there is a place to put all of your components.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

The various files and folders have different functions, listed in Table 2.x.

Table 2.x [caption here]

Folder	Description
<b>docroot</b>	This folder is the “root” directory of your application. As you can see, it has several subdirectories.
<b>WEB-INF</b>	This is the standard WEB-INF folder for web modules. Since portlets are web modules too, we will put our configuration files here. You can see that the directory contains several configuration files already. We will go over these separately.
<b>WEB-INF/src</b>	This folder contains the source code for the portlet.
<b>build.xml</b>	This is the Ant build script which you will use to compile and deploy your project.

The default portlet is configured as a standard Java portlet which uses separate JSPs for its three portlet modes (view, edit, and help). Only the `view.jsp` is implemented in the generated portlet; the code will need to be customized to enable the other modes. For Hello You, we'll only implement Edit mode.

In addition to the standard portlet configuration files, the Plugins SDK generates a project which contains some Liferay-specific configuration files:

Table 2.x [caption here]

Configuration file	Purpose
<b>liferay-display.xml</b>	This file describes for Liferay what category the portlet should appear under in the Add > More window.
<b>liferay-portlet.xml</b>	This file describes some optional Liferay-specific enhancements for Java portlets that are installed on a Liferay Portal server. For example, you can set whether a portlet is instanceable, which means that you can place more than one instance on a page, and each portlet will have its own data. The DTD for this file explains all of the possible settings.
<b>liferay-plugin-package.properties</b>	This file describes the plugin to Liferay's hot deployer. One of the things that can be configured in

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

<p>this file is dependency .jars. If a portlet plugin has dependencies on particular .jar files that already come with Liferay, you can specify them in this file and the hot deployer will modify the .war file on deployment so that those .jars get copied from Liferay into the <i>WEB-INF/lib</i> folder of the .war file. This prevents you from having to include .jar files (such as Spring, Struts, or Hibernate) in your portlet project that are already used by Liferay.</p>
--

Okay; enough lists of what is possible to do; let's put all of these files, directories, and code together to make something.

Open the Hello World project that you created in the last chapter. We're going to turn this into the Hello You portlet.

### 2.3.2 Configuring Hello You

The first thing you should take a look at is the `portlet.xml` file, which you'll find in the *WEB-INF* folder. This is the configuration file for the portlet. You will find a line in there that looks like this:

```
<portlet-class>com.liferay.util.bridges.mvc.MVCPortlet</portlet-class>
```

This defines what Java class implements the portlet. By default, projects are generated to use Liferay's `MVCPortlet`, which we'll get to later. It has certain benefits to the experienced developer, but it also abstracts much of the portlet lifecycle from you, which we don't want to do just yet. So we're going to implement our own portlet class the way the portlet specification recommends so that you can get familiar with the Portlet API and then move past it to the `MVCPortlet` and to other frameworks if you wish.

Since we aren't going to use `MVCPortlet`, we have to change this line to point to the portlet class we are going to create. So modify this line so that it looks like this:

```
<portlet-class>com.liferay.inaction.portlet.HelloYouPortlet</portlet-class>
```

Next, we need to tell the portal that our portlet implements edit mode as well as view mode (it assumes view mode; otherwise there would be no point to your portlet). So change the `<supports>` tag in your `portlet.xml` so it reads like this:

```
<supports>
  <mime-type>text/html</mime-type>
  <portlet-mode>view</portlet-mode>
  <portlet-mode>edit</portlet-mode>
</supports>
```

You may have noticed in `portlet.xml` another tag called `<init-param/>`. This tag, as you have probably figured out, defines initialization parameters which can be used in your portlet. The default project defines a parameter called `view-jsp` which defines the location of the JSP file that will be used to display the portlet in view mode. This parameter can thus be used in the portlet class to forward processing over to the `view.jsp` file in the project. This worked in the initial portlet because this functionality was implemented already in `MVCPortlet`; now that we're using our own portlet class, we will have to implement it ourselves.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>



Below your definition of your portlet class, add the additional initialization parameter below the existing one like this:

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

Imagine Darth Vader's voice: "All of our configuration is now complete." We can now start implementing the logic of the portlet.

Create a package in your *src* folder called `com.liferay.inaction.portlet`. In this package, create a Java class called `HelloYouPortlet.java`. This class will extend the `GenericPortlet` class, which is included with the Portlet API and is available in every portal implementation. So far, your class should look like the following code.

```
package com.liferay.inaction.portlet;

import javax.portlet.GenericPortlet;

public class HelloYouPortlet extends GenericPortlet {

}
```

#### PORTLET INITIALIZATION AND IMPLEMENTING VIEW MODE

The first thing we want to do is implement the Portlet API's `init()` method to pull the values from our initialization parameters into our portlet class. All we need to do is define two instance variables to hold these values and then implement the method:

```
protected String editJSP;
protected String viewJSP;

public void init() throws PortletException {
    editJSP = getInitParameter("edit-jsp");
    viewJSP = getInitParameter("view-jsp");
}
```

Easy, right? We get access to the `getInitParameter()` method because we're extending an existing class in the Portlet API. So anything that is in those initialization parameters gets pulled into these variables.

Now we can actually implement the default view of our portlet. This is done by implementing a method called `doView()`. As you can imagine, there are similar methods for the other portlet modes, and we will also be implementing `doEdit()`. The functionality for this mode is simple: we want to retrieve a preference that may or may not have been stored for the logged in user. If we don't find one, we'll simply print "Hello!" If we do find one, we will print a message that takes that preference and displays it in the message. So if the user's name is Mortimer Snerd, we'll print "Hello Mortimer Snerd!"

#### Listing 2.1 The default view of your portlet

```
public void doView(RenderRequest renderRequest,
                  RenderResponse renderResponse)
    throws IOException, PortletException {
    PortletPreferences prefs = renderRequest.getPreferences();
    String username = (String)prefs.getValue("name", "no");
    if (username.equalsIgnoreCase("no")) {
        username = "";
    }
    req.setAttribute("userName", username);
}
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

```

        include(viewJSP, renderRequest, renderResponse);
    }

```

In this code we first grab the `PortletPreferences` and check to see if the preference is there. The method for getting the preferences wants to know what the preference you're looking for is called as well as a value to return if it couldn't find the preference. So we provide *name* for the preference and *no* for the value to return if the preference wasn't found.

If we find a name, we then store it as an attribute in the request. If not, we simply store an empty string. We then forward to the `view.jsp` file which was previously defined in our `portlet.xml` file.

Note that we're using some objects here that are very similar to `HttpServletRequest` and `HttpServletResponse`, both of which you are likely already familiar with. These are very similar to their servlet counterparts in that you can set parameters and attributes, retrieve information about the portlet's environment, and more. In our case, we're retrieving an object called `PortletPreferences` from the portlet instance.

We're also calling a convenience method called `include()`, which looks like listing 2.2.

### Listing 2.2 Shortening code with an `include()` convenience method

```

protected void include(
    String path, RenderRequest renderRequest,
    RenderResponse renderResponse)
    throws IOException, PortletException {

    PortletRequestDispatcher portletRequestDispatcher =
        getPortletContext().getRequestDispatcher(path);

    if (portletRequestDispatcher == null) {
        _log.error(path + " is not a valid include");
    } else {
        portletRequestDispatcher.include(
            renderRequest, renderResponse);
    }
}

```

Just like you can in a servlet-based web application, you can forward request processing to a JSP from a portlet. And just like in a servlet-based web application, you have to chain a whole bunch of methods together in order to accomplish it. To make our code more neat, we can simply create a convenience method called `include()` which chains all those methods together for us so that all we have to do is call this one method (which has a nice, short name) to forward processing to a JSP.

We've included a check here to make sure that the `PortletRequestDispatcher` we get out of the `PortletContext` is not null. If it is null, we log that. To use this method, therefore, we'll also have to enable logging in our portlet. Liferay ships with Apache's Commons Logging classes, which make it easy to add log entries from our portlets. We only need to add an instance variable to the class for our `_log` object:

```

private static Log _log = LogFactory.getLog>HelloYouPortlet.class);

```

To complete the view mode of this portlet, we now have to implement the JSP to which we are forwarding, which you can see in its entirety in listing 2.3.

### Listing 2.3 Implementing the JSP for view mode

```

<%@ taglib uri="http://java.sun.com/portlet" prefix="portlet" %>

```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

```

<jsp:useBean id="userName" class="java.lang.String" scope="request"/>    #1

<portlet:defineObjects />

<p>This is the Hello You portlet.</p>
<p>Hello <%=userName %>!</p>

```

### #1 Gets the bean out of the request

In the first line, we declare the portlet tag library. This again is a part of the Portlet API, and is a standard across all portals. The next line should look familiar: we pull a bean out of the request that we had made available in the `doView()` method of our portlet (#1). There's no difference in how this is done versus doing it with a servlet. The `<jsp:useBean>` tag in the third line comes from the portlet tag library, which we declared in the first line. The `defineObjects` tag registers several objects with the JSP that may be useful: `renderRequest`, `renderResponse`, and `portletConfig`.

### Note

The tag library declaration above is the declaration for Portlet 1.0. This has been done for backwards compatibility, as we are not using any features in this first portlet that require Portlet 2.0. This portlet, therefore, can be deployed on any Portlet 1.0 container (such as Liferay 4.4.x) or any Portlet 2.0 container (such as Liferay 5.0 and above).

The Portlet API's `RenderRequest` and `RenderResponse` objects correspond to the `HttpServletRequest` and `HttpServletResponse` objects with which you may be familiar, and we've already used these in our portlet class. The `PortletConfig` object corresponds roughly with the `ServletConfig` object you would use in a Servlet, in that it holds data about the portlet and its environment. Because you may need to use these objects in your JSP's display logic, these three variables are defined through the `defineObjects` tag, and it's a good idea to always put this tag at the top of your JSPs.

Otherwise, the logic of this JSP is very simple. We pull the bean we stored in the request and print a "Hello" message. If the bean has a value, that value will be printed after the message; if not, nothing will be printed.

At this point, view mode is complete. You can actually deploy the portlet as it is and it will display a hello message. Since we haven't implemented edit mode yet, though, it won't have any functionality, so that's what we'll do next.

### IMPLEMENTING EDIT MODE

Before we jump into the code for edit mode, I need to tell you something about URLs in portlet applications. Most web developers are used to manipulating the URL directly. That is not something you can do in a portlet application. In most cases, portlet URLs are linked with Portlet actions which cause the portlet to do some processing. Portlet actions are defined by using a special portlet URL object called an `ActionURL`. Because a portlet is a fragment of a page that is assembled at run time by the portlet container, developers cannot simply define their own URLs as they can in regular web applications. Instead, they must be created programmatically. This is a bit of a paradigm shift, but it's fairly easy to make the transition. It's important that to know that the contents of URLs must be generated by the portal server at run time. Why? So that there are no conflicts between URLs generated by different portlets.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

For example, consider a search portlet that sits on a page. This portlet can search for any uploaded content. Say another portlet is placed on that page by a user. This other portlet contains a list of customers and has a search box at the top.

If developers knew ahead of time that these portlets would be placed on the same page, they could make sure that the URLs to the two separate search functions were different. But they didn't know ahead of time that a portal administrator would do this, and so both search URLs point to `/search` in the application's context. Can you see how there would be a conflict? Which application's search would get called?

For this reason, the Portlet API provides URL objects. Developers can create URL objects programmatically and even set parameters in them. This allows the portlet container to generate the actual URL strings at runtime, preventing any conflicts like the one mentioned above. Since our implementation of edit mode consists of a very simple form that users will fill out and submit, we need a URL for the submit button. For this reason, we will have to create one using the API. So our `doEdit()` method in listing 2.4 reflects this.

#### Listing 2.4 Implementing `doEdit()` in our portlet

```
public void doEdit(RenderRequest renderRequest,
                  RenderResponse renderResponse)
    throws IOException, PortletException {
    renderResponse.setContentType("text/html");
    PortletURL addName = renderResponse.createActionURL();
    addName.setParameter("addName", "addName");
    renderRequest.setAttribute("addNameUrl", addName.toString());
    include(editJSP, renderRequest, renderResponse);
}
```

When you add the code above, you will also have to add the import `javax.portlet.PortletURL`.

#### Note

The first line of code above—which sets the content type—is only required in the 1.0 (JSR-168) version of the API. It doesn't hurt anything, so we have left it in the method above to maintain backwards compatibility, but if you are using Portlet 2.0 (JSR-286), it's not necessary.

We have created an `ActionURL` and added a parameter to it called `addName`. We then put the URL in the request object so that it may be used in the JSP to which we will be forwarding the processing.

Let's implement the JSP for edit mode next. Create a file in `docroot` called `edit.jsp`. Remember that we earlier pointed to this file by using an initialization parameter in `portlet.xml`. Listing 2.5 shows the `edit.jsp` file.

#### Listing 2.5 The JSP for edit mode

```
<%@ taglib uri="http://java.sun.com/portlet" prefix="portlet" %>
<jsp:useBean class="java.lang.String" id="addNameUrl" scope="request" />
<portlet:defineObjects />

<form
  id = "<portlet:namespace />helloForm"
  action="<%=addNameUrl %>"
  method="post">
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

```

<table>
    <tr>
        <td>Name:</td>
        <td><input type="text" name="username"></td>
    </tr>
</table>
<input type="submit" id="nameButton" title="Add Name" value="Add Name">
</form>

```

In the code above we first declare the tag library for the Portlet API as we did before, and we then make our URL available to the JSP by using the `useBean` tag to retrieve it from the request object and place it in a variable called *addNameUrl*.

After this is a simple HTML form. Notice that we use a portlet tag called `namespace` in front of our form name. This is done for similar reasons that portlet URLs exist: you could have multiple forms with the same name on the page, and then they would conflict with each other. By using this tag, you allow the portal server to prepend a unique string to the front of your form's name, thereby ensuring that no other portlet will contain a form with the same name as yours.

Note also that the form's action is set to the value of the action name parameter in the `ActionURL` we created in the portlet class and then retrieved as a bean in the JSP. This URL has the parameter *addName* in it, which will be captured by our `processAction` method when this form is submitted so that processing can be directed to the right place.

Speaking of `processAction`, that is now the only missing element in our portlet. This method is part of the standard Portlet API. The `processAction` method is there to provide a place where portlet actions can be processed. A default implementation is provided with the API (which we will see in the next chapter), but we are going to override it for this portlet and provide our own implementation.

Portlet URLs can be of two types: a `RenderURL` or an `ActionURL`. A `RenderURL` simply tells the portlet to render itself again. Whatever parameters have been defined for the portlet at that time take effect, and the portlet re-draws itself. An `ActionURL` immediately forwards processing to the `processAction` method, where logic can be in place to determine which action has been taken by the user and then the appropriate processing can occur. We'll implement our own version of this so that we can see how it works, and then later on we'll take advantage of other (better) implementations. Since we have only one action, the logic will be pretty simple, as you can see in listing 2.6.

## Listing 2.6 Processing a portlet action

```

String addName = actionRequest.getParameter("addName");
if (addName != null) {
    PortletPreferences prefs = actionRequest.getPreferences();
    prefs.setValue("name", actionRequest.getParameter("username"));
    prefs.store();
    actionResponse.setPortletMode(PortletMode.VIEW);
}

```

You'll also need to add the following two imports to the top of your class:

```

import javax.portlet.PortletPreferences;
import javax.portlet.PortletMode;

```

This code searches for a parameter in the portlet's `ActionRequest` object called *addName*. We created this parameter earlier as part of an `ActionURL` in our `doEdit()` method. Simply by having the form's action point to this URL, it directs processing to the four lines of code in the `if` statement.

We could just as easily have called a different method instead, which is what you would normally do in a larger portlet so that you can keep only action processing logic here.

In any case, the parameter—called *username*—will be found in the request, because it was a field on the form. The Portlet API is then accessed in order to store a preference for this particular user. The preference is called *name* and we use whatever value was in the parameter. Yes, I know: we're not doing any field validation. Normally you would do that before storing anything, but I wanted to keep this example as simple as possible. This key/value pair will now be stored for that portlet/user combination.

The last thing this code does is set the portlet mode back to view mode. When that is done, `doView()` will be called, and the user will now get the "Hello <name>" message.

To summarize: we now have all of the processing for the portlet's edit mode in place. The `doEdit()` method creates a URL with an action name parameter of *addName*. Processing is then forwarded to the `edit.jsp` file where this parameter is used as the action of a form. The form contains one field, which the user will use to type his or her name. When the form is submitted, the portlet's `processAction` method will run and retrieve the action's *name* parameter, which will have the value *addName*. Checking for this value leads us to code which stores the name the user submitted as a `PortletPreference`. To keep the example simple, we have not implemented any validation on the data submitted.

Rather than look at it piecemeal as we did above, it is sometimes easier to see how things work by showing the whole example. So listing 2.7 shows what the entire portlet should look like.

#### Listing 2.7 The complete Hello You portlet

```
package com.liferayayinaction.portlet;

import java.io.IOException;

import javax.portlet.ActionRequest;
import javax.portlet.ActionResponse;
import javax.portlet.GenericPortlet;
import javax.portlet.PortletException;
import javax.portlet.PortletMode;
import javax.portlet.PortletPreferences;
import javax.portlet.PortletRequestDispatcher;
import javax.portlet.PortletURL;
import javax.portlet.RenderRequest;
import javax.portlet.RenderResponse;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

/**
 * The Hello You portlet, a simple example demonstrating
 * Portlet Modes, Portlet Actions, and Portlet Preferences.
 *
 * @author Rich Sezov
 */
public class HelloYouPortlet extends GenericPortlet {

    public void init() throws PortletException {
        editJSP = getInitParameter("edit-jsp");
        viewJSP = getInitParameter("view-jsp");
    }
}
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

```

    }

    public void doEdit(RenderRequest renderRequest,
                       RenderResponse renderResponse)
        throws IOException, PortletException {
        renderResponse.setContentType("text/html");
        PortletURL addName = renderResponse.createActionURL();
        addName.setParameter("addName", "addName");
        renderRequest.setAttribute("addNameUrl", addName.toString());
        include(editJSP, renderRequest, renderResponse);
    }

    public void doView(RenderRequest renderRequest,
                       RenderResponse renderResponse)
        throws IOException, PortletException {
        PortletPreferences prefs = renderRequest.getPreferences();
        String username = (String) prefs.getValue("name", "no");
        if (username.equalsIgnoreCase("no")) {
            username = "";
        }
        renderRequest.setAttribute("userName", username);
        include(viewJSP, renderRequest, renderResponse);
    }

    public void processAction(
        ActionRequest actionRequest,
        ActionResponse actionResponse)
        throws IOException, PortletException {
        String addName = actionRequest.getParameter("addName");
        if (addName != null) {
            PortletPreferences prefs =
                actionRequest.getPreferences();
            prefs.setValue(
                "name", actionRequest.getParameter("username"));
            prefs.store();
            actionResponse.setPortletMode(PortletMode.VIEW);
        }
    }

    protected void include(
        String path, RenderRequest renderRequest,
        RenderResponse renderResponse)
        throws IOException, PortletException {

        PortletRequestDispatcher portletRequestDispatcher =
            getPortletContext().getRequestDispatcher(path);

        if (portletRequestDispatcher == null) {
            _log.error(path + " is not a valid include");
        } else {
            portletRequestDispatcher.include(
                renderRequest, renderResponse);
        }
    }

    protected String editJSP;
    protected String viewJSP;

```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

```
private static Log _log = LogFactory.getLog(HelloYouPortlet.class);  
  
}
```

### 2.3.3 Deploying and testing your portlet

Because we used the Plugins SDK to create this project, it came with its own Ant script which makes it very easy to build and deploy the project. If your IDE supports Ant, you can deploy the project right from within your IDE. If you're not using an IDE, you can run a simple command to deploy your project.

First, start your Liferay server. Again, if you are using an IDE and have followed the instructions from the previous chapter, you can do this right from within your IDE. Once Liferay has started, keep a window open to Liferay's console so you can watch the deploy take place.

Next, run the *deploy* task from the Ant build script. To do this from the command line (on any operating system), issue the following command from the project folder:

```
ant deploy
```

Once you receive the BUILD SUCCESSFUL message, turn your attention to the Liferay console. You should see your portlet deploy right away.

Your new portlet is now indistinguishable from any other portlet deployed in Liferay. Go to <http://localhost:8080> and log in using the administrative credentials:

**User Name:** test@liferay.com

**Password:** test

Go up to the Dockbar and click *Add > More*. You will see many categories of portlets displayed. Open the one labeled *Sample*. You will see the Hello World portlet listed there. Drag it off the list and drop it in the right-most column, if it's not there already. If you followed along with the previous chapter, the portlet should already be there.

You will now see your portlet displayed. You can close the *Add > More* window by clicking the red X in its top right corner.

The default message of "Hello!" is being displayed in the portlet. This is the way it's supposed to function: we haven't set our Portlet Preference yet, so the portlet does not know our name. To get to Edit Mode, click the button in the title bar of the portlet that has a wrench on it, and then click the *Preferences* link (Liferay Portal displays a portlet's edit mode as a Preferences menu item). You will be brought to the portlet's edit mode, and your `edit.jsp` will be displayed.

Type your name and click the *Add Name* button. Your Portlet Preference will be stored, and because we changed the mode of the portlet back to View in our `processAction` method, the portlet will redisplay itself in view mode. Because our Portlet Preference is now set, the portlet will display the name we entered.

Congratulations! You have just written your first portlet!

### 2.3.4 Changing the portlet's category and name

Notice that when you added the portlet, you had to select it from the *Sample* category in the *Add > More* window. You also had to add the "Hello World" portlet, but now our portlet is called "Hello You." The portlet is in a suboptimal category because the generated portlet project defaults to this category, and the portlet has the wrong name because we created "Hello World" in the last chapter. If you would like to create your own category for your portlet, this is easy to do. At the same time, we also want to

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>



rename the portlet so that it is no longer called Hello World, since it is now the Hello You portlet. We can do these things by editing two XML files:

- portlet.xml
- liferay-portlet.xml
- liferay-display.xml

### RENAMING THE PORTLET

First, let's change the name of the portlet. Open portlet.xml, which is in your *WEB-INF* folder. There are two places in this file where we need to change the name. The first takes effect at a system level, and the second takes effect for the portlet title.

Find the two lines in the file that look like this:

```
<portlet-name>hello-world</portlet-name>
<display-name>Hello World</display-name>
```

Change them so that they read like this:

```
<portlet-name>hello-you</portlet-name>
<display-name>Hello You</display-name>
```

Next, find the the <portlet-info> section of the file:

```
<portlet-info>
    <title>Hello World</title>
    <short-title>Hello World</short-title>
    <keywords>Hello World</keywords>
</portlet-info>
```

Change it so that it reads like this:

```
<portlet-info>
    <title>Hello You</title>
    <short-title>Hello You</short-title>
    <keywords>Hello You</keywords>
</portlet-info>
```

Next, open liferay-portlet.xml, which is in the same folder. Find the following line, which is in the <portlet> tag:

```
<portlet-name>hello-world</portlet-name>
```

Change it so it reads:

```
<portlet-name>hello-you</portlet-name>
```

### 2.3.5 Creating a custom category

Finally, in the same *WEB-INF* folder, you will find a file called liferay-display.xml. This file controls the category under which your portlet appears in the *Add > More* window. Open this file and you will see the code in listing 2.8.

#### Listing 2.8 Changing the category for your portlet

```
<?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="category.sample">
        <portlet id="hello-you" />
    </category>
</display>
```

Change the category name to something else, like *My Portlets*. The code would then read:

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

```
<display>
  <category name="My Portlets">
    <portlet id="hello-you" />
  </category>
</display>
```

Now go ahead and deploy your portlet again. Refresh the page. You'll see that an interesting thing has happened to your portlet:

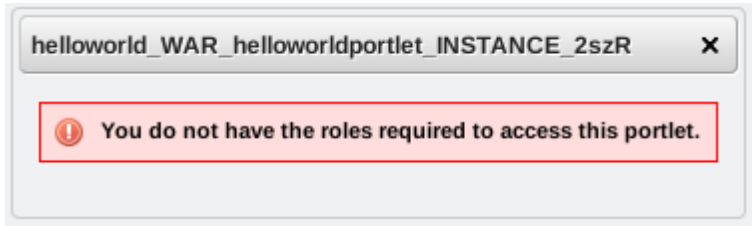


Figure 2.9 Aaarrgh! What happened to my portlet?!?

Your portlet no longer exists. You just renamed your portlet to Hello You, but Liferay thinks a Hello World portlet is on the page because, well, you put it there. Of course, now Liferay can't find it, so it displays this message. So what you'll have to do is remove the portlet from the page and add your newly renamed portlet to the page instead. To remove the portlet, just click the X.

Next, open the *Add > More* window. You will see your new category displayed, and your portlet is now displayed there:

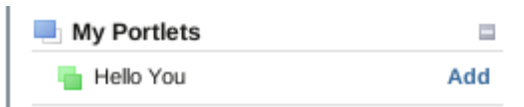


Figure 2.10 Your portlet is now both renamed and in a custom category which you have created.

You can now drag the portlet over to the page. When you do, notice what it says now:



Figure 2.11 The Hello You portlet seems to have amnesia. What happened to your portlet preference?

The preference you stored earlier when the portlet was named "Hello World" was saved *for that portlet*. Now that you have renamed it, that portlet no longer exists. For all intents and purposes, even

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

though the code is exactly the same, this is now a new portlet, according to Liferay, so Liferay treats it as such. Its title bar also now properly displays “Hello You,” because you configured it that way in the `portlet.xml` file.

## 2.4 Summary

You can see that the Plugins SDK provides a full development life cycle for all plugin types. Since plugins are simple .war files and the Plugins SDK uses Ant for all its operations, it is tool, application server, and operating system agnostic, so you are free to use the tools of your choice to create Liferay development projects. I'm sure that you'll enjoy using the Plugins SDK and that it will be a useful tool for you to create your Liferay plugins.

Liferay provides many ways of writing code on its platform. You can add your own applications by writing portlets, define the way it looks by using themes, and customize existing Liferay functionality with hooks and Ext plugins. We talked about a great way of determining when to use which, and this can be summarized very simply by saying, “use Ext sparingly.” The other plugins provide a lot more flexibility in terms of deployment options and development. Ext should be used only when you need to customize some core feature of Liferay that cannot be customized any other way.

Next, we created a portlet according to the Java standard. We saw how a portlet is structured, as well as the different configuration files that make up a portlet web module. We also saw how to deploy and test the portlet, as well as change which category it appears under in Liferay's Add > More menu. This portlet really doesn't do very much, but it serves as a good introduction to several highly used features of the portlet API, such as portlet modes, portlet actions, and portlet preferences. We have so far used only Portlet 1.0 features in this portlet, so you could deploy it on containers that support either Portlet 1.0 (like Liferay 4.4.x and below) or Portlet 2.0 (like Liferay 5.0.x and above). The skills you have learned here will provide a good foundation for creating more complex portlet projects.

We'll focus next on more complex projects like the ones you might encounter every day. To do this, we'll start building a site for a fictitious company. In that way, we'll be able to simulate the common challenges faced by developers and knock them out one by one.

# 3

## *A data-driven portlet made easy*

This chapter covers:

- Designing a portlet for database interaction
- Liferay's Service Builder code generator
- Architecting applications using DAOs and DTOs
- Defining relationships using Service Builder

For the rest of this book, we'll use a case study to illustrate the examples. This way, we avoid abstract examples (like the ones in the preceding chapters) and are able to build real-world solutions like the ones you will be building for your web site. We're going to start with a data-driven portlet, which arguably is the most common type of application that developers are working on every day. So without further delay, let's take a look at our case study.

### ***3.1 Introducing Inkwell: A Case Study***

For the bulk of this book, we'll be using a case study as the unifying example site we'll be building. This case study will help us apply concretely all of the concepts that we're going to cover. So first let's look at some background information about the case study, and then we'll go over the design of our first portlet.

#### ***3.1.1 Company Profile: Inkwell***

Inkwell is a company dedicated to bringing back fountain pen technology simply because it's better. In today's world of workers with desk jobs, repetitive stress injury is a real danger. Couple that with the wide use of workstations that are ergonomically incorrect, and you can spell disaster.

Inkwell is bringing fountain pen technology not only to the realm of physical pen and paper, but to the digital realm as well. Our patented handwriting recognition technology allows the information worker to write on a simple pad and paper, while all the time the words are being transcribed to a computer up

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>



Figure 3.1

Inkwell's PDA pen product image. They intend to use this image on the web site; currently it appears in paper brochures.

- Pen-Tops—a whole computer system consisting of a base model and a pen. This is a full-featured computer, and the pen can act as either a data entry tool or a mouse.
- Traditional fountain pens for everyday use

Now, you may be asking: aren't fountain pens messy? Not anymore. Our patented leak-proof technology is backed by a 5 year warranty protecting you from leaks. And many of our technology pens don't use ink at all! So you can feel safe knowing that there won't be a mess with our pens.

### 3.1.2 What Inkwell needs in a web site

Inkwell has several partnerships with various technology vendors. Rather than engineer new technology themselves, Inkwell employs several teams of programmers and hardware engineers who handle technology integration with fountain pens. Each of the vendors has a special relationship with Inkwell, but they shouldn't need to know about each other. For this reason, Inkwell needs a secure Extranet which can facilitate this relationship.

To enable its employees to better communicate, collaborate, and coordinate, Inkwell also needs a full-featured Intranet which provides all the communication and collaboration tools their users need. It needs to allow for easy sharing of data, but its security model must also allow for protecting certain sensitive data so that only authorized persons can view it.

Of course, Inkwell will also need a robust Internet site which can facilitate the marketing and support of their products as well as facilitate the relationship with their community of users.

To accomplish this, Inkwell has chosen Liferay Portal to handle all of its web-based activity. A single Liferay install will handle all three of its sites.

### 3.1.3 Inkwell's high-level portal design

The Inkwell web team went through the process outlined in Chapter 1 to design their portal. At the end of the process, they had a design which they could express in a diagram that looked like this:

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

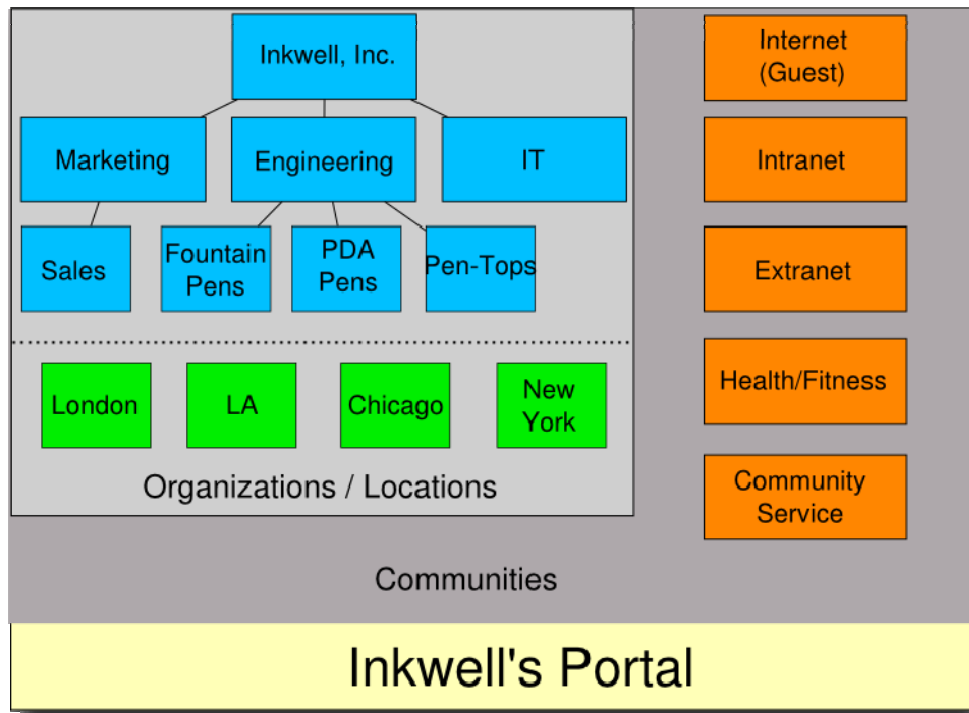


Figure 3.2 Inkwell's high-level portal design.

The diagram above represents the end state of Inkwell's portal project. We will be looking at only the first phase of this project, which is a subset of the eventual full functionality. This first phase is designed to produce two outcomes:

- Take advantage of the "low-hanging fruit," or functionality that can be implemented quickly that also provides core necessary features of the site.
- Help the members of the development team get up to speed on the various features of the Liferay platform by implementing site functionality that is less complicated first, and then moving to more complicated features toward the end of this phase and future phases of the project.

Inkwell's organization chart will be mirrored using Liferay's Organizations and Locations (Locations are just "leaf nodes" in the hierarchical tree of organizations). Because of space considerations, not all organizations are depicted in the diagram. Corporate users will be registered in the portal as members of Inkwell, Inc., the organization for which they work (such as Sales, IT, or a Product Team), and their corporate location. Inkwell is headquartered in London, with small sales offices in New York, Chicago, and Los Angeles, so most of the employees will be in the London location. Future locations are planned as the company grows.

Since both Organizations and Communities can have pages, this design allows individual organizations within the company to maintain their own web pages on their Intranet or Internet. For the

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

first phase of this project, however, organizations will not have any pages of their own. Instead, an Intranet community will be created for internal users to access.

### **3.1.4 Inkwell portal phase 1 requirements**

The Inkwell Portal will be a comprehensive web site covering all online needs of Inkwell, Inc. The current web site (developed in HTML with some separate web applications) no longer services the needs of the company, and Inkwell desires to have a much more interactive and easy to manage site. This will help the company to maintain better contact with its customers and will increase its ability to sell its products.

Additionally, Inkwell wishes to replace its Intranet with something that is a bit more interactive and can be maintained by non-developers. Currently the corporate Intranet consists of a home page done in HTML with links to many of the applications (hosted in separate environments and maintained by separate teams) that employees need. Inkwell will replace the main Intranet site with Liferay, and will over time migrate existing application functionality over to the portal. End users will be responsible for maintaining their own pages, freeing up Inkwell's web team to work on migrating existing applications and later, creating new applications.

The following components of the Inkwell Portal will be completed in phase 1:

- Internet-facing web site
- Intranet site
- Extranet site for partners

#### **INKWELL INTERNET**

Responsibility for creation of the Internet site will be divided into two teams: a content management team and a development team. The content management team will be responsible for designing the pages and page layouts, as well as the content types, structures, and templates. Content itself will be created in collaboration with the Marketing department. The development team will be responsible for creating the applications (i.e., portlets) that have been identified as in scope for phase 1.

#### **INKWELL INTRANET**

In the same manner as for the Internet site, the content management team will handle the management of the pages and site content on the Intranet. The development team will handle the applications.

#### **INKWELL EXTRANET**

The Extranet site will be a special community for interaction with Inkwell's suppliers and distributors. Currently, there are no applications slated for phase 1 of the Extranet. Liferay's Document Library portlet is sufficient to provide the patches, fixes, and data sheets that need to be provided to those companies with whom Inkwell collaborates.

Our first task, then will be to work on Inkwell's Internet site, and the Inkwell development team has decided that the Product Registration portlet is the number one priority.

## **3.2 Designing the Product Registration portlet**

The first application the development team will create is the product registration application. It was deemed by the lead developer on the team that this application would be a good starting point for the development team to get their feet wet with Liferay. Providing this application early in the development

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

phase of the project will also help the content management team as they determine where on the site the application will be placed.

- 
- 

### 3.2.1 A blue print of the portlet

The team gathered the requirements from Marketing and came up with the following design for the product registration form. This design is based on the registration card that is shipped with the product documentation.

#### Inkwell Product Registration Form

First Name:	<input type="text"/>	
Last Name:	<input type="text"/>	
Street Address 1:	<input type="text"/>	
Street Address 2:	<input type="text"/>	
City:	<input type="text"/>	
State:	<input type="text"/>	
Zip:	<input type="text"/>	
Country:	<input type="text"/>	
Email Address:	<input type="text"/>	
Phone Number:	<input type="text"/>	
Date Purchased:	<input type="text"/>	
Date of Birth:	<input type="text"/>	
Gender:	<input type="radio"/> Male	<input type="radio"/> Female
How did you hear about this Inkwell product?	<input type="text"/>	
Where did you purchase your Inkwell product?	<input type="text"/>	
Product Serial Number:	<input type="text"/>	
Product Type:	<input type="text"/>	

TV Advertisement  
Radio Advertisement  
TV News  
Magazine Article  
Retail Store  
Friend/Family member  
inkwell.com  
Other web site  
Trade Show  
Home Shopping

Retail Store  
TV Shopping Network  
Gift  
Catalog  
Online Retailer  
inkwell.com  
Other

Fountain Pen  
PDA Pen  
Pen Top

Figure 3.3 A design mockup of Inkwell's Product Registration form. This form appears on a business reply mail card which is shipped with all of Inkwell's products.

When the team began the design process based on this form, they realized that many of the values on this form are already captured during Liferay's registration process. So it was decided that the application would pre-fill the form with values from the Liferay database if a user with an ID fills it out.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>



Otherwise, the form will be blank. This prevents users from having to register on the site in order to register a product.

Additionally, every field on the form that Liferay requires for user registration will be required. This will allow the development team to add a check box later to the form asking if the user wants to register on the site using this information. Once the team is more familiar with the Liferay API, it is assumed that it will be a fairly easy process to just go ahead and register users who check this box. The next step was to pass the form to the database group so the table design could be done.

### 3.2.2 *Designing the database tables*

The form requirements were then passed on to the database analysts (DBAs) and they came up with the following table design:

Table 3.x Product Table

Field Name	Field Type	Length	Description
productId	Integer	19	Unique key for each entry
productName	Varchar	75	Name of product
serialNumber	Varchar	75	Serial number mask of product

Table 3.x Registered User Table

Field Name	Field Type	Length	Description
regUserId	Integer	19	Unique key for each entry
userId	Integer	19	Liferay user ID, if applicable
firstName	Varchar	75	First Name
lastName	Varchar	75	Last Name
address1	Varchar	75	Address, first line
address2	Varchar	75	Address, second line
city	Varchar	75	City
state	Varchar	75	State

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

postalCode	Varchar	75	Postal code
country	Varchar	75	Country
email	Varchar	75	Email address
birthDate	DateTime		Birth Date
gender	Integer	3	Gender

**Table 3.x Registration Table**

Field Name	Field Type	Length	Description
regId	Integer	19	Unique key for each entry
regUserId	Integer	19	Registered User ID
datePurchased	DateTime		Date the product was purchased
howHear	Varchar	75	How did you hear about this product?
wherePurchase	Varchar	75	Where did you purchase your product?
serialNumber	Varchar	75	Product Serial Number
productType	Varchar	75	Product type

Because it is likely that one customer may purchase several different products, the registrations are stored in a separate table from the users. The Registered User table will only be used for those who decide *not* to obtain a portal user ID. The portal data will be considered to “override” data in the Registered User table. The Product table will be used by site administrators to add the products that can be registered to the database. These will then be shown in a selection box at runtime.

Now that we know what we're going to need to build, let's get started building it.

### **3.2.3 Defining portlet modes and generating the project**

The Product Registration portlet will implement two modes from the Portlet API: view mode and edit mode. Edit mode will be completely hidden from regular users of the portal. Administrators will be able to use edit mode to add products for which users can register. These products will appear in the combo box on the form so that users can pick them when registering their purchased products.

The view mode of this portlet will display a button that says *Register*. If a user clicks the button, he or she will be brought to a form like the mock-up above. The user can then fill out the form, and the

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

data will be stored in the tables above. Once the data is stored, a message will be displayed to the user thanking him or her for registering the product.

Help mode will not be needed or implemented, as the portlet has been designed to be as self-explanatory as possible. It will be placed on Inkwell's public web site.

The first thing you want to do if you haven't already is to generate a new portlet project in the Plugins SDK. Use the following command on LUM or Windows to do this.

LUM:

