



The Liferay Cookbook

The Liferay Cookbook

M P Ahmed Hasan

Envisioned to serve as a accelerated yet complete learning experience for Software Engineers on advanced Liferay Portal Development Practices. Compatible to all versions of Liferay including the latest 6.1.1



Dedicated to my father Dr. Peer Mohamed

“O My Lord, Increase My Knowledge”

Disclaimer

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage and retrieval system, without written permission from the author.

Using Liferay Portal 6.1

By, Ahamed Hasan

Copyright ©2013 by mPower Global, Inc.

ISBN <pending>

This work is offered under the following license:

Creative Commons Attribution-Share Alike Unported



You are free:

1. to share – to copy, distribute, and transmit the work
2. to remix – to adapt the work under the following conditions:
 - a) Attribution. You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).
 - b) Share Alike. If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license.

The full version of this license is here: <http://creativecommons.org/licenses/by-sa/3.0>.

Table of Contents

Table of Contents.....	4
About mPower.....	12
About mPower's Liferay Expertise	12
Why Liferay Cookbook is Free?	13
Before Getting Inside	14
Preface	14
Evolution of this book	14
Who should read this book?	16
What is NOT covered in this book?	16
Conventions followed in this book.....	17
How to Approach this book?.....	18
About the author.....	19
System requirements	19
1. Introduction to Portal & Portlets	20
1.1 Definition of a Portal.....	21
<i>1.1.1 Formal Definitions.....</i>	<i>21</i>
<i>1.1.2 Types of Portals</i>	<i>22</i>
1.2 What are Portlets?	23
<i>1.2.1 Servlets and Portlets – Similarities</i>	<i>23</i>
<i>1.2.2 Frameworks and Configuration.....</i>	<i>24</i>
1.3 Portlet Standards and Specification (JSR-286)	25
1.4 Liferay Portal.....	26
<i>1.4.1 Why So Special.....</i>	<i>26</i>
<i>1.4.1 Liferay Enterprise Edition Advantages.....</i>	<i>27</i>
2. Setup and Configuration.....	28
2.1 Install Java 7	28
<i>2.1.1 Are you new to Java?</i>	<i>29</i>
2.2 Install Ant 1.9	29
<i>2.2.1 Maven Instead of Ant</i>	<i>30</i>
2.3 MySQL Database	30
2.4 Install Eclipse Juno.....	31
<i>2.4.1 Install Liferay IDE for Eclipse</i>	<i>32</i>
<i>2.4.2 Liferay Developer Studio</i>	<i>33</i>
<i>2.4.3 Install Subclipse for subversion</i>	<i>33</i>
2.5 Supporting Tools	34
<i>2.5.1 Mozilla Firefox Browser</i>	<i>34</i>
<i>2.5.2 Unzip Utility – WinZip or WinRar</i>	<i>34</i>
<i>2.5.3 SQL Query Browser</i>	<i>35</i>
2.6 Liferay Downloads	35
<i>2.6.1 Tomcat Bundle</i>	<i>35</i>
<i>2.6.2 Liferay Portal Source</i>	<i>36</i>
<i>2.6.3 Portal Source as Eclipse Project</i>	<i>38</i>
<i>2.6.4 Plugins SDK</i>	<i>38</i>
Summary	40
3. Setting the stage with Liferay IDE	41
3.1 New Liferay Server	42
3.2 Basic Configuration.....	44
3.3 Pointing Liferay to MySQL Database	45
<i>3.3.1 Initial Cleanup</i>	<i>47</i>

3.4 New Liferay SDK.....	48
3.5 New Liferay Project	49
3.5.1 Deploying “my-first-portlet” to Server.....	50
3.5.2 Anatomy of “my-first” Portlet	51
3.5.3 Visual Elements of a Portlet.....	52
3.5.4 Liferay Marketplace.....	54
Summary	56
4. Library Management System.....	57
4.1 New Liferay Project	58
4.1.1 Types of portlets with Liferay IDE.....	59
4.1.2 SVN Repository	59
4.2 Establishing Basic Page flow	60
4.3 Anatomy of a Portlet URL	61
4.3.1 Liferay and Search Engine Optimization.....	62
4.4 Creating a Simple Form.....	63
4.4.1 Avoid Hard Coding	64
4.4.2 RenderRequest Vs. ActionRequest	65
4.4.3 URL Formation.....	66
4.4.4 Usage of tag, <portlet:namespace/>	66
4.4.5 Methods of ParamUtil class.....	67
4.4.6 Implicit Objects.....	67
4.5 HTML form to AUI form	68
4.5.1 Setting focus on first field.....	69
4.5.2 Liferay’s Custom JavaScript	69
4.5.3 Form Validation.....	71
4.6 Injecting 3 rd Party Libraries	73
4.6.1 Injecting jQuery library	73
4.6.2 Injecting jQueryUI library and its CSS.....	74
4.6.3 Common JavaScript libraries	75
4.6.4 Making a library available for all portlets.....	75
Summary	76
5. Service Layer and Service Builder	77
5.1 Service Builder Explained.....	78
5.2 Generating our first Service Layer	79
5.2.1 New Liferay Service Builder	79
5.2.2 Files Generated by Service Builder	80
5.2.3 Files to be checked in to SVN.....	81
5.3 Invoking the Service Layer API.....	81
5.3.1 Making our code as a new method.....	82
5.3.2 Auto-generating the primary key	82
5.4 Pulling Data – Use another API	84
5.5 Avoiding Multiple Submits	85
5.5.1 Declarative way	85
5.5.2 Programmatic way.....	86
5.5.3 Redirecting to any other page	86
5.5.4 Attributes of actionRequest	87
5.6 Separating the Business Logic.....	88
5.6.1 Never write business logic in Portlet Class	89
5.6.2 Objects injected by default.....	90
5.7 Service Layer of Portal Source.....	90
5.8 Sharing a custom service layer	91
5.9 Caching to improve performance	91
5.10 “service.xml” DTD Explained.....	92

Summary	94
6. Improving the Book List.....	95
6.1 HTML table to Search Container	96
6.1.1 <i>The “search-container” tags</i>	97
6.1.2 <i>Fixing the Pagination issue.....</i>	98
6.2 Referring to Portal’s TLDs and JARs	99
6.2.1 <i>Other benefits of “liferay-plugin-package.properties”</i>	100
6.3 Adding link to delete book	100
6.4 Adding “Actions” on a Book.....	102
6.5 Editing a book	103
6.5.1 <i>Modifying the book information.....</i>	104
6.5.2 <i>Redirect to another page after update</i>	105
6.5.3 <i>Moving “modifyBook” to LocalServiceImpl Class.....</i>	105
6.5.4 <i>Introducing a new audit field – “modifiedDate”</i>	106
6.6 Viewing book details.....	107
6.6.1 <i>Back URL to list view.....</i>	108
6.6.2 <i>Objects injected by “liferay-theme” tag</i>	109
6.7 Showing details as a popup	109
6.7.1 <i>Implementing the popup.....</i>	110
6.7.2 <i>Removing header from popup</i>	111
6.8 Sortable Columns	111
6.9 Performing an action on a set of items	113
Summary	118
7. Data Retrieval Methods	119
7.1 Order By Clause	120
7.1.1 <i>API Level support for Sorting</i>	120
7.2 Finder Tags.....	121
7.2.1 <i>Adding New Method to Search.....</i>	123
7.2.2 <i>Pros and Cons of Finder tags</i>	125
7.3 Dynamic Query	126
7.3.1 <i>Scenarios while using DynamicQuery</i>	127
7.3.2 <i>Example of Dynamic Query API usage.....</i>	128
7.4 Custom SQL Statements.....	129
7.4.1 <i>Liferay’s use of custom SQL</i>	131
7.5 Some Real world use-cases	131
7.5.1 <i>Expressing Entity relationship through “service.xml”</i>	132
7.5.2 <i>Modeling Database View into service layer</i>	135
7.5.3 <i>Service Layer Dummy Entity.....</i>	136
7.5.4 <i>Connecting to different Data Sources</i>	137
7.5.5 <i>Splitting a large “service.xml” file</i>	138
7.5.6 <i>Invoking a Stored Procedure via Service Layer.....</i>	139
7.5.7 <i>Generating Exception classes</i>	140
7.6 Which mechanism is Best?.....	141
Summary	142
8. Remote Services and Beyond	143
8.1 SOAP / RPC based Web Services	144
8.1.1 <i>Web Services Exposed by Liferay</i>	144
8.1.2 <i>Invoking (Consuming) Existing Web Service</i>	144
8.1.3 <i>Few Important notes around Web Services.....</i>	147
8.1.4 <i>Exposing Web Service for a Custom Plugin.....</i>	148
8.1.5 <i>Consuming (Invoking) custom Web Service.....</i>	149
8.1.6 <i>Generating “library-service-client.jar”</i>	150
8.2 JSON Web Service.....	151