```
./create.sh product-registration "Product Registration"
```

Windows:

```
create product-registration "Product Registration"
```

Import the project into your IDE of choice. We're not going to touch the portlet just yet. Instead, we'll go right to using Service Builder to generate our persistence objects, and we'll implement the portlet once our database layer is complete.

### **3.3 Automating DB code with Service Builder**

If you are an experienced developer, it's likely that you've dealt with a data-driven application before. Liferay ships with a tool called Service Builder, which makes the creation of data-driven applications very easy. I highly recommend that you use Service Builder when writing applications on Liferay's platform, as it will really help to get you going very quickly. How? By generating a lot of the database plumbing code for you, so you can concentrate on your application's functionality.

We're going to take a look at the need which Service Builder is responding to, as well as how to configure and then run Service Builder to create the layer of code that handles your database transactions.

#### **3.3.1 Filling a definite need**

In the old days, we used to roll our own JDBC code by getting a connection to the database and implementing our own methods which had SQL embedded in them. Of course, this was fraught with problems. Maintenance of this code was difficult, because developers had to map predefined SQL statements with application functions which tended to change frequently. Another issue was that applications became tightly coupled to the database layer, presenting challenges to the portability, stability, and maintainability of the applications. Applications became susceptible to SQL injection attacks. Developers had to worry about opening and closing connections manually, managing transactions manually, and a slew of other things. Doing this well required a lot of knowledge of both databases and good programming practices.

The next thing everybody tried were Container Managed Enterprise Java Beans. Using these, developers could rely on the EJB container to manage connection pools, dynamically insert parameters, and more. Often, application server vendors would supply tools for generating these automatically, including the queries the application would need. The problem with this, of course, is that using these tools tied your application to that particular application server and the particular database against which you generated your EJBs—as well as to the tool that was used to generate the code. Additionally, developers found EJBs to be complex to write and to carry too much overhead in memory and processing power. Another solution was needed, and again, it was open source to the rescue.

The Hibernate project took a different approach. In order to offer the developer a simple API to accessing data from a database, Hibernate does what is called *object/relational mapping* (ORM). Using

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

this paradigm, database tables are *mapped* to Java objects. This frees the developer to simply work with the Java objects with which he or she is already familiar, and Hibernate will take care of persisting those objects to the database—any database which Hibernate supports. Your database code is now much more easily applied to multiple databases, allowing your application to be deployed on a wider variety of systems. This is sort of an over simplified description of Hibernate—because it includes a lot more—but it should be sufficient for our needs here. If you want to learn more about Hibernate, I strongly recommend you pick up *Java Persistence with Hibernate*, also from Manning Publications.

Another technology that is sometimes used with Hibernate but is also useful for many other things is Spring. If you've ever heard of Dependency Injection, then you've probably heard of Spring. And if you haven't heard of Dependency Injection, it's a totally different way of thinking about how your application is organized. For example, all Java developers are familiar with this sort of thing:

```
MyObject something = new MyObject();
```

This is one of the first things you learn in a Java class: using a constructor to create a new instance of an object. You do this all the time without thinking about it, if you're writing Java code. But what if, before writing something like this, you first asked yourself this question: Does the class I'm working on *depend* on a fully instantiated object of this type? In other words, when you create the instance of this object, do you have to then populate it with a bunch of stuff that you already have? If so, you are going to like Dependency Injection.

Interacting with a database manually requires that you juggle a bunch of objects that you set up once and then keep passing around: *Connection*, *DataSource*, *DriverManager*, etc. Whenever you make a query, you need to pass around these objects (some of which wrap the others) in order to get something out of the database. Hibernate requires the same sort of things, but the objects are slightly different (*SessionFactory*, *DataSource*, *HibernateProperties*, and more). Wouldn't it be great if you could set up all this stuff once and then *inject* already instantiated versions of these objects into the classes that need them? Wouldn't it be great if you could automatically instantiate objects with known parameters and then inject those objects into other objects that need them? With Spring, you can. Dependency injection allows you to define objects that depend on other objects, and Spring can inject instances of those needed objects automatically. All you need to do is configure an XML file which defines those dependencies. Using Spring, you can automatically inject a Hibernate session right into the code which queries the database, saving you from having to get an instance of the session every time you want to ask the database for data. The combination of Spring and Hibernate are used all the time, because it makes developers' jobs a lot easier.

But let's go one step further. What if you could simply define a database table in an XML file and from that definition, generate all of the Hibernate configuration, all of the Spring configuration, finder methods, your model layer, the SQL to create the table on all leading databases, and your entire Data Access Object layer in one fell swoop? That's what Service Builder gives you. Figure 5.4 illustrates this.

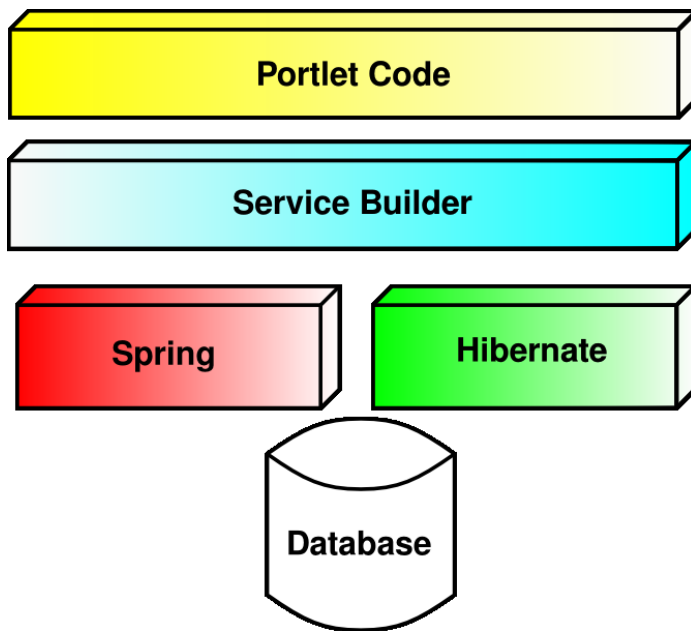


Figure 3.4 Service Builder is a tool that sits on top of Hibernate and Spring, automatically generating both configurations for you—along with the Java code necessary to persist your entities to the database.

This tool is an excellent database persistence code generator which makes it easy to define new tables and to manipulate the data through select, insert, update, and delete operations. It uses a combination of standard technologies that Java developers use every day—Hibernate and Spring—to do this. It is important to note that all of Liferay's internal database persistence is generated using this tool, so it is a proven tool which produces code that is suitable for enterprise deployments.

If you're like me, I know what popped into your head immediately when I said "code generator." It was, "Oh, no. Code generators are bad." And then you began justifying that statement with many sound, accurate, and excellent arguments. Believe me, I agree with you. But Service Builder is different. You'll see why in more detail as the rest of the chapter unfolds, but let me try to reassure you—from one code generator "hater" to another—Service Builder is designed to *enable* you to write custom code, not prevent it. It just takes care of the mundane stuff you hate writing anyway. *That* is a code generator I can get on board with, and I think you'll like it too.

We will use Service Builder to generate the database tables that have been defined above for Inkwell's Product Registration portlet, and then we will make use of it for our database persistence too. To start, we have to create that one xml file that is the key to generating the rest of the code.

### 3.3.2 Creating the service.xml file

Create a file called `service.xml` in the *WEB-INF* folder of your project. This file will contain your table definitions which have already been defined above. You will populate this file with all the information Service Builder needs to generate the SQL to create the table—for all the databases Liferay supports—as well as database persistence objects you can use in your Java code.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

We'll start with just the simplest of the tables above, which is the Product table. Use the code below to define the table for Service Builder.

### Listing 3.1 Defining a table using Service Builder

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE service-builder PUBLIC "-//Liferay//DTD Service Builder 6.0.0//EN"
"http://www.liferay.com/dtd/liferay-service-builder_6_0_0.dtd">

<service-builder package-path="com.inkwell.internet.productregistration"> 1

  <author>Rich Sezov</author> 2

  <namespace>PR</namespace> 3

  <entity name="PRProduct" local-service="true" remote-service="false"> 4

    <column name="productId" type="long" primary="true" /> 5

    <column name="productName" type="String" /> 6
    <column name="serialNumber" type="String" />

    <column name="companyId" type="long" /> 7
    <column name="groupId" type="long" />

    <order by="asc"> 8
      <order-column name="productName" />
    </order>

    <finder name="G_PN" return-type="Collection"> 9
      <finder-column name="groupId" />
      <finder-column name="productName" />
    </finder>

    <finder name="GroupId" return-type="Collection">
      <finder-column name="groupId" />
    </finder>

    <finder name="CompanyId" return-type="Collection">
      <finder-column name="companyId" />
    </finder>

  </entity>

</service-builder>
#1 Java package for code
#2 Author for JavaDoc
#3 Namespace for tables
#4 Name of entity
#5 Primary key
#6 Additional fields
#7 Foreign Keys
#8 Order data is returned
#9 Finder methods
```

## Cueballs in four paragraphs below

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

## SECTION 1 – GLOBAL INFORMATION

This file is pretty easy to read, so we'll attack it section by section instead of line by line.

You can define the Java package into which Service Builder generates the code that powers this database table, and you can see this in #1. Service Builder also generates JavaDoc, and the name you place in the Author tags in #2 will wind up in the JavaDoc as the author of the code. By default, tables you define with Service Builder will go in the Liferay database. To set them off from the rest of Liferay's tables, you can prefix them with a name space, as in #3. So this table, when created, will actually be called `PR_PRProducts` in the database.

## SECTION 2 – DEFINE AN ENTITY

Now we get to the cool stuff. The database entity—which for us, is a Product—is defined using the Entity tag that you see in #4. The two parameters in this tag define how you want Service Builder to generate the service that retrieves and saves these entities. You need to at least have a local service. But you can also have a *remote* service. This is not an EJB. It is instead a web service, complete with a WSDL document describing it, so that your service may participate as part of a Services Oriented Architecture (SOA).

## SECTION 3 – DEFINE COLUMNS

All that's left is to define the columns and finder methods. #5 is our first column and is defined as a primary key in the database. Other fields we want to store come next (#6). We are also defining two foreign keys in #7: a `companyId` and a `groupId`. Notice that the DBA team did not specify these two foreign key fields in the tables above, but we added them anyway. This has been done because the DBAs didn't know the internal workings of Liferay when they did their table design. These fields are internal to Liferay, and are used for context purposes in non-instanceable portlets. The `companyId` corresponds to the portal to which the user has navigated, and the `groupId` corresponds to the community or organization to which the user has navigated. Since we will be using these field values as parameters in all of our queries, our portlet will have different data in different portals, communities, and organizations. Quick test: without reading the next sentence, is our portlet an instanceable or non-instanceable portlet? It's a non-instanceable portlet, because we'll be using these fields to make sure that the portlet ties all data to the portal and to the community or organization upon which the portlet is placed.

#8 defines a default ordering of the entities when they are retrieved from the database. You can choose to have them returned in ascending or descending order. You aren't limited to one column here; you can specify multiple columns, and the ordering will happen by priority in order of the columns.

## SECTION 4 – DEFINE FINDER METHODS

The finder methods which actually go and retrieve the objects appear in #9. Specifying these finders means that Service Builder will automatically generate methods which retrieve objects from the database using the parameters you specify in the finder. For example, the first finder returns Products by `groupId` and `productName`. Now that we've defined our table, it's time to run Service Builder.

### 3.3.3 Running Service Builder

Save the `service.xml` file and then run the Ant task called `build-service`. You will see several files get generated and the task will complete with a BUILD SUCCESSFUL message. If you're using an IDE that compiles source files automatically, you'll notice that errors suddenly appeared in your project. Don't worry about this; it's easy to fix.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

What has happened is that Service Builder generated several Java source files. Some of them are in your existing `src` folder. Others are in a new source folder called *service* that it also generated. This folder will reside inside of `WEB-INF` like the `src` folder you already have. To fix the errors in your project, just use your IDE's facility to add another source folder to your project. You may also need to refresh your project. Once you do that, the errors will go away.

Service Builder divides the source it generates into two layers: an interface layer and an implementation layer. The interface layer gets generated in the aforementioned *service* folder. You will never change anything in the interface layer manually; Service Builder always generates the code that is found there. The implementation layer gets generated in your `src` folder and is initially just skeleton code which allows you to implement the functionality you need.

You will notice that there is also a new file in the root of your `src` folder called `service.properties`. This file was also generated by Service Builder. It contains properties that Service Builder needs at runtime in order to perform its functions. The most important of these properties is a list of Spring configuration files which were also generated.

Another new construct that was generated was a `META-INF` folder in your `src` folder. This folder contains all of the XML configuration files that Service Builder needs, including Spring configuration files and Hibernate configuration.

What all of this means is that Service Builder has taken care of all of your database persistence configuration for you. If you have ever used Hibernate alone or the combination of Hibernate and Spring, you know that there are multiple configuration files to juggle between the two. Service Builder automatically configures all of that for you and provides you with static classes that you can use to perform all of your database persistence functions. It provides both a Data Access Object (DAO) layer and a Data Transfer Object (DTO) layer for you automatically. Our next step is to provide the functionality in our DTO to keep our portlet code from being dependent on anything having to do with SQL databases.

### **3.4 Using DAOs and DTOs**

Liferay's Service Builder encourages the proper layering of functionality within a portlet application. Generally, when building an application, it is a good practice to separate the various layers of the application: UI, model, persistence, etc. This is sometimes called *Separation of Concerns*. By keeping the layers as separate as possible, you gain the ability to change the implementation of any one layer more easily—if for some reason you find a better way to do it later.

To some, this must seem like some unnecessary work, but let me give you an example as to why this is important.

#### **3.4.1 Why layering is important**

Consider this example: say you have a poorly designed application which consists of only two layers: your JSPs and your portlet class. You develop the application by creating action URLs in your JSPs which correspond to various methods within the portlet. These methods use JDBC to connect to a database and select, insert, update, and delete data. Your database, validation, and page navigation code is all in the portlet or the JSPs, resulting in very large files with lots of logic in them. You test the portlet as well as you can, and it then gets deployed to production.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>



Within the first day of its use, a bug is found. Your application has a last name field, and a user tried to insert the last name *O'Bannon*. Because you coded all of your database interaction manually, you forgot that sometimes you needed to escape out certain characters. In this case, the apostrophe (') in the name is causing your database insert to fail.

With the poor design you have used for your portlet, you will now have to modify the Insert and Update methods of your portlet class and add some code that checks the values your users are inserting for apostrophes. This is an example of a *tightly-coupled* design: your database code is inside your portlet, so your implementation of code which communicates with the database is intertwined with your business and display logic. Because you were recently working on this portlet, you know where the problem is and so you fix it and redeploy the application, and the users work with it for several months before another bug is found. It seems that somewhere in the portlet, a database connection isn't being closed and is causing the database server to run out of connections.

Now some significant time has passed and you're onto another project, and you don't remember exactly what you did when you wrote this one. You now have to slog through mounds of spaghetti code in your one portlet class looking for the problem: database code which is mixed in with all kinds of other functionality, such as JavaScript functions, field validation code, database access code, and so on. It's become a hard-to-maintain mess. After hours of debugging, you finally found the condition where the database connection wasn't being closed, and it was part of a long block of if-then-else conditions, which included field validation logic, business logic, and persistence logic. It took you almost an entire day to find this one problem, because it was buried in a lot of code that should have been separated out into different layers of code logic.

If you had properly layered your application so that the UI code was in one place, the business logic was another, and the persistence logic was in another, you'd have had a much more maintainable project. And you might decide at that point that you could use something like Hibernate for your database persistence, and you'd only have to replace a few classes in the one layer. Doing that would solve all of your persistence problems, because Hibernate escapes characters and closes connections automatically. You might then go another step further and replace your home grown application logic with an MVC framework like JSF or Struts. On each refactor, you can modify one layer of your application at a time. Let's see how to do that for the persistence layer.

### **3.4.2 Using two layers for persistence**

While the above is a simplistic example, it serves as an example we can use to show how Service Builder encourages proper separation of concerns. For database persistence, two of these layers are the *Data Access Object* and the *Data Transfer Object*. These two layers can be written by developers manually, and often a lot of time is spent writing and debugging code in this layer. Service Builder generates both layers for you, automatically.

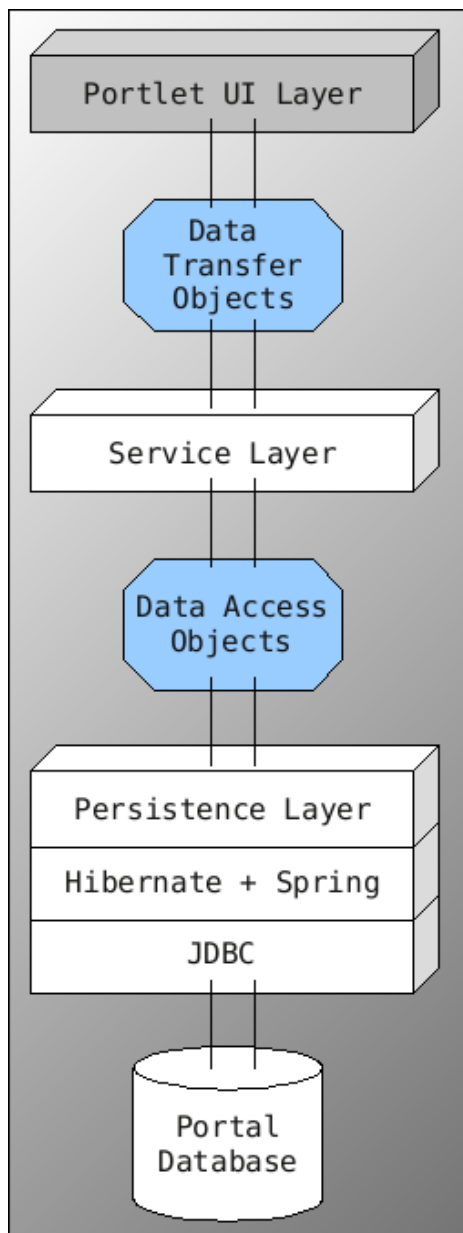


Figure 3.5 Service Builder generates for you everything from the Portal UI layer down. You will need to modify / customize some of this code, but you won't have to touch most of it. This significantly increases developer productivity.

The layer you will be working with for the most part is the Data Transfer Objects layer. This layer talks to the Service Layer, which is automatically generated and allows you to work with objects that

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

need to be persisted or have been retrieved from the database. These are the *-Impl* classes that you have generated, and this is the reason why they were placed in your *src* folder with the rest of your portlet code. You will call methods from the Data Access Objects in the persistence layer to do the actual persisting. These methods are generated from the *<finder>* tags you placed in your *service.xml* file.

Let's see how this actually works.

Though we haven't created it yet, our design calls for a form in Edit mode of the portlet. This form would allow users to enter products which will appear in a drop-down selection box in the actual registration form that makes up the portlet's View mode. Only two fields will be coming from this form (because the third is the primary key, which will be generated): 1) the name of the product, and 2) a serial number mask which is set by the manufacturing department and can be used for field validation. So how do we get the data the user entered into the database?

Remember: we are after the idea of *loose coupling*. This means that we don't want to have any database insertion code in our business logic. So our business logic (when we get to it) will need to call a generic method that has nothing to do with the database. This is where the DTO layer comes in. It acts as a buffer between your business logic and your underlying database code. Service Builder generates both layers, leaving the DTO layer as a stub for you to implement as you wish.

### **3.4.3 Implementing the DTO layer**

Presumably, we'll want a method that has two parameters in the method signature for our two values: the product name and the product's serial number mask. So let's create it. Open the file `PRProductLocalServiceImpl.java`. You will find this in a newly generated package called `com.inkwell.internet.productregistration.service.impl`. This class extends a generated class called `PRProductLocalServiceBaseImpl` (see figure 5.6). If you are using an IDE (such as Eclipse) which does not automatically recognize when another tool adds files, you may need to click on the project name and press F5 to refresh the project in order to see this package. The first thing we'll implement in `PRProductLocalServiceImpl` is adding a product.

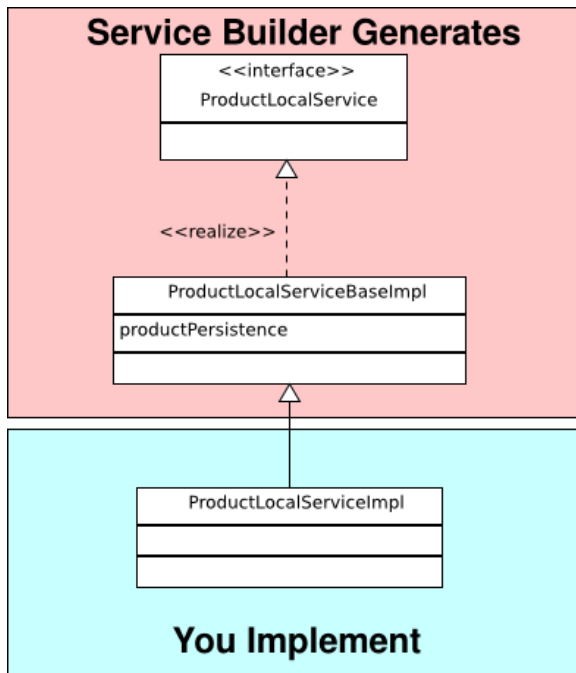


Figure 3.6 Service Builder generates both your DAO and DTO layer. Shown above is the DTO layer, which has an instance of the DAO layer (the `productPersistence` attribute) injected into it by Spring.

Once we implement adding a product, implementing the other methods will be very simple.

#### ADDING A PRODUCT

Another great thing about the design of Service Builder is that you never have to touch or modify any of the classes it generates. All of the Spring dependency injection, the Hibernate session management, and the query logic stays in classes you never have to touch. You add code in a class that extends the generated class, and if you add something which changes the interface/implementation contract, those changes are propagated up the chain so the contract is never broken. We'll see an example of this a bit later. For now, you will see that this file is just an empty stub. This is because you have yet to implement anything. So we'll implement our first method, which will take the values passed to it and call the underlying database code to persist the data to the database:

#### Listing 3.2 Adding a product to the database

```

public PRProduct addProduct(PRProduct newProduct, long userId)
    throws SystemException, PortalException {
    PRProduct product =
prProductPersistence.create(counterLocalService.increment(PRProduct.class.
    getName()));
    resourceLocalService.addResources(newProduct.getCompanyId(),
    newProduct.getGroupId(), userId,
    Product.class.getName(), product.getPrimaryKey(), false, true, true);
}
  
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

```

product.setProductName(newProduct.getProductName());
product.setSerialNumber(newProduct.getSerialNumber());
product.setCompanyId(newProduct.getCompanyId());
product.setGroupId(newProduct.getGroupId());
return prProductPersistence.update(product, false);
}

```

#### **#1 Create empty object**

#### **#2 Create permissions resources**

The first thing we do (#1) in this method is create a new Product object. Product is the Interface (inherited from PRProductModel), and PRProductImpl (which extends PRProductModelImpl) is the implementation. Note that all we have to do is specify the Interface here, as a Factory design pattern is used to create the object for us, using the prProductPersistence object which has been injected into this class by Spring. The interface and implementation of this object were both generated automatically by Service Builder, and the fields within the object map directly to the fields in the Product table that was defined. In order to obtain a new instance of this object, we call a create method which was also generated by Service Builder.

Because Liferay is database-agnostic, it does not use any database-specific means of generating primary keys for its database. Instead, it provides its own utility for generating primary keys. Since the create method that was generated requires a primary key, we need to call Liferay's Counter utility to generate this. Here's the cool thing about how this is done: the Counter is injected into the class we're working on, automatically. Why? Because if you are doing work with databases and Service Builder, you are 100% likely to need to use the Counter to create new objects that will be persisted.

Notice that in #2 we next make a call to resourceLocalService to persist resources. Resources are used to define permissions on the objects that are persisted. We added CompanyId and GroupId fields to our tables in order to both make our portlet a non-instanceable portlet and to enable us to later implement Liferay's permissions system.

These two variables track the portal instance and the community / organization, respectively. For example, say a user adds our portlet to a page in the *Guest* community of the default portal instance. Users then begin adding Products to the database. Because we have added the Company ID and the Group ID to the product entities, users can add our portlet to another community, organization, or portal instance, and the set of records returned will be different, based on the Company ID and Group ID. Our Finder methods filter the records being returned by Company ID and Group ID.

This is how Liferay allows you to place, for example, a Message Boards portlet in two different communities, and have completely different content in them. We will do the same thing as we write our portlet.

Once we have our object, it is a simple matter to set the proper values in the object and then persist it to the database. We then return the object that we have created back to the layer that called this method in the first place.

We aren't quite done yet. Remember when I said a couple of pages ago that changes to the interface/implementation contract get propagated up the chain automatically by Service Builder? We were just working in an implementation class which extends another class which we call a -BaseImpl class. All of the methods generated by Service Builder have entries in their Interfaces and actual implementations in their -BaseImpl class. If developers wish to add more, they are added in the -LocalServiceImpl class.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

Because we have just added a new method to a class which implements an Interface, there is no method stub in the Interface for the method we just created. So in order to continue without errors, we need to run Service Builder again. This, as we have seen before, can be done by running the *build-service* Ant task. When you run the task, Service Builder will regenerate the interface, which is called `PRProductLocalService`, to include a stub for the method you created in the implementation class.

This is generally the point where people scratch their heads and say, “Isn't this backwards? Aren't you supposed to define the Interface first and then write the implementation?”

In a sense, yes, that is correct. But remember that we are working with a code generator, and it is that code generator's job to make things easier for us as developers. By using Service Builder, we *are* defining the Interface first, as much as we can do that up front. We did this in one step, when we defined our database table in `service.xml`. Service Builder generated our Interface as well as our default implementation—as best as it could guess at what we needed. That's what went in `PRProductLocalServiceImpl.java`. If you need further customization—and you generally do—you can add your own methods. These need to be added to a class that is free of the code generator, so that your changes don't get overwritten. So `PRProductLocalServiceImpl.java` is provided as a DTO layer, which allows you to add any methods you may need.

You will generally only need to make customizations in one class: the `-LocalServiceImpl` class. In some cases, you may need to make customizations in `-Impl` classes in the `model.impl` package, but we will cover that later. For now, the only thing we need to worry about is the `-LocalServiceImpl` class.

Now that we're done with adding products, let's try deleting products.

## DELETING PRODUCTS

Since we can create products now, why not try deleting them? If you think about how you might want to delete products, there are two ways:

- You have a `Product` object already, and you just want to delete it
- You have a `Product`'s primary key, and you want to delete it that way without having to retrieve the whole `Product`

To make things easier for us in the Controller layer, we will overload the *delete* method by creating versions of it that can handle both cases:

```
public void deleteProduct(long productId, long companyId)
    throws NoSuchProductException, SystemException, PortalException {

    PRProduct product = prProductPersistence.findByPrimaryKey(productId);
    deleteProduct(product, companyId);
}

public void deleteProduct(PRProduct product, long companyId)
    throws PortalException, SystemException {

    resourceLocalService.deleteResource(
        companyId, Product.class.getName(),
        Resource.Constants.SCOPE_INDIVIDUAL, product.getPrimaryKey());
    prProductPersistence.remove(product);
}
```

What we have done here is enable us to delete a `PRProduct` using its primary key or the `Product` object itself. If we are using the primary key, we will first retrieve the `Product` object and then call

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

*deleteProduct* on it. Note that we delete the resource as well as deleting the product. This keeps Liferay's permissions tables free of "dead" data that might otherwise be left there simply taking up space. When you are finished modifying your `-Impl` class, run Service Builder again using `ant build-service`. As before, this will add your new methods to the interface.

The only thing remaining to do is to query the database for the products we have entered because presumably, users will want to view them and / or edit them.

#### QUERYING THE DATABASE

Just like adding and deleting, getting products out of the database is really easy. You'll probably remember that we had Service Builder generate a finder for this which queries the database for products by the `GroupId`. We always want to query by `GroupId` because our portlet is non-instanceable. So we'll implement this in our DTO layer like this:

```
public List<PRProduct> getAllProducts(long groupId)
    throws SystemException {

    List<PRProduct> products = prProductPersistence.findByGroupId(groupId);
    return products;
}
```

This will return a `List` of `PRProducts` which can be used in the UI layer to display products for viewing or editing purposes. And as you can see, whatever calls this method (it will be our portlet class) does not need to know anything about JDBC or databases. It's simply requesting a `List`. This frees you to do something like swap out Service Builder later if you find that it doesn't meet your needs.

We started with this table because it was a simple, standalone table. Now that you see how easy it is to use Service Builder, we can move on to looking at tables with relationships.

### 3.5 Service Builder in action

We did the `Product` table first for a reason: it's a simple table which has no relationships to any other tables (well, except for the implied relationships with Liferay's `Resource` and `Counter` tables). The other two tables that Inkwell's DBAs defined do have a relationship: there can be one-to-many Registrations per Registered User. Let's see how we define those two entities and their relationship to each other.

#### 3.5.1 Defining table relationships

Below is how you would set that up in your `service.xml` file:

##### Listing 3.x Remaining entities in `service.xml`

```
<entity name="PRUser" local-service="true" remote-service="false">

    <column name="prUserId" type="long" primary="true" /> #1

    <column name="firstName" type="String" />

    <column name="lastName" type="String" />

    <column name="address1" type="String" />

    <column name="address2" type="String" />

    <column name="city" type="String" />
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

```

<column name="state" type="String" />
<column name="postalCode" type="String" />
<column name="country" type="String" />
<column name="phoneNumber" type="String" />
<column name="email" type="String" />
<column name="birthDate" type="Date" />
<column name="male" type="boolean" />
<column name="userId" type="long" />
<column name="companyId" type="long" />
<column name="groupId" type="long" />
<column
    name="userRegistrations"
    type="Collection"
    entity="PRRegistration"
    mapping-key="prUserId" />
<order by="asc">
    <order-column name="lastName" case-sensitive="false" />
</order>
<finder name="G_LN" return-type="Collection">
    <finder-column name="groupId" />
    <finder-column name="lastName" />
</finder>
<finder name="G_E" return-type="Collection">
    <finder-column name="groupId" />
    <finder-column name="email" />
</finder>
<finder name="G_U" return-type="Collection">
    <finder-column name="groupId" />
    <finder-column name="userId" />
</finder>
</entity>
<entity name="PRRegistration" local-service="true" remote-service="false">
    <column name="registrationId" type="long" primary="true" />

```

A

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>



```

<column name="prUserId" type="long" /> #2
<column name="datePurchased" type="Date" />
<column name="howHear" type="String" />
<column name="wherePurchased" type="String" />
<column name="serialNumber" type="String" />
<column name="productId" type="long" />
<column name="companyId" type="long" />
<column name="groupId" type="long" />
<finder name="GroupId" return-type="Collection">
    <finder-column name="groupId" />
</finder>
<finder name="G_RU" return-type="Collection">
    <finder-column name="groupId" />
    <finder-column name="prUserId" />
</finder>
<finder name="G_DP" return-type="Collection">
    <finder-column name="groupId" />
    <finder-column name="datePurchased" />
</finder>
<finder name="G_SN" return-type="Collection">
    <finder-column name="groupId" />
    <finder-column name="serialNumber" />
</finder>
</entity>

```

#### **A Foreign Key relationship**

##### **#1 Primary Key of User**

##### **#2 Foreign key for User**

In examining the code above, you will see that there isn't much here that you haven't already seen. There is only one new concept here: relationships.

If you recall from the beginning of the chapter, Inkwell's DBAs defined a relationship between a registered user and a particular product registration. This is a one to many relationship: it is hoped that most customers will be repeat customers, and will own many Inkwell products.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

To reflect this in the data, there is a separate `PRUser` table and a separate `PRRegistration` table, with a relationship between them defined by the `prUserId` column. So we define that relationship in the `PRUser` table by using the code highlighted above. This will do two things for us:

- Service Builder will create a Spring configuration that injects the Registration persistence objects into the `regUser` DTO classes, so that all of the operations on Registration objects are available.
- Service Builder will generate methods which will allow us to pull all registrations by a specific `prUserId`. Your `PRUser` objects will have a `getRegistrations()` method which will let you seamlessly pull all registrations which belong to that user into a `List`.

We won't be using the latter query in this portlet, but it will be useful in another application that could be written for the marketing teams to use, to see how much repeat business Inkwell is getting. In fact, that brings up another point: Service Builder classes can be shared.

### 3.5.2 Sharing services

If you are building a larger web site in which several plugins need access to the same services, you can make those services available to the plugins pretty easily. Of course, one way would be to put all of your plugins in the same `.war` file, as we did with the two portlets in the IPC example in the previous chapter. Another way—and one which keeps your projects from being rather monolithic—is to simply make the services available on the classpath of your other projects.

Service Builder makes your services available in a convenient `.jar` file which is generated and placed in the `WEB-INF/lib` folder of your project. You can very easily take this `.jar` file and put it inside another `.war` file or on the global classpath of your application server. This has the effect of making the services in that `.jar` file available to other plugins that might need to access those services.

Inkwell obviously needs these services for the Product Registration portlet, which will be placed on their public web site. But they are also likely to need these services for an internal reporting portlet that can be used by the marketing and support teams. Because Service Builder makes it easy to share services by packaging them separately, this will be no problem for the development team to implement. All they have to do is take the `.jar` file from this project and put it on the global classpath of their application server.

But, of course, we're not done yet implementing our services, so they can't give away that `.jar` file just yet. We need to add the methods to our DTO layer which add registered users and registrations that go with them to the database.

### 3.5.3 Adding registered users and their products

Next, we implement the methods we will need in our DTO layer to add registered users:

#### Listing 3.x Maintaining registered users

```
package com.inkwell.internet.productregistration.service.impl;

import com.inkwell.internet.productregistration.model.RegUser;
import
com.inkwell.internet.productregistration.service.base.RegUserLocalServiceBaseImpl;

import com.liferay.portal.kernel.exception.PortalException;
import com.liferay.portal.kernel.exception.SystemException;
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

```

public class PRUserLocalServiceImpl extends PRUserLocalServiceBaseImpl {

    public PRUser addPRUser(PRUser user, long userId) throws SystemException,
        PortalException {
        PRUser prUser =
prUserPersistence.create(
    counterLocalService.increment(PRUser.class.getName()));           A

        resourceLocalService.addResources(PRUser.getCompanyId(), prUser.getGroupId(),
            PRUser.class.getName(), false);                             B

        prUser.setAddress1(user.getAddress1());
        prUser.setAddress2(user.getAddress2());
        prUser.setBirthDate(user.getBirthDate());
        prUser.setCity(user.getCity());
        prUser.setCompanyId(user.getCompanyId());
        prUser.setCountry(user.getCountry());
        prUser.setEmail(user.getEmail());
        prUser.setFirstName(user.getFirstName());
        prUser.setGroupId(user.getGroupId());
        prUser.setLastName(user.getLastName());
        prUser.setMale(user.getMale());
        prUser.setPhoneNumber(user.getPhoneNumber());
        prUser.setPostalCode(user.getPostalCode());
        prUser.setState(user.getState());

        return prUserPersistence.update(prUser, false);               C
    }
}

```

**#A Create empty entity**

**#B Add resources**

**#C Add filled entity to database**

We implement only an *add* method here, because this portlet is designed for end users to add their registrations to the database through the web site, rather than submit postcards by mail. For that reason, we don't need to implement anything but *add* functionality.

Next, we add our DTO methods for our registration objects:

### Listing 3.x Adding registrations

```

package com.inkwell.internet.productregistration.service.impl;

import com.inkwell.internet.productregistration.model.PRRegistration;
import
com.inkwell.internet.productregistration.service.base.PRRegistrationLocalServiceBaseI
mpl;

import com.liferay.portal.kernel.exception.PortalException;
import com.liferay.portal.kernel.exception.SystemException;

import java.util.List;

public class PRRegistrationLocalServiceImpl
    extends PRRegistrationLocalServiceBaseImpl {

    public PRRegistration addRegistration(PRRegistration reg)
        throws SystemException, PortalException {           A

```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

```

        PRRegistration registration =
prRegistrationPersistence.create(
            counterLocalService.increment(PRRegistration.class.getName()));

        resourceLocalService.addResources(registration.getCompanyId(),
            registration.getGroupId(), PRRegistration.class.getName(),
            false);

        registration.setCompanyId(reg.getCompanyId());
        registration.setDatePurchased(reg.getDatePurchased());
        registration.setGroupId(reg.getGroupId());
        registration.setHowHear(reg.getHowHear());
        registration.setProductId(reg.getProductId());
        registration.setPRUserId(reg.getRegUserId());
        registration.setSerialNumber(reg.getSerialNumber());
        registration.setWherePurchased(reg.getWherePurchased());

        return prRegistrationPersistence.update(registration, false);
    }

    public List<PRRegistration> getAllRegistrations(long groupId) throws
        SystemException {
        List<PRRegistration> registrations =
            prRegistrationPersistence.findByGroupId(groupId);
        return registrations;
    }
}

```

#### **A Adding a Registration**

#### **B Getting All Registrations by GroupID**

As you can see, in addition to the method for adding registrations, we have provided a method for getting all the registrations by `groupId` out of the database as an example. If the team writing the reporting portlet for marketing requires more services, it would be very easy to add them to this class. We will implement a rudimentary “view registrations” screen in the meantime while that other portlet is having its requirements gathered.

This is all the database interaction that the Product Registration portlet needs. From here, we get to move on to the portlet layer of the application, which is what we'll look at in the next chapter.

## **3.6 Summary**

After taking a look at the Inkwel development team's design for the Product Registration portlet, we saw that we could jump start development by making use of Liferay's code generator for database persistence, which is called Service Builder. This utility (which ships as part of Liferay) creates code and SQL for accessing database from within portlets. Since it uses Spring and Hibernate to implement this, it is not much different from what developers would already do manually, with the important exception that it does much of this “grunt work” automatically, freeing time for developers to implement their business logic.

Service Builder makes it really easy to generate a whole persistence layer of an application. Using well-known design patterns such as Data Access Objects and Data Transfer Objects, it not only implements a consistent design but also helps the developer to do so. Finders are automatically generated which allow developers to access their data as Java objects.

Stay tuned to the rest of this book, as this only scratches the surface of what Service Builder can do. Later, we'll see that we can use Liferay's Dynamic Query API, which implements the most widely used

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

portions of the Hibernate Criteria Query API, and we'll also see how you can use your own custom SQL with Service Builder. For now, we have an application to finish, so we'll be continuing with the Product Registration portlet in the next chapter.

# 4

## *MVC the Liferay Way*

I hope that you didn't begin to 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 over-used "buzzwords" (if you can call an abbreviation a buzzword) you'll see out there. There has been framework after framework 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 just seem to keep multiplying like some kind of virus.

With that in mind, what in the world is Liferay thinking by having their *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 which 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 out there: 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 which you've already seen. If you're going to get into Liferay—particularly by writing and / or customizing existing portlets written by Liferay, you might as well become familiar with Liferay's `MVCPortlet`, as you'll be seeing it anyway. And you may find that you like it once you start using it.

We're going to continue working with the Product Registration Portlet which we started in the last chapter. In that chapter, we used Service Builder to create the database persistence layer, or service layer, of our application. Now that all of that foundational code is done, we can concentrate on the layers of our application which interact with our users. These layers comprise the Model, the View, and the Controller.

### **4.1 Using Model-View-Controller**

By using this pattern, we are separating our concerns into various layers of the application, just as we did with our Data Transfer Objects and the Data Access Objects in our service layer. If you have 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

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

a way that is a bit easier to use. Let's look at the various components of the MVC design pattern and see how these are implemented.

**Model:** The model layer of the application holds the data of the application and contains any business rules for manipulating that data. Our Product object with its fields containing values for the name and serial number is part of our model, and was generated by Service Builder. Any logic which 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 of 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 all functions provided by the view layer. We will create JSPs which will handle our view layer, and we'll see some tools that Liferay provides which 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 on and then directs processing to the proper function which would 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).

These three layers are pictured in figure 4.1.

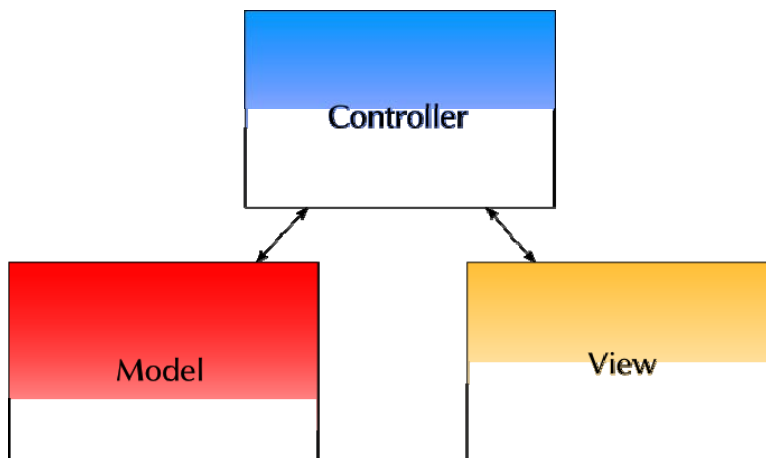


Figure 4.1 The MVC design pattern provides an easy way to structure your web application, by separating your concerns into easily digestible components.

So how would we implement this design pattern in a portlet? We already have our model layer generated for us by Service Builder from our table design. We have stated above that our JSPs will be our view layer. That leaves our portlet class as our controller. As you will see, this design will provide for us a good implementation of the MVC design pattern, allowing us to separate our concerns in a way that is maintainable and straightforward.

One nice thing we have seen already in the *ipc-baseball* portlet is that the Portlet 2.0 API already gives us a rudimentary controller mechanism by implementing a `processAction` method that forwards processing to other methods using annotations. You could easily use this mechanism for your controller

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

functionality. But Liferay has subclassed `GenericPortlet` to provide functionality which makes it even easier to create MVC applications using portlets. So rather than focusing our efforts on making cross-platform portlets, we're going to look at doing things the Liferay way. If you are interested in further information about creating portlets that work on every portal server, I highly recommend you check out *Portlets in Action*.

So, 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 very 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 does several things for you:

- You don't have to worry about `doView()`, `doEdit()`, or any other do- method.
- Your portlet class won't need to implement any portlet-specific APIs.
- Your portlet class will be very simple: all it will contain is action methods.
- If you wish, you can still 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.

Let's take a closer look at Liferay's way of doing MVC, and we'll apply that to our specific project.

#### **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 all that 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 for that thing that the inventors never dreamed of. Consider the case of Lawn Chair Larry. Larry always dreamed of flying, but poor eyesight kept him from service 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 which 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.<sup>2</sup> The point? I am sure that the inventor of the weather balloon never envisioned someone using it for this.

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 does not provide enough flexibility for the "wild, wild west" of custom designed websites all over the Internet whose designers don't want to be locked into the boxy, window-like interface of a portlet. Minimize, maximize, and close buttons are for desktop operating systems, not the web. And portlet modes have not been used very much. I have yet to see someone actually implement Help mode in a portlet. Why? Because there are much better ways to implement a help system in your application than by using Help mode.

---

<sup>2</sup> <http://darwinawards.com/stupid/stupid1998-11.html>



Edit mode, while the most useful of the portlet modes, 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 settings about the portlet. And of course, if you use edit mode, you are locked into that boxy design for your web site, because you have to provide that title bar so the icon can reside there.

Liferay has the unique position of being used for Internet-based websites just as often as it is used internally for large enterprises. For that reason, a slightly different (and we 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 just for those settings and embed it in Liferay's Control Panel. This puts all settings and options in one place, rather than having them scattered around your website. And 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 our Product Registration portlet, we'll create a separate portlet in the same application and embed it in the Control Panel.

There are benefits to this 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, it also gives your end users freedom to arrange your application on the page any way they like.

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

Content for **Guest** ▾
[Back to Guest](#)

Product Administration



[Display Registrations](#)

Product Name

Serial Number Mask

Product Name	Serial Number Mask	
Fountain Pens	PEN-	<a href="#">Actions</a>
PDA Pen	PDA-	<a href="#">Actions</a>
Pen Top	PTOP-	<a href="#">Actions</a>
USB Pen	USB-	<a href="#">Actions</a>

Showing 4 results.

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

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

### 4.1.2 MVC according to Liferay

When you first create a portlet in the Plugins SDK, you have 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 it do that?

If you take a 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 we've already been working with, but it contains enhancements. `MVCPortlet` can do all page management for us. It does this by providing a default view, which is defined as an `init` parameter for each portlet mode. Since 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 here:

```
<init-param>
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

```

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

```

So just by generating the project, you have a portlet that works (though it doesn't do much), and doesn't have a portlet class. To do anything interesting, you need to create a portlet class, as we have done in the previous chapters.

What we're going to do now is extend `MVCPortlet` instead of `GenericPortlet` so that we can take advantage of its page management features and thus make our portlet smaller and easier to work with. When we're done, we'll find that our portlet contains nothing but action methods. All page management is controlled through the mechanisms that `MVCPortlet` gives us. The first thing we need to do is configure some deployment descriptors to reflect the two portlets we'll have.

## 4.2 Configuring the portlet project

We'll be creating a project that has two portlets in it. We haven't worried so much about the portlet configuration so far, but we have generated all of the persistence code that we will be needing. The Plugins SDK generated for us a project that has a single portlet in it with the same name as the project, which we called *product-registration*. Since we will be having a Product Registration portlet, we'll keep that configuration, and we'll add a new portlet for our Product Admin portlet. To do this, we have to modify the deployment descriptors for our project.

### 4.2.1 Defining portlets in your deployment descriptors

Open the `portlet.xml` file and add the portlet configuration in listing 4.1 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.ProductAdminPortl
et</portlet-class>
  <init-param>
    <name>view-jsp</name>
    <value>/admin/view.jsp</value> 1
  </init-param>
  <init-param>
    <name>add-process-action-success-action</name> 2
    <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
  <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>

```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

```

</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>
#1 Moves JSP to admin folder
#2 Disables Liferay status messages
#3 Creates a multi-lingual portlet

```

Once you've added this portlet, you'll want to go ahead and create the *admin* folder under the *docroot* folder of the project, because #1 is telling MVCPortlet that our 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 will by default do this automatically after every portlet action. Since we don't want it to do this every time, we can use the initialization parameter in #2 to turn that functionality off. We are also going to be creating a multi-lingual portlet, so we'll need a language bundle for that (#3).

The next file we need to edit is `liferay-portlet.xml`. As you'll remember, this file is the Liferay-specific deployment descriptor. We'll be needing some custom settings for our administration portlet, which you can see in listing 4.2.

#### 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> 1
  <control-panel-entry-weight>1.5</control-panel-entry-weight> 2
  <header-portlet-css>/css/product-admin.css</header-portlet-css>
  <header-portlet-javascript>/js/test.js</header-portlet-javascript>
</portlet>
#1 Adds portlet to Control Panel
#2 Ordering info for Control Panel

```

To put our portlet in the Control Panel, we need only tell Liferay we want to do that via its deployment descriptor. We are adding our portlet to the Content area of the control panel (#1), and specifying a weight (#2), that determines where it appears in the list.

The Control Panel is divided into four areas which each have a particular purpose.

- **Personal**—Used for administration items which the logged in user needs to access, such as My Account.
- **Content**—Used to administer any type of content in the portal. Since we are defining Inkwell's products as content, we have added our portlet to this section.
- **Portal**—Contains administration tools which affect the portal globally.
- **Server**—Contains administration tools which affect the entire Liferay installation.

As you can probably tell, the Control Panel looks a bit different from the rest of the portal, but now you know it is populated entirely with portlets! So you already have the basic knowledge for adding anything you want to the Control Panel.

Defining the weight the way we have makes sense once you understand that the weights for the default items range from 1.0 to 11.0. So by making the weight for our portlet 1.5, we are making sure it appears second in the list.

Lastly, we have one final configuration file: `liferay-display.xml`. This is how we 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, we are creating a category for where Inkwell's portlets destined for its Internet site will go. We also have another category, called *category.hidden*. Any portlets in this category are prevented from appearing in the Add > More menu, so we will put our administrative portlet there. Since it appears in the Control Panel, we don't want users to also be able to place it on pages.

We've now finished configuring our 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 does not follow a pattern which 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. This is okay because a lot of the time you are using Liferay's tag libraries instead of standard markup. And in most cases, this division between designers and programmers simply doesn't exist. My goal here is not necessarily to evangelize one way over the other (that is up to individual developers who should do things the way they are most comfortable doing them), but instead to let you know before we get into the code that I am well aware that we're not doing things the way many others do them. My goal is to show you the Liferay way of doing things, as 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.

With that said, because there will be some Java code in our JSPs, we will 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 then created imports `init.jsp` so it can take advantage of all those declarations (see figure 4.3).

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

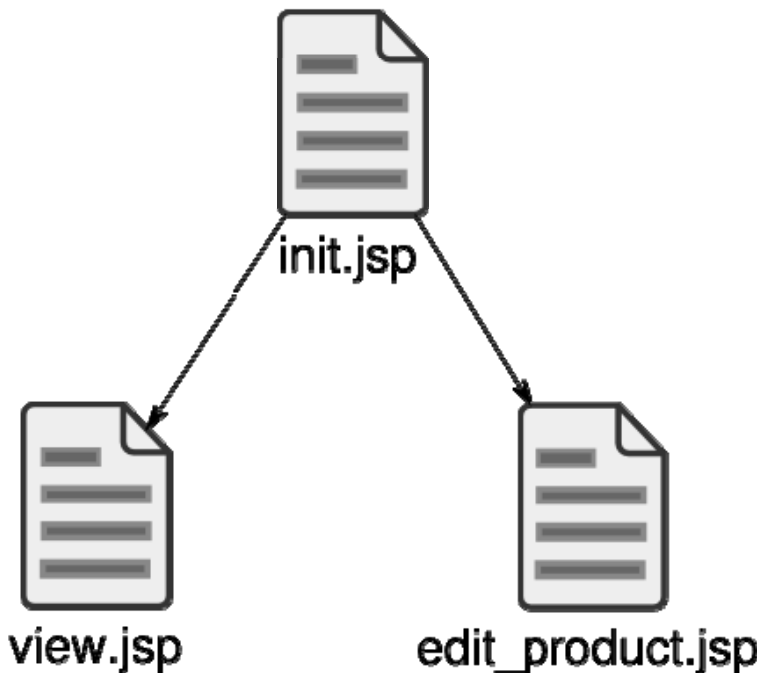


Figure 4.3 Every JSP file in the project will include `init.jsp` so that all initialization and configuration can be done in one easy to maintain file.

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

#### Listing 4.3 Putting initialization code in one place

```

<%@ taglib uri="http://java.sun.com/portlet_2_0" prefix="portlet" %>      A
<%@ taglib uri="http://java.sun.com/jstl/core_rt" prefix="c" %>
<%@ taglib uri="http://liferay.com/tld/au" prefix="au" %>
<%@ 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" %>                                     B
<%@ 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.dao.search.ResultRow" %>

```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

```

<%@ 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 />
```

C

```
<liferay-theme:defineObjects />
```

D

**A Tag library declarations**

**B Import statements**

**C Initializing portlet taglibs**

**D Initializing Liferay taglibs**

As you can see, this JSP is fairly simple: it declares all the tag libraries we will be using in this portlet application, imports all the classes we'll need in our scriptlets, and then initializes any tag libraries that need initializing. What makes this nice is that since we will be using all of this stuff in the rest of our JSPs, we don't have to bother re-importing classes or re-initializing tag libraries in every single JSP. We can simply add this at the top of any JSP we create:

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

The last tag in this file is a Liferay-specific tag. This tag, like the Portlet API's similarly used tag, makes several variables available to the page:

Object	Description
themeDisplay	A runtime object which contains many useful items, such as the logged in user, the layout, logo information, paths, and much more.
company	The current company object. This represents the portal instance on which the user is currently navigating.
account	The user's account object. This object maps to the Account table in the Liferay database.
user	The User object representing the current user.
realUser	When an administrator is impersonating a user, this variable tracks the administrator's user object.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

Object	Description
contact	The user's Contact object. This object maps to the Contacts table in the Liferay database.
layout	The set of pages to which the user has currently navigated. Generally, communities and organizations have two: a public set and a private set.
layouts	The list of layouts that exist in the community or organization (i.e., a <i>group</i> in the code) to which the user has currently navigated.
plid	A Portal Layout ID. This is a unique identifier for any page that exists in the portal, across all portal instances.
layoutTypePortlet	This object can be used to programmatically add or remove portlets from a page.
scopeGroupId	A unique scope identifier for custom scopes, such as the page scope which was introduced in Liferay Portal 5.2.
permissionChecker	An object which can determine given a particular resource whether or not the current user has a particular permission for that resource.
locale	The current user's locale, as defined by Java.
timeZone	The current user's time zone, as defined by Java.
theme	An object representing the current theme which is being rendered by the portal.
colorScheme	An object representing the current color scheme in the theme which is being rendered by the portal.
portletDisplay	An object which 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.

Because we have added this initialization tag to our *init.jsp*, these objects will be available on all of our pages.

Now we have one place where all initialization stuff can be maintained, and all of our JSPs can benefit from it. Since we have this in place now, we're ready to start adding functionality, and the first thing we'll add is a form which lets users add products.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>



### 4.3 Creating a form with AlloyUI taglibs

As we have seen, MVCPortlet uses an initialization parameter to forward processing to a JSP for the user to view. So our first step will have to be making this JSP display what we want to display to the user. If you refer back to figure 6.2, you can see that we want to show the user a form which allows them to very quickly add products. Under that form, we want to display a table of products which have already been added, so that they may be viewed, edited, or deleted.

In order to create our form, we'll use another tool that Liferay gives us: AlloyUI tag libraries. The form we're creating is very small, so you won't necessarily see the benefits of using AlloyUI taglibs just yet, but suffice it to say you'll be glad you're getting an introduction to it now, because it's going to make your life a lot easier later in the chapter.

But first of all, what is AlloyUI?

#### 4.3.1 Getting started with AlloyUI tag libraries

Put simply, AlloyUI is an *interface metaframework*. Okay, that probably sounded like gobbledegook, but bear with me for a second. Website front ends are created using a combination of three technologies: HTML, CSS, and JavaScript (see figure 4.3). These three technologies together comprise 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 interactive elements of any web page. If something moves, changes, can be dragged, dropped, resized, or removed, JavaScript is enabling that for the end user.



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

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

AlloyUI was designed because user interface developers tend to have to solve the same kinds of problems over and over. Rather than continuing in 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 that you know somebody out there 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. You use AlloyUI's taglibs for things like forms (which is what we're about to do), and AlloyUI will generate the proper HTML to lay your form out properly. And if you need them, it will even provide interactive widgets for your users to use to enable them to get those forms filled out properly.

Let's start very gently with AlloyUI. We have a very simple form which hardly needs it, but which serves as a great introduction to it. This form is below in listing 4.4.

#### Listing 4.4 An AlloyUI form

```
<portlet:actionURL name="addProduct" var="addProductURL"/>           A
<alui:form action="<%= addProductURL.toString() %>" method="post">   B
    <alui:fieldset>
        <alui:input name="productName" size="45" />
        <alui:input name="productSerial" size="45" />
        <alui:button-row>
            <alui:button type="submit" />
        </alui:button-row>
    </alui:fieldset>
</alui:form>
A ActionURL for form submission
B AlloyUI form
```

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

Liferay's `MVCPortlet` class enhances the `GenericPortlet` class that we've already seen in one important way: instead of needing to use annotations to mark out action methods corresponding to action URLs, simply naming the method with the same name as the action URL's name will cause the portlet to execute that method if that URL is clicked. So to enable our form, all we have to do is create a method in our portlet class called `addProduct`, and that method will get executed. So let's take a look at that in listing 4.5.

#### Listing 4.5 Adding products

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

```

public void addProduct(ActionRequest request, ActionResponse response)
    throws Exception {
    ThemeDisplay themeDisplay =
    (ThemeDisplay)request.getAttribute(WebKeys.THEME_DISPLAY);
    PRProduct product = ActionUtil.productFromRequest(request);      A
    ArrayList<String> errors = new ArrayList();

    if (ProdRegValidator.validateProduct(product, errors)) {          B
        PRProductLocalServiceUtil.addProduct(product, themeDisplay.getUserId());
    C
        SessionMessages.add(request, "product-saved-successfully");

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

```

**A Convenience method**

**B Validating input**

**C Calling service layer**

I don't know about you, but as you can see, 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 of this in the code above. First, we call a method in something called `ActionUtil` which can retrieve a `Product` object out of the form the user submits. Since it is likely we'll use this over and over, it's been placed in a class that can be shared by both portlets. Next, we call a method in something we're calling `ProdRegValidator`. This class will contain code which validates the input from the user-submitted form. Again, since we will likely reuse these validation routines, they're in a class which can be used by both portlets. We'll go over the validator in a section below, because that class uses Liferay's `Validator`, which you'll definitely want to see.

The next thing we want to highlight is our use of the service layer we created in the previous chapter. If the values on the form pass validation, we need to go ahead and save them to the database. We can do this very easily, as all of our services are available to us via static methods in a – `LocalServiceUtil` class. In our case, we are using `PRProductLocalServiceUtil`, since we're saving `PRProduct` entities.

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

### 4.3.2 *Providing feedback and messages*

You probably noticed that some vital information was missing from the form in listing 6.4. Generally when you have a field on a form, you also have a label for that field, like 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 pretty common

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

sense, but our code above doesn't reflect the fact that we have a label. Or does it?

Our code, however, looks like this:

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

Where is the text, then, that prints out the words "Product Name" for the field label? Thank you; I'm very glad you asked that question. Remember back when we were configuring our `portlet.xml` file, we 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 us not only to put all of our form messages in one place, but also to support multiple languages for our portlet application.

Create a folder called *content* in your *src* folder, and create a file called `Language.properties` there. The file for this project will have the contents in listing 4.6.

#### 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          A
productDeleted=The product has been deleted successfully.
product-name=Product Name
product-serial=Serial Number Mask
productUpdated=The product was updated successfully.
there-are-no-products=There are no products yet to display.

error-deleting=There has been an error deleting this product.  B
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                C
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
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

```

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?

```

```

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

```

**A Product Form Messages**

**B Product Form Errors**

**C Registration Form Fields**

**D Registration Form Errors**

**E View messages**

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

```
<liferay-ui:success key="productSaved" message="product-saved-successfully" />
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

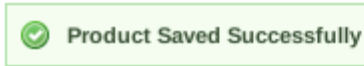
<http://www.manning-sandbox.com/forum.jspa?forumID=642>

```

<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 will activate and the actual message from the `Language.properties` file will be displayed to the user. If the messages aren't in `SessionMessages` or `SessionErrors`, nothing will appear on the page. If you've used Liferay



for any  
amount of  
time,

you've seen these messages. They look like figure 4.6.

Figure 4.6 When you save a product, the above message appears to show the user that everything's okay and the action he or she just 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 language of your user.

### 4.3.3 Translating messages to multiple languages

You can provide alternate translations for all of your message keys. Normally, you would 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 your 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 is generally a lot of work, as you would have to get someone to translate all of 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 have been using to deploy your project. You'll notice that it is pretty small, as all of its functionality is derived from other Ant scripts that are 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 have just created, using the following command on all OSes:

```
ant build-lang
```

You will see messages like the following ones, among others:

```

Translating en_it Product Name
Translating en_it Serial Number Mask

```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

```
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, take a look at your *content* folder. You will see that it now contains files for many languages.

What has just happened? The Plugins SDK contains Ant targets which have the ability to use the Babelfish (<http://babelfish.yahoo.com>) service to translate all of the keys in your file to multiple languages. All we did by providing the Ant task we created was to give those Ant targets the parameters they needed to do their work: namely, the folder where our `Language.properties` file exists and the name of the file. Once we did that, we were able to use Babelfish to generate all of the language files you now see in that folder.

You might run into a problem with Babelfish 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 just where it left off, and eventually, you'll get everything translated.

One thing that's very important to note is that since Babelfish is an automated translation service, sometimes the translations it provides are not optimal. But they certainly serve as a good starting point, and are better than nothing. If you are targeting your application for a specific audience in a specific language, you may want to send your generated translation file to someone who can go over the translation and make sure it is correct before releasing 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 do want to provide a translation yourself, you can do this by providing to your translator the file with the `.native` extension appended to it. You'll notice that Liferay generates these by default as well. This file overrides the automatic translations provided by Babelfish. Have your translator put his or her translations in this file and then copy it back to the `content` folder, and those translations will 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 that you do not have to redefine common actions (such as *Save* or *Cancel*) that are already used by the portal: you can use these in your portlets as-is. Your portlets will therefore inherit all the translations of these common labels that have been 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 that we haven't gone over yet, and that is field validation. Yes, Liferay has tools for that too.

#### 4.3.4 Validating user-submitted forms

Liferay includes a utility which can perform field validation. Rather than writing something yourself or using a framework, you can make use of Liferay's utility to easily validate the data your users enter. Often this is much easier to use than what comes with a particular framework—especially if you are just

learning the framework for the first time. It can be used for any portlet, whether you are using a framework or not, and is extremely simple to implement.

Liferay's validator is implemented in the `com.liferay.kernel.util.Validator` class. We will create a separate `Validator` class for our portlet which will contain validation logic for all of the objects we want to persist to the database. You've already seen this class in use in our `addProduct` method. Listing 4.7 contains the method we called 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; A  
  
    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;  
}
```

##### A Validation based on a boolean

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

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



**Add A Product**

Product Name

Serial Number Mask

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

The presence of the `SessionErrors` object triggers the first message, and the second message is the message we wanted to display to the user. You can have an individual message for every field, and we’ll see that when we get to the more complicated form for registering a product.

Our page is not yet complete, however. We still need a way of displaying the data we’re entering so that we can view it and edit it. For that, we’ll use another helpful utility from Liferay called Search Container.

## 4.4 Displaying data with 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 will be working with lists of objects. In our case, we are working with lists of `PRProducts`. We could conceivably create a way manually of iterating through `PRProducts` in a table and then create another one for `PRRegistrations`, etc. Liferay, however, has solved this problem for us through the creation of Search Container. This object wraps our list of objects (the type of object does not matter) and automatically provides for us 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 get our data displayed and editable using this component.

### 4.4.1 Using Search Container to present our data

Let’s jump right in so we can see how this works. The code in listing 4.8 should be placed directly under the form in `view.jsp`.

#### Listing 4.8 Search Container makes showing data easy

```
<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(),
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

```

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">
B

  <liferay-ui:search-container-column-text
    name="productName"
    property="productName"
  />

  <liferay-ui:search-container-column-text
    name="productSerial"
    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>

```

#### **A Display works from these values**

#### **B Columns we want to display**

The first thing we see here is the initialization of the search container. We set an empty results message from our `Language.properties` and we set the delta for pagination to a pretty low number, because we're not envisioning many products in this search container. We get the results from our `ActionUtil` class, which calls the service layer we generated in Chapter 5 to retrieve the products we need from the database. Once we've calculated the total and the delta, we set these attributes in the `pageContext` so that the search container can iterate over them.

Once we get to the rows to be displayed, all we need to tell search container is the name of the class that it is displaying, the property of the primary key, and the name of the variable to represent our model. From there, we can list the columns simply by their names (from `Language.properties`) and the property from the model bean.

Note the last column. It contains another JSP which will define our actions for each row. This JSP defines the actions button that will appear on every row of the search container table, allowing users to perform certain actions on `PRProducts`. Figure 6.8 shows this button.

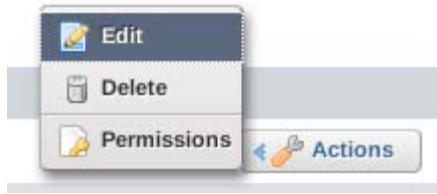


Figure 4.8 At the end of every row, an action button appears. When users click this button, they will be able to perform three actions on each record. They will be able to edit the record, delete it, or set permissions on it.

The action button is implemented in another JSP, which we will call `admin_actions.jsp` (see listing 4.9).

#### Listing 4.9 Adding actions to Search Container

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

<%
ResultRow row = (ResultRow)request.getAttribute(WebKeys.SEARCH_CONTAINER_RESULT_ROW);
PRProduct myProduct = (PRProduct)row.getObject();
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() %>" />
    </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 %>" />
    </c:if>
</liferay-ui:icon-menu>
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

```

    </c:if>
</liferay-ui:icon-menu>
A Get object out of row
B Delete is different

```

As Search Container is looping through records, certain attributes of that container are available in the request, such as the current row in the loop. So for each row, we can retrieve our `PRProduct` object and pull information about it that we will need to implement our Action button functionality.

For example, our database code operates from primary key values. The View layer will therefore need to provide a primary key to the action method so that the `PRProduct` to be operated on can be found in the database. We use the list of `PRProduct` objects that are wrapped in the Search Container to retrieve these keys, and then we define the key that is found as a parameter on each action URL that we generate. So what we are doing in this file is defining the action URLs that we will use to call functionality for specific `PRProduct` objects. These URLs are then used in Liferay tags that assemble the Action button with its links that fly out when the button is clicked.

Note that we have not used any JavaScript or CSS to generate this; all of that is automatically generated by the tag library. This makes it very convenient to use, and your code looks very 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 which pops up and asks that question, allowing users to cancel if they clicked it by mistake.

Pretty cool, eh?

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

#### 4.4.2 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 from what you might be used to. When a user clicks the Edit button in the Search Container, the `editProduct` portlet action runs. Since the action URL we used contained the primary key of the record we want to edit in a parameter, we 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, we validate the key first to make sure it has a value before we pass it on in our processing logic. Then we just grab the `PRProduct` from the database and put it in the request so we can display it on a form for editing. This form will be almost the same form we used for adding; in fact,

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

it could be the same form if we wanted it to be. But there's one more thing I'd like to point out before we get to the form, and that's the last line of the code above. Because we have defined the location of our edit form in the portlet's instance variable `editProductJSP`, we can point our page management render parameter, `jspPage`, to the page we want the user directed to when this action completes.

Let's take a look at that form in listing 4.10.

#### Listing 4.10 Editing a product

```
<%@include file="/init.jsp" %>
<jsp:useBean id="product"
type="com.inkwell.internet.productregistration.model.PRProduct" scope="request" />
<portlet:renderURL var="cancelURL"><portlet:param name="jspPage"
value="/admin/view.jsp" /></portlet:renderURL>                                A

<portlet:actionURL name="updateProduct" var="updateProductURL" />            B

<h2>Edit A Product</h2>

<auiform name="fm" action="<%=updateProductURL.toString() %>" method="post">

    <auifieldset>

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

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

        <auiformbutton-row>

            <auiformbutton type="submit"/>
            <auiformbutton
                type="cancel"
                value="Cancel"
                onClick="<%=cancelURL %>"
            />

        </auiformbutton-row>

    </auiformfieldset>

</auiformform>
A Cancel button is render URL
B Submit button is action URL
```

As you can see here, we use the `jspPage` functionality again in the form to let our Cancel button know where to send the user if he or she decides that editing this product is not what he or she wants to do. The action URL points to another action which actually goes and updates the product. Because the update method is so similar to the add method, there's really no point in outlining it here (though, of course, you can see it by downloading the source code to this chapter). The same goes for the delete method. Let's instead get to something a bit more interesting. You've already seen it sprinkled throughout the code above, but we haven't touched on it yet: permissions.

## 4.5 Protecting data with Liferay permissions

Liferay Portal has a robust permissions system which 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 which ship with Liferay. This means that you can implement security all the way down to the object level of your code.

This means that we can now implement security in our portlet to enable administrators to set permissions so that only the users they want will be able to add products. This is surprisingly easy to implement, particularly if you are already familiar with conditionally displaying HTML fragments to users. We will implement that part of it using JSTL for our example.

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

### 4.5.1 Pointing to your 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 which configure the portlet or override settings from `portal.properties`. Incidentally, this is how Liferay's social networking portlet modifies some settings inside of Liferay upon deployment. Be careful with this: Liferay does not guarantee in what order individual portlets will be registered, so if you need to override a setting, make sure another portlet does not override the same setting.

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

### 4.5.2 Configuring Liferay permissions

Go ahead and create this folder in your source folder, and create the `default.xml` file in this folder. You should have the contents of listing 4.11 in the file.

#### Listing 4.11 Defining Liferay Permissions

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

<resource-action-mapping>

  <portlet-resource>
    <portlet-name>product-registration</portlet-name>
    <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>

  </portlet-resource>
</resource-action-mapping>
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

```

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

</portlet-resource>

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

<model-resource>
    <model-name>com.inkwell.internet.productregistration.model.PRUser</model-name>
    <portlet-ref>
        <portlet-name>product-registration</portlet-name>
    </portlet-ref>
    <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>
</model-resource>

<model-resource>
    <model-name>com.inkwell.internet.productregistration.model.PRRegistration</model-
name>
    <portlet-ref>
        <portlet-name>product-registration</portlet-name>
    </portlet-ref>
    <supports>
        <action-key>DELETE</action-key>
        <action-key>PERMISSIONS</action-key>
        <action-key>UPDATE</action-key>

```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

```

        <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>
    </model-resource>
</resource-action-mapping>

```

**#1 Portlet resource actions**

**#2 Model resource actions**

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 (#1) 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. We don't have it here because our portlet is in the Control Panel and does not need a configuration screen.

The other default action we have added is *view*. This action allows the viewing of the portlet. Without this action, no one would be able to view the portlet, which would not make any sense.

The third action is not a default action, but one we have defined as functionality for our portlet: *add product*. By defining this action, we are saying that we will want to choose which users can add products and which users cannot.

These actions are portlet actions, because they belong to the portlet itself. The portlet is the authority for whether PRProducts (and PRRegistrations) 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 it by creating a role and then defining permissions for it. Figure 4.9 shows this for a role called Product Administrators.



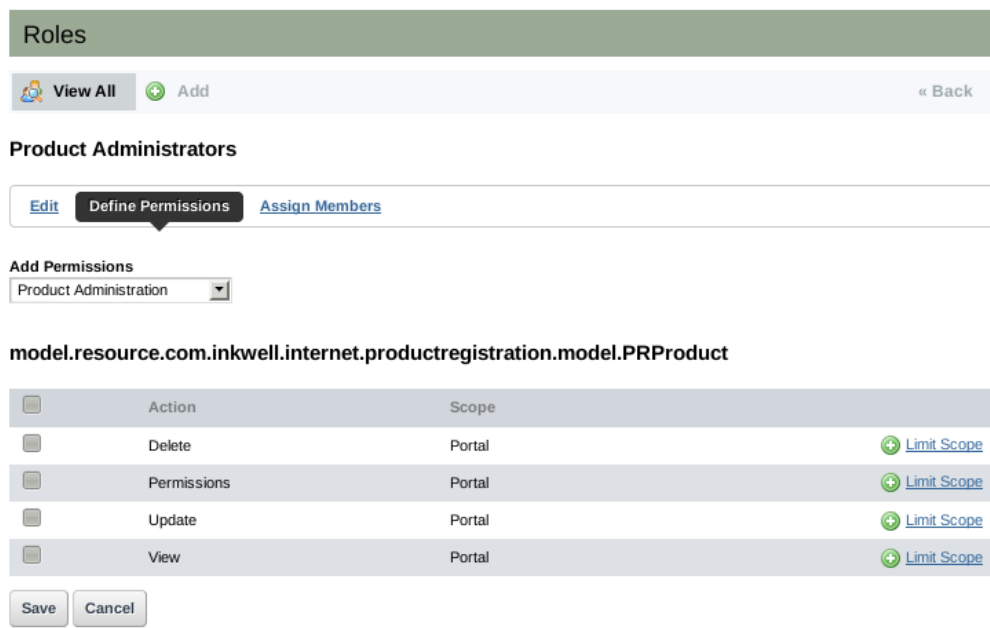


Figure 4.9 Any model resource you define can have permissions configured for it in Liferay. Here, we have created a role called Product Administrators, and can grant that role permission to delete, to define permissions, to update, or to view PRProducts.

## DEFINING THE MODEL RESOURCE

These resources are the objects you have created which get stored in the database. We have just been providing the functionality for creating, editing, and deleting PRProducts. If we want to wrap Liferay's permissions system around the objects that have been created, we 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 that we may want to grant permission to do, and those functions have already been created. Permissions is a new action which we are adding. This is the ability of a user to grant or take away permissions to do other actions. The really nice thing about this is that *no coding has to be done to enable this*. You simply need to provide a link to Liferay's permissions system—which will be covered next—and this functionality will be added to your portlet.

In both the portlet resources and model resources sections of the file, you can define default permissions for the community/organization in which the portlet is located or guests. For portlet resources, we have given everyone the *view* permission by default and have made sure that for guests, the *Add Product* permission is unsupported. For model resources, we have given everyone permission to view PRProduct records (otherwise, users who are not registered would not 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.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

All of these permissions can be overridden by administrators through the permissions system. These are just 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 we're not through yet: there are some more advanced things you probably need to do. So let's take a look at the registration form we need for our Product Registration portlet.

## 4.6 Generating different field types with AlloyUI taglibs

If you recall, our registration form has more than just text fields on it. Users are asked to pick from lists of things, and have a couple of date fields to fill out. Anyone who has written data driven applications knows that giving users a text field for these kinds of things leads 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 that we need to provide them ways of entering this data which both preserves the integrity of the data and is easy to do.

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

### 4.6.1 Generating date pickers

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

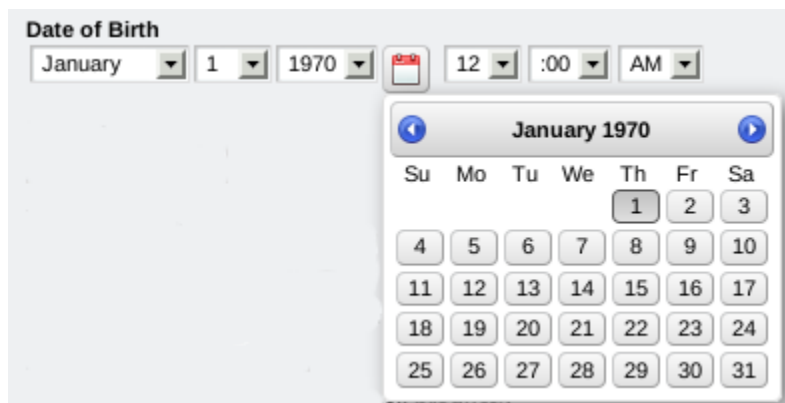


Figure 4.10 AlloyUI tag libraries make it easy for you to generate date pickers. No JavaScript required.

This great date picker can be generated automatically for you by crafting your field like listing 4.12.

#### Listing 4.12 Date pickers without JavaScript

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

<%
  Calendar dob = CalendarFactoryUtil.getCalendar();
  dob.setTime(regUser.getBirthDate());
%>                                B

<au:input
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

```

name="birthDate"
model="<%= PRUser.class %>"
bean="<%= regUser %>"
value="<%= dob %>" />

```

C

**A Error message from Language.properties**

**B Initialize Calendar object**

**C Tag contains model, bean, value**

Now, obviously, this can't be created without JavaScript, so it may be a bit of an exaggeration to say it was created without it. But *you* don't have to write any of it. Simply by using the tag, the same JavaScript that Liferay wrote to provide date pickers for their own portlets is used to provide your date picker.

There's one thing missing, though. Notice that we're working with a date field, and the date in figure 6.10 is from 1970. This is so that it's quite clear users should be picking 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 afterwards. It's just not as "real" to them. According to Unix, Time started on Thursday, January 1, 1970. Though it can represent time before that, it's not as "real," because it has to be represented as a negative number. Why? Because Unix (and most operating systems) represent time as "the number of seconds *since* midnight on January 1, 1970." This system of time works well, but has some problems, such as the Y2K38 bug, which happens on Tuesday, January 19, 2038 when the integer that's used to represent Time rolls over to zero.

So why'd we pick 1970? Convenience. It's somewhere in the middle of where people who are using this application are born. Plus we get geek cred for using January 1, 1970.

To make the tag display just a date, we 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." This is done using a file which was generated when Service Builder created Java classes to manipulate our data. Open the file called `portlet-model-hints.xml`, which you will 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>

```

What this does is tell the tag that for this field that we want a date range of 70 years and we want to disable the ability of a user to select dates which are in the future. And because it's a birth date, we 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, we have a Date Purchased field. For this field, all we need to do is turn off showing the time.

There's another field type that our form uses which we haven't covered, and that's a select box.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

### 4.6.2 Selecting data with AlloyUI taglibs

When the user gets to select which product he or she is registering for, we populate that selection field with values from the database.

This is a simple matter of using the AlloyUI tag for a select field along with some Java code which retrieves the data:

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

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

  <%
    for (int i = 0; i < products.size(); i++) {
  %>
  <alui:option
    value="<%=products.get(i).getProductId() %>">
    <%=products.get(i).getProductName() %>
  </alui:option>
  <%
    }
  %>

</alui:select>
```

We get the primary key as the value of the select, and we get the name of the product for display to the user. This populates our selection box, and AlloyUI takes care of all 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 won'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, that can be done in one place by styling them in your theme (see Chapter 7).

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

**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 6.11 No HTML, CSS, or JavaScript was used to generate 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.

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

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

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

## 4.7 Using Liferay's MVC makes your portlets simpler

The really great thing about Liferay's MVC framework is how it makes your portlet classes simpler. You don't have to write any page management or page flow code; MVCPortlet handles that for you. To demonstrate this, I'd like you to see the entirety of the Product Registration portlet class, which is in listing 4.13.

### Listing 4.13 Portlet classes are simpler with MVCPortlet

```
public class ProductRegistrationPortlet extends MVCPortlet {

    public void addRegistration (ActionRequest request, ActionResponse response) {
A
        ThemeDisplay themeDisplay = (ThemeDisplay) request.getAttribute(
            WebKeys.THEME_DISPLAY);

        PRRegistration registration = (PRRegistration)
request.getAttribute("registration");
        PRUser prUser = (PRUser) request.getAttribute("regUser");

        if (registration == null || prUser == null) {

            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.getClass().getName(), user.getUserId());
                } catch (SystemException ex) {

                }
                if (addresses.size() > 0) {
                    homeAddr = addresses.get(0);
                }
                // populate what we can of our registration
                prUser.setFirstName(user.getFirstName());
                prUser.setLastName(user.getLastName());
                prUser.setEmail(user.getEmailAddress());
                try {
                    prUser.setBirthDate(user.getBirthdate());
                    boolean male = user.getMale();
                    if (male) {
                        prUser.setGender("male");
                    } else {
                        prUser.setGender("female");
                    }
                }
                prUser.setMale(male);
            }
            catch (PortalException e) {
                prUser.setBirthDate(new Date());
            }
        }
    }
}
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

```

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

        registration = new PRRegistrationImpl();
        registration.setDatePurchased(new Date());
        prUser = new PRUserImpl();
        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 {
    B
    PRUser regUser = ActionUtil.prUserFromRequest(request);
    PRRegistration registration = ActionUtil.prRegistrationFromRequest(request);
    ArrayList<String> errors = new ArrayList();
    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;

        // check to see if user is a guest
        if (liferayUser.isDefaultUser()) {
            userId = 0;
            user = PRUserLocalServiceUtil.addRegUser(regUser, userId);
        } else {

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

```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

```

        if (user == null) {
            user = PRUserLocalServiceUtil.addRegUser(regUser, userId);
        }
    }

    registration.setPrUserId(user.getPrUserId());
    PRRegistration reg =
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);
}

}

protected String viewAddRegistrationJSP =
"/registration/view_add_registration.jsp";
protected String viewThankYouJSP = "/registration/view_thank_you.jsp";
private static Log _log = LogFactory.getLog(ProductRegistrationPortlet.class);
}

```

**A Users click Register button**

**B Users click Submit button**

Note that I've removed all ancillary stuff like import statements, JavaDoc, comments, and package declarations. The important point is this: our entire portlet class has been reduced to only two methods, and both of them are the result of actions that the user performs. The first is when users click the *Register* button to display the form, and the second is when users click the *Submit* button to submit the registration. The Product Administration portlet is the same: it consists only of action methods that users trigger by clicking on 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.