8.2.1 JSON Services Exposed by Liferay	151
8.2.2 JSON Services Exposed by Our Plugin.....	152
8.2.3 Consuming the JSON Service – Legacy Way.....	153
8.2.4 Consuming the JSON Service – Modern Way.....	156
8.3 RESTful Web Services.....	157
8.3.1 What's available by default?.....	158
8.3.2 Programmatically invoking a RESTful Service.....	158
8.3.4 Passing credentials to a RESTful API.....	159
8.3.5 Processing the Response of RESTful API	160
8.4 The “Beyond” Part	161
8.4.1 Saving the Address information	161
8.4.2 Points to Ponder.....	165
8.4.3 ListType and drop-downs.....	167
8.4.4 Here are few challenges for you!	167
8.5 Calling Portal’s JSON Web Service.....	169
8.6 Service Context and its significance.....	171
8.6.1 Service Context Fields.....	172
8.6.2 Creating and Populating a Service Context.....	172
8.6.3 Accessing Service Context data	173
8.6.4 A Simple usage of Service Context – UUID.....	173
Summary	175
9. More Features and API's	176
9.1 Portlet Filters	177
9.2 Implementing Friendly URL's	179
9.2.1 “portlet.xml” verses “liferay-portlet.xml”	181
9.3 Encrypting & Decrypting Portlet Data	182
9.3.1 Encrypting the Data.....	182
9.3.2 Decrypting the Data.....	183
9.3.3 Modifying at the Model Level – Audited Model.....	184
9.3.4 <i>toString()</i> and <i>toXMLString()</i> methods	186
9.4 Enabling Logger at Portlet Level	187
9.4.1 Integrating with Log4J Logger	187
9.4.2 Controlling Logging Levels.....	188
9.5 Portlet Internationalization (I18N)	189
9.5.1 Illustration of Internationalization.....	189
9.5.2 Internationalizing our Library Portlet.....	189
9.5.3 Accessing the entries from Portal Source	191
9.6 Portlet and WCM Marriage	192
9.6.1 Embedding Web Content in a Portlet.....	192
9.6.2 Embedding Portlet inside Web Content.....	193
9.6.3 Accessing Service Layer API's from Web Content	194
9.6.4 Accessing Theme Resources from Our Portlet.....	196
9.7 Making Our Portlet appear in Control Panel	197
9.7.1 More about Control Panel	198
9.7.2 Making our Library unique for every Department.....	198
9.7.3 Introducing “groupId” for LMSBook	199
9.7.4 Liferay and Multitenancy	201
9.7.5 Check with Existing Applications.....	202
9.8 Running Backend Jobs through Quartz.....	202
9.8.1 Implementing the actual Job	203
9.8.2 Contents of “message” object.....	204
9.8.3 Winning of the Portlet Contest.....	205
9.9 Firing Emails from Portlet.....	206
9.10 Getting Direct Access to Database	208

9.10.1 Obtaining a Connection from Pool.....	208
9.10.2 Max Database Connections	208
Summary	209
10. Configuration & Communication	210
10.1 Static Portlet Preferences.....	211
10.1.1 Specifying preferences via “portlet.xml”.....	211
10.1.2 Disabling “success” messages post an Action	212
10.1.3 Accessing a preference inside JSP.....	213
10.2 Dynamic Portlet Preferences	213
10.2.1 Dynamic Preferences through edit mode.....	213
10.2.2 Setting preferences the Liferay Way	216
10.2.3 Archival and Restoration of Preferences (settings)	218
10.2.4 Preferences Entries in Liferay Deployment Descriptor.....	219
10.3 Reading from “properties” files.....	221
10.4 Firing Emails from Portlet – Continued	222
10.4.1 Replacing the values for Subject Line.....	222
10.4.2 Mail Body template – Use ContentUtil	223
10.4.3 Storing email template as a Web Content	224
10.4.4 SMTP settings for actual email	225
10.5 Portlet Data Handlers – Export / Import.....	226
10.5.1 Export / Import on Signin Portlet.....	226
10.5.2 Export / Import on Library Portlet.....	227
10.5.3 Packaging Portlet Data.....	227
10.6 Inter-Portlet Communication.....	232
10.6.1 Server-Side Eventing.....	233
10.6.2 Public Render Parameters (PRP)	237
10.7 Non-Standard ways of IPC.....	239
10.7.1 Portlet Session Sharing	239
10.7.2 Sharing session variable across WARs	240
10.7.3 Obtaining a handle to HttpSession	241
10.7.4 Client-side Eventing	241
10.8 Portlet URL Invocation	242
10.8.1 Using the Tag	242
10.8.2 Using the Util class	243
10.8.3 Namespace problem during ActionURL call	243
10.8.4 Calling Portlet In Another Page	244
10.8.5 Dynamic URL Formation thru JavaScript.....	245
10.9 Customizing Portlet Based on Query String.....	246
10.9.1 Another Classical Usecase	247
Summary	248
11. Liferay Frameworks – Part 1	249
11.1 Security and Permissions.....	250
11.1.1 Overview of Portlet 2.0 Security	250
11.1.2 Overview of Liferay’s Permission System.....	251
11.1.3 Liferay’s Permissioning Algorithm	252
11.2 Portlet Permissions	253
11.2.1 Understanding the backend tables	253
11.2.2 BitWise permissioning unleashed	254
11.2.3 Adding New Portlet Permissions	256
11.3 More Security Layers and Custom Utility.....	259
11.3.1 Level-2 Security for Protecting Resources.....	259
11.3.2 Level-3 Security Protection for Local / Remote API’s.....	260
11.3.3 Writing a Custom Permission Class	260

11.4 Model Permissions	262
11.4.1 Defining and Registering Model Permissions.....	262
11.4.2 Setting Permissions While Adding Book.....	263
11.4.3 UI to Control Permissions of Every Book in List.....	264
11.4.4 Showing Only those Books with VIEW Permission.....	265
11.4.5 Permissions filtering and “filterFindBy” Methods.....	265
11.4.6 Hiding an Action Based on Permission.....	266
11.4.7 Removing a Resource During Delete.....	267
11.4.8 Exploring Permissions from Admin UI.....	267
11.5 Liferay Asset Framework.....	269
11.5.1 Library Book as an Asset	269
11.5.2 Resources Verses Assets.....	270
11.5.3 Summary of Parameters passed to updateEntry	271
11.5.4 Proper Deletion of an Asset	272
11.6 Attaching Tags and Categories.....	272
11.6.1 Changes to the User Interface.....	273
11.6.2 How the information is stored?.....	273
11.6.3 Showing Summary in Book Details Page.....	274
11.7 Publishing Assets (Books) through Asset Publisher.....	275
11.7.1 Configuring Asset Publisher Portlet	275
11.7.2 Asset Publisher Display – Basic Infrastructure	276
11.7.3 Actual Implementation of Showing the Books.....	277
11.8 Asset Publisher for Books – Finishing Touches.....	279
11.8.1 Improving abstract view and full view pages.....	279
11.8.2 Making book “addable” from Asset Publisher.....	280
11.8.3 Enabling the “Print” option	280
11.8.4 Links to comments.....	281
11.8.5 New Friendly URL through Asset Publisher.....	281
Summary	282
12. Liferay Frameworks – Part 2	283
12.1 Liferay Storage Framework.....	284
12.1.1 Uploading Cover Page of the Book	284
12.1.2 Image upload – Backend Impacts	287
12.1.3 Jackrabbit and Document Library	287
12.2 MVCPortlet and Large Applications.....	288
12.2.1 The two main reasons	288
12.2.2 Moving our code to a new File	289
12.3 Showing Book Cover in the List Page	290
12.3.1 Direct fetching of an Image.....	291
12.3.2 Serving the Image through serveResource.....	291
12.3.3 Scaling the image as a Thumbnail	293
12.4 File Processing and Liferay	294
12.4.1 Simple Storage to DLStore.....	294
12.4.2 Retrieving a file from DLFileStore	297
12.4.3 Data Backup Strategy on a Real Server.....	298
12.5 Server Side Validation.....	300
12.5.1 Client Side validation to check file Extension.....	300
12.5.2 Server-Side Validation	300
12.5.3 Various types of “aui:input” fields	301
12.6 Custom Fields Framework	303
12.6.1 Control Panel and Custom Fields.....	303
12.6.2 Impact to the backend tables.....	304
12.6.2 Activities for You.....	305
12.7 Enabling Custom Fields for Library Portlet	306