## 4.8 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 leverage 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 developers in performing field validation for forms. AlloyUI tag libraries assist developers in creating good looking, dynamic forms complete with CSS styling and auto-generated JavaScript widgets. Liferay also provides developers with a way to automatically translate language bundles so they can support multiple languages very easily. And finally, Liferay permissions are easy to

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>



integrate into any application simply 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, which let you completely control the way Liferay looks.

# 5

## *Using themes and layout templates*

Liferay themes allow developers and designers to completely customize the look and feel of Liferay Portal. They have been designed to integrate nicely with all of the web technologies you already know: HTML, CSS, and JavaScript. As a theme developer, you won't need to know the ins and outs of Java. Instead, Liferay makes a great number of variables available to you using the Velocity or Freemarker templating languages. Both of these are easy-to-use languages which integrate with HTML in a similar manner to PHP. In fact, PHP developers should find it very comfortable to work with either one.

Once you have your theme designed, you can implement any number of layout templates. These enable you to provide custom page layouts into which your portlets can be placed. Though Liferay ships with many layout templates already included, many times when working with a particularly complex page design, only a custom layout will do.

In this chapter, you'll learn the components of Liferay themes so that you can use your artistic and design skills to make your site look—well, any way you want it to look. You'll also learn how to create layout templates to go with your themes that let you place portlets anywhere you want. Let me start with a couple of examples to show you what I mean.

### **5.1 Understanding themes and their structure**

Believe me when I say you can make Liferay look like *anything*. Don't believe me (and after all this time we've spent together)? Okay, then. I'll show you two web sites which both run on Liferay but which could not be more different; nor could they be targeted to more different audiences.

The first is the web site for Sesame Street. Behold figure 5.1.



Figure 5.1 You can find this site at <http://www.sesamestreet.org>. Targeted to children, this site won the first Emmy award, called New Approaches, for digital media. Can you tell it's running Liferay? No, I didn't think so.

Obviously, the Sesame Street web site is geared for children. Using large fonts, minimal text, lots of colors, and running video, this site is very easy to navigate for small children. And if you don't believe that, I have a five year-old daughter who would be glad to speak to you.

Okay, on to the next example, shown in figure 5.2, which could not be more different.



Figure 5.2 You can find this site at <http://www.monsterenergy.com>. Quite obviously *not* targeted for children, this site also makes use of video, has articles about nightlife, as well as “Monster Girls,” which must be pictures of female monsters. Or something like that.

Black. Slash. Skull and crossbones. Motorcycles. This is not the stuff of children, yet Liferay deals with it as effortlessly as it deals with children's content. And again, there's no way you can tell that this site is running on Liferay Portal.

The reason I wanted to show you these sites is simple: if *they* can do it, *you* can do it too. The goal of this chapter is to give you what you need to write your own themes, making your web site look however you want it to look, to target your particular audience.

### 5.1.1 Generating a theme project

Themes are plugins, and are therefore hot-deployable just like portlet plugins. You will be able to use the Plugins SDK (see Chapter 2) to build your themes automatically so that they can be deployed to any Liferay instance. The Plugins SDK packages a theme into a .war file just like a portlet, and this .war file can then be deployed to Liferay in the same manner in which you would deploy a portlet.

Liferay makes writing themes as straightforward as possible. If you have experience in coding HTML, CSS, and JavaScript, you should be right at home with Liferay themes. One warning, however, about this chapter needs to be stated: like the previous chapters where I assumed knowledge of Java web applications, here I'm assuming knowledge of HTML, Cascading Style Sheets, and JavaScript. If you don't have familiarity with these technologies, there are many great books on the subject, including David Flanigan's *JavaScript: The Definitive Guide* and John Resig's *Secrets of the JavaScript Ninja*.

But before we get into all that complicated stuff, let's look at the structure of a theme.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

### 5.1.2 Deconstructing a theme

Let's jump right in and take a look at the directory structure of a theme using Inkwell as an example. You would create a new theme in exactly the same way as you create a portlet. The Inkwell developers want to create a theme called Inkwell Internet. This is how they'd do it in LUM:

```
./create.sh inkwell-internet "Inkwell Internet"
```

Or for Windows:

```
create.bat inkwell-internet "Inkwell Internet"
```

This will create your new theme project, which has a surprisingly small and simple directory structure.

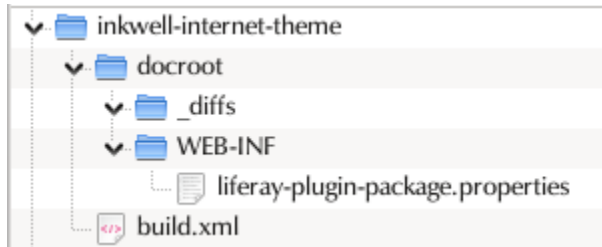


Figure 5.3 The structure of a newly generated theme. You can see that it doesn't contain any code, yet you can deploy it and get a copy of the default theme. What gives? See below.

As you can see, initially, there isn't much in a theme project. In order to build our theme, we want to take a look at how theming works in Liferay.

If you were to look inside Liferay's .war file or into the *webapps* folder of a Tomcat installation that contains Liferay, you will find that Liferay contains a few themes bundled in by default. Two of these are important and form the basis of all other themes: *\_styled* and *\_unstyled*.

- *\_unstyled*—This theme contains all of the default Velocity templates and image files necessary for a full theme, but it is *unstyled* in the sense that there is no CSS styling applied to the layout. The CSS files are there, but only the descriptors are in the file—there is no style information added for any of them.
- *\_styled*—This theme contains nothing but CSS styling (and a screen shot image of the theme) which applies a basic look and feel. Liferay's default theme, called *Classic*, is just a combination of the *\_unstyled* theme and the *\_styled* theme. This is done for every theme you write.

When you create a new theme, you concentrate on the *differences* between the default theme and your theme. That is why the *\_diffs* folder is there, and that is where you will put your theme code. You can use the *\_unstyled* theme as a template for your own theme, mirroring the directory structure that you find there under the *\_diffs* folder.

So to get started, create the following folders under *\_diffs*:

- *css*
- *images*
- *javascript*

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

- templates

Then examine the code in the `_unstyled` theme and decide if you need to change the markup at all. For example, the default page contains a div for a top banner which pulls the logo that was uploaded to Liferay. Perhaps you want your theme to randomly choose a different banner for each page visit, and none of the banners will use the uploaded logo. In this case, you would need to modify `portal_normal.vm` and change the markup to support the functionality you want. You would also likely need to add some JavaScript functions in the `javascript` folder to implement the actual banner switch. Note that your markup goes in the `templates` folder under `_diffs`.

Now that you understand where everything goes, it's important to understand how you can make use of the three components of a theme, which we'll look at next.

## 5.2 Understanding theme markup, CSS, and JavaScript

In this section, we'll take a closer look at the markup (HTML/Velocity/Freemarker), the styling (CSS), and the scripting (JavaScript).

### 5.2.1 How markup works in a theme

The purpose of each of the markup files is described in the table below. If you need to customize any of the standard components on a Liferay Portal page, you would create your own implementation of that particular file in the `_diffs` folder.

File	Purpose
<code>init-custom.vm</code>	Allows you to add your own custom Velocity variables
<code>init.vm</code>	In conjunction with <code>VelocityVariables.java</code> in Liferay, sets many Velocity variables that correspond to Liferay Java objects.
<code>navigation.vm</code>	Implements the page navigation within the theme
<code>portal_normal.vm</code>	The overall template for all pages the theme implements. This file includes the other files.
<code>portal_pop_up.vm</code>	The overall template for any portlets which implement pop-up windows.
<code>portlet.vm</code>	The template for portlet windows within the theme.

The global page markup is handled by `portal_normal.vm`. This page defines several layers of the page where various page elements reside. Figure 5.4 shows the structure of the default Liferay page.

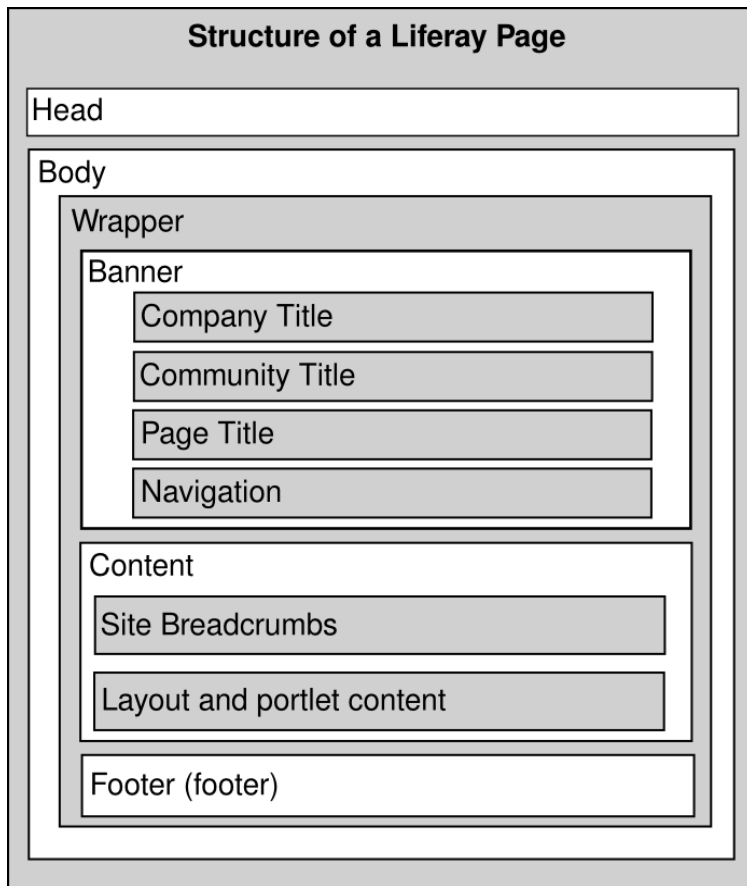


Figure 5.4 The default structure of the Liferay page. This shows all of the elements which can be styled via CSS. You may decide to include all of them, some of them, or only the portlet content elements in your theme.

All of these elements are styled via Cascading Style Sheets, using styles that have been predefined. Of course, you are free to add your own as well. Next, we'll see how to use the styles provided and where to put your custom styles.

### 5.2.2 Using Cascading Style Sheets in themes

The default theme contains multiple .css files that handle every aspect of Liferay's UI, and are divided up in order to keep styling for similar components all in the same place. When you create a theme, it is likely that most of the styles that are already defined will not need to be modified. There is only a small subset of styles that are used by the theme markup files, and for most themes, you won't have to go beyond the styling of that markup.

For this reason, a best practice is to create a single file in your `_diffs/css` folder called `custom.css`. This file will contain all of the styles you want to implement which will override the default styles provided by Liferay. Because this file is processed last by the browser (due to the order of the import

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

statements in `main.css`), all styles in this file are guaranteed to take precedence over any of the styles provided by default within Liferay. So any of the styles in multiple files can be overridden within `custom.css`, and this file contains only the styles you need to modify from the default implementation.

Below is a table describing the organization of the styles within the various `.css` files provided by Liferay.

File	Purpose
<code>application.css</code>	Contains styles having to do with components of applications. This includes the tabs, expandable trees, dialog overlays, and the results grid (i.e., Search Container). Much of the markup for these styles is created by Liferay's tag libraries.
<code>base.css</code>	Contains styling for standard HTML tags, such as paragraphs, headings, tables and more. This file also contains styling for some Liferay-specific page elements, such as errors, warnings, tool tips, the loading animation, and more.
<code>dockbar.css</code>	Contains styling for the Dockbar, which floats at the top of the page when a user is logged in.
<code>custom.css</code>	This is a blank file which is loaded last. Theme developers put their custom styles here which override styles provided by Liferay.
<code>forms.css</code>	Contains styling for all form elements.
<code>layout.css</code>	Contains styling used by layout templates.
<code>main.css</code>	Contains no styling, but imports the rest of the files.
<code>navigation.css</code>	Contains styling for the main navigation elements.
<code>portlet.css</code>	Contains styling for the portlet windows.

You can override any style in any of these files inside `custom.css`. The table above should help you to find a particular style that you may need to override. As you can see, Liferay's styles are well-organized and should be easy to find.

Once you've got the styling down, you may want to provide some dynamic functionality using JavaScript, so next we'll take a look at where and how you can plug your JavaScript functions into your theme.

### 5.2.3 Using AlloyUI and JavaScript in themes

JavaScript goes inside a `javascript` folder in your `_diffs` folder. Liferay provides a default file which contains no implementation, but which defines three events (see listing 5.1).

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>



## Listing 5.1 Liferay JavaScript events

```
AUI().ready(  
    function() {  
        }  
    );  
Liferay.Portlet.ready(  
    function(portletId, node) {  
        }  
    );  
Liferay.on(  
    'allPortletsReady',  
    function() {  
        }  
    );  
A Runs when HTML is loaded  
B Runs when each portlet loads  
C Runs when all portlets have loaded
```

Each of these events fire at a certain point of the page loading. The first fires when the HTML is delivered to the browser, *sans* the portlets, which are loaded dynamically. The second is executed for each portlet when it loads. The third runs once all the HTML has been loaded and all the portlets have been initialized.

Besides theme-wide JavaScript, there is also support for page-specific JavaScript. When end users create a page using Liferay's GUI, the Page Settings form provides three separate placeholders for JavaScript that can be inserted anywhere in your theme. Use the following in your theme markup to include the code from these settings:

```
$layout.getTypeSettingsProperties().getProperty("javascript-1")  
$layout.getTypeSettingsProperties().getProperty("javascript-2")  
$layout.getTypeSettingsProperties().getProperty("javascript-3")
```

The content of the JavaScript settings fields are stored in the database as Java Properties. This means that each field can have only one line of text. For multi-line scripts, the newlines should be escaped using `\`, just as in a normal `.properties` file. For these reasons, the default themes provided by Liferay do not implement this feature. I'm just letting you know it's there in case you want to use it.

Let's next take a closer look at the JavaScript library called Alloy UI that you get out of the box with Liferay.

### 5.3 Reaping the benefits of AlloyUI

AlloyUI is a user interface web application framework. In other words, it's a framework which you can use to build user interfaces in web applications. Underlying AlloyUI is YUI, which is a well known JavaScript library developed by Yahoo!, Inc. AlloyUI solves common problems faced by developers across the spectrum of HTML, CSS, and JavaScript, and it does this by combining the best of existing solutions under one consistent API that is easy to use.

You can use AlloyUI's components instead of hacking together your own markup, figuring out how to style it properly with CSS, and then coding up your own dynamic behavior with JavaScript. AlloyUI gives all of that to you.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

First, its components are made using HTML 5 and common markup patterns that work. That means you don't have to experiment with your markup until you come up with something usable. For non-front end people, AlloyUI provides a tag library which generates this—we've already seen this in the Product Registration portlet in chapter 6, because we used these tag libraries to generate our forms.

Second, AlloyUI provides styling using CSS3 for layouts and forms; yet its widgets gracefully degrade for browsers which don't support the latest features of CSS3. This means that no matter which browser your end users are using, they will see the best possible result which their browser is capable of rendering.

Third, AlloyUI's components are fully JavaScript enabled, complete with their own documented APIs. The JavaScript is based on YUI3, which is a well known, widely used JavaScript library. This means that the JavaScript is distributed across the spectrum of browsers and environments, and therefore is well tested and robust. This, however, does not make the page heavy; instead, AlloyUI's JavaScript has built-in lazy loading, which means that components don't load until they are needed. So if you have a portlet on a page using one of AlloyUI's components, but the user is interacting with another portlet, Liferay doesn't have to waste time assembling all the resources of the portlet the user isn't using—instead, those resources will only load once the user begins interacting with that portlet.

So what does all of this mean practically to you? Much. Two things come to mind right away.

▪

### **5.3.1 You get components**

A major benefit of using AlloyUI is its components. These are working pieces of code which already solve the problems you are likely trying to solve. You can make use of them in your themes and in your portlets to provide the best possible interface for your users. Figure 5.5 shows some of the components available with AlloyUI.

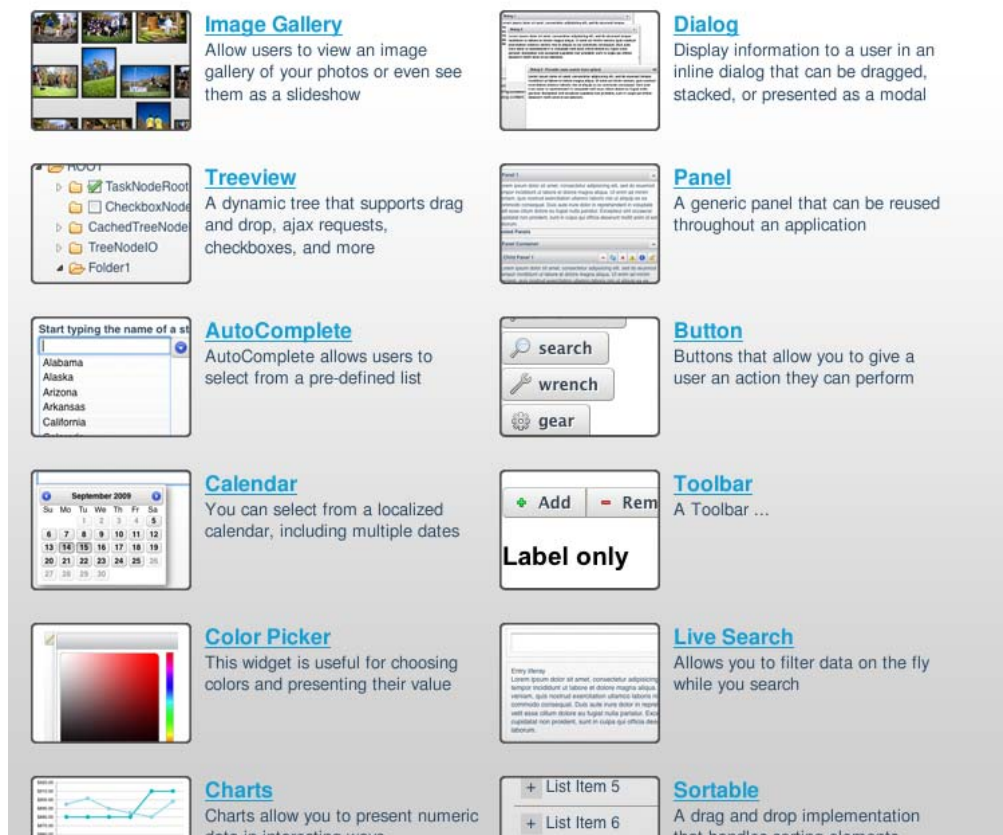


Figure 5.5 This is a small amount of the growing list of components available for AlloyUI. I had to cut the image off somewhere, but check <http://alloyui.liferay.com> for a complete, current list.

Here's a small list of the kinds of components AlloyUI gives you:

- Autocomplete
- Charts
- Calendars
- Data grids
- Tabs
- Menus
- Paginator
- Trees
- Toolbars
- Slider

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

- Advanced layout
- Panels, dialogs, and overlays
- Ajax History management
- Lazy loading of modules and components

As you can see, there are plenty of components you can use which will enable you to be more effective in your development projects. There's one more benefit, though, that I want to highlight.

### **5.3.2 *You get good design***

I can't tell you how many times I've written some application, deployed it, and then found that when used in the real world, it either didn't perform to expectations or the user did something I didn't foresee which completely borked the system.

If you've ever had to figure out why something was slow or if you've ever asked a user, "Why'd you click on that?!?" then you probably know what I mean. This can be a rude awakening for developers (like me), who sometimes assume everyone else's machine is configured the same way as theirs. For example, I once implemented a calendar which popped up different events when the user moused over dates on the calendar. I completed the feature and users began to test. Immediately it became apparent that if your screen and browser weren't sized to the proper resolution, the pop ups (especially those on the edges of the calendar) would appear outside the visible area of the browser. Needless to say, I had to add some logic which placed the pop up in a different location depending on the date in the calendar.

You won't have this problem with AlloyUI. AlloyUI is class based and uses inheritance. The benefit to this, of course, is that you are able to add features to the components very easily simply by inheriting all the functionality of the AlloyUI component and then adding your own methods to that. Since you'll likely be adding some small piece of functionality to a pre-existing, well-tested component, it's less likely that you'll be able to add something that'll completely bork the system or that will cause a performance issue.

If you use AlloyUI's forms or form tag libraries, you get consistent forms throughout your site. If you use AlloyUI's layouts, you get a great framework for organizing your content on the page, and a consistent way to manipulate the objects on your pages, if that is necessary. Regardless, you get rich components that are well designed and well tested which you can use on your website.

AlloyUI is a generic, open source JavaScript library, and as such it can be used outside of Liferay. In fact, a whole book could probably be written on AlloyUI alone, so we're only really going to be able to scratch the surface of it here. Liferay, however, also has custom JavaScript which is based on AlloyUI, so let's take a look at that in some more detail.

### **5.3.3 *Using Liferay custom JavaScript***

Liferay includes a lot of custom JavaScript (much of it dependent upon AlloyUI) which can help you to create a dynamic, modern web site. You will find that a lot of the common things you would want to do are already covered by Liferay's included JavaScript functions, keeping you from having to reinvent the

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

wheel. These are all properly name spaced, so even if you do not wish to use Liferay's implementations of certain pieces of functionality, you are free to go ahead and write your own.

Note also that if you are a Java developer, many of the tag libraries you'll use implement their functionality using AlloyUI's JavaScript. You have already seen an example of this with the `<au:input />` tag we used as a date picker in the Product Registration Portlet (see Chapter 6). This is an incredible time saver for you, as you don't have to spend a lot of time developing highly functional web widgets in JavaScript: Liferay has already done it for you.

The exhaustive list of what JavaScript functionality is available can be found in Liferay's source under *portal-web/html/js/liferay*. What follows is a list of some common things web developers may want to do.

- **Liferay.AutoFields**—Contains functions for forms which have fields that repeat. A well-known example of this sort of thing can be found in Google's Gmail: when a user wants to attach a file, initially there is only one field available. If a user wants to attach another file, there is a link stating "Attach another file" which the user can click. Once this link is clicked, another file browser is created, allowing the user to attach another file to the message. Users can duplicate the field as many times as they want to attach as many files as they want. Liferay includes similar functionality which is used in places such as the CMS, where repeatable fields can be created by users. Using this class, you can implement this feature as well.
- **Liferay.ColorPicker**—This class creates a small div-based pop up which allows users to pick a color that is already being displayed, and have that color's hexadecimal value inserted into a field.
- **Liferay.Language**—This class allows you to get the values of language keys by using JavaScript instead of by using the tag library. It defines a single method, *get()*, which requires the key. In themes, you can get the global Liferay Portal keys; in portlets, you can get those or your custom keys you have created for your portlet.
- **Liferay.Notice**—This controls the notification area that appears at the top of the screen. You may have seen this if you've let your session time out: Liferay will display a countdown and if it reaches zero, you will be automatically logged out. You can use this notice area for your own notifications. Be aware that Liferay uses this notification area as well, so you don't want to use it a lot, or you might run into instances where you've accidentally covered up a notification that Liferay is trying to display to the user.
- **Liferay.Panel**—A Panel is a basic container for content. If you place your markup inside a panel, that panel can then be manipulated by the functions in this class (i.e., it can be collapsed and certain events can be sent to it).
- **Liferay.Upload**—This is the widget used by the Document Library portlet to upload files.
- **Liferay.Util**—This class, as its name describes, contains many utilities for working with document elements, especially form elements. For example, there are methods for dealing with check boxes (e.g., *checkAll()*), for enabling and disabling elements, for escaping HTML so it can be inserted into a database, for focusing fields, and much more. You would be more likely to use these within your portlets than in a theme.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

Okay, so the question then becomes, how do you use this cool stuff? Let's take a look at some code that uses JSP to call Liferay JavaScript objects :

```
<au:script use="liferay-autofields,liferay-notice,liferay-upload">      A
    new Liferay.Upload();
    new Liferay.AutoFields();
    new Liferay.Notice();
</au:script>
```

#### **A Lazy loading**

Notice that, similar to a Java import statement, you have to declare what you're going to use first? This is because the modules are not initialized right away; they're lazy loaded. This speeds up the rendering of the page.

This same thing can also be done via JavaScript. This code uses JavaScript to call Liferay JavaScript objects :

```
AUI().use('liferay-autofields', 'liferay-notice', 'liferay-upload',
function(A){
    new Liferay.Upload();
    new Liferay.AutoFields();
    new Liferay.Notice();
});
```

As you can see, it is done in a very similar way. The AUI() object is the global AlloyUI object, and it is used to do the initialization. The one exception to this initialization process is the Liferay.Util class. You don't have to initialize that class in order to use it; it's available by default on every page.

Suffice it to say that AlloyUI gives you a lot, and hopefully this information is enough to get you started using it. As I have stated, a whole book could be written on AlloyUI, but we have only space here to sprinkle it throughout *Liferay in Action*.

Next, we'll see how you can configure a theme by using the liferay-look-and-feel.xml configuration file.

## **5.4 The liferay-look-and-feel.xml file**

You can do all sorts of advanced things with themes by creating a liferay-look-and-feel.xml file according to the DTD which can be found in the portal source of the version of Liferay upon which you are building your site. Let's take a look at some of the configuration parameters you might want to consider using with your themes.

### **5.4.1 Theme Availability**

You can limit the use of a particular theme to a company, community, or organization. This can become useful, for example, for corporate sites that want to enforce a particular look and feel.

This is done via includes and excludes. Consider the following configuration:

```
<company-limit>
  <company-includes>
    <company-id>liferay.com</company-id>
    <company-id>yoursite.com</company-id>
  </company-includes>
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

```

<company-excludes>
  <company-id>mysite.com</company-id>
</company-excludes>
<company-limit>

```

With this configuration, company IDs liferay.com and yoursite.com would be allowed to use the theme, but mysite.com would not. The same configuration can be done for communities and organizations using the <group-limit> tags defined in the DTD.

This allows you to have several different themes installed, but allow only certain themes that contain the proper “branding” to be used in particular areas of your portal. Company IDs are created in the Control Panel when you create a new portal instance. Select Server > Portal Instance to create a new one (see figure 5.6).

Portal Instances				
<button>Add</button>				
Instance ID	Web ID	Virtual Host	Mail Domain	# of Users
<a href="#">10123</a>	<a href="#">liferay.com</a>	<a href="#">localhost</a>	<a href="#">liferay.com</a>	<a href="#">1</a>
<a href="#">11306</a>	<a href="#">yoursite.com</a>	<a href="#">yoursite.com</a>	<a href="#">yoursite.com</a>	<a href="#">1</a>
<a href="#">11433</a>	<a href="#">mysite.com</a>	<a href="#">mysite.com</a>	<a href="#">mysite.com</a>	<a href="#">1</a>

Showing 3 results.

Figure 5.6 Creating new portal instances is easy in Liferay's Control Panel. Each portal instance acts as though it's a completely separate installation of Liferay Portal, with a different set of users, communities, organizations, user groups, and more.

Simply click the *Add* button to create another instance. You are limited only by the traffic coming to your server hardware in the amount of portal instances you can create.

### 5.4.2 Modifying the Default Paths

The default paths specified above in section 5.1 are just that: defaults. You can modify the paths to your markup, your images, your CSS, or your JavaScript by specifying the appropriate settings in liferay-look-and-feel.xml.

You could use this if for some reason you don't like the organization of the files the way that Liferay has them. But more importantly, you can use this setting programmatically in your theme code, using Velocity variables.

For example, Liferay's Browser Sniffer utility is available in your themes via the \$browserSniffer variable. This might allow you to write something like this in your theme:

```

#if ($browserSniffer.isMobile())
  #parse ("{$full_template_path/mobile_portal_normal.vm")
#else
  #parse ("{$full_template_path/desktop_portal_normal.vm")
#end

```

You can then present your web site in one way to users visiting with a regular browser, and a different, more lightweight way to users visiting with a mobile device. Of course, just the \$isMobile check may not be sufficient for all mobile browsers (particularly for some of the more modern devices which

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

can display pages like desktop browsers), but this is a way for you to make your themes a bit more dynamic. It really only scratches the surface of what Velocity can do.

### 5.4.3 Template Extension

Don't like Velocity? You can use JSP or Freemarker to develop your themes, simply by specifying

```
<template-extension>jsp</template-extension>
```

or

```
<template-extension>ftl</template-extension>
```

Velocity is the default and is likely to remain so, but JSPs and Freemarker are just as good as Velocity is to develop your themes. And it is likely in the future that Liferay will support other technologies for building themes. For example, Freemarker was not available in Liferay 5.2 and below; it was added for Liferay 6.

### 5.4.4 Conditional Settings

Themes can define individual settings. Using settings enables you to programmatically access the settings in your theme template and then take action based on the settings you have defined.

Settings are key / value pairs and can be defined in the `liferay-look-and-feel.xml` using the following syntax:

```
<settings>
  <setting key="my-setting" value="my-value" />
</settings>
```

After defining a setting, you can access the setting in your theme code fairly easily using the method below:

```
$theme.getSetting("my-setting")
```

For example, some web sites like to have a large banner at the top which prominently displays the site's logo and some navigation, allowing users new to the site to become familiar with the navigation. As a user delves deeper into the site, the content becomes more important than the huge banner, and it may be wiser to shrink the site logo and navigation on these pages.

You can do this by creating two themes which are exactly the same except for the header. But if you use theme settings, you can instead create only one theme and use a setting to choose the appropriate header.

In the `liferay-look-and-feel.xml` file, create two different entries that refer to the same theme but have a different value for the *header-type* setting:

#### Listing 5.x

```
<theme id="beauty1" name="Beauty 1">
  <root-path>/html/themes/beauty</root-path>
  <templates-path>${root-path}/templates</templates-path>
  <images-path>${root-path}/images</images-path>
  <template-extension>vm</template-extension>
  <settings>
    <setting key="header-type" value="detailed" />
  </settings>
  <color-scheme id="01" name="Blue">
    <css-class>blue</css-class>
    <color-scheme-images-path>${images-path}/color_schemes/${css-class}</color-
scheme-images-path>
  </color-scheme>
  ...
</theme>
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>



```

</theme>
<theme id="beauty2" name="Beauty 2">
  <root-path>/html/themes/beauty</root-path>
  <templates-path>${root-path}/templates</templates-path>
  <images-path>${root-path}/images</images-path>
  <template-extension>vm</template-extension>
  <settings>
    <setting key="header-type" value="brief" />
  </settings>
  <color-scheme id="01" name="Blue">
    <css-class>blue</css-class>
    <color-scheme-images-path>${images-path}/color_schemes/${css-class}</color-
scheme-images-path>
  </color-scheme>
  ...
</theme>

```

In the `portal_normal.vm` template use the following code:

```

#if ($theme.getSetting("header-type") == "detailed")
  #parse("${full_templates_path}/header_detailed.vm")
#else
  #parse("${full_templates_path}/header_brief.vm")
#end

```

When this theme is deployed to Liferay, it will display to the user as two different themes. When your content managers are setting up the pages for your web site, all they need to do is choose the detailed theme for the top-level pages and choose the brief theme for the pages that are “deeper” in the site. You as the theme developer only need to maintain the one theme to enable your users to use both.

#### 5.4.5 Theme Security and Roles

You can limit which themes are available based on roles defined in your portal. For example, if you have a portal installation which serves both an Intranet and an outward-facing Internet site, you may want to make it so that only the managers of the external site can choose the theme for the external site. Other users could only choose internal themes which are available to them.

By default, no role names are set, so anyone with access to the *Manage Pages* function for a community or organization can use your theme. To limit your theme usage to specific roles defined in the portal, use the following code in your `liferay-look-and-feel.xml` file:

```

<roles>
  <role-name>Internet Admins</role-name>
  <role-name>Admins</role-name>
</roles>

```

This will limit the usage of this particular theme only to users who have either the Internet Admins or the Admins role.

#### 5.4.6 Color Schemes

Liferay themes support different *color schemes*. You can therefore define a theme that has an overall style, but which can present itself in several different colors.

Color schemes are specified using a CSS class name which allows you to change colors, choose different background images, different border colors, and more (see figure 5.7).

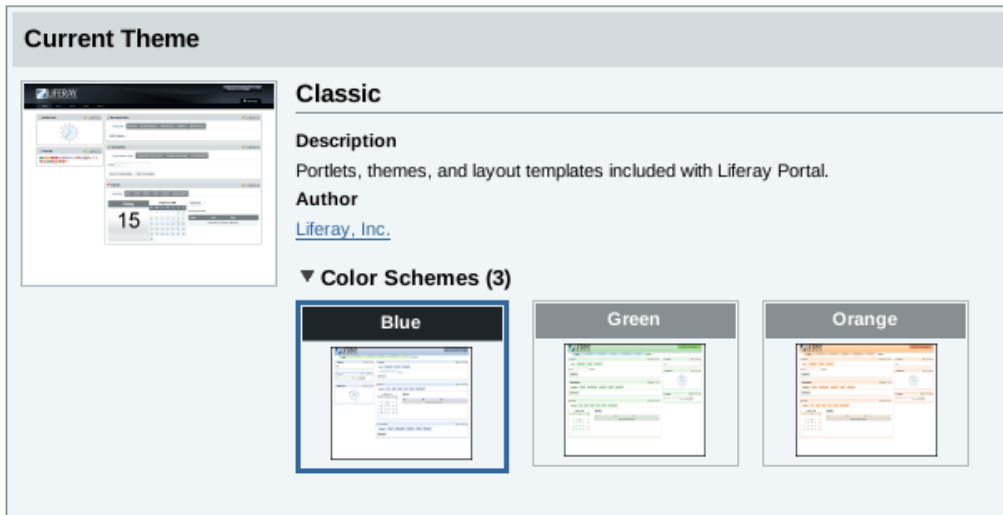


Figure 5.7 Color schemes can be added to themes to give your users more options without changing the entire look of the theme.

In your *css* folder, create a folder called *color\_schemes*. In that folder, place a *.css* file for each of your color schemes. If you were to have two color schemes—one for green and one for blue—in your theme, you could implement *green.css* as the default and specify blue as an alternative color scheme.

Inside your *custom.css* file, import the color schemes you will need:

```
@import url(color_schemes/blue.css);
@import url(color_schemes/green.css);
```

When defining your styles, use prefixes for the color schemes. For example, in *blue.css* you would prefix all of your *css* styles like this:

```
.blue a {color: #06C;}
.blue h1 {border-bottom: 1px solid #06C}
```

And in *green.css* you would prefix all of your *CSS* styles like this:

```
.green a {color: #06C;}
.green h1 {border-bottom: 1px solid #06C}
```

To enable Liferay to recognize your color schemes, modify your *liferay-look-and-feel.xml* file to define the class name prefixes you used in your *CSS* styles:

```
<theme id="a_cool_theme" name="A Cool Theme">
  <color-scheme id="01" name="Blue">
    <css-class>blue</css-class>
    <color-scheme-images-path>
      ${images-path}/color_schemes/${css-class}
    </color-scheme-images-path>
  </color-scheme>
  <color-scheme id="02" name="Green">
    <css-class>green</css-class>
  </color-scheme>
  <color-scheme id="03" name="Orange">
    <css-class>orange</css-class>
  </color-scheme>
</theme>
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

When you deploy your theme, Liferay will make it available as usual. However, when you go to select the theme for your pages, you will get a choice of color schemes to go along with it.

Let's next take a look at the styling conventions used in Liferay themes.

## 5.5 Understanding Theme Conventions

As with any complex system such as this, conventions are used which make reading the code and performing certain repeated functions easier. Theme conventions are used throughout Liferay's code and I wanted to list them here so that if you ever have to modify a Liferay theme or want your code to conform to Liferay's conventions, you'll have an easier time of it. Conventions also provide you with a nice structure for the look and feel of your site, and they speed up development because you don't have to reinvent the wheel for every site design.

The conventions are divided into two categories: one for styles and one for CSS coding.

### 5.5.1 Using Liferay's styling conventions

While it would be beyond the scope of this book to go over CSS and styling in general, I can give you an introduction to Liferay's styling conventions so that you can see how they are generally organized.

Liferay's various styles are divided up into several different CSS files which group the styles together under relevant categories. Additionally, there are some global styles which you can use to apply browser or OS specific styling to your themes. This, of course, will enable you to easily support your look and feel on multiple browsers on multiple operating systems using multiple renderers to render the same page.

[LW: Add lead-in sentence to table(s).

#### Rendering Engine

CSS Selector	Description
.gecko	Defines the Gecko rendering engine, used by Mozilla Firefox and its variants.
.webkit	Defines the WebKit rendering engine, used by Safari, Google Chrome, Konqueror, and others.

#### Browsers

CSS Selector	Description
.aol	The browser embedded within the America Online client.
.camino	A Mac-based browser which uses the Mozilla Gecko rendering engine.
.firefox	Mozilla Firefox, one of the most popular browsers in use today. Uses

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

CSS Selector	Description
	the Gecko rendering engine.
.flock	A browser which focuses on support for social networks, based on the Gecko rendering engine.
.icab	A shareware browser for Mac-based systems.
.konqueror	The default browser on the Linux-based KDE desktop. This browser originated WebKit and is still WebKit-based.
.mozilla	A catch-all for Mozilla-based browsers.
.ie	Microsoft Internet Explorer, one of the most popular browsers in use today, and the default on Windows systems.
.netscape	The Netscape browser. Recent versions are based on the Mozilla Gecko rendering engine.
.opera	Opera is a browser used on both the desktop and many mobile devices, as well as the Nintendo Wii.
.safari	Apple Safari is the default browser on Apple Macintosh systems. This browser is based on WebKit.
.browser	A catch-all for all browsers.

## Operating Systems

CSS Selector	Description
.win	Microsoft Windows operating systems
.mac	Apple Macintosh operating systems
.linux	Linux operating systems
.iphone	The iPhone operating system
.sun	Sun Solaris operating systems
.os	A catch-all for other operating systems

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

Using these selectors, theme developers can create styles which apply just to these browsers, rendering engines or operating systems. The really nice thing about that is you can enforce a consistent look and feel regardless of browser. Your users will then—regardless of their browser choice—have the same experience using your site. Liferay makes this very easy for you to do.

If, for example, you need to create a style for a specific browser and version, you might use the following code:

```
.ie6 a { }
```

This style will take effect for the anchor tag only for users who visit your site on Internet Explorer version 6.

You can also combine the selectors. This becomes useful when you want to target users of specific browser and operating system combinations. Since several of the above browsers are cross-platform, you might want to style something for particular users. For example, if you wanted to target Firefox users who are running on Microsoft Windows, you might use the following code:

```
.firefox .win #hidden-message {  
    visibility: visible;  
}
```

Not only can you do that, but you can also have styles specific to color schemes:

```
.firefox .blue #hidden-message {  
    visibility: hidden;  
}
```

I hope you can see that using these selectors makes handling multiple browsers a lot easier, and that it frees you from having to deal with the contortions of other, less elegant means of browser detection and styling.

### **5.5.2 Using Liferay's CSS Coding Conventions**

Liferay adheres to certain code conventions when providing CSS. These are designed to make the code easy to read. If your organization already has code conventions for CSS, they may be very similar to Liferay's. Please note that if you wish to contribute changes to Liferay's CSS, you will need to follow these conventions when you submit your code.

#### **OUTSIDE OF SELECTOR BODIES:**

- Group selectors under the common element that they style using comment tags.
- Keep global selectors toward the top and more specific selectors toward the bottom.
- Keep only one end line between all selectors and comments.

#### **INSIDE SELECTOR BODIES:**

- Insert only one space between selector name and the opening bracket ({}).
- If you are using multiple selectors for the same body of declarations, separate selectors with a comma and one end line.
- All declarations should be indented by one tab.
- Keep all declarations in alphabetical order within the declaration body.
- For each declaration, be sure to put one space between the property and the value, after the colon.
- Colors should only be specified by using their hexadecimal value.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

- Use all caps for hexadecimal values, and condense to 3 digits whenever possible.
- For the *background* property, make sure there is no space between the *url* attribute and the opening parenthesis.
- No quotes are required when using *url()* or for font names, unless you are using fonts that aren't browser-safe.
- Insert spaces after commas between font names. This enhances readability.
- When using URLs, always use a relative address instead of an absolute one.
- Condense all padding, margin, and border values whenever possible, leaving out measurement units on 0 values (i.e. *px*, *pt*, *em*, %).
- Comments are only used to head and divide each section of the code appropriately. Comments without any content below them are not necessary.
- Use shorthand properties where applicable.

Now that you've had an opportunity to see how you can take your design skills to the max with Liferay themes, we can look at something really easy: layout templates.

## 5.6 Designing a page with layout templates

Layout Templates are ways of choosing how your portlets will be arranged on a page. They make up the body of your page, the large area where you drag and drop your portlets to create your pages. Liferay Portal comes with several built-in layout templates, but if you have a complex page layout (especially for your home page), you may wish to create a custom layout template of your own.

Layout Templates are the easiest plugins to create. They comprise only a few files with simple table or CSS-based containers into which portlets can be dropped, as well as a thumbnail image of what the layout looks like. This thumbnail is displayed when the end user clicks *Layout Template* from the Dock menu.

### 5.6.1 Creating Layout Templates

Creation of layout templates is done in a similar manner to the creation of portlets and themes. There is a *layouttpl* folder inside the plugins SDK where all new layout templates reside. To create a new layout template, you run a command in this folder similar to the one you used to create a new portlet or theme. For LUM, type:

```
./create.sh <project name> "<layout template title>"
```

For example, to create a layout template with a project folder of *3-columns* and a theme title of *3 Columns*, type:

```
./create.sh 3-columns "3 Columns"
```

On Windows, you would type:

```
create.bat 3-columns "3 Columns"
```

This command will create a blank layout template in your *layouttpl* folder.

### 5.6.2 Anatomy of a Layout Template

Layout Template projects are very simple. Say, for example, you wanted to create the above *3 Columns* layout template, with left and right columns that take up 20% of the available space, and a middle

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

column that takes up 60%. This is a very common site layout and surprisingly, Liferay does not ship with it (though it does have the aforementioned three evenly-spaced columns template).

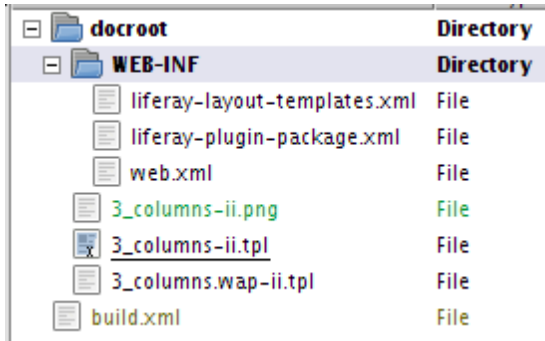


Figure 5.8 Layout Templates are perhaps the simplest of Liferay plugins. They contain only a few files and are very easy to configure.

To separate our template from the one that ships with Liferay, we'll call it "3 Columns II" as it will be the newer, slicker sequel. After running the above *create* Ant script, we have a new project with the layout pictured at the right.

Our next step is to open the *3-columns-ii.tpl* file and create our three column template. This file is for regular web browsers to use; Liferay will automatically detect the client being used to connect to the site and serve up the appropriate template. If the client is a phone, it will serve the *3\_columns.wap-ii.tpl* file.

Open the *3-columns-ii.tpl* file in your text editor of choice. You will see that by default, it is a completely blank file. In future versions of the Plugins SDK, there may be a commented out template in there to help you get started. For now, though, we'll provide the code for you to use.

Paste the following code into the file:

```
<div class="columns-3" id="content-wrapper">
  <table class="lfr-grid" id="layout-grid">
    <tr>
      <td class="lfr-column twenty" id="column-1" valign="top">
        $processor.processColumn("column-1")
      </td>
      <td class="lfr-column sixty" id="column-2" valign="top">
        $processor.processColumn("column-2")
      </td>
      <td class="lfr-column twenty" id="column-3" valign="top">
        $processor.processColumn("column-3")
      </td>
    </tr>
  </table>
</div>
```

You'll note that each table cell has a CSS class associated with it, as well as an ID. These may be customized by modifying the theme you are using to display the layout.

For the WAP version of the file, we'll use simpler syntax:

```
<table>
<tr>
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

```

<td>
    $processor.processColumn( "column-1" )
</td>
<td>
    $processor.processColumn( "column-2" )
</td>
<td>
    $processor.processColumn( "column-3" )
</td>
</tr>
</table>

```

WAP doesn't have the benefit of CSS, so we will have to settle for three evenly-spaced columns.

The other file you need to customize is the *3-columns-ii.png* file. The default file in the project is a blank layout template preview. You'll need to customize it in an image manipulation program such as the GIMP (<http://www.gimp.org>), Adobe Photoshop, or whatever it is that you use. This file can be modified so that it looks like your layout. This should make it blend in somewhat with the other layout template preview icons.

Once you have customized the icon, all that is left is to deploy the layout template, as the various configuration files have been already generated properly by the Ant scripts in the Plugins SDK.

## 5.7 Inkwell Implementation

Using all of this information, the design team at Inkwell created a theme for their Internet which they thought would appeal to their customers. Since Inkwell is a company that successfully mixes digital technology with the real-world, analog feel of fountain pens, it was thought that their web site should reflect that as well.

So the designers set out to design a web site that would mimic the experience many of their customers have of writing with their products on a desk, which you can see in figure 5.9.

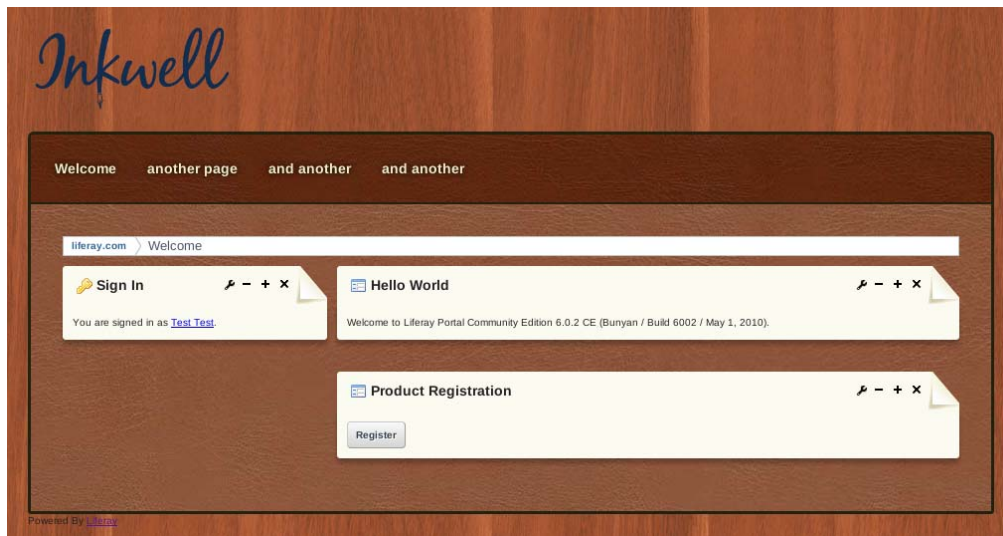


Figure 5.9 The Inkwell theme features a wood grain desktop surface, a leather blotter, and various “pages” laying on the blotter. Of course, we know those pages are really portlets, don't we?

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>



So the Inkwell web site is starting to take form a bit. Using their existing HTML, JavaScript, and CSS skills, the design team was able to create this theme very quickly, using the tools they were already comfortable using. And once the content creators begin creating content for the site, it will really begin to shape up.

## 5.8 Summary

This chapter has introduced you to the world of creating themes for Liferay Portal. You have learned how to create theme projects in the Plugins SDK, and have seen what the overall structure of a theme looks like.

Themes are based on differences from a base that is provided by Liferay, and this enables you to keep your themes small and light and rely on the baseline that Liferay has provided. From there, you learned how Liferay's CSS files are organized, and where you can find the styles you need to override in `custom.css`.

Next, you learned how and where to implement your own JavaScript functions in your themes. AlloyUI plays a prominent role in the JavaScript that is available to you within Liferay, not only for themes, but also for portlets. You were also introduced to some of the JavaScript that is provided by Liferay in addition to AlloyUI. This allows you to take advantage of functionality that is already provided out of the box and, more importantly, is maintained by the Liferay development team.

You then learned the ins and outs of the `liferay-look-and-feel.xml` file, which allows you to provide many features in your theme, including security, settings, color schemes, and more. You also learned how Liferay provides you an easy way to style your theme elements so that your theme can be supported by all browsers.

Finally, you saw how simple layout templates are to create, enabling you to arrange your portlet applications on your pages in any way that suits you. Coupled with your theme, you get the flexibility to present your content and applications however you have designed them.

All of this can be brought together to implement a look and feel that makes Liferay appear to be anything. Inkwell's design team used this information plus Liferay's CSS style conventions to implement a look and feel that is appropriate for their business.

Now armed with the information from this chapter, you should be able to go ahead and create beautiful themes for Liferay. We'll move from here into some of Liferay's specific APIs, starting with its API for social networking.

# 6

## *Making your site social*

This chapter covers

- The growth of social networking on the web
- Integrating your Liferay-based site with social networks
- Liferay's social networking plugin
- Using Liferay's Social API to connect users
- Checking for social relationships
- Publishing activities from your applications

Unless you've been hiding under a rock for the past several years, you've probably heard of the term *Social Networking*. Web sites like Facebook and MySpace have made the term almost ubiquitous, and now we have a craze on our hands similar to the general Internet craze of the '90s. It used to be that you would meet and interact with different people on different web sites, primarily using discussion forums to talk about topics of common interest. Nowadays, social networking brings that idea to a whole new level, allowing users to put more of their activities online, granting others permission to see what's going on in their lives.

My first high school reunion was, uh, before the advent of social networking. It was pretty much as has been described in countless movies, TV shows, and novels—except, of course, for its size. Though I grew up in a Jersey shore town (no jokes, please) which is ostensibly larger than Superman's small town upbringing, my reunion was not as big as the one in Smallville depicted in Superman 3. We didn't rent out any big halls in a hotel, restaurant, or even in a fire station. No, we had ours in a bar. Did I get to see some people I hadn't seen since high school? Yes. Was it my whole class? Not by a long shot. Nowadays, I guess, people get so spread out that there's little chance that, once they get disconnected, they'll find each other again.

That is, until the advent of Facebook.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

I'm obviously older (not *that* much older) than the original Facebook generation, which I define as those who were able to access Facebook back when it required a University email account. So I resisted joining Facebook for a while, because, in my infinite wisdom, it was something "those kids" used. I certainly wasn't going to find anybody I knew on there—except, of course, for the people younger than me who were telling me I should join Facebook.

Boy, was I wrong.

People I never thought would be on Facebook were there. Heck, people I wasn't sure knew how to use the internet were there. And, of course, far more people that I knew from high school were there than were at my reunion. I could see what they were doing, could contact them and talk to them directly, see pictures of them and their kids, and become somewhat of a participant in their lives again. And of course, they could do the same with me. I had to admit: that was pretty cool. And that's the power of social networking.

For these reasons, social networking has become very popular, and all kinds of web sites are building social features into their previously un-social experience. If you're building your site on Liferay, you will be happy to know that Liferay contains a whole API for building social web sites. This API is tried and tested and is in use today, powering sites which connect users for everything from exercise routine accountability to making friends with one another's pets. So let's take a deeper dive into this concept and see what it really entails.

## **6.1 Social Networking: why is it important?**

There are all kinds of social networking sites which are used for different purposes. Sites like Facebook and MySpace are general purpose social networking sites—anybody can get an account and start posting stuff there to share. Other sites have a more focused purpose. For example, LinkedIn is meant for those work relationships you've built over the course of your career. You won't be posting family pictures there, but you might post your resumé and current activities, and you certainly might want to keep connected to current and former colleagues in order to take advantage of opportunities to work together again.

As you can see, the same exact features power all of these varying purposes. And this list of features can be used to expand your reach, by using your connections. This translates into a more positive user experience, because users are presented with the connections they want and the information they want. We're about to see how that works in more detail.

### **6.1.1 Allowing users to connect with each other**

Quick: picture someone you haven't seen or talked to in 5 years. What's the first thing you might want to know about what's happened to this person in the intervening time? It all depends on your relationship, doesn't it? If you had a professional relationship with the person, you might wonder if he or she is still working in the same place or in the same position. If you had a friendship with the person, you might wonder what he or she is up to and who he or she is hanging out with now.

Regardless of the reason, people connect with each other all the time in real life. Social networking tries to mirror that in the online world. This becomes especially useful when people are separated by distance or life patterns that now differ (such as not working at the same physical location anymore).

Allowing users to connect with each other on your web site will make your site more popular with your users. When users connect and interact online, they are in control. If users are in control, they will

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

be far more likely to participate. Users participate because of the relationships they have with other people online. If they can be free to pursue those relationships on *your* site, they will keep coming back. To misquote James Carville, "It's the relationships, stupid."<sup>3</sup>

Liferay has a social networking API which you can use to allow your users to make connections with each other. Once those connections are made, you can use the API to check whether connections exist, query a list of connections, and a whole lot more. We'll be going through examples of using this API as the rest of the chapter unfolds.

First, though, we'll look at how you can increase participation on your site by connecting it to other social networks.

### 6.1.2 Expanding your reach beyond your own site

If you've used Facebook, chances are you've played at least one game on the site. Though I don't frequent the games, I have to confess I played one or two before the novelty wore off. Did you know that the games on Facebook really aren't on Facebook? Instead, they're served from their own servers, by the entities who created them. And many of these entities operate their own sites which allow you to go play the games directly. It's just that Facebook gives them a visibility they wouldn't have if they didn't make their games available there. And because of that visibility, more people play their games.

Liferay Portal allows you to serve your Liferay-based applications on Facebook, as a Google Gadget, on Netvibes, or any other web site. You don't have to do anything extra to make this happen. In fact, you can add the Product Registration portlet you wrote in chapter x and y to any external web site. Simply click the *Configuration* button and you have options for Facebook or any other site, as seen in figure 8.x.

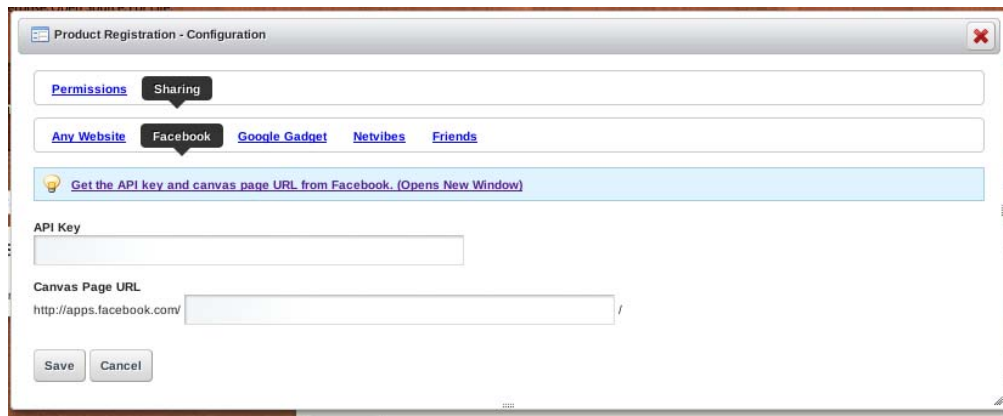


Figure 8.x. You can add your Liferay-based applications to other social networking sites, such as Facebook. You can also let friends share your application. Who knows? Maybe your application can become viral!

There's absolutely nothing stopping you from writing an awesome role-playing game on Liferay, serving it up on Facebook, and making millions. Nothing, of course, except for your own ingenuity. This

<sup>3</sup> [http://en.wikipedia.org/wiki/It%27s\\_the\\_economy,\\_stupid](http://en.wikipedia.org/wiki/It%27s_the_economy,_stupid)

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

brings me to my next point, about how social networking gives users a more positive experience by bringing their many interests together.

### **6.1.3 *Creating a dynamic, more positive user experience***

Way back, I used to use browser bookmarks extensively. I'd find a site, realize I wanted to visit it again, and bookmark it. I even spent a lot of time organizing my bookmarks into folders because I had so many and I was afraid I wouldn't be able to find a particular site again. Over time, I used bookmarks less and less, to the point where I still have that highly organized folder structure of bookmarks to sites that mostly don't exist anymore. I've found that I tend to visit only a few sites repeatedly now, and very rarely do I come across something that I'd like to bookmark for later.

What's the point of all this? The point is that aggregation is key on the web in the 21<sup>st</sup> century. We tend to visit fewer individual sites because the sites we do visit aggregate so much content that we don't have to go all over the place to find what we're looking for. And we now get to that content because of social networking features that are built into these sites.

It's ridiculously easy to upload and share videos on Facebook or YouTube. Additionally, those sites have the bandwidth to serve up those videos optimally so that people can view them without a lot of buffering and stuttering. If I come across an interesting blog post or news article, I can post a link to it on my Twitter feed if I want to share it. Because I am sharing these links via social networking, anyone connected to me can see it. And I have things set up so that I can post simultaneously to Twitter, Facebook, and LinkedIn if I want to. That's way better than spamming all my friends via email.

All of this is to say that social networking creates a more positive user experience than the siloed applications we used to have. It's much easier to upload your own content, share links, and generally communicate using social networking tools than it is by using separate applications like email, static web sites, or single-purpose web applications. Social networking combines all of that into one dynamic experience. And Liferay Portal is an engine that can power it all. So now that we understand what social networking is and what we can use it for, let's get started building those features into Inkwel's web site.

## **6.2 *Installing Liferay's social networking portlets***

Liferay by default contains all you need to build a social networking web site from the ground up. You could use the social API to build exactly the experience you want. Liferay has, however, also created a default implementation (using this API) of several social networking applications that tend to be common across social sites. If you use these, you'll get a jump start on your site, and you'll only need to concentrate on the features you need to build. Of course, if you don't like the way Liferay has implemented things using its API, you can provide your own implementation, because it's all done with plugins.

To install the Social Networking portlet plugin, choose Plugins Installation > Install More Portlets from the Control Panel. Once installed, the Social category of the Add > More menu displays additional portlets, as you can see in figure 8.x.

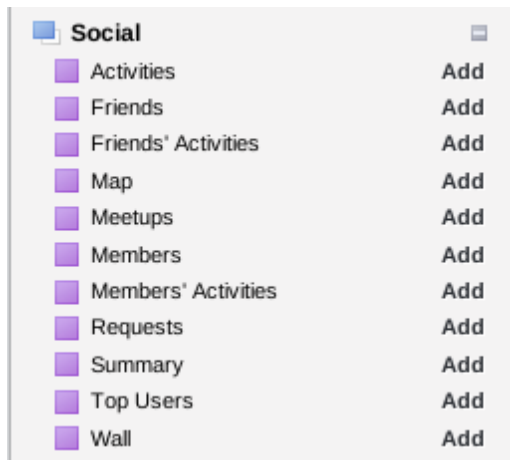


Figure 8.x. Liferay provides many social portlets which you can use as a starting point to get your social networking site running.

Before we start putting these portlets on pages, let's take a step back and see what they do. That way, we can make the best assessment we can of how Inkwell might want to use them.

### 6.3 Understanding Liferay's social features

As has been already stated, Liferay Portal contains an API specifically designed for building social web sites. This API enables you to implement the following features:

- Relating with others
- Publishing activities
- Determining online presence (in the case of the Chat portlet)

These functions can power a whole host of features, as we are about to see.

#### 6.3.1 Relating with others

Friends, connections, buddies—whatever you want to call them, social networking is powered by relationships. Those relationships are defined as connections between people. Liferay calls these *social relations*, and so the methods in the API reflect that terminology. This means you'll be seeing methods with names like `addRelation`, `isRelatable`, and such. Of course, Liferay goes further than simply providing the API; it has the Summary portlet which implements this API to create those social relations in ways you would expect. Though you might want to implement your own version of this portlet, you've got a great starting point in the Summary portlet, which we'll see in a bit.

Once you've established relationships with people, the next step in social networking is to keep up with their activities.

#### 6.3.2 Publishing activities

I'm sure you remember that way back in Chapter 1, I showed you a little thing called the Activities portlet. This portlet is a one-stop shop for all of your activities. In that example, I showed how adding a

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

wiki portlet to a page generated an activity, because whenever you add a wiki to a page, the default wiki article is created at the same time.

Of course, the wiki is an internal portlet that comes with Liferay. This, however, doesn't mean that you can't publish activities in your portlets too. We will see how to do exactly this, as we'll be enabling the Product Registration portlet for social networking. If a user with an account registers a product, we'll go ahead and make that an activity which is displayed in the Activities portlet.

Activities are important, but they are only visible to those who are socially related. Users become related by responding to social requests.

### 6.3.3 Sending social requests

Liferay's social API allows users to send other users social requests. Liferay's social networking portlets have an implementation of this in the Summary portlet and the Requests portlet. So one way to implement this right out of the box is to put the Summary portlet on users' public profile pages, where other users can see it (see figure 8.x).

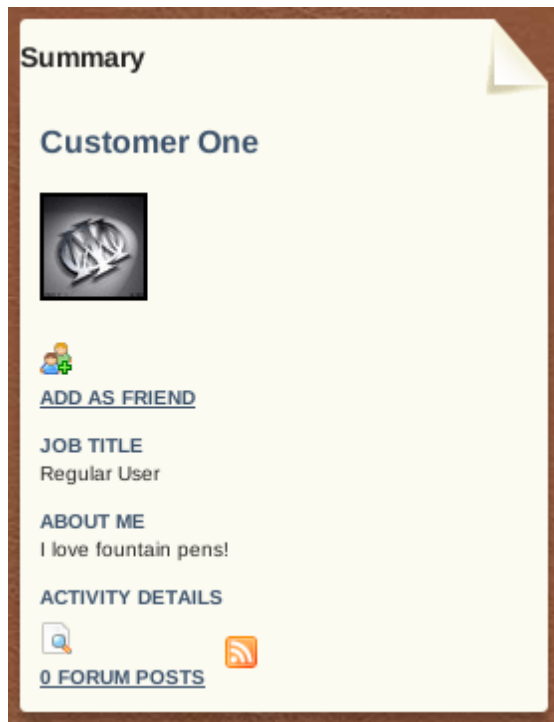


Figure 8.x If the Summary portlet is placed on a user's public profile page, it displays a link labeled *Add as Friend*. To Liferay, this is just a social relation, so you can label it anything you like. For example, if you were doing a site for fantasy role-playing gamers, you might label it *Add As Fellow Warrior*.

You, of course, don't have to use Liferay's Summary portlet; it just happens to already include a user interface for adding a social relation. You can implement any portlet you like and use Liferay's social API

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

in the same way that the Summary portlet does. But let's continue with how this flows, so we can get a high level understanding of how it works before we dive in to the low level. Clicking the *Add as Friend* link will send a social request to that user, and the user can then choose whether he or she wants to confirm that the requestee is indeed somebody with whom it is worthwhile to have as a social relation.

Social requests are captured by the Requests portlet, which ships by default in Liferay. When you place it on a page, if the user has no requests, it's invisible. When a user has a request, it displays that request, as you can see in figure 8.x.

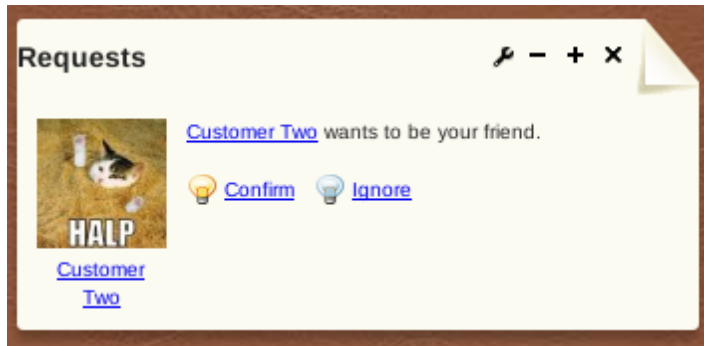


Figure 8.x The Requests portlet shows a list of those who have requested to be a social relation. You can see their profile pictures and links to their profiles, and you can either confirm the request or ignore it.

Obviously, all of these social portlets are designed to be placed on users' individual profile pages. Once there, they'll display information appropriate for that user. Inkwell wants to do this for their web site, so let's delve into the configuration for Inkwell's site. We'll get the out of the box social networking up and running first, and then we'll modify the Product Registration portlet so that it displays activities whenever a user registers a product.

## 6.4 Using profile pages

Inkwell wants users to have profile pages, but they don't want them using the *Add > More* menu to put whatever portlets they want there. Instead, they'd rather define a static layout for users' pages that the users cannot modify later. This is very easy to do with a few custom properties added to the `portal-ext.properties` file. You may remember that we used this file in Chapter 2 to connect Liferay to a MySQL database. There are many, many configuration options for Liferay which you can put in this file, and we're going to use a few to control how users' profile pages work.

Everything you put in `portal-ext.properties` overrides the default in Liferay's `portal.properties` file. You can find full coverage of this file in Liferay's documentation or in the file itself if you download Liferay's source code. For us, we're interested in the following properties:

```
layout.user.private.layouts.enabled=true
layout.user.private.layouts.modifiable=false

layout.user.public.layouts.enabled=true
layout.user.public.layouts.modifiable=false
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>



The first two properties control the private layouts, which are the user's private pages. The second two properties control the public layouts, which are the user's public pages. We're making sure the layouts are enabled, and we're making them so they're unmodifiable. This will keep the users from being able to add portlets or move them around on their pages.

Next, we need to provide defaults for users' profile pages. Inkwel management has determined that the public pages should have the two column, 30/70 layout. The first column (the thinner one) should contain the Summary and Search portlets, and the second column should contain the Activities and Wall portlets. The private pages will contain the defaults, except the Requests portlet will be added to the top of the left column and the Friends' activities will be added to the top of the right column. We can configure this through properties as well. But first, a note about portlet IDs.

### 6.4.1 Identifying a portlet

Like secret agents, portlets in Liferay are identified by a code number, not a name. We didn't adhere to this convention when we did our portlets in the earlier chapters because frankly, it's confusing. Let me show you what I mean.

We identified our Product Registration portlet like this:

```
<portlet>
  <portlet-name>product-registration</portlet-name>
  <display-name>Product Registration</display-name>
  ...
</portlet>
```

Makes sense, right? The portlet is given a name without a space, and the display name (i.e., what appears in the portlet window) looks the way a human would want to read it.

If we look at the way this is configured in the portlets that ship with Liferay, we see something else entirely:

```
<portlet>
  <portlet-name>121</portlet-name>
  <display-name>Requests</display-name>
  ...
</portlet>
```

As you can see, the portlet is given a code number for a name, while its display name is a human readable name telling us that this is the Requests portlet. I don't know about you, but I can translate *Requests* to *requests* in my head in order to come up with what the portlet name for a given portlet might be, but I'll never be able to translate *Requests* to *121*. I'll always have to go look it up.

For portlets that are parts of plugins, they also are identified by their portlet name, and yes, Liferay numbers those too. And the numbers overlap. So if, for example, you wanted to refer to the Summary portlet, you'll see that it is defined in the plugin's `portlet.xml` file like this:

```
<portlet>
  <portlet-name>1</portlet-name>
  <display-name>Summary</display-name>
  ...
</portlet>
```

Why am I going over this? Because we are going to define certain portlets to be on users' profile pages by default, and we need to be able to refer to them by their portlet names in order to do this. When we do this for a portlet that ships with Liferay, we refer to it simply by its portlet name, which Liferay defines as a number. When we do this for a portlet that is in a separate WAR file (i.e., a plugin), we refer to it by its name and then we append `_WAR_[name of war]`. We'll set this up next.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

## 6.4.2 Defining content for public and private pages

Once we know which portlets we want to be on users' pages by default, we use some properties to set up this configuration. Since Liferay already provides defaults for this, we override what Liferay ships with by placing these properties in our `portal-ext.properties` file.

```
default.user.public.layout.column-1=1_WAR_socialnetworkingportlet,3
default.user.public.layout.column-2=116,3_WAR_socialnetworkingportlet
```

```
default.user.private.layout.column-1=121,2,23,11
default.user.private.layout.column-2=4_WAR_socialnetworkingportlet,29,8
```

As you can see, we set up the public pages so they look as described above by defining portlets by their ID in the columns in which they belong.

When we do this for public pages, we get the result we wanted, shown in figure 8.x.

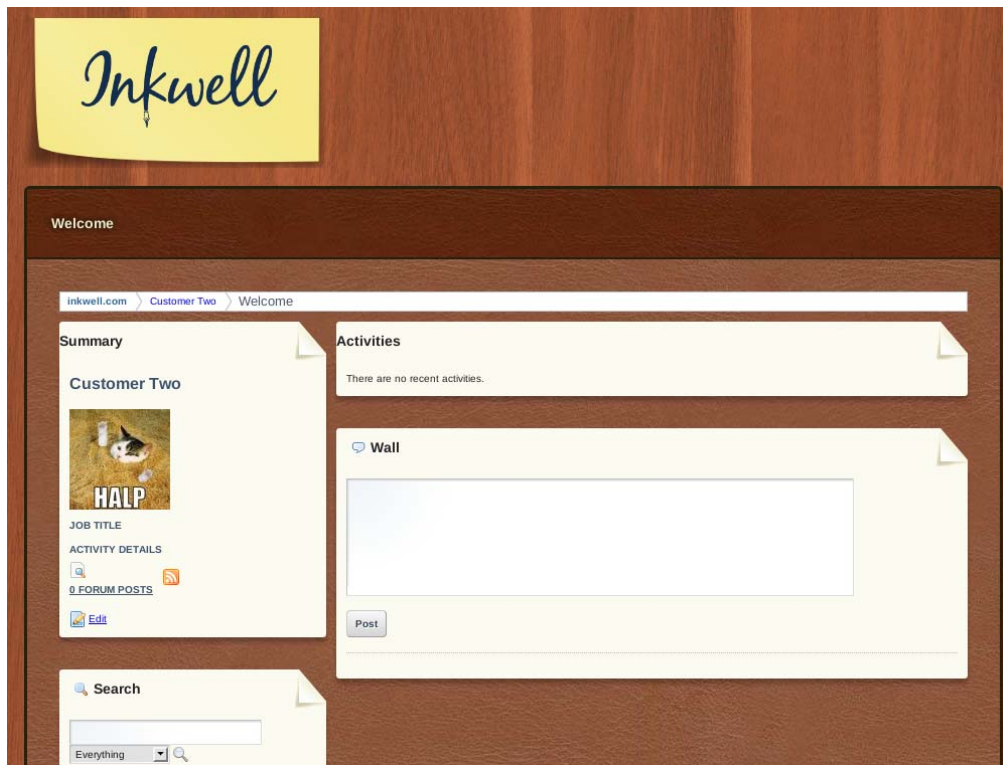


Figure 8.x. Just as we wanted, users' public pages contain the Summary, Activities, and Wall portlets, and they are placed right where we wanted them. The Search portlet was there by default, and we've left it there, as it'll likely be useful.

Similarly, we configure users' private pages so that they contain the portlets we want, using similar properties. Notice that we are using the convention outlined above: for the portlets that ship with Liferay, we use just their numbers, which really are the portlet names according to the spec. For the portlets that are contained in the social networking plugin, we use the number plus the suffix.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

We could, of course, have done the same thing with our Product Registration portlet, but that portlet is not something we'd want on users' profile pages, is it?

In any case, the result for the private pages is shown in figure 8.x.

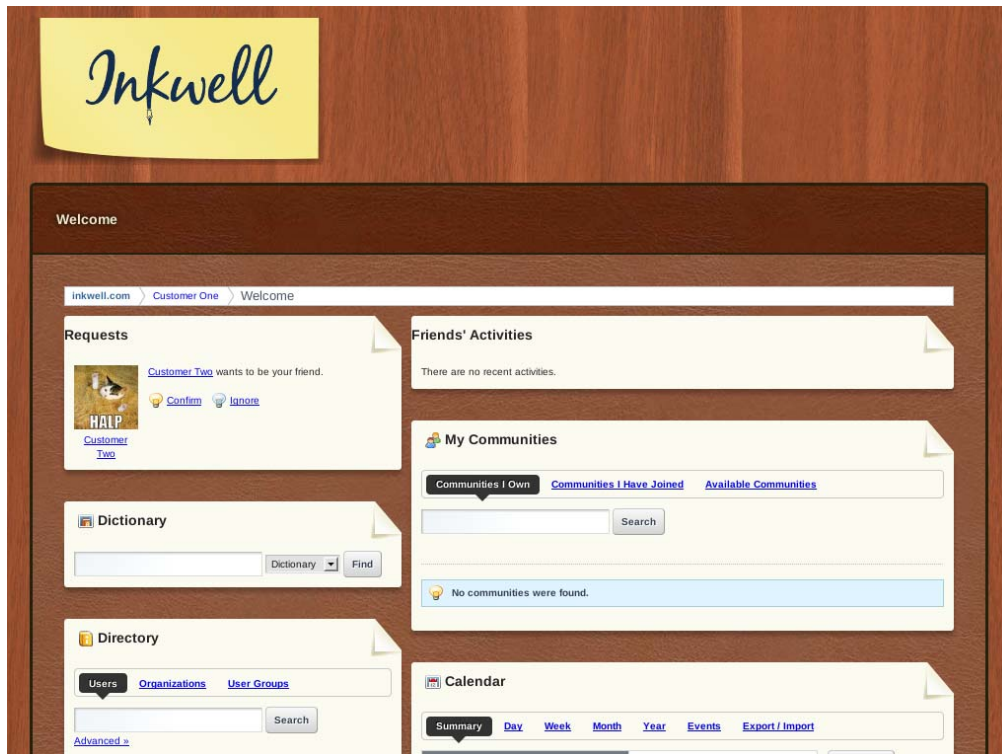


Figure 8.x. Users' private pages, in addition to the defaults provided by Liferay (which we've left in place), contain the Requests portlet and the Friends' Activities portlet. You can see that Customer Two has sent a social relation request to Customer One.

As you can see, using Liferay's out of the box social networking features is quite easy. The product provides you with all the tools you need to set up social networking in a generic way. Your next task, of course, will be to integrate your own applications with your budding social network.

## 6.5 Friends, Romans and Countrymen: All Social Relations

You may have noticed in figure 8.x above that the Requests portlet says "Customer Two wants to be your friend." This comes out of the `Language.properties` file in the portal, not from the portlet itself. Remember: Liferay calls this concept internally a "social relation," in order to keep the concept as generic as possible, and to let *you* decide what you want to call these things in your own social network. They can be friends, Romans, countrymen, connections, relatives, coworkers, or whatever.

We will see when we go over Hooks in Chapter 10 how to customize the `Language.properties` file in the portal so that the Requests portlet displays whatever you want it to. If you want your social

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

relations to be “homeys,” so be it. But one last important factor about social networking must be mentioned before we delve deeper into the code.

### 6.5.1 This is not security

It is important to note that social relationships are not security. Yes, portlets can be written to display or not display their contents based on whether the viewing user has a social relationship with the user who “owns” the portlet, as you can see with the Wall portlet in figure 8.x. But social relationships are not built into the security framework of Liferay, and so they don't inherit any of the role-based security and permission checking features of the platform.

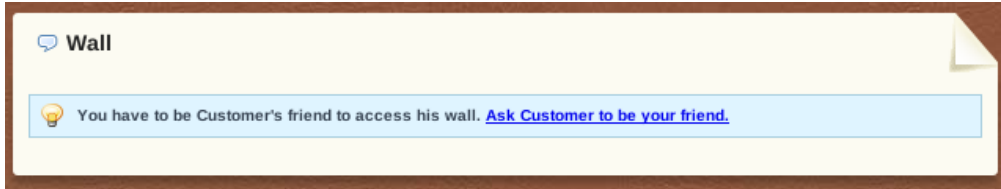


Figure 8.x The Wall portlet specifically checks to see if the current user hasRelation with user whose profile is being viewed. If the user does not have a social relation, the wall portlet will not display its contents. This is a different check than the permissions check we saw in Chapter 6.

In short, if you want to protect the contents of a portlet based on whether the current user has a social relation with another user, you will need to make a separate check. The Wall portlet does this, and any portlets you write which will take advantage of Liferay's social API will need to do the same. So let's see how the Wall portlet does this so you can get a feel for it.

### 6.5.2 Coding for relationships

Social portlets which display data specific to a particular user are designed to be placed on users' profile pages. So rather than using the `PermissionChecker` to determine whether certain data should be displayed (like we did in Chapter 6) we will instead check a couple of other things:

- Is the portlet placed on a user's profile page?
- Does the user viewing the portlet have a social relationship with the user upon whose page the portlet has been placed?

Checking whether a portlet is on a user's page or a community / organization's page is fairly simple using the `scopeGroupId`. You may remember this from Chapter 6: you can get it out of Liferay's `themeDisplay` object, and it denotes the ID of the community or organization to which the user has currently navigated. Internally, Liferay calls communities and organizations *groups*. Using the ID, you can get the Java object representing that group, and from there, get the group type. This is how the Wall portlet does it:

#### Listing 8.1 Checking the group type

```
Group group = GroupLocalServiceUtil.getGroup(scopeGroupId);           A
Organization organization = null;
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

```

User user2 = null;

if (group.isOrganization()) {
    organization = OrganizationLocalServiceUtil.getOrganization(group.getClassPK());
}
else if (group.isUser()) {
    user2 = UserLocalServiceUtil.getUserById(group.getClassPK());
}

```

#### **A Getting group**

#### **B Checking type**

This code checks to see if the current group—which is retrieved using the `scopeGroupId` found on the current page—is a user's group or is an organization. If the group is a user group, we need to retrieve that user, because a little later we're going to check to see if the current user has a social relation with the user who owns this page.

Next, the Wall portlet uses a small block of code to check to make sure the portlet itself has been placed on a user's profile page:

```

<c:choose>
    <c:when test="<%= user2 == null %>">
        <div class="portlet-msg-error">
            <liferay-ui:message key="this-application-will-only-function-
when-placed-on-a-user-page" />
        </div>
    </c:when>
    <c:otherwise>
        <%@ include file="/wall/view_wall.jspf" %>
    </c:otherwise>
</c:choose>

```

We know we're on a user's page if the `user2` variable was populated in the previous code block. So here, all we have to do is check to see if it has a value. If it has a value, we'll include the JSP fragment which displays the wall. If it doesn't have a value, we show a message from the `Language.properties` file telling users to place this portlet on a user's page.

Here comes the fun part. When it's time to display the wall, we need to make sure that the user attempting to view the wall has a social relation to the user upon whose profile the wall resides. And these social relations can be of different types, which allows you to support a wide variety of relationships, both bi-directional and uni-directional. We'll get to that in a minute; right now, let's look at what the social relationship check looks like.

```

<c:choose>
    <c:when test="<%= themeDisplay.isSignedIn() && ((user.getUserId() ==
user2.getUserId()) || SocialRelationLocalServiceUtil.hasRelation(user.getUserId(),
user2.getUserId(), SocialRelationConstants.TYPE_BI_FRIEND)) %>">

[logic for displaying the wall goes here]

    </when>
</choose>

```

We've got several checks in one line here. We check for the following stuff:

- Is the user signed in?
- Is the user viewing his / her own profile, OR
- Does the user viewing the profile have a social relation with the user who owns the profile?