12.7.1 Defining the custom fields.....	306
12.7.2 Code Level changes.....	308
Summary	309
13. Liferay Collaboration Frameworks	310
13.1 Liferay Subscription Framework.....	311
13.1.1 Knowing the basics	311
13.1.2 Enabling Subscription for Library Portlet.....	311
13.2 Liferay and AJAX using AlloyUI	314
13.2.1 Getting the ground ready for AJAX	314
13.2.2 AJAX Loading Library Welcome Message.....	317
13.2.3 Liferay JavaScript Loading Sequence.....	318
13.3 Commenting and Rating Framework	319
13.3.1 Enabling Comments on a Library Book.....	319
13.3.2 Liferay's Rating System.....	322
13.3.3 Stand-alone Rating System.....	322
13.3.4 Inserting Social Bookmarks	324
13.4 Social Activity Framework.....	325
13.4.1 Transformation to a Social Activity	325
13.4.2 Social Activity Intrepreter and Tracking.....	326
13.5 Social Equity Framework	328
13.5.1 Understanding "Community Equity" Model	329
13.5.2 Measuring Social Activity.....	330
13.5.3 Enabling Social Activity for Library Portlet.....	331
13.5.4 User Statistics and Top Users Portlets	333
13.5.5 Group Statistics Portlet.....	334
13.6 Business Process (BPM) Framework	335
13.6.1 Liferay and Workflow.....	335
13.6.2 Our Library and Workflow.....	336
13.6.3 Database and UI changes.....	338
13.6.4 Custom Handling of workflow events	340
Summary	341
14. A Tour of Advanced API's	342
14.1 Portlet Messaging – MessageBus	343
14.1.1 Integrating Subscriptions to MessageBus.....	343
14.1.2 MessageBus Usage Situations.....	346
14.2 Indexing and Search	348
14.2.1 Enabling the Indexer Class	348
14.2.2 Indexing during Book insertion.....	350
14.2.3 Writing new Search Functionality	353
14.3 Advanced Search Features	355
14.3.1 Search with multiple fields.....	356
14.3.2 Faceted Search.....	358
14.3.3 Open Search Integration	360
14.3.4 Fixing the URL Issue.....	362
14.4 Device Detection API.....	363
14.4.1 Insalling the Device Detection plugin.....	364
14.4.2 Using the Device API	364
14.4.3 Device Capabilities	364
14.5 Creating a Custom Taglib.....	365
14.5.1 Creating a custom tag for Library Portlet	366
14.6 More on Portlet + Web Content Integration.....	369
14.7 Form Navigator Tag	372
Summary	374

Appendix	375
A – Liferay Portlet XML Entries.....	375
A.1 “action-timeout”	375
A.2 “add-default-resource”	375
A.3 “ajaxable”	375
A.4 “atom-collection-adapter”	375
A.5 “autopropagated-parameters”	376
A.6 “facebook-integration”	376
A.7 “include”	376
A.8 “layout-cacheable”	376
A.9 “maximize-edit”	376
A.10 “parent-struts-path”	377
A.11 “permission-propagator”.....	377
A.12 “poller-processor-class”.....	377
A.13 “pop-message-listener-class”	377
A.14 “pop-up-print”	377
A.15 “portlet-layout-listener-class”	377
A.16 “portlet-url-class”	377
A.16 “private-request-attributes”.....	378
A.17 “private-session-attributes”	378
A.18 “remoteable”	378
A.19 “render-timeout”.....	378
A.20 “render-weight”	378
A.21 “restore-current-view”	378
A.22 “show-portlet-access-denied”	378
A.23 “show-portlet-inactive”	379
A.24 “social-request-interpreter-class”	379
A.25 “struts-path”	379
A.26 “system”	379
A.27 “url-encoder-class”	379
A.28 “use-default-template”	379
A.29 “user-principal-strategy”	380
A.30 “virtual-path”	380
A.31 “webdav-storage-class”	380
A.32 “webdav-storage-token”	380
A.33 “xml-rpc-method-class”	380

About mPower

mPower is an IT services company specialized in the field of Liferay and headquartered in Bangalore, India. This company was incorporated in the year 2004 by 4 committed young individuals and now has offices across the globe - India, USA, Middle East and Malaysia.

mPower has a decade long story of delighting customers through robust, efficient and cost effective IT solutions. mPower started its IT journey by empowering fresh graduates from tier 2 and tier 3 cities who lacked IT and communication skills to enter into the competitive corporate IT world. Within 2 years of incorporation we began to excel in different verticals like IT services, Product Development and consulting! mPower now specializes in the field of web portal development, and stands out as the leader in Open Source technology.

About mPower's Liferay Expertise

When it comes to Liferay, mPower is a pioneer and is a name to reckon with. We achieved success over the last 8 years through large-scale liferay implementation. Starting from Liferay 3x version mPower has built unmatched expertise in the technology by nurturing a remarkable team of liferay experts through which the company obtained many large accounts like Cisco, Xerox, Capgemini, and Bosch. Successive integrations with enterprise solutions like SAP, Oracle Service Bus, Novell Access Manager, Oracle Financials and an outstanding track records of migrations from legacy systems confirms mPower's position as leaders in high end and complex delivery space.

mPowers expertise is endorsed by various international recognitions including the Lockheed Martin Innovation Award and International Portlet Contest Award. These stand testimony to its passion, innovation and commitment to excellence. With mPower's leadership every customer gets a delighting experience, cost effective solutions, outstanding quality and on time delivery.

“Think Liferay. Think mPower”

Why Liferay Cookbook is Free?

14 expert software engineers', 8 years of Liferay Architecture and 5 months of rigorous content writing comes to you for free!! Wonder Why?

The Author, Ahamed Hasan expressed his dream that this book should be the 'Most referred Liferay book in the world'. Hence, he approached me to make this book freely available to all without any dollar barrier.

However, it was a big challenge for the management to agree to this free idea as the cost incurred to launch this book was huge. After many discussions and debates I considered this noble idea of "Making it free" to the mass audience. This idea was further endorsed by Mr. Brian Chan the founder of Liferay.Inc when I met him in USA to discuss the same, together the decision was finalized keeping in mind the larger benefit of the liferay Community.

Hence, "The liferay Cookbook – Free for life"!

Happy Reading



Akbar Ali
CEO – mPower Global



The first copy of "The Liferay Cookbook – Free for Life" was handed over to Mr. Brian Chan – Founder and Chief Architect of Liferay.Inc by Mr. Akbar Ali CEO of mPower at the Liferay Head Quarters in LA, USA

Before Getting Inside

Preface

January 15, 2013 – The day I wrote the preface for my first book on Liferay is the most memorable day in my life. Reason – this is the same day I got married exactly seven years back in 2006. Immediately two days after our simple marriage at my wife's native place, I took her back to Bangalore to report back to work, as those were very challenging days in my life – managing the business at mPower Global which was still at its infancy stages, managing a big project with one of our clients where we were implementing Liferay for the first time ever and managing my young wife who had to cope up with my late working hours.

This is how my encounter with Liferay started seven years back. It was never planned. It all happened as an accident. I developed a component very much similar to Hibernate when I was still working with General Electric and built a Struts kind of web framework around that. I nicknamed it as "**nCash**". It was a big thing at that time when no major frameworks existed. When I was made to work on Liferay along with a bunch of young engineers whom I've trained on our first project on Liferay, I started loving Liferay so much that I had to make my own framework rest in peace eternally. Our first project was a Travel Portal for ABB built on a very early version of Liferay 3.2. Even Liferay was at a very nascent stage at that time. But I could foresee a big future for this fantabulous product. Now continue to read to see the history of this book.

Evolution of this book

The ABB project was my turning point that has kept me glued to Liferay even after seven long years. I've seen Liferay evolving and improving version after version and gaining market acceptance. Being an open source product, Liferay always lacked good documentation. Thus I thought why not come up with some tool or a product that will simplify the Liferay development and make life easier for new adaptors of this great framework. The thought further crystalized and it gave rise to one of the great products built on top of Liferay called "**mPire**".

Though this tool positioned our company as one of the leaders in this space, yet we were not able to fully monetize it for various reasons. Apart from securing a position in the Liferay's ecosystem, this tool got us the prestigious Lockheed Martin Innovation award. I personally travelled all the way to Dalian in China during March 2008 to meet Brian Chan, the Chief Architect of Liferay to show case this tool. He was simply amazed and gave lots of appreciation. No doubt, he is one of most amazing guys I've ever met in my life. Yes, a person who build such a massive product and knows every single line of its code can't be an ordinary human being. He has to be extraordinary. This is the picture myself and Chan took together at Liferay Dalian office. It is my honor.



Months passed and I became too busy working on various client projects. In between I had to do lots of trainings on Liferay to make both our ends meet. These trainings gave me a much stronger understanding of Liferay. The experience I got executing various projects gave me an opportunity to do these training with great comfort and confidence. I trained more than thousand individuals both within mPower Global and many big corporates across the globe (around 40) including USA, Middle East, India and Europe. I trained people from companies like Capgemini, Intuit and Grass Roots (UK). Many of these companies engaged me more than once to train their IT staff / developers on Liferay.

I always wanted to convey all my training material in the form of a book. But writing a book on Liferay still remained a dream for me as I was busy with multiple projects at the same time. The training requirements kept growing but many times I had to forgo the opportunities, as I was either busy on some consulting role with a customer or working on a Liferay project. Until the middle of 2012, I was never able to give complete attention towards writing a book on Liferay. This is the time I became serious about writing a book on Liferay that is based on my experience of working on this great framework for the past seven years. I never wanted the knowledge to stay with me alone.

Another major reason for the book is that formal training on Liferay has always remained a distant dream for many people in the developing and third world countries. It was my desire to make the Liferay learning affordable and at the same time enjoyable. I found huge scope for a book that covers the basics of Liferay as well as more advanced topics in a step by step manner. Thirdly, the huge gap between the demand and supply in the market for people with Liferay expertise, further encouraged me to write this book. Liferay popularity and acceptance has been gaining momentum each passing day amongst the CIO's of major companies, yet there aren't enough skilled people to work on the Liferay projects. This book is a humble attempt to bridge this demand vs. supply gap. My dream is to make this book part of the computer science curriculum in all universities worldwide and make young people learn this splendid technology right from their school days, enabling them to be prepared for the tough challenges of the IT industry.