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

Notice the social relation check: it includes as part of the check a social relation *type*. In this check, we're checking for a type of *friend* that is *bi-directional*. That's the definition of a friend, right? If you're my friend, I must be your friend, otherwise the friendship doesn't exist. This is only one type of social relationship that's available in the API, and this gives you options you might not have considered. For example consider the possibility of *another* Wall portlet—this one coded to check for the uni-directional relationship type of parent or child. This way, content uploaded by users can be targeted to the people with the appropriate relationships.

For example, you might want to create a portlet for uploading photos. This portlet would use the back-end API of Liferay's Image Gallery for the actual photo storage. But on the front-end, you could filter photo folders by the type of social relationship users have. Some pictures would be appropriate for co-workers. Other pictures would be appropriate for family. All of the relationship types are defined in a file called `SocialRelationConstants.java` in the Liferay source. The types are:

- `TYPE_BI_COWORKER`
- `TYPE_BI_FRIEND`
- `TYPE_BI_ROMANTIC_PARTNER`
- `TYPE_BI_SIBLING`
- `TYPE_BI_SPOUSE`
- `TYPE_UNI_CHILD`
- `TYPE_UNI_PARENT`

If you look at the request that was sent in the Summary portlet, you'll see the following code:

```
SocialRequestLocalServiceUtil.addRequest(
    themeDisplay.getUserId(), 0, User.class.getName(),
    themeDisplay.getUserId(), FriendsRequestKeys.ADD_FRIEND,
    StringPool.BLANK, user.getUserId());
```

That `FriendsRequestKeys.ADD_FRIEND` variable maps to the type of social relationship that is being requested. This variable is defined in `FriendsRequestKeys` in the social-networking-portlet project, and its value is 1. If, however, you look at the types defined in Liferay's `SocialRelationConstants.java` file, you'll see that 1 is actually `TYPE_BI_COWORKER`. So while the Summary portlet and the Requests portlet are communicating to the user that he or she is making a "friend" request, the actual request is a co-worker request.

Why is this? Because the portlets in the social-networking-portlet project were originally designed specifically for use on Liferay's web site. On Liferay's web site, Liferay employees are made "friends" with each other automatically—and of course, now we know that the relation type is actually co-worker. So unlike some of the other portlets that are available in the repository, these are provided as examples of using the Social API, not necessarily as drop-in functionality like the Mail portlet or the Chat portlet. But think of the possibilities: you could write a portlet which allows users to request all different types of social relationships. And once your users organize themselves into these relationships, you can provide all sorts of social functionality based on these relationships.

Now that you understand how all these social relationships are created, we need to figure out what to do with them. Users perform *activities* on your web site, don't they? So next we'll turn to how we can publish these activities to those with whom we have a social relationship, and we'll do this specifically for Inkwell's web site.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

## 6.6 Implementing social activities in your portlets

We wrote a portlet for Inkwell in [Chapters 5 and 6](#) that allows their customers to register products that they purchase on their web site. This portlet is a perfect candidate for social networking, because the act of registering a product can be made easily into a social networking activity. So in a nutshell, here's what we'll need to do:

- Modify our service layer so that it adds an activity when it adds a registration
- Create a `SocialActivityInterpreter` in our portlet project to help the Activities portlet display our custom activity

Cosmetically, this won't change the Product Registration portlet at all. But if Inkwell wants to use the social networking features of Liferay for their web site, they'll find that whenever a user registers a new product, the Activities portlet (which could be placed on users' profile pages) will display something like figure 8.x.

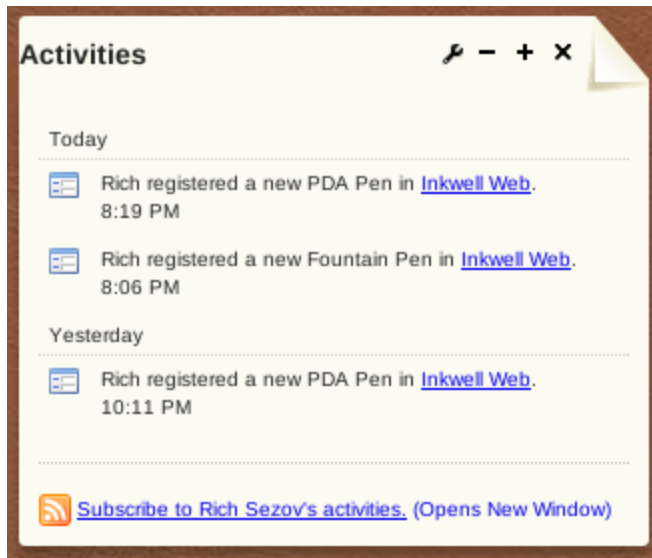


Figure 8.x The Activities portlet can show any number of activities that are possible in your web site. If you've written some games portlets, activities in those games can be published here. Users can even subscribe to an RSS feed of individual users' activities and read them offline.

Notice also in the figure above that the Activities portlet displays the icon for the portlet that published the activity. What we can see above is telling us that yes, it's important to customize the portlet icon, and that maybe we should get our designers on that task, pronto.

### Note:

There's a nasty bug in Liferay 6.0.5 CE and Liferay 6.0 EE that prevents messages from a plugin's `Language.properties` from appearing in the Activities portlet. Instead, what will appear is the

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>



key. To get around this bug, place your key/value pairs in a file called `Language-ext.properties` in your Liferay install's `WEB-INF/classes/content` folder. This bug is fixed in later versions of the product.

The code I'll show below gives you the pattern that Liferay uses for their own portlets that publish social activities. I'm telling you this because it may seem overly modularized for our simple use case, but the benefit is you'll be able to immediately pick apart what Liferay is doing when examining their code, *and* you'll have a great model for more complicated portlets that publish lots of activities, such as those games I keep mentioning.

### 6.6.1 Adding an activity in the service layer

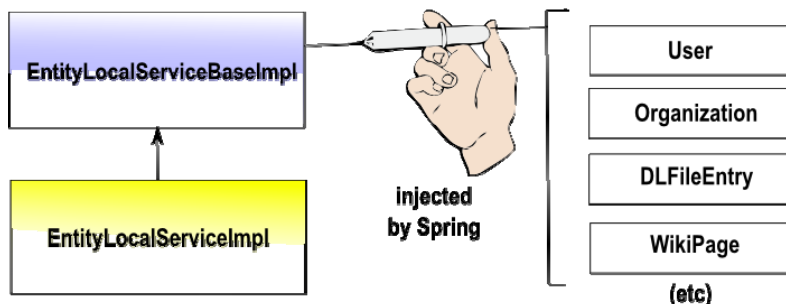
The first step is to add the activity when you're saving your entity, in a similar fashion to the way you added a resource in order to use Liferay's permissions system. We want, of course, to do this in as efficient a manner as possible, which means we have to think about the transaction, since we're introducing another database call in order to do this. Service Builder makes it easy to wrap all your database calls in a single transaction. We already took advantage of this when we made the call to add the permissions resources to the database at the same time we added our entities, but the configuration for that transaction was built in to Service Builder. This time, we'll have to configure it ourselves. Thankfully, this is really easy to do—it's only two lines of code in `service.xml`:

```
<entity name="PRRegistration" local-service="true" remote-service="false">
  .
  .
  .
  <reference package-path="com.liferay.portlet.social"
    entity="SocialActivity" />

  <reference package-path="com.liferay.portal" entity="User" />
</entity>
```

You can reference any package which contains Service Builder entities, and Service Builder takes care of all the Spring / Hibernate configuration file wiring for you. What happens under the hood is the same thing that happens with resources and the counter, except this time you are defining which entities get injected. We do this through *references*.

By defining a reference in your entity, you'll cause Service Builder to generate a Spring configuration that injects an implementation of the entity you have referenced. In plain terms, this means that there will be an instance variable of the referenced service in your implementation class.



©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>



Figure 8.x. You can inject any Service Builder service into your service simply by providing a reference in `service.xml`. The injection occurs in the base class, which your implementation class inherits, so you don't have to touch anything in your code to make this happen.

Why do we do this? Wouldn't it be easier to simply use the generated static `-Util` class that Liferay provides for its services? If you were working up in your portlet layer, the answer would be yes. Since you're down in the service layer, the answer is no. The reason this is the case has to do with transactions.

Pretend that you (or if this is too painful, a friend) have purchased a really cheap hosting plan for your MySQL database. You have connected Liferay, running on a different system, to this database. Unbeknown to you, the reason the database hosting service is so cheap is that the servers are comprised of racks of white box machines running in somebody's basement. There's no backup battery, generator, or even surge protection. To top it off, even the network connection is provided by a residential cable service. If you want your data backed up, well, back it up yourself.

Obviously, this situation is untenable, and needs to be set right. No one should be able to get away with taking advantage of people in order to make huge profits on such a badly implemented service. Additionally, I'm sure the cable company wouldn't take an especially kind view to this sort of thing. So because I believe in justice (and I'm the one making up this story), say a thunderstorm to end all thunderstorms comes, and in a single, blazing flash of light and high-decibel boom, it puts this particularly nefarious scheme to a fiery end.

Unfortunately, at the same time this was happening, your Liferay-based web site was chugging along, saving data to the database in multiple tables. If you called one Liferay service from within another service using the regular `-Util` class, each call is a separate transaction. This means that you could likely have a situation where entities that should be saved together are not, especially in the event of a service interruption. If, however, you created a reference from one service to the other and you used the injected service instead of the static `-Util`, the whole operation becomes a single transaction which can be rolled back if the service or network gets disrupted. Figure 8.x shows how this works.

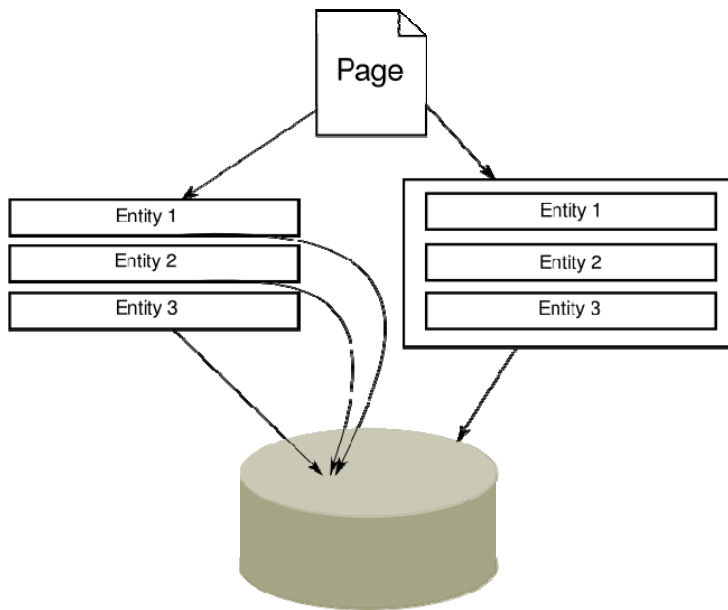


Figure 8.x If data coming from the same web page gets saved to the database as three different entities, it is best to group the entire operation in one transaction. That way, if there is an interruption in service, you don't wind up with corrupted or orphaned data in your database.

Of course, if your whole data center is disintegrated in a blinding explosion, this isn't going to help you very much. So we'll say that in the explosion the hard drive containing your (or your friend's) database is miraculously blown clear out of the "data center," gets tangled up in a bunch of helium balloons which were accidentally released at a child's birthday party, and lands gently on your porch. When you pick up that hard drive and access its data, if you have used references in Service Builder, you will find that no part of the transaction that was in the process of being saved was committed, and your database is clean. If you used the `-Util` class directly, you'll find that you now have orphans in your tables.

In any case, I highly recommend that you (or your friend) back up that data immediately and find a better database hosting service.

Now that we have injected both the `SocialActivity` service and the `User` service into our `PRRegistration` service, we can use them to make a call to add the activity when a registration is added to the database. We'll put this code into our `PRRegistrationLocalServiceImpl` class.

### Listing 8.2 Adding social activities

```
public PRRegistration addRegistration(PRRegistration reg)
    throws SystemException, PortalException {

    PRRegistration registration =

    prRegistrationPersistence.create(counterLocalService.increment(PRRegistration.class.get
    etName()));
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

```

registration.setCompanyId(reg.getCompanyId());
registration.setDatePurchased(reg.getDatePurchased());
registration.setGroupId(reg.getGroupId());
registration.setHowHear(reg.getHowHear());
registration.setProductId(reg.getProductId());
registration.setPrUserId(reg.getPrUserId());
registration.setSerialNumber(reg.getSerialNumber());
registration.setWherePurchased(reg.getWherePurchased());

resourceLocalService.addResources(
    registration.getCompanyId(), registration.getGroupId(),
    PRRegistration.class.getName(), false);

PRUser prUser = prUserLocalService.getPRUser(registration.getPrUserId());
User user = userLocalService.getUser(prUser.getUserId());      1

if (user != null) {

    socialActivityLocalService.addActivity(
        user.getUserId(), reg.getGroupId(), PRRegistration.class.getName(),
registration.getPrimaryKey(),
        ProductActivityKeys.ADD_REGISTRATION, StringPool.BLANK, 0);    2
    }

    return prRegistrationPersistence.update(registration, false);
}

```

#### **1 Get User from PRUser**

#### **2 Add social activity**

Adding a social activity obviously requires a user ID. And since we coded our Product Registration Portlet to be able to accept registrations from anybody—whether they were logged in or not—we have to check to see if we have a Liferay user ID mapped to the user information we saved with the registration. We can only add an activity if we have a mapped portal user who was logged in saving the registration.

You'll probably also notice that while we're adding an activity, we make a call to another class called `ProductActivityKeys`. This class holds nothing but the list of possible activities for this portlet. It's a bit overkill for our portlet since it has only one type of activity, but I wanted to give you an example of the pattern Liferay uses for this. All this line of code does is add an entry in Liferay's `SocialActivity` table for this activity, for this company, for this group, for the logged in user, and for the class we have specified.

It's obvious that if we've added a call to add the activity, we need to add a call to delete the activity as well, so I won't bore you with the details of that. Next, we'll look at how to make the Activities portlet interpret our activity correctly so that it displays what we want.

### **6.6.2 Giving the activities portlet an interpretation of a custom activity**

The Activities portlet is kind of a strange bird. When it's placed on a page, it looks around to try to figure out where it is, and then it displays activities based on where it finds itself. For this reason, we need to handle one of two conditions: whether the activities portlet is in the same community / organization in which the activity occurred, or whether it's in a different one.

For example, many Liferay social networking implementations place the Activities portlet on users' profile pages. If a user performs an activity in another community, the Activities portlet will display a message like:

Some Dude did this thing in Community X.

The words "Community X" will be a link over to that community. This lets anyone browsing a social relation's profile hit the public pages of that community in order to look at the content and maybe even decide whether he or she wants to join that community.

If the Activities portlet is in the same community in which the activity occurred, it'll display a message like:

Some Dude did this thing.

For this reason, we're going to have to create not one, but two keys in our `Language.properties` file for our activity message:

```
activity-product-registration-add-registration-in={0} registered a new {1} in {2}.
activity-product-registration-add-registration={0} registered a new {1}.
```

As you can see, we've got an "in" property and a (for lack of a better term) "not in" property, complete with some tokens which represent what will get placed there.

The next thing we need to do is create our interpreter class.

### Listing 8.3 Interpreting activities

```
public class RegistrationActivityInterpreter
    extends BaseSocialActivityInterpreter {

    public String[] getClassNames() {                                1
        return _CLASS_NAMES;
    }

    @Override
    protected SocialActivityFeedEntry doInterpret(
        SocialActivity activity, ThemeDisplay themeDisplay)
        throws Exception {                                          2

        PermissionChecker permissionChecker =
            themeDisplay.getPermissionChecker();

        PRRegistration registration =
            PRRegistrationLocalServiceUtil.getPRRegistration(activity.getClassPK());

        if (!permissionChecker.hasPermission(
            registration.getGroupId(), PRRegistration.class.getName(),
            registration.getPrimaryKey(), ActionKeys.VIEW)) {

            return null;                                            3
        }

        String link = StringPool.BLANK;                             4

        String key = "activity-product-registration-add-registration";
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

```

        PRProduct product =
PRProductLocalServiceUtil.getPRProduct(registration.getProductId());

        String title = getTitle (activity, key, product.getProductName(), link,
themeDisplay);

        String body = StringPool.BLANK;

        return new SocialActivityFeedEntry(link, title, body);
    }

    protected String getTitle(
        SocialActivity activity, String key, String content, String link,
        ThemeDisplay themeDisplay) {
        String userName = getUserNam(activity.getUserId(), themeDisplay);

        String text = HtmlUtil.escape(cleanContent(content));

        if (Validator.isNotNull(link)) {
            text = wrapLink(link, text);
        }

        String groupName = StringPool.BLANK;

        if (activity.getGroupId() != themeDisplay.getScopeGroupId()) {
            groupName = getGroupName(activity.getGroupId(), themeDisplay);
        }

        String pattern = key;

        if (Validator.isNotNull(groupName)) {
            pattern += "-in";
        }

        return themeDisplay.translate(
            pattern, new Object[] {userName, text, groupName});
    }

    private static final String[] _CLASS_NAMES = new String[] {
        PRRegistration.class.getName()
    };
}

```

**1 Entities to interpret**

**2 Overridden from parent class**

**3 Return null if private**

**4 Activities have link, key, title, body**

**5 i18n of activity**

Every `SocialActivityInterpreter` needs a #1: list of class names for which this interpreter is responsible. This way, you can use `doInterpret()` to check for class names with an `if` statement and then have separate methods for interpreting each type of activity. Since we only have one class we're interpreting, we can use `doInterpret()` directly. #2 shows that we're overriding from a parent base class, rather than implementing the interface directly. This gives us access to some helpful methods (used in `getTitle()`) that we otherwise wouldn't have. Next, #3 checks to make sure

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

that the current user has permission to view the entity for which an activity has been published. If you can't view the entity, you really shouldn't be able to view the activity either.

#4 has the contents of the activity. Activities have various properties: a link to the activity, the key from `Language.properties` that should be used, and a title and body which can be displayed. Since this is a registration, we're not using a link (which would provide a link to that user's personal registration information) or a body, but as you can see, they're handled similarly to the title—and I've put some code in `getTitle()` that would handle a link if you had one. If you're interested, you can take a look at how Liferay's message boards portlet publishes an activity with a link to the relevant message boards post. You'll need to know how to do `FriendlyURLs`, which we'll be introducing in Chapter 12.

Finally, #5 is the `getTitle()` method, which uses Liferay's internationalization API to pick the proper translation of the key and populate it with the data from the activity. Notice the code which checks to see if the community in the activity matches the community the user is currently browsing. If they don't match, we pull the "in" key, and then make the call to translate it. The translator wants the pattern (the value from the key) and the values to insert in place of the tokens. We pass these values in as the name of the user, the name of the product (which could have been displayed as a link), and the community where this activity took place. Our `doInterpret()` method then returns a fully interpreted `SocialActivityFeedEntry` that the Activities portlet can display.

That's all there is to publishing an activity. As you can see, activities are very simple, but very powerful. They give developers a common interface to publish to their social relations what users are doing across the entire portal installation. You can use them to great effect to provide a dynamic experience for your users.

## 6.7 Summary

Liferay Portal is a great platform for social web sites. Not only does it integrate nicely with other social sites such as Facebook, it also provides a comprehensive API of its own which allows you to build your own social web site. This API provides everything you need to manage different types of social relationships, query for those relationships, allow sharing of information between users who have relationships, and more.

You can also integrate your own Liferay portlet applications with the social API to provide your users with a running list of all the activities their friends are performing on your site. Whether your site contains games, collaboration, or any other type of application, you can publish that application's activities so all your users' networks can see them. Liferay Portal gives you an entire social platform upon which to build your site.

# 8

## *Hooks*

This chapter covers

- Hooks, and why they're necessary
- Hooking into Liferay properties
- Hooking into Liferay JSPs
- Hooking into services
- Customizing a Liferay portlet with a hook

Open source is making great headway in the business world today, but some organizations are still wary of it, and they have their reasons. One of the reasons is the maintenance problems you run into if you customize open source software with stuff that's meant for your own organization, but is not really applicable to the project as a whole. Now you've got an issue: you have to maintain this customization through every new release of the open source software yourself. This means you probably have to understand that open source project at a much deeper level than you may have been prepared for, and you need to be ready at each release to keep track of where your customizations go in order to reintroduce them into the base project.

Let's take a concrete example, and we'll keep it simple. Say there's an open source image gallery application that you really like and want to use, but you have to integrate it with some custom software written by your development team. The custom software is responsible for keeping track of employee ID cards, and is the gatekeeper for other systems. When a user is created in the custom system, it gets propagated out to other systems—including the image gallery—and the image gallery can be used to browse not only employee mug shots for the ID cards, but also galleries that employees create. To integrate the image gallery with this system, you customized the code in the image gallery's security module so it will create a user ID in the image gallery for every user who gets an ID card in the other system.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

Now say that some security problem has been found with the image gallery, and the developers of the open source project produce another release to fix the issue. Of course, since the problem was with the security module, you now have to figure out how to integrate your custom code with the new code—and it may not be that easy. It's possible that the fix for the security flaw impacts the customization you made. Now you have to figure out a way to code around the new security implementation, or continue using software which has a security flaw.

Liferay hooks are designed to solve exactly this problem. With a hook, you can deploy your customization without touching Liferay's source code at all. This provides a level of separation between your customizations and the Liferay core, giving you freedom to upgrade whenever you want. You won't have to worry about trying to maintain customized versions of any of Liferay's source files and then trying to integrate those customizations back into an upgraded version of Liferay. Instead, you can take advantage of the fact that Liferay is built to be easily customized.

Liferay Hooks are the newest type of plugin which Liferay Portal supports. They were introduced late in the development cycle for Liferay Portal 5.1.x, and are now the preferred way to customize Liferay. As with portlets, layout templates, and themes, they are created using the Plugins SDK, and are used for multiple scenarios. Let's take a closer look at hooks so we can see what they're useful for, and then we'll jump in and start customizing Liferay.

## **8.1 What is a hook?**

So what are hooks? Aptly named, they are pieces of code which are designed to “hook” into Liferay and take over certain pieces of functionality. They allow you as a developer to override parts of core Liferay with your own implementation.

Liferay Portal has had functionality like this for a long time. The Ext plugin (formerly the Ext environment) was designed for exactly the same use case: overriding and customizing Liferay itself. So the first question usually asked when experienced Liferay developers are presented with hooks is why? Why did we need another way to customize Liferay? Isn't the Ext plugin enough? Are you purposefully trying to confuse me?!?

### **8.1.1 An easier customization paradigm**

The Ext plugin was always positioned as an easier way to customize Liferay than modifying the Liferay source code itself, and this is true. It does keep your custom code separate from the portal, so there is a clear delineation as to where Liferay's code leaves off and your code picks up. Customers have used the Ext environment to do many amazing (and unexpected) things with Liferay. This, however, doesn't come without a cost in complexity.

Because the Ext plugin gives the developer complete access to the internals of Liferay Portal, custom code becomes tightly coupled to particular implementations of functionality within Liferay, and these implementations often change from version to version—or even point release to point release. This has caused developers unnecessary headaches, as whenever upgrade time came around, very often they would encounter a lot of changes within the Liferay code base: method signature changes, package refactoring, class renaming, and so on.

Additionally, Ext can be hard to work with, particularly for a team of developers. It is a large, monolithic environment that cannot be deployed piecemeal and is not easily divided up into subprojects.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>



So if multiple developers are working on different pieces of functionality, everybody has to wait until all pieces are relatively stable before doing a deploy to test.

To relieve the pressure on developers, a different paradigm was needed, a paradigm which allowed for customization against a stable API that was guaranteed not to change between versions—and if it needed to change, functionality would be deprecated appropriately so that developers would have a chance to update their code on a schedule that was more conducive to their projects. The new paradigm also needed to be smaller and more nimble, with the ability to cobble together multiple deployable assets to build out the whole feature set. Hence, hook plugins were born.

### **8.1.2 Hook basics**

Hook plugins are hot deployable, just like portlets, layout templates, and themes, so you can add them to and remove them from your portal at will. You can divide functionality up into multiple hooks written by multiple developers, allowing for a more dynamic development environment. And hooks are written to Liferay's public API, which is properly deprecated when it changes. As you can see, hooks were designed to overcome many of the limitations of using Ext.

Even though hook plugins have all these advantages over Ext, the Ext plugin also has some advantages over hooks. Using Ext, you can customize anything in Liferay, because you are working in the same class loader as the portal. Hooks are in the plugin class loader, so it does not have access to all those core classes and thus, the Liferay engineering team has had to choose the extension points which are available. In other words, you cannot customize as much with hook plugins as you can with the Ext plugin; however, the list of what you *can* customize grows with each release of the portal as Liferay receives feedback from users.

So how do you go about choosing which to use? Simple: use hooks wherever you can. They are much easier to write, to deploy and undeploy, and to maintain. If you run into something which cannot be customized with a hook, then use the Ext plugin.

### **8.1.3 Creating a Hook**

You can create a hook in exactly the same manner in which you create portlets or themes. Go to the *hooks* folder in your Plugins SDK, and in LUM, type:

```
./create.sh my-hook "My Hook"
```

In Windows, it would be:

```
create my-hook "My Hook"
```

Easy, right? And wonderfully consistent.

I must mention one caveat now. Usually the next step for a developer is to import the project into an IDE, and I do not want to discourage you from doing this. I have, however, never met an IDE that likes customizing Liferay's JSPs in a hook. Why? Well, because a hook is one project, right? And the portal (if you have the source) is another project. So if you have a JSP with dependencies from an included JSP in another project (as Liferay does often with its pattern of using `init.jsp` for imports), the IDE doesn't know where to find the included file. What you wind up with is a file full of red Xs and little squiggles underneath all of the objects that the IDE doesn't understand.

For that reason, when writing hooks, I just use a text editor (jEdit is my current favorite all-purpose text editor). It's much simpler that way.

Now that we have a hook project, let's take a look at what hooks can do.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

## 8.2 Types of Hooks

Hooks are designed to customize four main features within the portal:

- portal properties
- language properties
- JSP files
- services

In order to do this, you will have to create a configuration file called `liferay-hook.xml` and place this file in the *WEB-INF* folder of your hook project in your Plugins SDK. The following code creates a skeleton of `liferay-hook.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hook PUBLIC "-//Liferay//DTD Hook 6.0.0//EN"
"http://www.liferay.com/dtd/liferay-hook_6_0_0.dtd">

<hook>
</hook>
```

Once you've got this file skeleton set up, you're ready to tell Liferay what it is that you want to customize. We'll start with the simplest customization.

### 8.2.1 Customizing Portal Properties

Hooks can override portal properties in the same way in which you override them via the `portal-ext.properties` file. Not all properties can be overridden, but the properties which can are listed in the DTD for the `liferay-hook.xml` file for the version of Liferay you are running. You will find this DTD in the *definitions* folder of the portal source for your version of Liferay.

Let's take a very simple property and customize it. As you probably know, the first time someone logs into Liferay Portal, he or she is presented with a "terms of use" page. The user must agree to the terms of use in order to continue. This feature is controlled by a property in Liferay's default `portal.properties` file that looks like this:

```
terms.of.use.required=true
```

We can configure a hook to turn this feature off by changing the property from true to false. This is extremely easy to do and requires no code. First, we configure our hook to override properties by editing our `liferay-hook.xml` to look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hook PUBLIC "-//Liferay//DTD Hook 6.0.0//EN"
"http://www.liferay.com/dtd/liferay-hook_6_0_0.dtd">

<hook>
  <portal-properties>portal.properties</portal-properties>
</hook>
```

As you can see, we have defined a file called `portal.properties` in the hook where our overridden properties will be placed. This file must be on the classpath of the project, so the best place to put it is in the *src* folder. All we need to do to override the property is put the key / value pair we want to override in the file:

```
terms.of.use.required=false
```

Save and close the file and deploy your hook in the same manner in which you deploy other plugins:  
`ant deploy`

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

Your hook will be deployed and this property will now be overridden—dynamically. New users will not need to accept the terms of service, and you won't need to restart your portal for this to take effect. By the same token, if you wanted to revert this property back to the default behavior, all you would need to do is undeploy the hook, and the portal will immediately revert, without the need for a restart.

See? Hooks are easy. Let's now try something a little more complex.

### 8.2.2 Customizing Portal Event Properties

Some properties within Liferay are not meant to have a single value. Instead, they comprised a list of multiple values. In this case, any values you specify in your hook are appended to the list. For example, you can provide a value in your hook project that appends a value to the `application.startup.events` property. That property defines a list of classes which override one of the Action classes in `com.liferay.portal.kernel.action`, allowing you to fire your own event which will run when the portal starts.

[LW: Diagram of how this works?]

You can then implement this event by adding a package and a class which extends a Liferay Action in the `src` folder of your hook project and then deploy it to Liferay. Your action will then be appended to the list of actions in `application.startup.events`, and will fire when Liferay Portal starts. This enables you to create these kinds of events which were once only possible to do using the Ext plugin.

Any of these events are defined as Liferay actions, as stated above. When you extend an Action class, you put your functionality in a `run()` method which you must override. Here's a very simple example of how to define the class:

```
package com.liferay.test.hook.events;

import com.liferay.portal.kernel.events.ActionException;
import com.liferay.portal.kernel.events.SimpleAction;

public class StartupAction extends SimpleAction {

    public void run(String[] ids) throws ActionException {           #1
        System.out.println("### StartupAction");
    }

}
```

#1 company ID ...

The String array of IDs (#1) are Company IDs. This value is populated when the event is fired by the portal, and is not something you will need to populate yourself. Events get fired once for each portal instance that's been defined, which is tracked by Company ID.

Beyond event properties, there are other kinds of properties which you can customize.

### 8.2.3 Customizing Listener Properties

Hooks also support overriding the `value.object.listener.*` properties. This is a very powerful feature of hooks, allowing you to add your own custom listeners for any model class in Liferay.

For example, say you wanted to trigger the sending of an email whenever a new blog entry is created. You would first, as you have already done above, define the `portal.properties` file in `liferay-hook.xml` so that you can override the appropriate value object property for the class to

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

which you want to attach a listener. Then you simply define your listener on the property in the `portal.properties` file:

```
value.object.listener.com.liferay.portlet.blogs.model.BlogsEntry= \
    com.inkwell.liferay.portlet.blogs.NewBlogEntryListener
```

Once you've defined the listener, all you need to do is implement it. All model listeners must implement the `com.liferay.portal.model.ModelListener` interface. So you will find that you have a class that contains the following methods:

```
public void onAfterCreate(BaseModel arg0) throws ModelListenerException {
    BlogsEntry entry = (BlogsEntry)arg0;

    /* Code for sending an email goes here */
}

public void onAfterRemove(BaseModel arg0) throws ModelListenerException {
}

public void onAfterUpdate(BaseModel arg0) throws ModelListenerException {
}

public void onBeforeCreate(BaseModel arg0) throws ModelListenerException {
}

public void onBeforeRemove(BaseModel arg0) throws ModelListenerException {
}

public void onBeforeUpdate(BaseModel arg0) throws ModelListenerException {
}
}
```

You can see that there are a lot of events in the listener where you can add some custom code. There's one other type of property that I want to make sure you know is available.

## 8.2.4 Customizing Language Properties

In a similar fashion to portal properties, you can override language resource bundles by using a hook. The syntax in `liferay-hook.xml` is identical:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hook PUBLIC "-//Liferay//DTD Hook 6.0.0//EN"
"http://www.liferay.com/dtd/liferay-hook_6_0_0.dtd">

<hook>
    <language-properties>content/Language_en.properties</language-properties>
</hook>
```

Note that you can add as many language bundles as you like in order to override as many different languages as you like. Your changes will overlay the values from the portal, meaning that anything you override gets overridden, but you don't have to override everything. For example, say for whatever reason you don't like the word "Save." This word is used in the language bundles throughout Liferay for saving message board posts, blog posts, documents, and pretty much everything else. If you wanted to change this word to "Store," all you would have to do is define a language file as above and then change the value for that key. So your language file would have this in it:

```
save=Store
```

Deploy your hook, and everywhere Liferay uses the value of the `save` key, it will display the word "Store" instead of "Save."

Let's go beyond properties and look at some more interesting things we can customize.

## 8.2.5 Customizing JSP Files

Hooks allow you to replace any of Liferay's JSP files with your own implementation. Again, this was once only possible to do with Ext. What this enables you to do is modify the functionality that exists within the JSP files of Liferay's core portlets. If, for example, you don't like the way Liferay's Document Library presents itself, you can modify its JSPs with a hook and include styling in your theme to go along with the modifications. And again, if you undeploy your hook, Liferay will revert back to its default behavior without the need to restart.

Let us take a look at how we would configure our hook for custom JSPs. All we need to do is tell Liferay in our `liferay-hook.xml` file where the custom JSPs are:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hook PUBLIC "-//Liferay//DTD Hook 6.0.0//EN"
"http://www.liferay.com/dtd/liferay-hook_6_0_0.dtd">

<hook>

    <custom-jsp-dir>/META-INF/custom_jsps</custom-jsp-dir>

</hook>
```

Under the `custom_jsps` folder, we can create the same folder structure Liferay Portal uses to store its JSP files. So if we wanted to customize the `view.jsp` file for the Blogs portlet, we would create it here:

```
custom_jsps/html/portlet/blogs/view.jsp
```

When you deploy a hook that modifies JSPs, behind the scenes Liferay renames the original JSP file from `[filename].jsp` to `[filename.portal].jsp`. So the original file is always still there and still accessible. If you are good at string manipulation, this allows you to make your modifications to the page in a way that is easier to maintain than by simply providing a new implementation. Why is it potentially easier to maintain this way? Well, it depends on what you need to change.

If you need to make a major change to the markup or logic of the JSP, you will want to go ahead and reimplement the JSP. If you only need to change a header, a field title, or something minor like that (perhaps in conjunction with a change to the language properties), you can do string manipulation on the original JSP and simply replace the Liferay label with yours. This protects you from Liferay's upstream changes to the JSP. It is not likely that headers and labels will change much. It is much more likely that logic in the JSP will be modified in order to take into account additional bean fields or other business logic. If you've simply done string manipulation to replace a default label with yours, you can preserve Liferay's logic—however it's implemented—and only update the label. Here is a simple example of how you might do that:

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

<h6>Some Special Header</h6>

<liferay-util:buffer var="html">
    <liferay-util:include page="/html/portlet/blogs/view.portal.jsp" />
</liferay-util:buffer>

<%
html = StringUtil.replace(html, "hello", "hola");
%>

<%= html %>
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

Using the `<liferay-util:buffer />` tag, the entire contents of the original Liferay JSP are placed in a string called `html`. Then there is logic which replaces any instance of the character string "hello" with another value. The result of this is then written out to the page. Obviously, this can get unwieldy if there are a lot of changes that you need to make, but it is a strategy that has been used successfully.

Now let's look at customizing some really interesting stuff: Liferay services.

### 8.2.6 Customizing Services

Remember how we generated our own services for our Product Registration portlet in Chapter 5? All of Liferay's services are generated in exactly the same way, and you can override them using hooks. You won't do that in a hook by using Service Builder itself; instead, you will use the decorator pattern to add the functionality you need to the service, while leaving the rest of the functionality alone. This is best done for read-only attributes or attributes that are stored in a separate system, such as LDAP.

Liferay provides a wrapper class for all generated services. This class is aptly named `[Model]LocalServiceWrapper.java`. Since Liferay's Service Builder utility uses Spring's dependency injection for much of its functionality, the wrapper class is a convenient place in which to inject an actual implementation of the interface. Liferay uses `[Model]LocalServiceImpl` as this implementation: you created these in Chapter 5 when you implemented the service layer of your portlet.

There is actually a chain of injections which allow you to customize things via the wrapper class: the `-Impl` class is injected into the wrapper, and the `-Wrapper` class is injected into the `-Util` class. Figure 10.1 shows this more clearly, I think, than describing it does (a picture is worth a thousand words, after all). Of course the `-Util` class is the one that end users call when they want to access the service. This means that you have an opportunity to *decorate* the original implementation with your changes by extending the wrapper class. When the `-Util` class is called, your decoration then becomes the implementation (inheriting functionality from its parent) that has been injected into the `-Util` class.

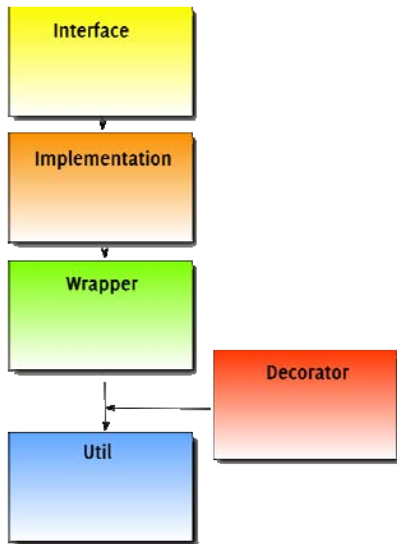


Figure 8.1 This shows how the class inheritance works within Service Builder, and how the decorator pattern is used to enable you to customize Liferay's services.

Keeping all of this in mind, we now have the tools to implement an override of certain functionality in a service. As with the other features of hooks, the first thing we need to do is define it in our `liferay-hooks.xml` file:

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hook PUBLIC "-//Liferay//DTD Hook 5.2.0//EN"
"http://www.liferay.com/dtd/liferay-hook_5_2_0.dtd">

<hook>
  <service>
    <service-type>
      com.liferay.portal.service.UserLocalService
    </service-type>

    <service-impl>
      com.liferay.test.hook.service.impl.MyUserLocalServiceImpl
    </service-impl>
  </service>
</hook>

```

We first define the service we want to override and then the implementation class which will contain the new logic we are providing. All that is left, then, is to write the class.

Let's show a very simple example. We'll say we want to add an attribute to the User class called `FavoriteColor`. So let's create that class in the same package in which it appears in Liferay:

```

package com.liferay.test.hook.model.impl;

import com.liferay.portal.model.User;
import com.liferay.portal.model.UserWrapper;

public class MyUserImpl extends UserWrapper {

```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

```

    public MyUserImpl(User user) {
        super(user);
    }

    public String getFavoriteColor() {
        return "My favorite color is green.";
    }
}

```

Now we'll decorate the wrapper class for the service and make sure that it returns the new implementation of User instead of the default one:

```

public class MyUserLocalServiceImpl extends UserLocalServiceWrapper {

    public MyUserLocalServiceImpl(UserLocalService userLocalService) {
        super(userLocalService);
    }

    public User getUserById(long userId)
        throws PortalException, SystemException {

        System.out.println("## getUserById " + userId);

        User user = super.getUserById(userId);

        return new MyUserImpl(user);
    }
}

```

Since the `MyUserImpl` class is the one being returned instead of the `User` class, it will have the extra attribute in it. Of course, this was a very simple example that returned a static value. Your `getUserById()` method could have created a new `MyUserImpl` object and then queried LDAP, a custom service, or some other system to populate the custom attribute before returning the object.

Speaking of a custom service, you can create services within hooks using Service Builder. This is done in the same exact manner as you would do it in a portlet. If, however, you wanted to use the service you create in a hook in a separate plugin, you will have to copy the .jar that is generated in your hook's `WEB-INF/lib` folder to the plugin in order to make the classes available to it. This is the same procedure you would use if you had generated the service in your portlet. Alternatively, the .jar files for shared services can be copied to your application server's global classpath, and then the services will be available to all plugins.

Now that you have the necessary background on what is possible with hooks, let's take a concrete example and see what the Inkwell development team did with a hook.

### ***8.3 Hooks in action: Inkwell's shopping cart***

Inkwell, like other manufacturers of electronic equipment, sells its products through brick-and-mortar retailers, online retailers, and its own web site. One of the reasons Inkwell chose Liferay was for its built-in functionality, and this will enable the Company to replace its aging PHP-based shopping cart with Liferay's.

The only problem is that Liferay's shopping cart looks like the rest of Liferay's portlets: it uses the Search Container to display categories of products in a table. While this is a good, generic implementation, it is not the way Inkwell wants to display its product categories. Since Inkwell doesn't

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>



have a lot of categories of products, they would like to display the categories graphically. It is hoped that presenting them in this way will make it easier for users to find what they are looking for and will ultimately lead to more orders from the web site.

The design team came up with a mockup for what this would look like, which you can see in figure 8.2.

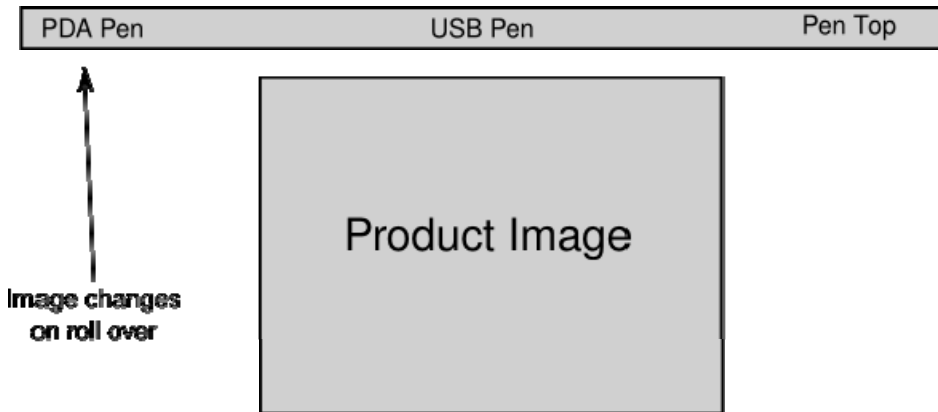


Figure 8.2 The Shopping interface will be replaced for end users with a menu that has the three product categories of Inkwell's products. As users roll over the links, the product image changes to a picture of that product in the image area.

A hook is a very good way to implement this. The development team has identified three areas where they can hook into Liferay's functionality and provide their own customization:

- Create their own version of `categories.jspf`, which is the page that displays the categories and items in the Shopping Cart
- Add a field to the `edit_category.jsp` file, which contains the form users access to add categories. This field will contain a URL to a picture in Liferay's Image Gallery portlet. This picture will be used for the category pictures.
- Add a very small entity called `ShoppingCategoryImage`, using Service Builder. This entity will have three fields: a primary key, a foreign key relationship to the shopping category key, and a field to hold the Image Gallery URL.

We'll do these in the same order in which we built our Product Registration portlet: we'll start with the service layer and then move on to the front end.

### 8.3.1 Generating a service layer in a hook

Let's take a look at the back end code first. We'll define our entity using Service Builder. Our `service.xml` file looks like this:

#### Listing 8.x

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

<!DOCTYPE service-builder PUBLIC "-//Liferay//DTD Service Builder 6.0.0//EN"
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

```

"http://www.liferay.com/dtd/liferay-service-builder_6_0_0.dtd">
<service-builder package-path="com.inkwell.internet.shopping.sb">

    <author>Rich Sezov</author>

    <namespace>Inkwell</namespace>

    <entity name="ShoppingCategoryImage" local-service="true" remote-service="false">

        <!-- PK Fields -->

        <column name="imageId" type="long" primary="true" />

        <!-- Foreign Key -->

        <column name="categoryId" type="long" />

        <!-- Other Fields -->

        <column name="imageUrl" type="String" />

        <!-- Order -->

        <order by="asc">
            <order-column name="imageUrl" />
        </order>

        <!-- Finder Methods -->

        <finder name="CategoryId" return-type="Collection">
            <finder-column name="categoryId" />
        </finder>

        <!-- Reference -->

        <reference package-path="com.liferay.portlet.expando" entity="ExpandoValue" />
        <reference package-path="com.liferay.portlet.expando" entity="ExpandoRow" />

    </entity>

</service-builder>

```

We've configured a small table with three columns, two of which are keys. We'll store the category ID from Liferay's shopping cart as a foreign key, and we'll store the URL to the image in our own field. This will require end users to upload their images to Liferay's Image Gallery first and then copy / paste the URL to the image into the field. Since there will be only three categories, this can be done once without much difficulty.

We're using references as we did in Chapter 6 to pull in Liferay's Expando services so we can use them in our transactions.

As before, when we generate the service, we are given a DTO layer with which to work, and we can add the methods we need to the layer. These methods are in listing 10.1.

### Listing 8.1 Shopping image service layer methods

```

public class ShoppingCategoryImageLocalServiceImpl
    extends ShoppingCategoryImageLocalServiceBaseImpl {

```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

```

public ShoppingCategoryImage addShoppingCategoryImage (
    long categoryId, String url) throws SystemException {
    A

        ShoppingCategoryImage image =
            shoppingCategoryImagePersistence.create(
                counterLocalService.increment(
                    ShoppingCategoryImage.class.getName()));

        image.setCategoryId(categoryId);
        image.setImageUrl(url);

        return shoppingCategoryImagePersistence.update(image, false);
    }

public ShoppingCategoryImage getShoppingCategoryImageByCategory (
    long categoryId) throws SystemException {
    B

        List<ShoppingCategoryImage> images =
            getShoppingCategoryImagesByCategory(categoryId);

        return images.get(0);
    }

public List<ShoppingCategoryImage> getShoppingCategoryImagesByCategory (
    long categoryId) throws SystemException {
    C

        List<ShoppingCategoryImage> images =
            shoppingCategoryImagePersistence.findByCategoryId(
                categoryId);

        return images;
    }

public ShoppingCategoryImage updateImage(ShoppingCategoryImage image)
    throws SystemException {
    D

        image = shoppingCategoryImagePersistence.update(image);

        return image;
    }

public void deleteImage(long imageId)
    throws NoSuchShoppingCategoryImageException, SystemException {
    E

        shoppingCategoryImagePersistence.remove(imageId);
    }

public void deleteImages (long categoryId) throws SystemException {
    F
        List<ShoppingCategoryImage> images =
            getShoppingCategoryImagesByCategory(categoryId);

        if (images.size() > 0) {
            for (ShoppingCategoryImage image : images) {
                try {
                    deleteImage(image.getImageId());
                } catch (NoSuchShoppingCategoryImageException nsscie) {
                    throw new SystemException(nsscie);
                }
            }
        }
    }

```

```

    }
}

}

}

public ShoppingCategoryImage getEmptyImage() {
    ShoppingCategoryImage image = new ShoppingCategoryImageImpl();

    return image;
}
}

```

**A Adding an image**

**B Get an image by Shopping Category**

**C Get all images by Shopping Category**

**D Update an image**

**E Delete an image**

**F Delete all images by Shopping Category**

**G Gets image object for filling**

As usual, once you've created the methods you need in your `-Impl` class, run Service Builder again and the methods will get propagated out to the interface.

Now we need to configure the other parts of our hook.

### 8.3.2 The Configuration File

We now need to glue all the different parts of our hook together by creating a `liferay-hook.xml` file. Create this file in the `WEB-INF` folder of your project with the following contents:

#### Listing 8.x

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hook PUBLIC "-//Liferay//DTD Hook 6.0.0//EN"
"http://www.liferay.com/dtd/liferay-hook_6_0_0.dtd">

<hook>
  <portal-properties>portal.properties</portal-properties>
  <language-properties>
    content/Language_en.properties
  </language-properties>

  <custom-jsp-dir>META-INF/custom_jsps</custom-jsp-dir>

  <service>
    <service-type>
      com.liferay.portlet.shopping.service.ShoppingCategoryLocalService
    </service-type>

    <service-impl>
      com.inkwell.internet.shopping.custom.InkwellShoppingCategoryLocalServiceImpl
    </service-impl>

  </service>

</hook>

```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

Okay, what have we done here? First, we're telling Liferay that we're going to override some portal properties, and those modifications will be in the file `portal.properties` on the class path of this project (we'll put it in the `src` folder and the build script will move it to `WEB-INF/classes` when we build the project). The next thing we're telling Liferay is that we're going to have one or more language keys. This will be for our additional field that will store the URL to the shopping category image—we have to call that field something, via a label. We'll put that something in a language file so that it can be translated if we want to do that.

After this, we tell Liferay we're also going to customize some core JSPs, and we're going to put our customizations in `META-INF/custom_jsp`s. And finally, we tell Liferay that we're also customizing a service. We define what service we want to override and then we define the class—which must extend the wrapper of the overridden service—which will provide the implementation we want.

Now we can move on to overriding the service.

### 8.3.3 Overriding Liferay's service

Now comes the fun part. We need to actually override Liferay's service that deals with shopping categories in order to insert our functionality for adding and removing images. We will do this by extending the wrapper class that comes with Liferay. Since this class is in the `portal-service.jar` file, it is on the global class path of the server, so it's accessible from our hook plugin.

There are only three points at which we need to extend the class: adding categories, updating categories, and deleting categories. For adding categories, we will need to make it so that we create a `ShoppingCategoryImage` object after the category is created. These two objects will be linked by `categoryId`. For deleting categories, we do something similar, but in reverse: we first delete the image associated with the category, and then call the super class's method to delete the category itself. Since Liferay's `delete` methods are overloaded twice, we will need to implement both versions of the method, because we don't know what else in the portal calls them or which version is called.

But wait a minute. All this time we've been assuming something. We've been assuming that we can get a value from a field on a form all the way down into the service layer. To do that, we'll need access to the `HttpServletRequest` or the `PortletRequest` object, right? That's where all the form field values from the browser go. But we don't have access to that object in the service layer.

Uh oh. We're in trouble. Our design is flawed.

Or is it?

One thing I've learned while working with Liferay: it is well-designed. The core engineers have already thought of that and have provided a solution. After all, hooks were *designed* to customize Liferay core functionality, so all the tools for doing so are provided. We're going to use a service called the `ExpandoBridge`, which can take custom fields and put them into Liferay's `ServiceContext`. This is why we included services for this as references to our entity. We'll come back to this a bit later.

So the first thing we need to do is to create our extension, which you can see in listing 10.2.

#### Listing 8.2 Extending Liferay's shopping categories

```
package com.inkwell.internet.shopping.custom;                                     1

public class InkwellShoppingCategoryLocalServiceImpl
    extends ShoppingCategoryLocalServiceWrapper {
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

```

public InkwellShoppingCategoryLocalServiceImpl(
    ShoppingCategoryLocalService shoppingCategoryLocalService) {

    super(shoppingCategoryLocalService); 2

}

@Override
public ShoppingCategory addCategory(
    long userId, long parentCategoryId, String name,
    String description, ServiceContext serviceContext)
    throws PortalException, SystemException {

    String imageUrl = (String)
        serviceContext.getExpandoBridgeAttributes().get("image-url"); 3

    ShoppingCategory shoppingCategory = super.addCategory(
        userId, parentCategoryId, name, description, serviceContext);

    ShoppingCategoryImageLocalServiceUtil.addShoppingCategoryImage(
        shoppingCategory.getId(), imageUrl); 4

    return shoppingCategory;
}
}

```

- 1 Package outside Service Builder**
- 2 Constructor from superclass**
- 3 Get field value from ServiceContext**
- 4 Call service to add entity**

Make sure you create the class in a package that is not within the Service Builder hierarchy(#1), so that it gets packaged with your hook rather than with the persistence classes.

As you can see here, we have created a class which extends the `ShoppingCategoryLocalServiceWrapper` that ships with Liferay. We are also making sure that we call the constructor from the super class (#2) so that any referenced services are injected into the class when we need them. Once we have the class set up like this, we can override the methods we need in order to provide some extra functionality.

Next, we override Liferay's `addCategory()` method. To get the URL for the image in (#3), we call something out of Liferay's `ServiceContext` called an `ExpandoBridgeAttribute`. When we get to the customization of the JSP, we will use a Liferay tag to render our field so that Liferay treats it as a custom attribute. Liferay takes any fields defined as custom attributes and makes them available in the `ServiceContext`. This is how we get the field value from the form all the way down into the service layer.

So why the weird name? The first time I heard the term `Expando`, I was immediately transported back to my comic book reading days, and thought that `ExpandoMan` might be a good name for a super hero who could stretch his limbs like a rubber band. One of Liferay's core engineers has defined `Expandos` as meaning "to attach additional properties to an object."<sup>1</sup> Liferay has a set of tables which allows developers to do just that.

---

<sup>1</sup> <http://www.liferay.com/web/raymond.auge/blog/-/blogs/expandos-what-are-they-and-how-do-they-help-me-liferay-portal-5-0-1>

Because Java is not a dynamic language like JavaScript or Python, Expandos proper are difficult to implement. The `ExpandoBridge` class is a helper class which allows for the creation of `Expando` objects in Java. Additionally, Liferay allows you to persist those `Expandos` in a set of tables. For our hook, we're borrowing some functionality from the `Expandos` API. This functionality allows us to create an arbitrary attribute in the JSP and get it down to the service layer by embedding it in `ServiceContext`. You'll see how that is done when we get to the JSP. For now, just understand that the image URL, which is the attribute we are looking for, is in the `ServiceContext` object, so we have access to it. We could do some validation on the value we get, but for this example, we haven't done any validation.

After this, all we need to do is add the category, and we do that by calling `addCategory()` from the super class. This returns to us the category that was added, which makes it a simple matter to then add the `ShoppingCategoryImage` object, linking the two by `categoryId`. We then return the `ShoppingCategory` object as the super class would have.

The `updateCategory()` method in listing 8.3 is very similar.

### Listing 8.3 Updating a category

```
@Override
public ShoppingCategory updateCategory(
    long categoryId, long parentCategoryId, String name,
    String description, boolean mergeWithParentCategory,
    ServiceContext serviceContext) throws PortalException, SystemException {

    String imageUrl =
        (String) serviceContext.getExpandoBridgeAttributes().get("image-url");

    ShoppingCategoryImage image = ShoppingCategoryImageLocalServiceUtil.
        getShoppingCategoryImageByCategory(categoryId);           1

    if (image == null) {                                          2

        ShoppingCategoryImageLocalServiceUtil.addShoppingCategoryImage(
            categoryId, imageUrl);

    } else {

        image.setImageUrl(imageUrl);
        ShoppingCategoryImageLocalServiceUtil.updateImage(image);

    }

    ShoppingCategory shoppingCategory =
        super.updateCategory(
            categoryId, parentCategoryId, name, description,
            mergeWithParentCategory, serviceContext);

    return shoppingCategory;
}
#1
#2
```

The only different things we do here are to #1 get the `ShoppingCategoryImage` object, #2 set whatever value is in `ServiceContext` in the object, and then save both the image and the `ShoppingCategory` to the database.

For the delete function, the underlying class provides two implementations. One of these implementations calls for a `categoryId`, and the other implementation calls for a `ShoppingCategory` object. Since we don't know what else in Liferay calls these methods, we need to override both.

```
@Override
public void deleteCategory(long categoryId)
    throws PortalException, SystemException {

    ShoppingCategoryImageLocalServiceUtil.deleteImages(categoryId);

    super.deleteCategory(categoryId);
}

@Override
public void deleteCategory(ShoppingCategory category)
    throws PortalException, SystemException {

    this.deleteCategory(category.getCategoryId());
}
```

As you can see, there's really nothing special here either. We delete the image first and then call the super class to delete the category.

Since this is all we need to override, we're done with the service layer and can move on to the view layer.

### 8.3.4 Overriding the Shopping portlet's interface

Now that we have everything implemented in the back end, we need to implement the front end. Liferay's front end is implemented in JSP files. As we learned earlier in the chapter, if we want to override a core JSP with one of our own, we need to place it in exactly the same path as the one that ships with Liferay. Since we defined the location of our customized JSPs as `META-INF/custom_jsp`s in our `liferay-hook.xml` file, we will need to duplicate Liferay's path in this directory.

In looking at the Liferay source, it looks like we'll have to customize two JSPs:

- `categories.jspf`
- `edit_category.jsp`

Both of these files are found in the `html/portlet/shopping` folder, so to start, we'll simply copy Liferay's version of these files into:

```
META-INF/custom_jsp/html/portlet/shopping
```

Now we can begin customizing the files. We'll start with `edit_category.jsp`, so that we can test our service layer customizations by entering image URLs. At about line 107 of the file, add listing 8.4.

#### Listing 8.4 Adding the image URL field to `edit_category.jsp`

```
<%-- Adding Custom URL Field --%>

<%
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>



```

ShoppingCategoryImage catImage =
ShoppingCategoryImageLocalServiceUtil.getEmptyImage();

try {
    catImage =
ShoppingCategoryImageLocalServiceUtil.getShoppingCategoryImageByCategory(category.get
CategoryId());

    if (catImage != null) {

        // excellent, don't do anything

    } else {

        catImage = ShoppingCategoryImageLocalServiceUtil.getEmptyImage();
    }
} catch (Exception cie) {
}
}

%>

<auri:input name="<portlet:namespace />ExpandoAttributeName(image-url)" type="hidden"
value="image-url" />
A

<auri:field-wrapper label="image-url">

    <liferay-ui:input-field model="<%= ShoppingCategoryImage.class %>"
fieldParam="ExpandoAttribute(image-url)" bean="<%= catImage %>" field="imageUrl" />
B

</auri:field-wrapper>

```

#### **A Defining Expando attribute**

#### **B Using Expando attribute for field**

All of this makes sense up until you get to the tags, right? Here's where we are borrowing a bit from the Expando API that I mentioned earlier. Let's dig in to how this works.

### **8.3.5 Expandos, Service Context, and tokens, oh my!**

We're going to take a short roller coaster ride which will navigate you quickly through the twists and turns of a couple of Liferay APIs that make developers' lives easier. So picture yourself sitting in a seat next to somebody you really like, with anticipation building because you know this is going to be a really cool ride. The operator comes down the line of cars and pushes the restraining loop down over your head, and you feel yourself securely pressed into the seat. After making sure everyone is securely in their seats, the operator then goes to a control panel, and you feel the anticipation building.

Pressing a button, the ride starts on its way, and you feel the pull of gravity as you begin ascending a steep incline in the tracks. It doesn't matter which car you're in; whether you're in the front or in some other car, the incline is such that the view ahead of you is hidden either by the path of the tracks or by the backs of the people's heads in front of you. Soon, however, you feel the car leveling out, and there's a perceptible slowing down as you move from an incline, to being level, to a slight decline, to—

An HTML input field that's hidden is still a part of the DOM; it's just not displayed to the user. This makes it useful as a convention when processing a form if you want to differentiate some kinds of fields from others. Normally, you use the hidden input field to include data that the user doesn't care about or

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

doesn't need to see. We used it this way ourselves in Chapter 6, when we needed to include the primary key of a record in a form, but didn't want to display it to the user.

Liferay's form processing engine recognizes some patterns in order to implement Expando fields, which are dynamic fields that can be created at runtime by users. If you have ever used Custom Attributes in the Control Panel to add attributes to User records, you have already used this functionality from an end-user perspective. We are in a similar way adding an extra attribute to Shopping Categories: an image URL. We could implement this with the full Expando API, but we have instead used Service Builder to create real entities for our images. To support our additional attribute, we're just going to use the part of the Expando API that handles the front end and then persist the field using a custom service rather than through the Expando back end.

This highlights an important distinction between hooks and Ext. Hooks target specific areas of customization that have been identified by Liferay as the most frequent "hot spots" of Liferay Portal customization. In our case, we are using hooks to customize both the front end (JSPs) and to override a core Liferay service which manipulates Shopping Categories. In our JSPs, we are having to borrow from the Expando API for one simple reason: Liferay's portlet actions are not customizable by hooks; they can only be customized from Ext. This is one limitation of hook plugins which you will need to consider as you think about the implementation of your site. Normally, because we don't have access to the `PortletRequest` object from a hook, we wouldn't be able to retrieve values from a form. We can still accomplish what we want, however, because the Expando API is there. This API provides an easy way of getting our form data from the browser down into the service layer.

Because Expando fields can be defined at run time by users, Liferay needs to be able to differentiate them in the front end from the regular fields that will be handled in the standard way. In order to do this, a convention has been defined: first a hidden field with the attribute name (i.e., the key in a key/value pair) is defined, and then a Liferay tag which maps that key to a Service Builder bean is used. Tokens are used in order to accomplish this. When Liferay processes this form, it looks for any field names which correspond to the predefined tokens. The Expando API defines two tokens: `ExpandoAttributeName()` for the key and `ExpandoAttribute()` for the value. The first token is used by the API to tell Liferay what field to look for, and the second marks the field in the code so Liferay can find it when processing the fields.

Any fields which are marked as Expando attributes wind up in `ServiceContext`, which is passed down to the service layer by the portlet action for the Shopping Cart portlet.

The `ServiceContext` object is like one of those loops on the roller coaster ride. It's one of the most exciting changes that have been made to Liferay recently. Normally, you would not consider an object used as a parameter on a method to be something exciting. But consider this: Liferay's service methods used to (necessarily) have lots and lots and lots (and lots) of parameters. They had so many parameters that they were an incredible pain to use, especially for developers who had customized Liferay via Ext. Why? Because not only were there lots of parameters, it was necessary as features were added to Liferay to *change* the parameters all the time, even between point releases.

What a nightmare. It made it very difficult to keep in step with the changes being made in the core (because, as McCoy said about Scotty, engineers love to change things), and to write code against methods that had 15 parameters in their signatures (no joke). So the Liferay engineers created the `ServiceContext` object, and this solved a lot of these problems. Many of the common parameters which the service layer (such as the `companyId` or the `scopeGroupId`) used to require can now be

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

found in one object which can be passed down to the service layer. And, as we are seeing in this example, the `ServiceContext` object can also be a handy place to temporarily store custom attributes so they can be persisted, either as Expandos or as bona fide Service Builder entities.

Now, is that exciting, or what? You don't have to deal with any of the headaches that earlier Liferay platform developers have had to worry about. Congratulations! You've successfully hit that loop and gone all the way around without tossing your cookies. And because you did that, you've found that it was the most exciting part of the ride.

Combining these two concepts, then, Liferay knows that when it encounters a field which uses the predefined token for an Expando attribute, not to put it in the `PortletRequest` object where all of the regular form fields will be placed, but to instead put it in `ServiceContext` as a key/value pair. All of this is done by a simple form convention, which you can use to put any String value you like into `ServiceContext`. You have just gained a field processing engine from the browser down into your service layer, for free.

When you deploy the hook to Liferay, the JSP that Liferay normally uses to display the form for editing shopping categories is replaced, and your form is used instead. That form will render with an additional field, as you can see in figure 8.4.

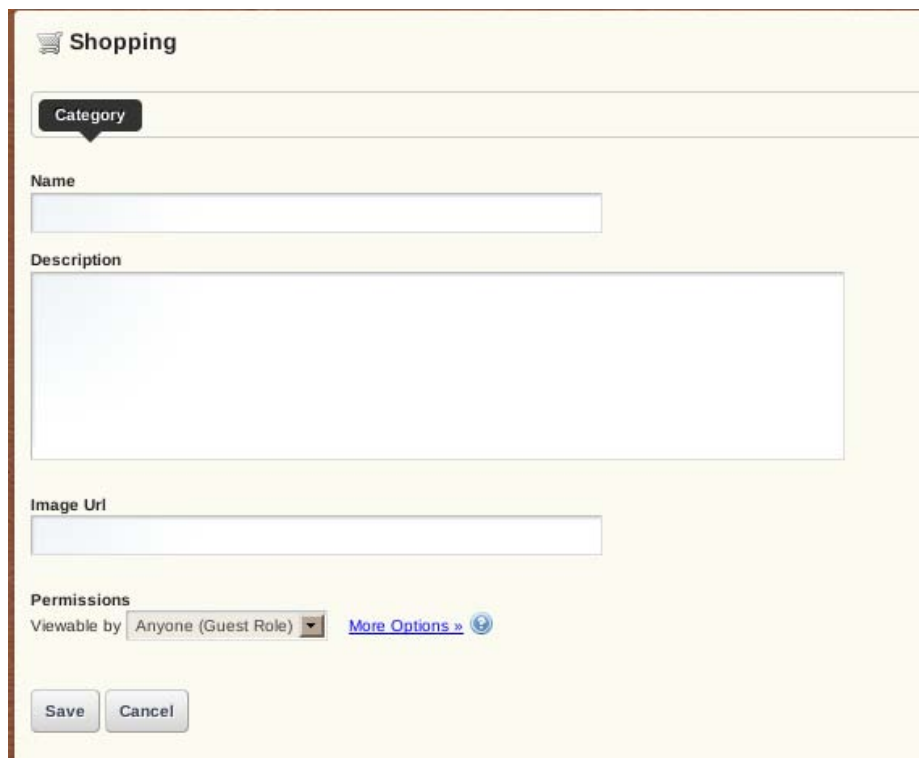
The image shows a web form titled "Shopping" with a shopping cart icon. It contains several input fields: a "Category" dropdown menu, a "Name" text field, a "Description" text area, and an "Image Url" text field. Below these is a "Permissions" section with a "Viewable by" dropdown set to "Anyone (Guest Role)" and a "More Options" link. At the bottom are "Save" and "Cancel" buttons.

Figure 8.4 Liferay's Shopping portlet now contains an extra field—your field—when a user goes to add a category.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

When a user pastes a value into the *Image Url* field, that field value will get down into your service layer via the roller coaster ride. You can then persist the field normally.

The Inkwell development team elected not to provide any field validation on the Image Url field so that they could finish the project faster. This is fine for this particular application: only users with the rights to add products and categories to the shopping cart will ever see this field. But if you are doing something like this for public users of your web site, you will definitely want to add a validation class to your hook in the same way we did with the portlet in Chapter 6. If you don't, you are only asking for trouble.

The easiest workflow for this, as defined by the Inkwell development team, is to have the users responsible for the shopping cart add a folder to Liferay's Image Gallery portlet called *Shopping Images*. Once added, they can copy / paste the URLs to those images when they create shopping categories. Since there are only going to be three categories, this should be pretty simple and straightforward.

What about the presentation? Surely, these images are to be presented to end users; otherwise, why would we be going to all this trouble to add them? Sure enough, the Inkwell design team has designed a new user interface for navigating the shopping categories. Users will be able to select product categories from a menu. When they roll their mouse over menu options, an image of that particular product category will be displayed. Here is what they have designed (figures 8.5, 8.6, and 8.7):



Figure 8.5 Categories appear in a horizontal menu. When users roll over menu items, the product image is displayed. The PDA Pen is by far Inkwell's hottest selling item, so it appears first in the list.



©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

Figure 8.6 The USB Pen is Inkwell's second best selling item, so it is second in the list.



Figure 8.7 The Wireless Pen, which is really a full-fledged computer, is Inkwell's newest item, and appears third in the list. This interface can handle a few more categories before it will need to be redesigned.

The beauty of this design is that it doesn't interfere with the existing interface for categories, which can be re-used by administrative users who need to add, edit, and delete categories. We will see how this is done shortly.

The design team first did a proof of concept of this design in a static web page, and then placed the resulting Javascript in the Inkwell Internet theme which was presented in Chapter 7. This necessitated the creation of a new file called `javascript.shopping.js`, which resides in the `_diffs/javascript` folder of the project. In this folder, we have a simple JavaScript function which defines the CSS background image attribute with whatever image URL is passed to the function:

```
function showProduct(p, pic) {  
  
    document.getElementById(p).style.backgroundImage="url('" + pic + "')";  
  
}
```

This file is included in the `portal_normal.vm` file by using a script tag in the `<head>` section:

```
<head>  
    <title>$the_title - $company_name</title>  
  
    <script src="$javascript_folder/javascript.shopping.js"  
type="text/javascript"></script>  
  
    $theme.include($top_head_include)  
</head>
```

This will make the script available to every page in the portal, so it could theoretically be used anywhere. We will make use of it on the page which displays the categories in the Shopping portlet, which is `categories.jspf`. This page is located in the portal in the folder `html/portlet/shopping`. Since we'll be overriding this page in our hook, we will start by copying this file into our hook project, using the exact same folder structure. Since we have already defined the

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

location of our JSP files as META-INF/custom\_jsps in our liferay-hook.xml file, we will create the proper folder structure there and copy the original file into it.

At approximately **line 116**, we can implement our new user interface using code in listing 8.5.

### Listing 8.5 Customizing the shopping interface

```
<%-- Category customizations here --%>

<%

StringBuilder catMenu = new StringBuilder();
catMenu.append("<div id='pen-menu'> <ul>");

for (int j=0;j<results.size();j++) {

    ShoppingCategory cat = (ShoppingCategory)results.get(j);
    String catTitle = cat.getName();
    String catJSTitle = JS.getSafeName(catTitle);
    ShoppingCategoryImage catImage =
        ShoppingCategoryImageLocalServiceUtil.getShoppingCategoryImageByCategory(
            cat.getCategoryId());
    String catImageUrl = catImage.getImageUrl();

    PortletURL catURL = renderResponse.createRenderURL();

    catURL.setWindowState(WindowState.MAXIMIZED);

    catURL.setParameter("struts_action", "/shopping/view");
    catURL.setParameter("categoryId", String.valueOf(cat.getCategoryId()));

    catMenu.append("<li><a href=\"" + catURL + "\""
onmouseover=\"showProduct('shopping-image','" + catImageUrl + "');\"
onmouseout=\"showProduct('shopping-image','/inkwell-web-site-
theme/images/custom/INKWELL_logo_V3.png')\">" + catTitle + "</a></li>");

}

catMenu.append("</ul></div>");

%>

<div id="pen-menu-container">

    <%=catMenu.toString() %>

</div>

<div id="pen-container">

    <div id="shopping-image">

<%

    if (category != null) {

        ShoppingCategoryImage chosenImage =
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

```

ShoppingCategoryImageLocalServiceUtil.getShoppingCategoryImageByCategory(category.get
CategoryId());
%>


<%
    } else {
%>

    &nbsp;
<%
    }
%>

</div>

</div>

```

Let's take a look at the overall strategy here, and that should help to make sense of the code above. If we think of our algorithm in a step-by-step fashion, we come up with the following steps:

- Create a String that's going to turn records from the database into HTML markup.

- Loop through each Shopping Category. For each one, get its:

- Name, so we can use the name as our link

- Image URL, so we can display the image when the mouse rolls over the link

- As we loop, we'll also create a URL which will comprise the actual link.

- Once we have all of these items, we can concatenate a String which can use them in the markup to create the menu. This menu will be created as an unordered list in HTML, which is easy to style via CSS.

- When we're done with the loop, we close the tags at the end of the String.

- All that remains is to spit out that String in the appropriate place on the page.

That's all there is to it. The result is an HTML unordered list that contains menu items built from our database data. After this is a `<div>` which we are calling `pen-container` which contains another `<div>` called `shopping-image`.

The links contain `onmouseover` and `onmouseout` events which call the JavaScript function that was defined in the theme. This function receives the `shopping-image <div>` and swaps the `background-image` attribute with the one passed to the function. Since we built each link as we were looping through the data from the database, each link will contain the image URL that was stored with the `ShoppingCategoryImage` entity. This URL, of course, points to an image stored in Liferay's Image Gallery portlet.

The supporting CSS for this code was also placed in the Inkwell Web Site theme, and it styles the menu and images according to the look and feel defined by the design team.

The `<div>` containing the image also contains some code. The reason for this is that the same JSP fragment is used to display the categories and to display the category menu. So when the category is

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

chosen, we want to display that category's image. For that reason, we fill the `<div>` with the image if the user has chosen a category; if not, we just have a non breaking space.

We can't just simply replace the old interface with this one, however. The old interface catered to two audiences: shoppers coming to the site to buy something, and site administrators who need to maintain the site. This new interface caters only to shoppers. So we need to preserve the old interface in some way, but make sure that shoppers never see it. It turns out that this is very easy to do.

Notice that all the way up around **line 111**, a boolean called `showAddCategoryButton` is created, based on the permissions of the current user. Obviously, regular users won't be allowed to add categories, so we already have the permission check that we need built in to the page!

The panel that is used to display the Search Container comes next in the markup. We can wrap that panel in a check on the `showAddCategoryButton` boolean, and thereby hide it for regular users and show it for administrators:

```
<c:if test="<%= showAddCategoryButton %>">
  <liferay-ui:panel collapsible="<%= true %>" extended="<%= true %>"
  persistState="<%= true %>" title="<%= LanguageUtil.get(pageContext, "categories")
  %>'>
    <c:if test="<%= showAddCategoryButton || showPermissionsButton || showSearch
    %>">
      <au:fieldset>
        <c:if test="<%= showSearch %>">

[ ... ]