This is how the seed was sown for this book. I engaged all the fresh interns who come to mPower Global to write the initial draft. In between when I got time out my busy schedule, I used to personally check the contents and correctness of the examples. Finally when I came back from a client engagement of two months in November 2012, I sat at a stretch to write the contents of this book myself and make it available to you. I know this is just the beginning and it is a long way to go to improvise the chapters in this book. I am sure I will be able to do that with all your feedback and support.

You're most welcome to send your valuable suggestions and feedback to my email, ahmed@mpowerglobal.com.

Who should read this book?

This book is targeted at the following categories of people.

- Those who are new to Liferay with some knowledge of Java and J2ee.
- Those who are new to Java itself who would like to get an understanding of Liferay and are willing to go back and learn Java after reading this book.
- Those who are new to programming itself and are serious about pursuing a career in a technology that is ruling the portal world today.

Whatever be the category, I tried my level best to make this book as lucid as possible making it easy for a novice programmer while at the same time keeping it interesting for a seasoned programmer. While I strongly encourage you to read other books on Liferay, with this book I have aimed to give you a thorough understanding of all the underlying concepts that form the building blocks of Liferay Portlet development. I am confident, if you master all the topics in this book, you will be on your way to becoming an expert in Liferay. This book can serve as a good reference for many of the Portlet related topics.

What is NOT covered in this book?

It is always very important to set the right expectations. The same rule applies here as well. Before I elaborate what is covered in this book, I will have to state what is NOT covered in this book. This does not mean that you will not find a mention about these topics in this book. It just implies that these topics lie out of the scope of this book and hence will not be covered in great detail. You can always refer to below mentioned sources to understand these topics better.

- **Liferay Portal Administration** – Liferay's official documentation is the best source of information on this topic. Bookmark this link, <http://bit.ly/Xr4v3U>.
- **Liferay Theme and Layout Development** – this is another great topic in and of itself that would require a separate book to do justice to it. I personally hope to make available a resource on this subject. For the time being, you can always refer

to the resources that are available out there to know more about Theme and Layout development.

- **Liferay Ext and Hooks development** – this topic is all about customizing the Liferay's core functionality through extension (Ext) plugins and Hook plugins. While hooks can be hot deployed and un-deployed on the fly, the changes applied by an Ext plugin can't be reversed (un-done) so easily unless and until you manually cleanup the server folders where Liferay is deployed. You can get a hang on Ext and Hook development again from the Liferay's official documentation. Read chapters 7 and 8 from <http://bit.ly/14HgAWN>.
- **Liferay Server / Deployment Administration** – this is a different animal altogether. This topic covers the deployment of the production Liferay server either as a single node or a cluster of nodes. As for deploying Liferay on a standard server or on the cloud (EC2 or Rackspace), this book speaks nothing about this topic, as the normal developers need not have to really worry about this subject.

Now let me come to what is really covered in this book – **Portlet development**. Yes pure Portlet development based on the Liferay's MVC framework that is very much like the younger sister of Struts framework. But the concepts you learn in this book are very important. Once you understand these concepts you can apply them to any MVC framework of your choice – Struts, JSF, Spring MVC among others. Yet the purpose of this book is NOT to teach you any of these present times MVC frameworks. Rather The Liferay MVC framework is chosen because it is default with Liferay and it supports all other frameworks and libraries.

For a fresher it is always easy to start learning Liferay through this simple framework rather than taking the complicated route of learning a MVC framework first and then moving on to learn Liferay. This will be a redundant exercise. All you need is some basic knowledge of Java and JSP in order to get started with learning Liferay. I should also mention here that Liferay is a technology loved by developers because of its elegance and simplicity. It has its own coding standards and best practices. In the course of reading this book you will discover those subtle secrets.

The first few chapters in this book talk about the initial setup and configuration. From chapter 4 onwards we'll be taking up a hypothetical application – “**Library Management System**”. Starting with a basic screen of this application we'll keep building complexity as we move forward. Though you can randomly jump to any chapter in the book, you should know that certain chapters are dependent on the ones preceding it. Thus I would suggest that you read the book in its entirety atleast once to get a better grasp

Conventions followed in this book

Being a techie whose main job is to write code, initially I wanted this book to be drafted by a professional technical writer. I tried a couple of them and then realized that it was not an easy task. But unfortunately, I just couldn't convey seven years of Liferay expertise to them in a matter of days and make them write a book on Liferay. Though the contents might be retained, I feared losing the originality and natural

flavor. Not wanting to take chances, I ended up writing the contents myself without relying on technical writers. Hence I may not have used many so-called conventions in this book. The biggest convention in this book is its originality and simplicity. Just to keep the publishers happy though, I have used a few conventions in this book.

- The main text is “Times New Roman” 12 font size. I selected this font because it is good both on print and on screen. It also consumes less space.
- The code examples are usually copied and pasted to this book from Eclipse IDE. The code copied from Eclipse IDE will appear in “Monaco font size 10” and it will be mostly syntax highlighted
-
- Wherever there is no copy/paste, I’ve used “courier font size 11” for any code examples.
- The references to folder names are always made bold and double quoted for e.g. “**liferay-portal-6.1.1-ce-ga2**” in order to make them prominent.
- Whenever there is a need to express the command prompt, a dark grey background is given as you find in `ant -version`

Couple of more conventions, I would like to highlight here. One is the **interactivity** – I have tried my best to make this book very interactive. This is the reason you will find over usage of words “I” (myself) and “you” (yourself, the reader). While reading the book, I want to make the reading experience very lively and mimic the real world experience of myself standing in front of you and conducting a course which you are attending. The other convention is usage of **plain English**. One of the greatest objectives of this book is to teach a great subject to people far and beyond the English speaking countries, viz. Indians, Chinese, Germans, Arabs, Africans and Malaysians. Hence, I’ve kept the language very plain and simple that can be understood by even a tenth grade kid whose mother tongue is not English.

How to Approach this book?

Before I conclude this chapter and move on to the next, I would like to give you few tips on approaching this book. This book is meant primarily as an e-book that can be read online using a computer or a smart phone. You can keep this book open using your favorite PDF reader and carry out all the examples here on the same computer as if you’re reading a manual. You’re hereby strongly encouraged to avoid taking the full printout of this book, giving due respect to our environment, unless and until this is inevitable.

About the author

This is the quick background of me. You can also refer to my Linkedin profile page <http://linkd.in/Yf5nrb>.

Name	Ahmed Hasan
Education	M.S (Software Systems) from BITS, Pilani, India
Current Role	Chief Architect at mPower Global Inc.
Founded	mPower Global Inc. by end of year 2005
Liferay experience	Seven+ years
Worked for	Aztec and General Electric
Family	Wife and three young daughters
Date of Birth	21-Oct-1973

System requirements

Nothing great, you can execute all that is explained in this book if you have a normal laptop / desktop with a 2 GB RAM and minimum hard disk space. I would recommend though, that you upgrade your system to 4 GB as you will be running many applications simultaneously. Since Liferay is based on Java which is platform independent, you literally can use any operating system – Windows, Linux or MacOS. I wrote this book and all the code presented using my MacBook. If you use Windows, you might find minor differences in appearance from the book. But you can very comfortably ignore these differences. As a Java developer, it shouldn't matter what OS you use.

1. Introduction to Portal & Portlets

This Chapter Covers

- Definition of a Portal
 - What are Portlets?
 - Portlet Standards and Specifications
 - Liferay Portal
-

This chapter is a very basic introduction to Portal, Portlets and Liferay. I've kept this chapter as short as possible as the contents are generic in nature. We start this chapter with a broader understanding of a web portal. Then we move on to discuss Portlets and the specifications that standardize them. In the final section of this chapter, we get into the details of Liferay Portal and the factors that sets it apart from its competitors.

1.1 Definition of a Portal

This is something not really new for you. If you're regularly surfing the Internet for the past many years, then probably you would have come across portals of all kinds. You may be using some of them on a daily basis to check the news, to know the stock price, to collaborate with your friends in realtime, etc. Having said this, the intent of this section is to give you a more formal introduction to portals and the frameworks that are used to develop / assemble them.

Portal is a term, generally synonymous with gateway. A portal is a web application that commonly provides personalization, single sign-on, content aggregation from different sources, and hosts the presentation layer of information systems. Aggregation is the act of integrating content from different sources within a web page. The idea of a portal is to collect information from different sources and create a single point of access to information - a library of categorized and personalized content. It is very much the idea of a personalized filter into the web.

1.1.1 Formal Definitions

A web portal or public portal refers to a web site or service that offers a broad array of resources such as e-mail, forums, search engines, and online shopping malls. An enterprise portal is a Web-based interface for users of enterprise applications. Enterprise portals also provide access to enterprise information such as corporate databases, applications (including Web applications), and systems. The key features of a portal are – security, diverse data access, transactions, simple and federated search, content publishing and personalization.

You have already encountered a comprehensive example of a web portal if you have used Yahoo! – one of the world's popular and widely used portals. This is what exactly any portal does. It provides a single point of entry to widely distributed information on the web, and it offers a unified way to access that diverse information. Some portals allow users to decide what they want to display on their portal pages. In many of these cases, the portal designer will customize the user's page contents and generate them dynamically. Regardless of whether the customization is done by the portal designer or the user, portals provide an easy way to configure desired content on a personal web page. On top of this, portals provide a consistent look and feel. Users can take advantage of diverse applications in the same manner, making it easy for them to access information from various sources.

Now let's look at the formal definition of the term "portal". Wikipedia the popular free encyclopedia provides the following definition:

A web portal is a site that provides a single function via a web page or site. Web portals often function as a point of access to information on the World Wide Web. Portals present information from diverse sources in a unified way. Apart from the search engine standard, web portals offer other services such as email, news, stock prices, infotainment, and other features. Portals provide a way for enterprises to provide a consistent look and feel with access control and procedures for multiple applications, which otherwise would have been different

entities altogether.

The Wikipedia definition is probably the most comprehensive one. As it states, a web portal gives a user access to contents generated by diverse applications in a unified way. Here's another definition from Sun Microsystems, which defines "portal" in its Java Portlet Specifications (JSR 286) as follows:

A portal is a web based application that commonly provides personalization, authentication, [and] content aggregation from different sources and hosts the presentation layer of information systems.

This definition states that a portal is a kind of web application that aggregates content from different sources, viz. web sites or web applications. The content generated by these web sites can be static or dynamic. For example, a travel related portal might generate a web page that aggregates and presents information from several travel web sites. If a user decides to gather further information from one of the displayed web sites, he can simply visit that web site by navigating to it from the portal page. After doing that, he can return to the portal page with ease and continue navigating to the other web sites if desired.

Different web sites offer several other definitions, all of which describe portals as user- customizable web sites that serve as gateways to diversified content arising from various sources. However, these definitions neglect to describe an important feature of today's portals – they provide collaboration among their users. Most of the Web 2.0 features such as wikis; blogs, video sharing, and even social networking are available on today's portals. Usually, these new types of portals give users tools and applications to create sites for social networking and collaboration. The Liferay portal that we'll explore in this book falls into this new category of portals.

1.1.2 Types of Portals

Portals can be classified into many different types based on their applicability and the need they are addressing. The types are:

- **Personal Portals** – the ones used by an individual
- **Academic Portals** – used by schools and universities \
- **Government and Regional Portals** – owned by government and local bodies
- **Domain Specific Portals** – like travel, sports, news, shopping, etc.
- **Collaboration Portals** – social networking and collaboration between users
- **Business-to-Business Portals** – one that connects businesses
- **Business-to-Customers Portals** – one that connects businesses with customers

The above list is not an exhaustive one. It is just indicative. Everyday new types of portal are emerging to address diverse needs.

1.2 What are Portlets?

Portlets are pluggable user interface software components that are managed and displayed in a web portal. Portlets produce fragments of markup code that are aggregated into a portal. If you have already worked with Servlets, then you should be aware that there are many similarities between them and Portlets. Similar to servlets, portlets are web components that are deployed inside of a container and generate dynamic content. On the technical side, a Portlet is a class that implements the `javax.portlet.Portlet` interface and is packaged and deployed as a WAR file inside of a Portlet container (Portal Server). The JSR-168 Specification defines a portlet as:

Portlets are web components – like servlets – specifically designed to be aggregated in the context of a composite page. Usually, many portlets are invoked to in the single request of a portal page. Each portlet produces a fragment of markup that is combined with the markup of other portlets, all within the portal page markup.

1.2.1 Servlets and Portlets – Similarities

Making the switch to portlet development is easier than you think. That's largely because the writers of the initial specification (JSR-168) used the Servlet specification as the basis for their documentation. Thus portlets work very similarly to servlets except with additional modifications to account for the change in technology. Here are a few things you should know when preparing to make the switch to portlet development:

1. Both servlets and portlets use the “`web.xml`” deployment descriptor. However, portlets also have an additional deployment descriptor, `portlet.xml`, which must declare all of the portlets that will be used by the portlet container.
2. Instead of `HttpServletRequest` and `HttpServletResponse` objects, portlets expect either an `ActionRequest` or `RenderRequest` object and `ActionResponse` or `RenderResponse` objects. There are limitations in the objects that portlets manage such as not having direct access to original servlet request parameters (JSR 168). But check your portlet specification and portlet container implementation to verify which features may or may not be available.
3. Portlets have a `Context` interface that allows developers to access the Portlet application under which a portlet is running. This is similar to the servlet's `context` interface.
4. Portlets have their own config object, `PortletConfig`, which can be used to access properties within the `portlet.xml`.
5. Portlets use the concept of portlet modes that allows developers to provide varying levels of end-user control for viewing, editing, or accessing help.
6. Portlets have window states of normal, maximized or minimized that determine how much a portlet can be seen within the portal window.
7. Listeners and Wrappers used by portlets are still declared within the “`web.xml`” deployment descriptor.
8. Sun provides a Portlet Tag library that is used for accessing portlet features from markup such as Java Server Pages or Velocity templates.

-
9. Portlets can expect to be invoked multiple times by the portlet container to update its area of the display. This means that rendering methods must contain idempotent code. When performing actions on a portlet, portlets should expect that the action will be performed only once and therefore should be non-idempotent.

Check the portlet specification that you're using to see the complete list of features supported. Also, review your portal documentation to see how closely the portlet container follows the specification.

In Summary, Portlets are **similar** to servlets, in that:

- A specialized container manages Portlets.
- Portlets generate dynamic content.
- The container manages a portlet's life cycle.
- Portlets interact with web client via a request/response paradigm.

Portlets are **different** from servlets, in that:

- Portlets only generate markup fragments, not complete documents.
- Portlets are not directly URL addressable. You can't send somebody URL of a portlet. You can send him the URL of the page containing a portlet.
- Portlets cannot generate arbitrary content, since the content generated by a portlet is going to be part of portal page. If a portal server is asking for html/text, then all portlets should generate html/text content. On the other hand, if the portal server is asking for WML, then each portlet should generate WML content.

1.2.2 Frameworks and Configuration

Additionally, if you're accustomed to using development frameworks such as Struts or Spring, you'll find these have customized class and tag libraries to allow you to continue to use their features within portlets. Usually, the libraries will have portlet class names and methods that correspond exactly to their servlet implementation. For instance, Spring provides a servlet front controller named **DispatcherServlet** and a portlet front controller similarly named DispatcherPortlet. As I mentioned in the beginning of this book, we'll use Liferay's MVC Portlet framework in this book to develop our Portlet.

Also, additional configuration files are usually required by Liferay when using portlets in addition to those used by frameworks. For instance, Liferay expects developers to define a "**liferay-portlet.xml**" file that declares the name of portlets to be used by Liferay. Another example is the Spring framework that expects the declaration of portlet specific configuration file that declares additional beans and properties to be used by the Spring controller when initializing the portlet.

1.3 Portlet Standards and Specification (JSR-286)

Portlet standards are intended to enable software developers to create portlets that can be plugged into any portal supporting the standards. The purpose of the Web Services for Remote Portlets protocol (*WSRP*) is to provide a web services standard that allows for the "plug-n-play" of remote running portlets from disparate sources. Many sites allow registered users to personalize their view of the website by turning on or off portions of the webpage, or by adding or deleting features. This is sometimes accomplished by a set of portlets that together form the portal.

The Java Portlet Specification (JSR-168 & JSR-286) enables interoperability for portlets between different web portals. This specification defines a set of APIs for interaction between the portlet container and the portlet addressing the areas of personalization, presentation and security. The first one is called as Portlet 1.0 and the latest one is called as Portlet 2.0 Specification. Liferay is the first portal server / Portlet container to completely support the JSR-286 specification. Liferay has also added many new features / extensions on top of the standard specifications. The specification defines a contract between the portlet container and portlets and provides a convenient programming model for Java portlet developers.

The Java Portlet Specification achieves interoperability among portlets and portals by defining the APIs for portlets that are defined by JSR 168 and JSR 286. By adhering to the standards, you can build portlets that can run in portals, irrespective of their vendors. Portlets are web-based components that enable integration between applications and portals and thus delivery of applications on portals. The JSR-286 has evolved, now allowing users to implement most of the use cases without the need to have vendor extensions. Generally speaking, the JSR-286 provides users with events and public render parameters, so that the users can build larger composite applications out of the portlets and reuse the portlets in different scenarios. It also allows the users to serve resources directly through the portlet, for example, AJAX and JSON data serving.

JSR-286 is the Java Portlet specification v2.0 as developed under the JCP and created in alignment with the updated version 2.0 of WSRP. It was developed to improve on the short-comings on version 1.0 of the specification, JSR-168. Some of its major features include

- Inter-Portlet Communication through events and public render parameters
- Serving dynamically generated resources directly through portlets
- Serving AJAX or JSON data directly through portlets
- Introduction of Portlet filters and listeners

Liferay portal is designed to deploy portlets that adhere to the Portlet API (JSR-286). Many useful portlets are bundled with the portal, for example, Image Gallery, Document Library, Journal, Calendar, Message Boards, Manage Pages, Communities, and so on. Before customizing these portlets in sync with the Liferay portal, we are going to work with the JSR-286 portlets.

1.4 Liferay Portal

Liferay Portal is a free and open source enterprise portal written in Java and distributed under the GNU Lesser General Public License and proprietary licenses. It is primarily used to power corporate intranets and extranets. Liferay Portal allows users to set up features common to websites. It is fundamentally constructed of functional units called portlets. Liferay is sometimes described as a content management framework or a web application framework. Liferay's support for plugins extends into multiple programming languages, including support for PHP and Ruby portlets.