</c:if>
```

At this point, the Inkwell Shopping hook is feature complete, and can be deployed to Liferay. Since it works in conjunction with Inkwell's theme, it will only work when that theme is being used.

## 8.4 Summary

You have just experienced a comprehensive presentation on Liferay hooks. Hooks are a powerful way to customize some core Liferay functionality. You can override some properties and append your own values to list properties, enabling you to define your own Liferay startup events, listeners, authentication events, and more. You can override and define your own Language properties as well, enabling you to use your own terminology or to provide alternate translations of specific keys.

More powerfully, you can override Liferay's JSP files with your own implementation, and also override Liferay's service implementations with your own. Combining these features gives you as a developer some powerful ways to change the core behavior of Liferay, simply by deploying a plugin.

Using our case study, you then saw a practical example of how to bring all of these concepts together in an example hook which overrides a core Liferay service, provides a new entity of its own, contains a Language property, and customizes some core Liferay JSP files in the Shopping Cart portlet.

Now you are empowered to go and create hooks which customize Liferay to your needs. You will find that this is an easy and convenient way to make Liferay sing to your tune, without ever having to touch the Liferay source code.

If you've got a customization you need to make which is just not possible using a hook, read on, because we'll be diving deeper into customizations with Ext plugins next.





## *Liferay and IDEs*

Though there are developers today who prefer nothing more than a command line interface to a build tool such as Ant coupled with a good text editor, they aren't me, even though I hope someday to aspire to such guru-ship. It can be argued that most developers use an Integrated Development Environment (IDE) to produce code. IDEs offer several benefits over a command line interface and text editor, such as:

- Code lookup and completion
- Project and file browsing
- Refactoring support
- Integrated debugging
- Integrated interface to Source Code Management (SCM) software, complete with diff tools
- Quick, built-in interface to Javadoc and in-line documentation
- Depending on the IDE, more; sometimes much more

From the information in Chapter 2, it should be easy to see how to use Liferay's SDKs with just a text editor and a command line interface to Ant. Since many if not most developers today use an IDE, it's important to spend some time covering how you'd set up Plugins SDK projects in an IDE, but the topic wasn't necessarily germane to Liferay specifically. For this reason, I'll demonstrate how to work with Liferay using the two most popular open source IDEs available at the time of this writing: Eclipse (<http://www.eclipse.org>) and NetBeans (<http://www.netbeans.org>), as well as with Liferay's flavor of Eclipse, Liferay IDE.

Another note about agnosticity: because developers are most effective in the tool of their choice, I'm not recommending any IDE over another; I chose Eclipse and NetBeans because they are both open source IDEs and thus both represent zero barrier to entry solutions for state-of-the-art development (and it should be obvious as to why Liferay IDE was chosen). With regard to the two generic IDEs, I will attempt to be fair and present them in alphabetical order, making no recommendations as to which may be most appropriate for your project, as both are adequate and ideal solutions in their own right. I'll present Liferay IDE first, as it has been specifically designed for Liferay.

All of these products have documentation of their own, so if you plan on using one of these IDEs to create your portlets, themes, or to extend Liferay, you will be best served by referring to the

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

documentation for these projects. I'm assuming you already know how to use your IDE, so we'll cover only the basics of getting up and running here. If you are using a commercial IDE such as IntelliJ IDEA, Borland JBuilder, or Oracle JDeveloper, the concepts in this chapter apply as well, and you should be able to use this information to do Liferay development in your IDE of choice.

## A.1 Liferay IDE

[Section to be written]

## A.2 Using Eclipse

Eclipse is an open source IDE that was originally created by IBM to replace their aging Visual Age for Java product. IBM's goal is stated as follows:

We wanted to establish a common platform for all IBM development products to avoid duplicating the most common elements of infrastructure. This would allow customers using multiple tools built by different parts of IBM to have a more integrated experience as they switched from one tool to another. We envisioned the customer's complete development environment to be composed of a heterogeneous combination of tools from IBM, the customer's custom toolbox, and third-party tools. This heterogeneous, but compatible, tool environment was the inception of a software tools *ecosystem*.<sup>2</sup>

IBM built the initial version of Eclipse and released it in 2003. Afterward, they created the Eclipse Foundation for two reasons: 1) to provide an open, collaborative organization to oversee the future development of Eclipse, and 2) to remove the perception that Eclipse was under IBM's control, as the goal was always to provide an open platform that was vendor-neutral.

This strategy worked, and today Eclipse in some incarnation (whether it be Eclipse itself or one of many re-branded versions that ship with various plugins) is one of the most widely used IDEs on the market today.

Eclipse comes in several downloadable versions for various operating systems. Because Liferay is a Java web application, I recommend that you download the version that is labeled for *Java EE Developers*. This gives you the basic features you need for Java EE applications like Liferay.

If you're going to work with Liferay's source code, you may find it useful to have *Subclipse*, which is an Eclipse plugin that interfaces with Subversion code repositories. Since Liferay's source code is stored in a Subversion repository, it can be more convenient to update from the repository from within the IDE. Instructions for installing Subclipse can be found at the product's web site at <http://subclipse.tigris.org>.

Speaking of plugins, yes, Eclipse also has the concept of plugins. These have nothing to do with Liferay's plugins, but are instead ways of extending Eclipse to do things other than what it does when you first download it. For example, RedHat provides a whole business modeling tool based on Eclipse plugins.

It is beyond the scope of this book to go over all of the Eclipse plugins that you might want to use as a developer. Suffice it to say that there are plugins for Facelets, Velocity templates, additional

---

<sup>2</sup> <http://www.ibm.com/developerworks/rational/library/nov05/cernosek/index.html>

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

application servers, diagramming, and much more. The configuration outlined above (Eclipse for Java EE plus optionally a plugin for Subversion) is the minimum you'll need to be effective in developing for Liferay. With this configuration, you should be able to do everything from implementing your own projects to contributing to Liferay's core.

## ECLIPSE AND WORKSPACES

Eclipse uses the notion of *workspaces* (something it inherited from Visual Age for Java) to store Java projects. Unlike Visual Age for Java, an Eclipse workspace is just a folder on your file system with projects in it. Eclipse creates various hidden folders (using Unix-style dot-notation) which make the IDE look at the folder structure as a workspace. Eclipse also allows you to import projects from your file system. In that case, only Eclipse's configuration files for the various projects are stored in the workspace. I think it's better to use Eclipse this way, because you're free to put your projects anywhere you want (which makes it easier to open your projects in other editors or IDEs).

The first time you launch Eclipse, it will ask you for a workspace location. If you are developing on a Windows platform and you plan to store projects in the workspace, I recommend that you place your workspaces somewhere close to the root folder of your drive in [Code Home], for reasons mentioned at the beginning of this chapter. Users of LUM operating systems do not have this problem.

Longtime Eclipse users generally have the practice of keeping separate workspaces for separate groups of projects. For example, if a developer were working on a set of portlets that made up a Customer Relationship Management system, that set of projects might exist in one workspace. Say a bug was found that affected another set of projects the developer completed a month ago. This developer could then switch workspaces and open the other set of projects all at once.

Developers who use Eclipse have learned to do this because Eclipse compiles code in the background from time to time. If there are a lot of projects in the workspace, this process can take a long time and slow down your system. Liferay is a very large project with tens of thousands of source code files, so you can imagine how Eclipse might be affected by just having the Liferay source code project in its workspace. As a plugin developer, this won't affect you as much, because you won't necessarily need access to the Liferay source. But if you want to hack on Liferay itself, it can be an issue, especially if you mix Liferay and a whole bunch of other projects in your workspace.

One other way to accomplish almost the same thing is the concept of *working sets*. These are sets of projects that can be opened and closed as a group. If you use working sets, you can generally keep all your projects in one workspace, and only open the group of projects with which you are working at any one time.

More recently, Eclipse allowed developers to open projects that exist outside the workspace. In this case, Eclipse keeps only its own configuration files in the workspace, and the source code files stay wherever they were. Since we use the Plugins SDK to generate portlet, theme, layout template, and other projects, we won't be using Eclipse to create new projects. For this reason, it will be better to use Eclipse to point to projects in your Plugins SDK, rather than putting your whole Plugins SDK in the workspace. You can still have different workspaces for different project sets; you'll just need to install a different Plugins SDK somewhere else on your system and point to those projects in a new workspace.

This may sound complicated to the uninitiated, but it's really not. We'll see how to get all of this set up in the next section.

## SERVER RUNTIME

Launch Eclipse and connect to the workspace of your choice. If you have never used Eclipse before, the default location is generally fine, unless you are on a Windows system—remember that 256 character limit on the total path of an NTFS file system.

Our first task in Eclipse will be to connect Eclipse to our installed Liferay bundle. We need to do this so that we can start it in debug mode in order to debug our code.

If you have installed Eclipse for Java EE developers, then Eclipse will by default start in the *Java EE* perspective. This is a version of the Eclipse UI that is optimized for writing Java EE applications. Eclipse has many other perspectives (which can be defined as versions of the UI), but this one is the one most appropriate for writing portlets.

At the bottom of the perspective is a tab called *Servers*. Click this tab to activate it, and then right click in a blank space in the body of the tab. Select *New > Server*. From the dialog box that appears, choose the type of server bundle you have installed. If you're following along with our recommended setup, this will be a Tomcat v6.x bundle, but you can use any bundle that Eclipse supports. If your server is not listed, you can click the link labeled *Download Additional Server Adapters* and install an adapter for your application server of choice.

At the bottom, you can change the server name so that it better reflects what server it is (see figure 2.6). For example, for a Tomcat v6.x bundle (the default at the time of this writing), you could change the name to *Liferay-Tomcat 6.0*.

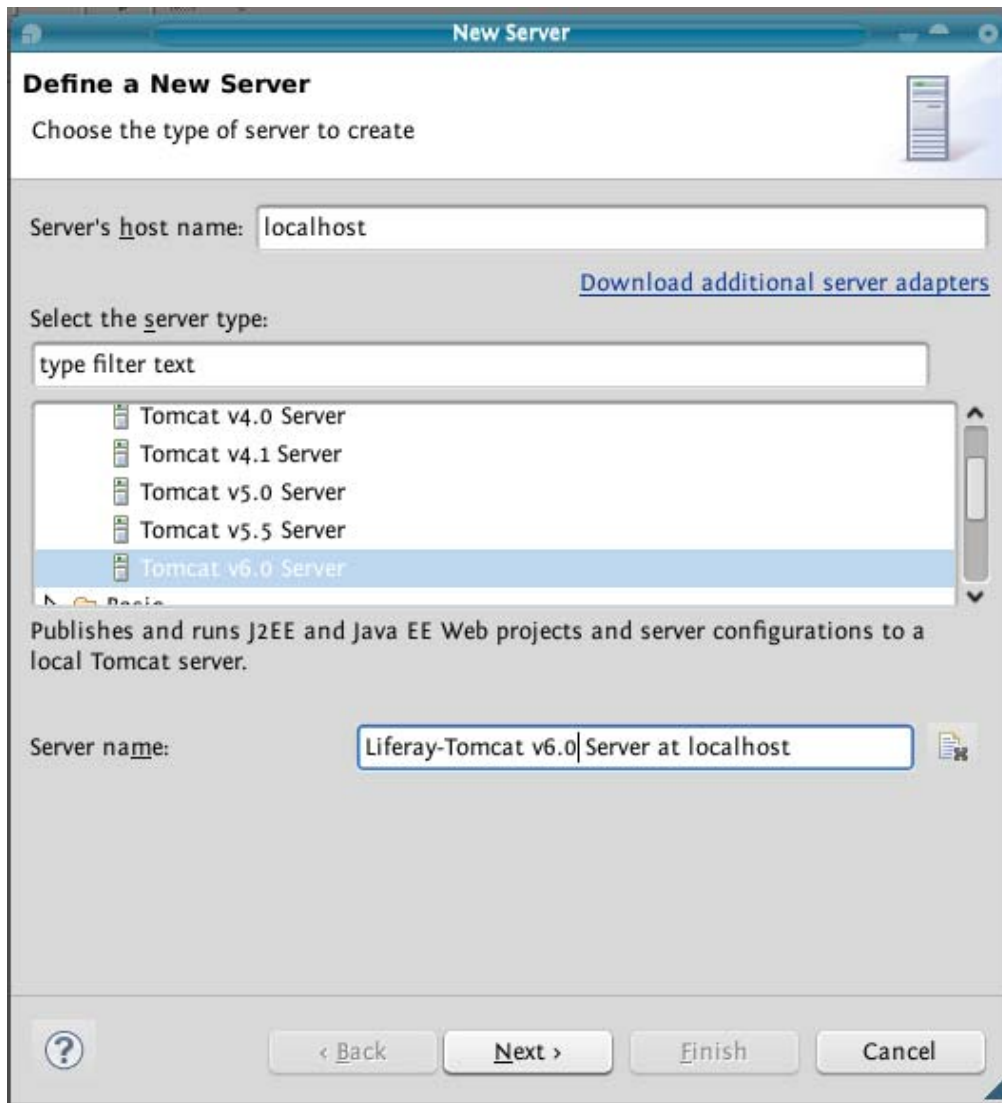


Figure 2.6 Setting up your Liferay runtime is easy because Eclipse contains support for many different kinds of application servers, including the one in the default Liferay bundle.

Click *Next*. In the dialog box that follows, you can click a *Browse* button to browse to the location where you have installed your Liferay bundle. Browse to this location and click *Finish*. The dialog will disappear and the server runtime will now appear in the Servers tab.

Right-click on your new server runtime and select *Open*. You will see the Eclipse Configuration page for your application server. By default, Eclipse creates a whole separate configuration for your application server in its workspace. We don't want it to do that; we want it to use the server

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

configuration that already exists in the application server, because the bundle is already pre-configured to run Liferay.

Under *Server Locations*, select *Use Tomcat Installation*. Under *Deploy Path*, select the *Browse* button and browse to your bundle's webapps folder.

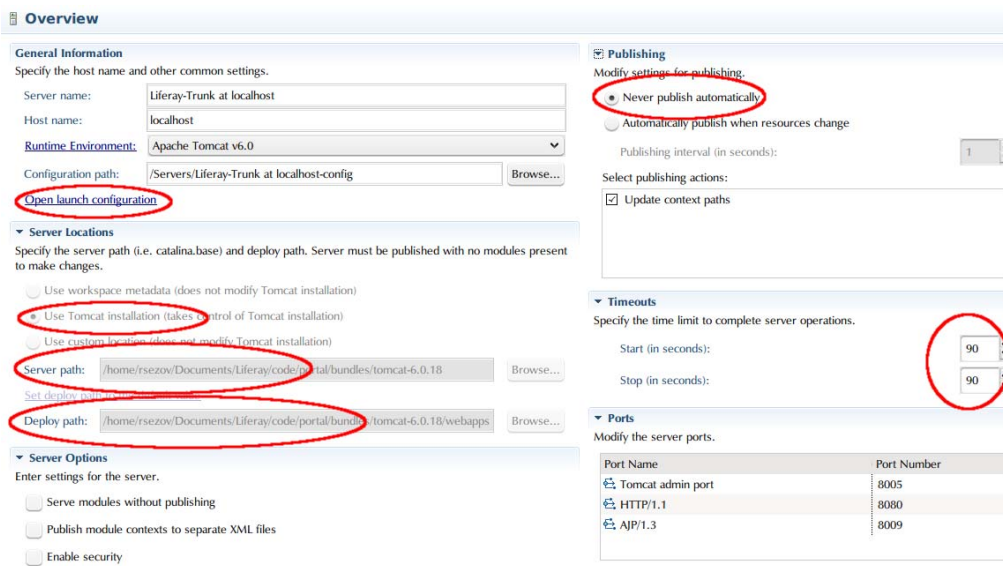


Figure 2.7 This shows all of the locations on the server page where settings need to be changed in Eclipse.

On the right side of the screen, open the section labeled *Publishing* and select *Never Publish Automatically*. Open the section labeled *Timeouts* and change both the start and stop timeouts to 30 seconds.

Finally, on the left side, select *Open Launch Configuration*. Select the *Arguments* tab and add the following arguments to the end of the list:

```
-Dfile.encoding=UTF8 -Duser.timezone=GMT -Xmx1024m -XX:MaxPermSize=256m -Dexternal-properties=portal-developer.properties
```

Click *Apply* and then *OK*. Close and save the configuration window by clicking on the X icon in the tab.

You're now ready to start your Liferay bundle in Eclipse! Right-click on it and select *Start* or select it and click the green *play* icon in the *Servers* window border. You should see the server startup messages scroll in Eclipse's console window. When Liferay has finished starting, the console window will switch back to the *Servers* tab, and the server's status will be labeled "started." You can always switch back to the *Console* tab to see the server console messages. In fact, Eclipse will do this automatically if another message is generated.

For now, right-click on the server in the *Servers* tab and select *Stop*. This will shut down Liferay.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

### SETTING UP A PLUGIN PROJECT

Since we've already generated a portlet project in the Plugins SDK called Hello World, let's set up that project for use in Eclipse. In Eclipse, select *File > New > Dynamic Web Project*. A new dialog box will come up asking you for the name of your project. Type *hello-world-portlet* in the **Project Name** field and then uncheck the box labeled *Use Default Location*.

Use the *Browse* button to navigate to the new project you just generated in your Plugins SDK. When you are finished, you should have a dialog box that looks like this:

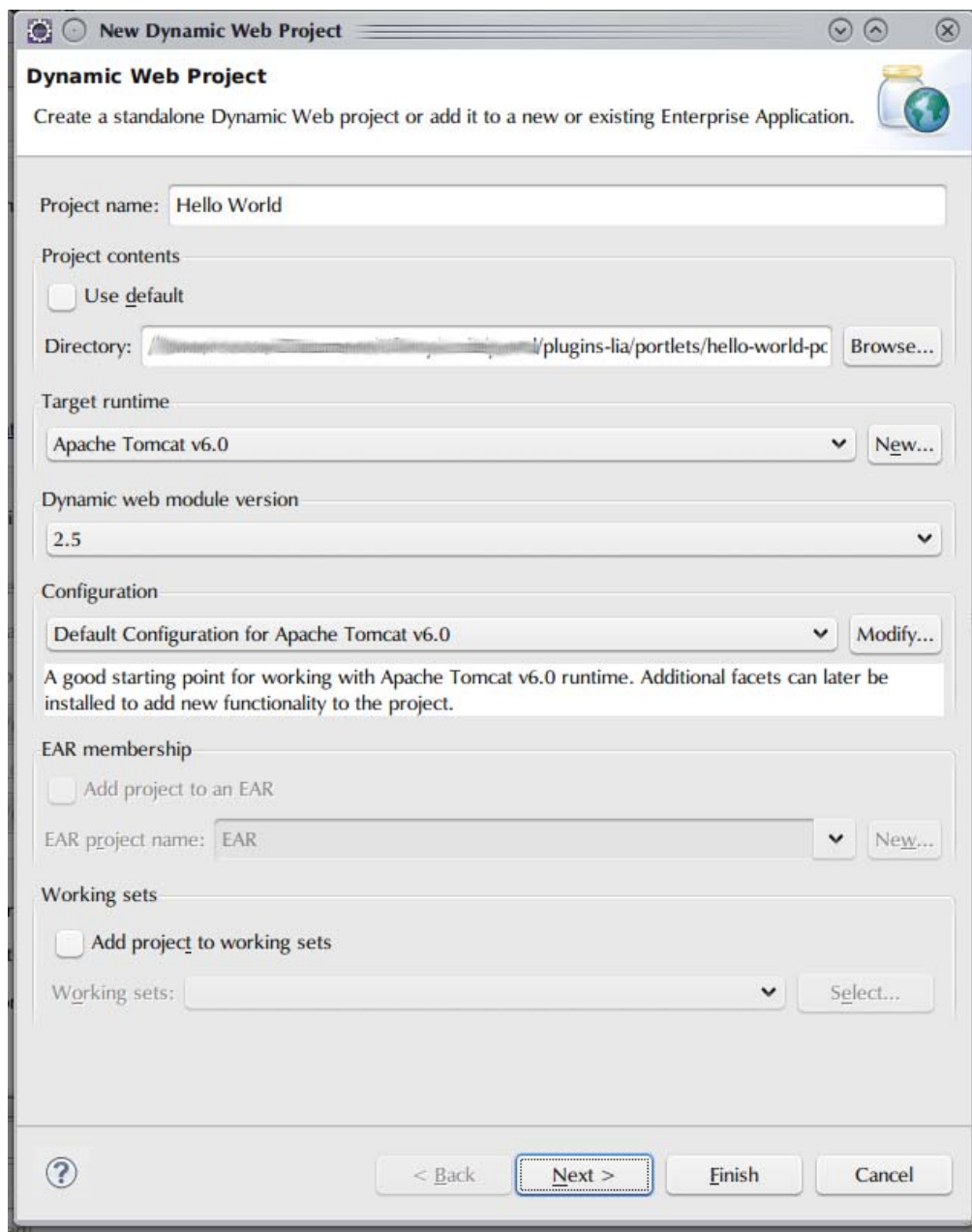


Figure 2.8 The first dialog box in setting up your Hello World project in Eclipse. I have smudged out the actual path on my system because I'm paranoid. Please don't be offended.

Select the web module version appropriate for your application server and click *Next*.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>



Eclipse will then guess at where your source code folder is and get the guess wrong. In the dialog box, select the source folder and click the *Edit* button. In the dialog box that appears, type *docroot/WEB-INF/src* and click *OK*. At the bottom of the dialog box, change the output folder to *docroot/WEB-INF/classes*. Click *Next*.

You are now given an opportunity to select your project structure. Plugins SDK portlet projects differ from Eclipse's default project structure in that the name of the web module content folder is called *docroot*. Eclipse by default uses *WebContent*. So under **Content Directory**, type *docroot*. If the check box for *Generate web.xml* is checked, uncheck it.

When you are done, click *Finish*. Eclipse will then open your project, and you will see it in the Project Explorer window on the left. You would use the same technique to set up a Liferay theme, layout template, web, or hook project.

### FIXING PROJECT DEPENDENCIES

You have probably noticed that your project as-is in Eclipse is showing that it has errors in it. The reason for this is that the dependencies for the project have not yet been configured.

Portlet projects require several dependencies on the class path. Eclipse has an easy way of setting this up for your project. Right-click on your project and select *properties*. From the list on the left select *Java Build Path*. Select *Server Runtime* and then click *Next*. Select the Liferay bundle runtime you configured earlier and click *Finish*.

This puts the Java classes that will be on the application's class

Next, select *Add External Jars* and then browse to your Liferay-Tomcat bundle folder. Browse to *webapps/ROOT/WEB-INF/lib* and then select the following .jars:

- commons-logging.jar
- util-java.jar
- util-taglib.jar
- util-bridges.jar

Click *OK* to clear the dialog box and after a brief pause (to allow Eclipse to attempt to compile the source code again), the errors should disappear. If you have errors in JSP or TLD files in your project, don't worry about these. Sometimes Eclipse cannot parse them properly.

### DEBUGGING AND DEPLOYING

You can now use Eclipse to debug and test portlet projects. You already have a Liferay bundle set up as a server in Eclipse. You can start this server in debug mode, set a break point in your portlet code, and step through to watch the functionality. This is as simple as right-clicking on your server in the *Servers* tab and choosing *Debug* instead of *Start* as you did earlier.

One other thing to note is that you should never use Eclipse to build and deploy your projects. Always use the Ant script that was generated with the project. Eclipse has very good integration with Ant. You can enable the Ant view by selecting *Window > Show View > Other > Ant*. This will place an *Ant* tab in the same location as your *Servers* and your *Console* tab. To use it, simply drag your *build.xml* script from your project to this view. It will be automatically parsed, and you will be able to select any ant task to run by simply double-clicking on it.

Eclipse users should now be comfortable working with Liferay projects—to say any more here would start delving into documenting Eclipse, and you would be much better served by official Eclipse

documentation. Suffice it to say that the Eclipse IDE is a great environment for working on Liferay projects, and lots of Liferay developers use it every day for just that purpose.

### A.3 Using NetBeans

NetBeans is an open-source IDE which is dual-licensed under the Common Development and Distribution License (CDDL) and the GNU Public License version 2 (GPLv2). Originally created by students at Charles University in Prague as a closed-source product, the IDE was purchased by Sun in 1999 and then subsequently open-sourced.

NetBeans has gone through many changes since its humble beginnings, and is now the flagship IDE offered by Sun, often bundled along with Java Development Kit downloads. At the time of this writing, Oracle has announced continued support for it, so it isn't going anywhere either. NetBeans is very powerful and supports all of the features you would need for Liferay development out of the box. NetBeans, like Eclipse, can also be extended by using plugins. Additional plugins are available from its plugin repository to support additional application servers, importing projects from other IDEs, Facelets, UML diagramming, and more. It is a cross-platform project, implemented in Java, using Swing as its widget set for its graphical user interface.

Projects in NetBeans are configured using its GUI, and settings are stored in XML files in a special *nbproject* folder which is created in the root folder of the project. Projects can be stored anywhere on your system and are opened using a typical *File > Open Project* command. Ant is integrated with the IDE at all levels: if you create a new project using NetBeans, it will use Ant to build it behind the scenes. You can use your own Ant scripts as well, and this is how we will be using NetBeans, since projects built out of the Plugins SDK already have their own Ant scripts.

To install NetBeans, download the installer and run it on your system. It's best that you download either the *Web and Java EE* distribution or the *All* distribution. On all operating systems, the installer is a typical wizard-based installation routine, and it allows you to install NetBeans anywhere on your system. When the installer has finished, you should have a NetBeans icon on your desktop. Double-click this icon to start NetBeans.

#### SERVER RUNTIME

Our first task in NetBeans will be to set up the Liferay bundle we installed earlier. On the left side of the NetBeans work area three tabs are displayed: *Projects*, *Files*, and *Services*. Click the *Services* tab. The bottom node in the tree is labeled *Servers*. If you open this node, you will see that there are one or more application server runtimes already registered with NetBeans, depending on which version of NetBeans you downloaded. We'll add another one specifically for your Liferay-Tomcat bundle.

Right click on the node labeled *Servers* and select *Add Server*. Since the bundle we're working with uses Tomcat 6.x as an application server, choose that. Note that you can also choose any application server that NetBeans supports a runtime for, but we will stick with Tomcat in our examples.

Once you choose a server, change its name to something that will help you recognize it, such as *Liferay-Tomcat-6*. When finished, click *Next*.

In the dialog box that follows, click the *Browse* button and browse to the location of your Liferay bundle. At the bottom of the dialog you are asked for a user name and password for the Tomcat manager role. Enter *tomcat* for both. When you are finished, you should have a dialog box that looks like figure 2.9.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

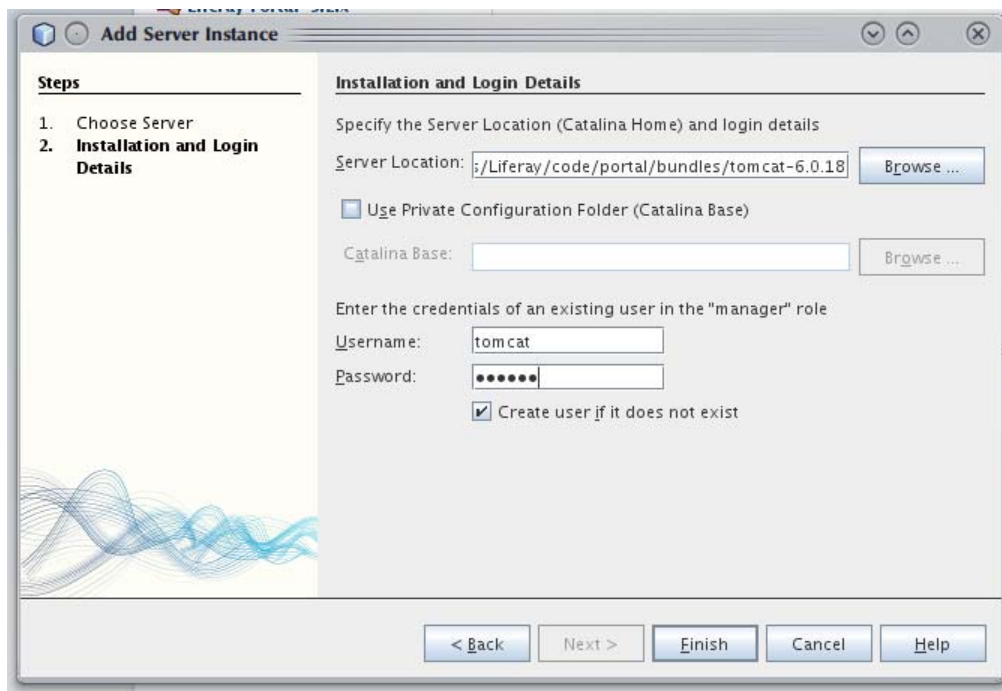


Figure 2.9 One plus for NetBeans: setting up a server runtime is a lot easier than it is in Eclipse.

Click *Finish*. The Liferay bundle will now be listed in the tree under *Servers*. To start it, right-click on it and select *Start*. You should see the Liferay server console messages begin to scroll in the console window at the bottom of your NetBeans screen. When the server has completed its startup sequence, it will be marked with a green “play” icon in the Servers tree on the left.

For now, right-click on your server and select *Stop*. Liferay will shut down, and the green “play” icon will disappear.

### SETTING UP A PROJECT

Since we already generated our Hello World project in the Plugins SDK, let's get it set up in NetBeans. Select *File > New Project*. The New Project Wizard will appear. From the left column, choose *Java Web* and from the right column choose *Web Application with Existing Sources*. This indicates to NetBeans that you want to create a project that already exists. Click *Next*.

In the dialog box that follows, click the *Browse* button next to the **Location** field and browse to the location inside your Plugins SDK of your *hello-world-portlet* project. The rest of the fields should then be automatically filled out. When finished, the dialog should look something like figure 2.10.

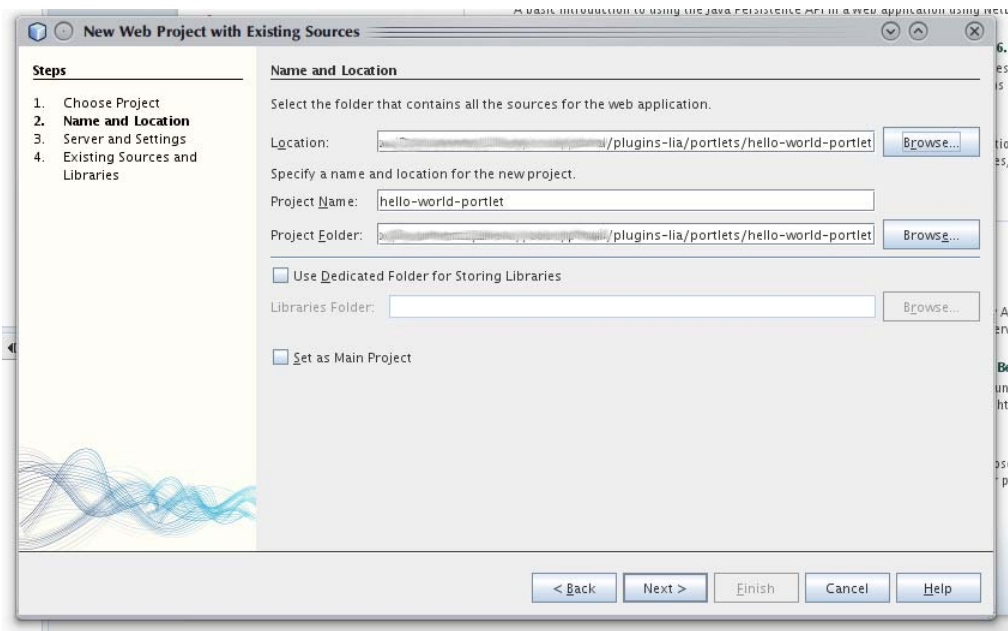


Figure 2.10 Browse to the path of your project here, and the rest of the fields should be filled out automatically. I have smudged out the actual path on my system because I'm paranoid. Please don't be offended.

A new dialog box will appear which tells you an Ant script already exists in the project. NetBeans uses Ant internally to build projects, so every NetBeans project needs to have an Ant script. This dialog box says that since the project already has an Ant script called *build.xml*, the IDE is going to generate one for NetBeans called *nbbuild.xml*. Click *OK* to let NetBeans do this—we will likely never use the script that NetBeans generates anyway.

The dialog box that follows asks you to map the project to a server runtime. Choose the Liferay runtime that you created previously. Choose the Java EE version that your application server of choice supports. Click *Next*.

The final dialog box is pre-filled with values for the location of your web pages folder and your WEB-INF content. It generally guesses these correctly, but the values should be *[Project Home]/docroot* for Web Pages, *[Project Home]/docroot/WEB-INF* for WEB-INF Content, and *[Project Home]/docroot/WEB-INF/lib* for the Libraries Folder, where *[Project Home]* is the location of your project in your Plugins SDK. You'll also need to add the folder where your Java source will be stored, and this should be *[Project Home]/docroot/WEB-INF/src*. The final dialog should look like what is pictured below. Click *Finish*.

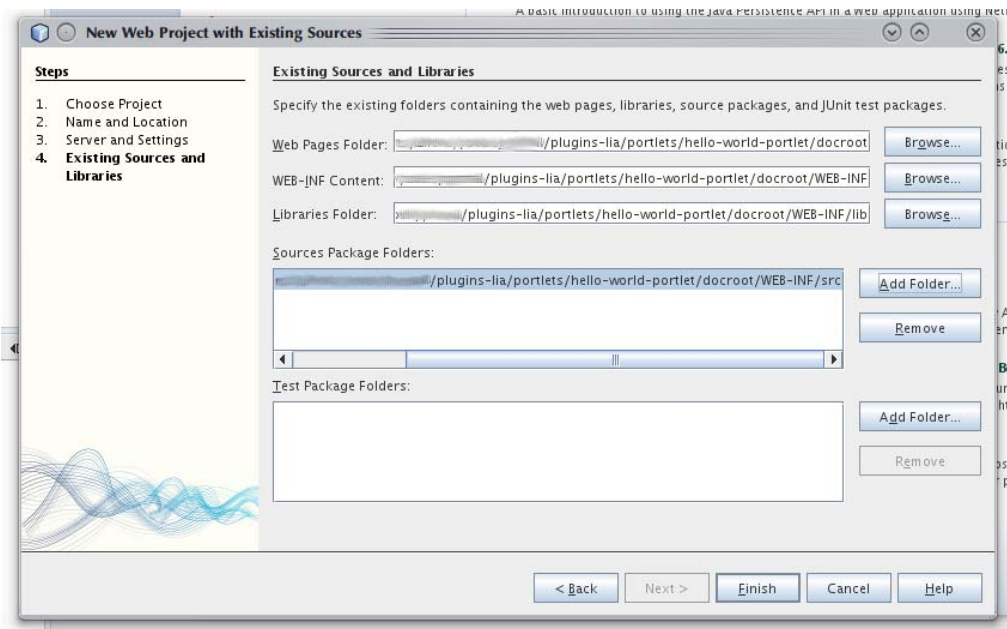


Figure 2.11 This is the final dialog for creating a project in NetBeans. Even if you get these folders wrong, you can always correct them later.

Your project should be automatically created. You may, however, have errors in the project because it requires certain .jar dependencies to be on the class path. To fix this, right-click on the project and click *Properties*. Click the *Libraries* category.

This area allows you to choose your project dependencies. Portlet projects require several dependencies on the class path. You will need to place the Java classes that will be on the application's class path at runtime here, as well as any other dependencies that will be deployed with the project. Below is the list of dependencies required. Add these using the *Add Jar/Folder* button:

```
[Liferay Install Location]/lib/servlet-api.jar
[Liferay Install Location]/lib/jsp-api.jar
[Liferay Install Location]/lib/ext/portal-kernel.jar
[Liferay Install Location]/lib/ext/portal-service.jar
[Liferay Install Location]/lib/ext/portlet.jar
[Liferay Install Location]/lib/ext/activation.jar
[Liferay Install Location]/lib/ext/annotations.jar
[Liferay Install Location]/lib/ext/hsqldb.jar
[Liferay Install Location]/lib/ext/jms.jar
[Liferay Install Location]/lib/ext/jta.jar
[Liferay Install Location]/lib/ext/mail.jar
[Liferay Install Location]/lib/ext/mysql.jar
[Liferay Install Location]/lib/ext/postgresql.jar
[Liferay Install Location]/webapps/ROOT/WEB-INF/lib/util-taglib.jar
[Liferay Install Location]/webapps/ROOT/WEB-INF/lib/util-bridges.jar
[Liferay Install Location]/webapps/ROOT/WEB-INF/lib/util-java.jar
[Liferay Install Location]/webapps/ROOT/WEB-INF/lib/jstl-impl.jar
[Liferay Install Location]/webapps/ROOT/WEB-INF/lib/jstl.jar
[Liferay Install Location]/webapps/ROOT/WEB-INF/lib/commons-logging.jar
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

Once you've added these and clicked *Ok*, all of the errors in your project should disappear. You're now ready to work on your project in NetBeans.

### DEBUGGING AND DEPLOYING

You now have a portlet project set up properly in NetBeans. You could now begin using the IDE to build your application. At some point, you will want to deploy your application to your Liferay server in order to debug it. This is very easy to do in NetBeans.

First, start your server in debug mode. You can do this by clicking the *Services* tab on the left side of the screen, right-clicking on your server, and then selecting *Start in Debug Mode*. The server should then start in debug mode.

To deploy your portlet, click on the *Files* tab on the left side of the NetBeans window. Expand your project and click on the *build.xml* file, which is your Ant script. In the window below (labeled *Navigators*), all of the Ant tasks are displayed. Right click on the *deploy* task and click *Run Target*. Your project will be deployed to Liferay.

To debug your code, set a breakpoint in a source code file by clicking in the margin of the Java editor on the line of code you wish to start debugging. Then add your portlet to a page in Liferay.

Debugging in NetBeans is a two-step process. First you start the server in debug mode, and then you attach the debugger. The console window contains a log of your server's startup. At the top of this log, the debugging port is displayed. You will likely have to scroll all the way to the top of the log in the console window in order to see it. Take note of this port number. Then click *Debug > Attach Debugger* and enter this number into the **Port** field. Click *OK* and the IDE will go into debug mode, where you'll be able to step through code, view variables, and so on. When your portlet reaches the code at which you have set a breakpoint, processing will stop and you'll be able to use the IDE to step through the processing.

### PROJECT SETTINGS

If you need to change the settings of your project (for example, to add a source code tree for unit tests), you can right-click on your project and select *Properties*. Here, you can step through some of the dialog boxes you used in the *New Project* wizard. This allows you to easily set up new project dependencies, map IDE tasks to Ant tasks, and more.

# B

## *Introduction to the Portlet API*

To help you understand the foundation undergirding Liferay's platform and get started writing portlets according to the Java Portlet Standard, this appendix takes a closer look at the Portlet API, which you may not be familiar with. For a more in-depth view of the Portlet API, see the Manning title, *Portlets in Action*.

### ***B.1 Portlets as fragments of a web page***

Portlets are web applications that run in a portion of a web page. Rather than being responsible for (and taking up) the whole page, they instead are responsible only for their own functionality. If you were expecting me to say something more profound or mysterious, I'm sorry to disappoint you: that's all they are. But this really gains you so much.

As a developer, you are free to concentrate only on your application and its functionality, and you don't have to worry about all the ancillary things like users, user management, registration, layout, permissions, and the like. That stuff is built into the portal, and you get to simply take advantage of it in your application. Figure B.1 shows four portlets running in a single web page.

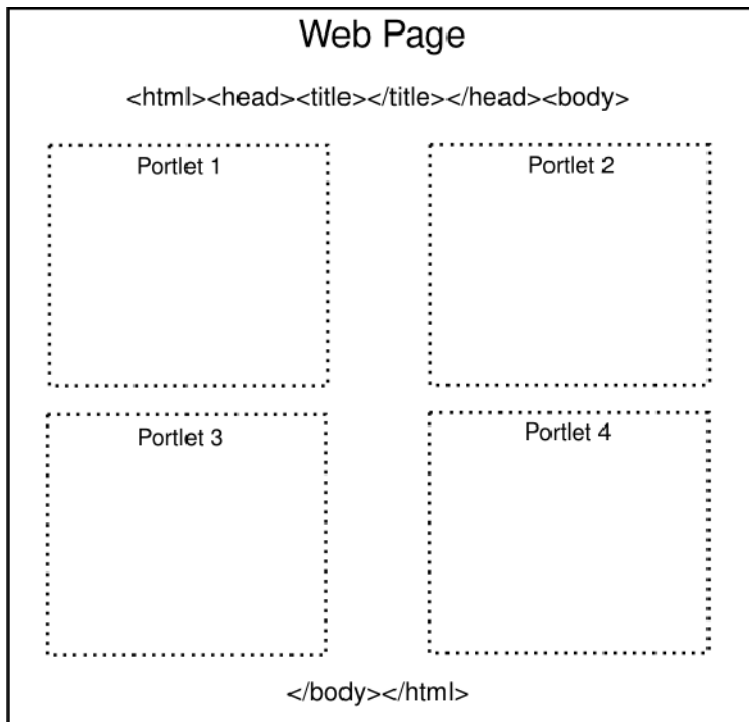


Figure B.1 Portlets appear within the body of a web page and thus are only responsible for producing the markup that makes up their individual section of the page. The portal will compose the page based on the theme, the layout, and the portlets that make up the page.

Having “that stuff” taken care of for you by the portal also gives you freedom in the way you think about your application. If you write a full web application, you have to be responsible for every page and its layout; portions of the page that need to update each other are things you'll have to manually manage. With portlets, you can approach the design of your application differently. Suppose you want a search bar at the top of the page that shows results below it. Searching and displaying results, though, are really two separate functions, aren't they? In a web application, you'd have to compose that page manually to include both functions, which is unnatural. So imagine you've done this and then you show your client what it looks like. What if the client doesn't like it and wants the search moved over to the left side instead? You then have to go back and modify the page styling to move the search element over to the left.

With portlets, you can write a search portlet that takes the search request and then passes it to a separate display portlet as an event, which then displays the result. If your client doesn't like the search at the top, fine: before their eyes, while you're in the midst of the demo, drag that search portlet over to the left and drop it there. No code has to be modified. Why? Because the portal is responsible for aggregating the portlets on the page. You've just saved yourself some work—especially if you can think of other ways of breaking up the functionality of your application.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>



So the heart of any portal implementation is its portlets, because portlets are where the portal's functionality resides. Liferay's core is a portlet container, and the container's job is to manage the portal's pages and to aggregate the set of portlets that are to appear on any particular page and display them properly to the user. All of the features and functionality of your portal application must reside in its portlets.

Portlet applications, like servlet applications, have become a Java standard, which various portal server vendors have implemented. The JSR-168 standard defines the Portlet 1.0 specification, and the JSR-286 standard defines the Portlet 2.0 specification. A JSR-168 or JSR-286 standard portlet should be deployable on any portlet container that supports those standards. Portlets are placed on the page in a certain order by the end user and are served up dynamically by the portal server. And, of course, you can lock down pages so that portlets can only be moved around by administrators.

I can almost hear your next question: what about my favorite framework? I don't want to give that up! Rest easy: you don't have to.

## ***B.2 Portlets, frameworks, and other languages***

Most developers nowadays like to use certain frameworks to develop their applications because those frameworks provide both functionality and structure to a project. For example, Struts enforces the Model-View-Controller design pattern and provides lots of functionality, such as custom tags and validation, that makes it easier for a developer to implement certain standard features. With Liferay, developers are free to use all of the leading frameworks in the Java EE space, including Java Server Faces (JSF), Struts, and Spring. This allows developers familiar with those frameworks to more easily implement portlets and also makes it really easy to port an application which uses those frameworks over to a portlet implementation.

Additionally, Liferay allows for the consuming of PHP and Ruby applications as "portlets," so you do not need to be a Java developer in order to take advantage of Liferay's built-in features (such as user management, communities, page building and content management). You can use the Plugins SDK to deploy your PHP or Ruby application as a portlet, and it will run seamlessly inside of Liferay. Liferay has plenty of examples of this; to see them, check out the Plugins SDK from Liferay's public code repository.

Does your organization make use of any Enterprise Planning (ERP) software that exposes its data via web services? You could write a portlet plugin for Liferay that can consume that data and display it as part of a dashboard page for your users. Do you subscribe to a stock service? You could pull stock quotes from that service and display them on your page, instead of using Liferay's Stocks portlet. Do you have a need to combine the functionality of two or more servlet-based applications on one page? You could make them into portlet plugins and have Liferay display them in whatever layout you want. Do you have existing Struts, Spring MVC, or JSF applications that you want to integrate with your portal? It is a straightforward task to migrate these applications into Liferay, and then they can take advantage of the layout, security, and administration infrastructure that Liferay provides.

Let's drop down from this high level and see what components make up a portlet.

## ***B.3 Understanding the structure of a portlet***

Portlets are web components that process requests and generate content fragments which are then aggregated into a full web page by a portal server. They are written according to an overall standard

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

which is then implemented by multiple portal vendors. In this way, portlets can be cross-platform and deployed on any number of servers that adhere to the standard.

From a deployment standpoint, an application server considers a portlet application to be just another web application that is packaged in a web archive (.war file). Though the file structure inside this archive is the same as that of any web application, a portlet application contains at least one more deployment descriptor than a standard web application. This deployment descriptor is called `portlet.xml` and it is stored in the WEB-INF directory along with the `web.xml` file. This descriptor tells the portlet container what portlets are included in the application, what roles they support, what portlet modes they support, and more.

Portlets, unlike servlets, have more than one phase of operation. In a servlet, the `service()` method processes all requests, often divided between `doGet()` and `doPost()` requests. A portlet has several phases:

- **Render Phase:** runs whenever the portlet needs to re-draw itself on the page
- **Action Phase:** called as the result of an `ActionURL`. This allows the portlet to do some processing to change its state, which is then reflected when the portlet is rendered again.
- **Event Phase:** called as the result of an event being fired. Events can be fired in the Action phase of the portlet and are processed during the Event phase.
- **Resource Serving Phase:** called by the `serveResource` method. This is used for directly serving a particular resource without calling any other part of the lifecycle, and was particularly designed with AJAX in mind.

Liferay 5.0 and greater support the Portlet 2.0 specification (JSR-286). This standard was approved and published on June 12, 2008. It retains backward compatibility with the first version of the specification while also adding new features. This means that any portlet which ran on a Portlet 1.0 container ought also to run on a Portlet 2.0 container. The Event Phase and Resource Serving Phases above are new features added to the specification. The Event phase in particular is a welcome addition to the spec, as it now provides a standard method of doing Inter-Portlet Communication (i.e., allowing two or more portlets to communicate with one another).

Portlets have additional characteristics that make them different from servlets. Portlets have three standard *Portlet Modes*, which indicate the function the portlet is performing:

- View mode—the standard mode of the portlet when it is first displayed. This can consist of one or more screens of functionality within the portlet window.
- Edit mode—a mode in which portlet configuration can be done. For example, a weather portlet can be placed into edit mode to allow a user to enter a zip code that causes the portlet to show the forecast for that location when it is in View mode.
- Help mode—a separate mode that can be used to display help text about the portlet.

Portlets also have *Window States* in which they can be displayed. Portlets can be maximized, minimized, or in regular mode. A maximized portlet takes up the whole portlet area. A minimized portlet shows only its title bar. And a portlet in regular mode can be on a page with several other portlets at the same time.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=642>

There is much more to the portlet specification than has been mentioned here, but hopefully this is a good introduction to the concept of a portlet without getting too detailed. For further details, you can read the specification itself—particularly if you're looking for a cure for insomnia.

C