Although Liferay offers a sophisticated programming interface for developers, no programming skills are required for basic website installation and administration. Liferay Portal is Java based and runs on any computing platform capable of running the Java Runtime Environment and an application server. Liferay is available bundled with a servlet container such as Apache Tomcat so it is easy to use and maintain. Liferay ships with over a hundred pre-built applications under various categories.

1.4.1 Why So Special

Liferay Portal is a JSR-286 enterprise portal which includes a suite of applications (e.g. Content Management System, blogs, instant messaging, message boards, etc.). It is distributed in two different editions:

- **Liferay Portal Community Edition** – a version with the latest features and support through the active community.
- **Liferay Portal Enterprise Edition** – a commercial offering that includes services including updates and full support. This release goes through additional quality assurance cycles and is usually available around 1 or 2 months after the Community Edition and comes under a non-free license.

Liferay also provides a collaboration suite based on the Liferay platform called Liferay Social Office - a social collaboration suite for enterprises. Liferay comes with certain portlets pre-installed. These comprise the core functionality of the portal system. They are listed in the following table:

Table of Portlets / features that come bundled with Liferay	
Tags and Categories	Document and Image management
Document Library / Recent Documents	CMIS integrations
WebDAV integration	Website tools
Web Form Builder	Breadcrumbs and Navigation
Page Ratings and Flagging	User Directory
LDAP Integration	Software Catalog
Blogs and blog aggregation	Calendar and Events
Chat and Forums	Mail and Personal Messaging
Polls and Polls display	Wiki and Knowledge base
Social Equity	Themes and Layouts
Web Content & Site Map	Asset Publishing

1.4.1 Liferay Enterprise Edition Advantages

Liferay Portal Enterprise Edition is the best way for demanding enterprises to take advantage of Liferay's market-leading innovation with the reduced risk exposure and long-term stability of professionally supported software. Your subscription for the Liferay Enterprise Edition entitles you to regular service packs, a commercial service level agreement, and much more. There are four striking features that make the Liferay Enterprise Edition far ahead of the corresponding Community Edition.

Reliability – Liferay Portal Enterprise Edition (EE) is configured for enterprise environments that require support for redundancy, failover and load balancing to ensure maximum uptime of your Liferay-based solution. Exclusive EE features like improved memory management and Terracotta support allows you to scale your system efficiently as your user base grows.

Security – Liferay Portal was benchmarked as one of the market's most secure portal platforms with its use of industry standard, government-grade encryption technologies. Subscribers to Liferay Portal EE benefit from additional security patches that are discovered by the customer network delivered via regular service packs. For browser-level security, Liferay Portal EE implements the Top 10 recommended best practices published by the OWASP organization. You can see the details at <http://bit.ly/WB8TjV>.

Performance – Liferay Portal Enterprise Edition is tuned against all major application servers and databases for optimal performance under load. Custom-configured to meet the mission-critical speed and scalability needs of large enterprise deployments, EE offers all the essential capabilities of the core Liferay product in production-ready form.

Stability – To ensure a stable product for enterprise customers, no new features are added to Liferay Portal Enterprise Edition, eliminating the risk of new defects being introduced by new features. Each Liferay Portal EE service pack is thoroughly tested using additional quality assurance processes to ensure that the product becomes more stable with each release.

For the purpose of learning Liferay, in this book, we'll use Liferay Community Edition that comes bundled with Tomcat. You have all liberty to try out the examples presented in this book, on the Enterprise Edition as well, without any problems. You can download the trial version of Liferay Enterprise Edition from this link, <http://www.liferay.com/products/liferay-portal/get-it-now>.

2. Setup and Configuration

This chapter covers

- Installing Java 7 JDK
 - Installing Ant 1.9 (Optional)
 - Installing MySQL Database
 - Installing Eclipse (Juno)
 - Installing Supporting Tools
 - Getting Liferay Downloads
-

The only thing required getting Liferay up and running is a JRE (Java Runtime Environment). Usually in a production server we'll setup only a JRE to run Liferay. As in our case, we are about to do proper development; we'll require a JDK (Java Development Kit) as well. Mere JRE installation will not be sufficient. This chapter details the installation of JDK and all other software's required for the purpose of a full-fledged development. We'll start with the installation of latest version of JDK. Getting the right tools and mastering how to effectively use them is the secret of any successful trade.

2.1 Install Java 7

Download the latest version of JDK from Oracle's site. The main download URL is <http://bit.ly/WykKyo>. Once you're in this page, you can “**Accept License Agreement**” and start downloading a particular version of JDK7 based on your underlying operating system and its type (*32 or 64 bits processor*).

After downloading, double click the executable and follow the installation steps to setup JDK on your computer. Usually the JDK is installed under “**c:/Program Files**” folder. This location is referred to as “JAVA_HOME”. A system environment variable should be created mapping this name with the actual location. After the successful installation don't forget to modify the “PATH” system environment variable to include “%JAVA_HOME%\bin”. There are many resources in the web that describe how exactly to do this. One such link is <http://bit.ly/14XpwIE>. To verify that the installation is successful, open a command prompt and issue the following commands:

Command	Output
java -version	java version "1.7.0_07" Java(TM) SE Runtime Environment (build 1.7.0_07-b10) Java HotSpot(TM) 64-Bit Server VM (build 23.3-b01, mixed mode)
javac -version	javac 1.7.0_07

Note: The actual output might be different depending on the operating system and whether your computer is 32 or 64 bits.

If you're using an operating system other than windows, you have to refer to the documents that talk about installing JDK on that particular operating system, e.g. Linux or MacOS. At the end of JDK installation process, you will have to run the above commands to ensure the JDK installation is proper and successful.

2.1.1 Are you new to Java?

If you're very new to Java, don't worry, you can learn it very easily. You're the most apt person to read this book and become a Liferay Expert. Unfortunately I cannot recommend you one comprehensive book on Java as a resource. But I've certainly heard people talking about "Head First Java" published by O'Reilly as a very good option for people who want to quickly get started with Java programming. The other good Java books are "Effective Java" and "Thinking in Java". You can pick any one of them from a bookstore and quickly get yourself acquainted with Java before embarking onto learning Liferay.

If you don't want to spend money buying a book on Java, you can browse through the official Java documentation from <http://bit.ly/146QSgw>. They provide a quick insight into Java language. If you have worked on a Java project in the past and now want to learn Liferay, it is recommended to quickly have a glance at the online tutorial before moving ahead with the rest of this book.

2.2 Install Ant 1.9

Note: You can skip this step as ANT comes bundled with Eclipse, which is going to be our development environment. Still it is recommended to download and install ANT. Whenever we get to a situation where we have to run some build scripts from the command prompt we can do it without depending on Eclipse.

Apache Ant is a Java library and command-line tool whose mission is to drive processes described in build files as targets and extension points dependent upon each other. Ant is commonly used to build Java applications. Ant supplies a number of built-in tasks allowing user to compile, assemble, test and run Java applications.

Ant can also be used effectively to build non-Java applications, for instance C or C++ applications. More generally, Ant can be used to pilot any type of process that can be described in terms of targets and tasks. Ant is written in Java. Users of Ant can develop their own "antlibs" containing Ant tasks and types, and are offered a large number of ready-made commercial or open-source "antlibs". Ant is extremely flexible and does not impose coding conventions or directory layouts to the Java projects that adopt it as a build tool. Download the latest version of ant from <http://bit.ly/XqSqQK>.

After downloading unzip the contents to a folder usually referred to as "**ANT_HOME**". Update the "**PATH**" system environment variable to include

“%ANT_HOME%\bin”. To confirm that ant is installed properly, open a terminal window and give the following command:

You will see the output as below,

```
Apache Ant(TM) version 1.9.0 compiled on June 20 2012
```

By default Liferay plugins SDK works based on Ant build tool. Liferay also unofficially supports an SDK that is based on another build tool called Apache Maven [<http://maven.apache.org>]. Since, Ant is widely used; we'll be using Ant to build our Portlet plugins throughout this book. When you install Eclipse, it automatically loads the Ant libraries based on the environment variable that is already set.

2.2.1 Maven Instead of Ant

Apache Maven is a software project management and comprehension tool. Based on the concept of a project object model (POM), Maven can manage a project's build, reporting and documentation from a central piece of information. Apart from Ant based SDK, Liferay provides a Maven based SDK to quickly develop Liferay plugins. We'll not get into the details of setting up and using this SDK as it not in the scope of this book. If you really want to know the details, you can check the blogs written by Mika Koivisto, one of Liferay's core engineers.

- <http://www.liferay.com/web/mika.koivisto/blog/-/blogs/15470275>

2.3 MySQL Database

Liferay by default comes bundled with a lightweight, embedded database called HypersonicSQL (HSQL), <http://hsqldb.org>. In reality Liferay can work seamlessly with any database as it uses the Hibernate as the ORM (Object Relational Mapping) layer to the underlying database [<http://www.hibernate.org>]. For this reason Liferay is called as database agnostic. The following are the list of the relational databases that Liferay successfully connects to:

IBM DB2	MS-SQL Server	MySQL
Oracle	Sybase	PostgresSQL

Apart from the relational databases listed above, Liferay can nicely work with modern databases (NoSQL) like MagnoDB, Hadoop, Cassandra, Amazon SimpleDB, etc. All we need to make use of any database is to have the appropriate driver for that database and the same has to be placed inside the “lib” folder of the Liferay tomcat bundle. By Default, Liferay has the drivers for some of the popular databases like MySQL and Apache Derby.

For the purpose of our book and all its examples, we'll connect Liferay to MySQL database that is the most widely used open source relational database. Download the latest version of MySQL from <http://bit.ly/XVp7RH>. Select the operating System for which you want to install MySQL. Then follow the installation steps as guided by the installer program. Once successfully installed, you will have to verify the same by issuing the following command in a terminal window.`-u root -p<password>`

After hitting enter you will get into the MySQL prompt. From this prompt, enter the following command:

You will see the list of databases that are available by default. This confirms that the MySQL database has been installed successfully. You have to enter “**exit**” to come out of the MySQL command window back to the normal command window.

Note: The contents for the above three section has been purposefully kept as minimal as possible for the simple reason that this book has been mainly targeted at JAVA/J2EE developers and they must be already well-verses with the installation of Java, Ant and MySQL and quite conversant with setting the appropriate environment variables like JAVA_HOME, ANT_HOME, ANT_OPTS & MYSQL_HOME and appending their respective bin folders to the system “PATH” variable.

Fresher’s and people who are doing these installations for the first time can always refer to the umpteen numbers of articles that are available all over the Internet. As the purpose of this book is not to get into the minor details of these installations, I’ve tried to keep these sections as short as possible.

2.4 Install Eclipse Juno

After the successful installation of JDK 7, ANT 8 and MySQL 5, let us go ahead and install the latest version of Eclipse that will help us easily develop our own applications to be deployed on Liferay. Eclipse – developed and maintained by Eclipse Foundation – is the most powerful and free IDE predominantly used for any JAVA development. You can download the latest version of Eclipse from the link, <http://www.eclipse.org/downloads/>.

Choose the version according to your operating system and type (*32 or 64 bit processor*). Eclipse comes in multiple versions After installing; Eclipse will open with the default welcome screen as shown here.

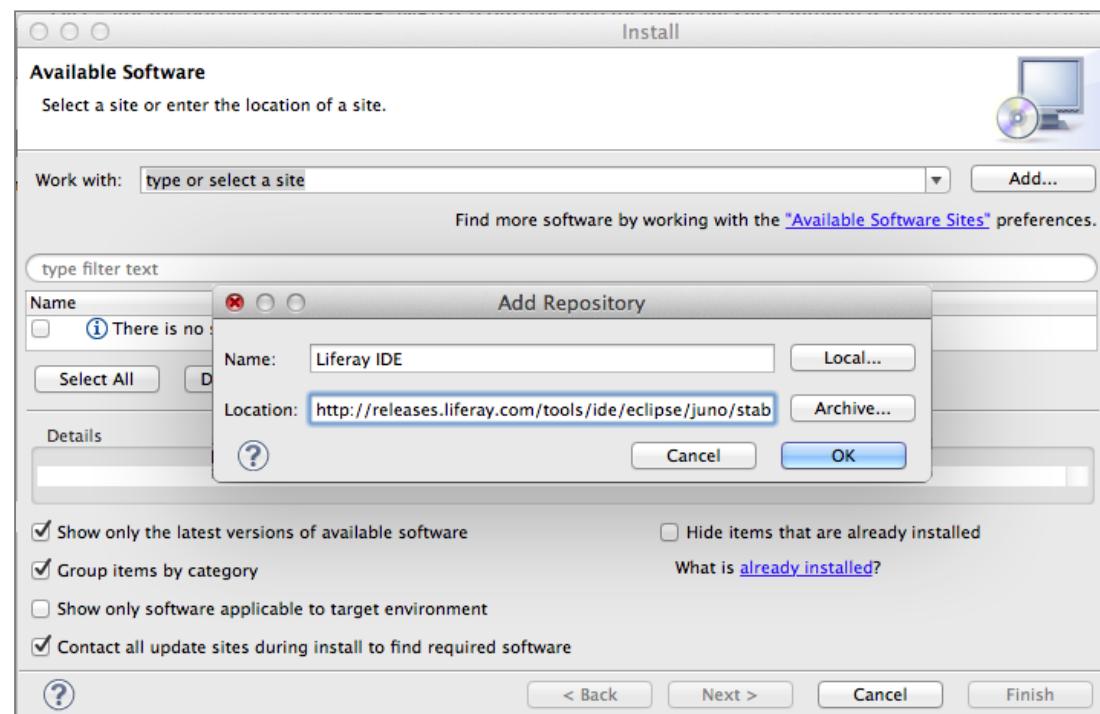
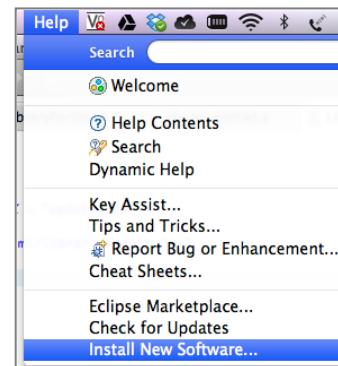


You can close this welcome window by clicking the X mark on the tab. This screen confirms that Eclipse has been installed successfully on your computer. The subsequent sections describe how to install some additional plugins on top of Eclipse we have just installed in order to make our development much easier and enjoyable. These include installation of Liferay IDE for eclipse and Subclipse – a plugin to hook to an active subversion repository.

2.4.1 Install Liferay IDE for Eclipse

This sub-section describes how to install Liferay IDE for Eclipse. This Eclipse plugin for Liferay will greatly enhance the productivity of developing various Liferay plugins – *Ext, Portlets, Hooks, Themes* and *Layouts*. Before Liferay came up with this plugin in mid 2011, “**mPire**” was the only ant-based tool that existed for quickly auto-generating LiferayMVC portlets. It was originally conceptualized by me and took over a year to develop. Finally “**mPire**” won the prestigious “**Lockheed Martin Innovation Award**”.

Let's see how to install the Liferay IDE for eclipse. Go to **Eclipse Menu → Help → Install New Software...** as shown in the last page. In the resulting window, click “Add” and specify the location of plugin repository that we want to install to. The URL is <http://releases.liferay.com/tools/ide/eclipse/juno/stable/>. The same is shown here.



Just follow the steps as guided by Eclipse by clicking on the appropriate options to install the plugin. It will ask to restart Eclipse after the successful installation of Liferay IDE. After you restart Eclipse, you will find these icons on the top navigation bar.



This confirms that you have successfully installed the Liferay IDE and you can now Perform other related setup.

2.4.2 Liferay Developer Studio

The Enterprise Edition of Liferay has an official IDE that comes bundled with Liferay called as “**Liferay Developer Studio**”. This has some more features when compared to the normal plugin for Eclipse. The features available ONLY in this version are:

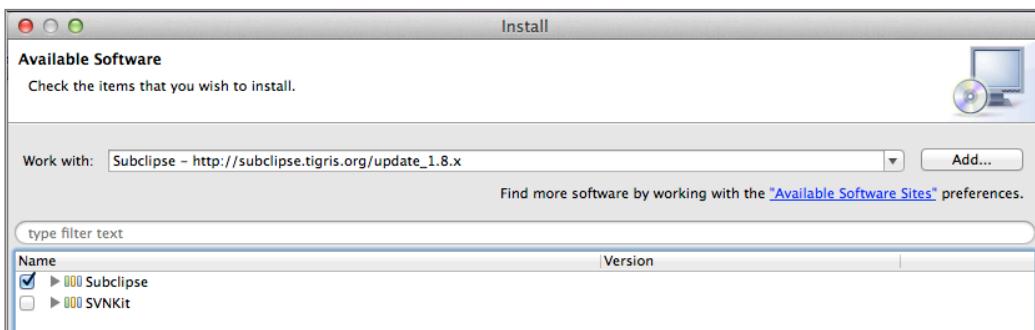
- Eclipse all-in-one bundle packaging
- Bundled Portal Server EE
- Bundled Plugins SDK EE
- Example Project wizard for bundled example projects
- WebSphere 6.x/7.0 Server development support
- Kaleo Designer for Java

Note: I would like to mention a word of caution here. Don't confuse Liferay plugin for Eclipse and Liferay plugins SDK (covered in section 2.5.3). These are two different things.

2.4.3 Install Subclipse for subversion

Though this step is not mandatory for carrying on with other chapters in this book, it is still recommended to install Subclipse that will help us check-out and check-in our codebase with any Subversion (SVN) repository of our choice. Subversion is a popular source code versioning system very similar to CVS and VSS. Google Code provides a free Subversion repository. You can always create a free repository for your project at <http://code.google.com/hosting>. In order to install Subclipse, you have to follow exactly the same steps you followed in the last section while installing Eclipse plugin for Liferay. The only difference is the repository URL, which in the case of Subclipse is http://subclipse.tigris.org/update_1.8.x.

In the “Available Software” dialog, select “Subclipse” and click “Next >”. This will install Subclipse. After the successful installation you have to restart Eclipse as usual.



Note: All code examples in this book have been checked-in to a Subversion repository hosted with Google Code. If required, you can check out the code from this repository in order to refer to the examples. The URL is <http://lr-book.googlecode.com>. We'll see the exact steps to checkout from this URL as we move forward.

Subclipse Verses Subversive

Like Subclipse, **Subversive** [<http://www.eclipse.org/subversive/>] is another Eclipse plugin for connecting to a Subversion repository. Though it is officially supported by Eclipse, I personally prefer using Subclipse as it is much more user-friendly and robust when compared to Subversive. The choice to use one of these is purely yours. If you're already using Subversive and you are comfortable working with it, you can continue to do so. But if you're new to SVN itself, it is better to start with Subclipse.

GitHub

Git is another distributed version control and source code management (SCM) system with an emphasis on speed. GitHub [<https://github.com>] is a platform that hosts millions of Git repositories. If you want to use GitHub for your project you have the liberty to go with it after creating an account there. Liferay's source code is now completely on the GitHub. The details of this repository will be explained later.

2.5 Supporting Tools

It is time to install some supporting tools that will make our life easier. We don't want to spend more time on this section as you may already have these tools installed in your machine. If they are not installed already, this section will serve as a checklist to install them quickly and proceed further.

2.5.1 Mozilla Firefox Browser

To test Liferay applications / portlets Mozilla Firefox is the preferred browser for various reasons. Though you can use other browsers, I personally recommend Mozilla Firefox. At the same time, I would also like to caution you that things that are working perfectly fine in Firefox may not work with other browsers, especially Internet Explorer. Before releasing your work to production, you should always ensure that the functionality you develop is compatible with all browsers.

To install Firefox just visit <https://www.mozilla.org/en-US/> and click on the appropriate download link. Immediately after the successful installation of Firefox, go ahead and install a very powerful plugin called FireBug from <http://getfirebug.com>. Firebug has many powerful features that will help you to create a super clean Portlet from browser's (UI) perspective.

2.5.2 Unzip Utility – WinZip or WinRar

In the sections that are going to follow, we have to unzip the files we'll download from Internet. You need a good unzip utility. You might go for WinZip (<http://www.winzip.com>) or WinRAR (<http://www.rarlab.com>). I personally recommend WinRAR that is free and fast. You can also use the java's "jar" utility to quickly unzip your files. Open a command window and go to the location where you

want to unzip your file, e.g. “**c:\liferay-workspace**”. Then give the below command and it will start inflating the contents of zip file.

```
>jar xvf [path of the zip file]
```

2.5.3 SQL Query Browser

We need a good tool for browsing the database and its tables. Sometimes, through these tools we will have to directly update data into our database that Liferay will be pointing to. I recommend the following tools in the order of my preference. It is up to you which one you want to choose.

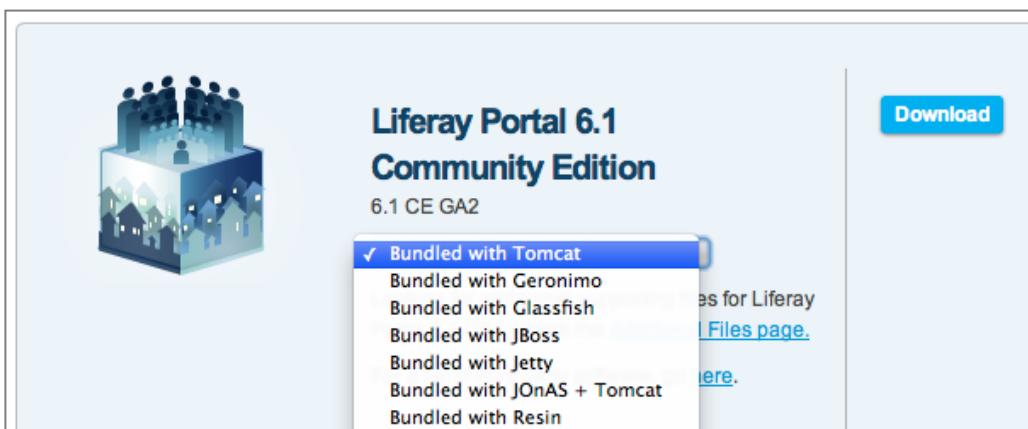
1. MySQL Query Browser (<http://dev.mysql.com/downloads/gui-tools/>)
2. Squirrel-sql (<http://squirrel-sql.sourceforge.net>)
3. PhpMyAdmin (<http://www.phpmyadmin.net/>). This is usually installed on the server where Liferay is running, so that you can have a web-based interface to the remote database that can be accessed via the browser.

2.6 Liferay Downloads

In this section, we’ll download all necessary artifacts from Liferay’s official website [<http://liferay.com>] in order to get our portal up and running. Installing Liferay is the simplest and easiest of all portal servers available in the market today. We can download and get Liferay up and running in matter of just few minutes whereas this is not the case with other portal products. As I mentioned before, all we need to get Liferay running is a JRE (Java Runtime Environment). In the following sections we’ll see the steps for installing a Liferay bundle and all other related software’s that will help in our development.

2.6.1 Tomcat Bundle

Liferay can be downloaded and installed either as a stand-alone WAR file OR it comes bundled with various web and application servers. The below screenshot has been taken from <http://www.liferay.com/downloads/liferay-portal/available-releases>.



<p>Bundles available by default are:</p> <ul style="list-style-type: none"> • Tomcat Web Server • Geronimo Application Server • Glassfish Application Server • JBoss Application Server • Jetty Web Server • JOnAS + Tomcat App Server • Resin Web Server 	<p>Though you can download any bundle of your choice, the most preferred and recommended flavor is the one that comes bundled with Tomcat. Let's go ahead and download this bundle and keep the downloaded file inside a folder “c:\liferay-downloads”. Unzip this file onto the folder that was created earlier – “c:\liferay-workspace”. After unzipping this file, you will see the folder “liferay-portal-6.1.1-ce-ga2”.</p>
---	--

Note: At the time of writing this book, the latest version of Liferay Community Edition is “6.1.1 GA2”. I have tried to keep the contents of this book as independent of the Liferay version as possible, so that the readers will have the maximum benefit. All the examples in this book are developed according to this version of Liferay. All these examples should also seamlessly work with the equivalent EE version of Liferay as well. Readers are encouraged to write to me if they find any discrepancies while trying out these examples in a different version of Liferay.

2.6.2 Liferay Portal Source

In this section let's download the complete portal source code and see its various components. We'll have to refer to the Liferay source code during many occasions while going through the contents of this book. The Liferay source code is well documented and is a rich source of information. We can learn many things from the Liferay source code itself. Form the list of available downloads, select “**Portal Source**” and click download. Save the downloaded file inside “**c:\liferay-dowloads**”. Once the download is successful, extract the zip file onto “**c:\liferay-workspace**”. You will get a folder “**liferay-portal-src-6.1.1-ce-ga2**” inside this workspace.

After extracting the zip file, you will find the following folders inside the main folder. The contents of the folder have been detailed below. We will get into the details of each of these folders as we move forward with other chapters in this book.

Folder	Details
benchmarks	Contains the scripts for testing the performance of Liferay portal using a tool called Grinder, http://grinder.sourceforge.net
classes	Contains the compiled class files. Not significant.
definitions	Contains the DTD files for the various XML files that are used by Liferay. All previous versions of these files are also maintained for backward compatibility. The DTD files present inside this folder are for following types of XML files: <ul style="list-style-type: none"> • liferay-display.xml • liferay-friendly-url-routes.xml • liferay-hook.xml • liferay-layout-templates.xml • liferay-look-and-feel.xml

	<ul style="list-style-type: none"> • liferay-plugin-package.xml • liferay-plugin-repository.xml • liferay-portlet.xml • liferay-portlet-app.xml • liferay-resource-action-mapping.xml • liferay-service-builder.xml • liferay-social.xml • liferay-theme-loader.xml • liferay-workflow-definition.xml
lib	Contains all “jar” files required for development, global and portal. You can read the “ readme.txt ” to know the difference between these types of jars. Also “ versions.html ” has the version information for each of these “jar” files.
nbproject	A folder created by NetBeans IDE. You can ignore this.
portal-client	Contains all generated web-services stubs that will be used by any client application in order to make a SOAP based web service call to Liferay.
portal-impl	The most important folder that contains the implementation (java) classes for various portal and portlet level functionalities.
portal-service	Contains mostly the files generated by the Service Builder. The Service Builder takes a “ service.xml ” as the input and generates the Service Layer. Chapter 5 covers this in great detail.
portal-web	Contains all CSS, Images, Theme related files, JSP files and taglib JSP for various portal features and portlets that come bundled with Liferay.
sql	Contains various migration scripts for migrating Liferay from one version to another version. The files are for various popular databases.
support-tomcat	Some files to support Tomcat class loading feature. You can ignore this folder.
tools	Contains various third party tools that are used by Liferay. Some of the tools are: <ul style="list-style-type: none"> • Maven (Build tool) • Putty (SSH access) • Jalopy (Code beautification) • Selenium (UI testing framework) • YUI Compressor (for CSS and JS compressing) • Grinder (For load and performance testing)
util-bridges	Contains portlet connectors for various frameworks and technologies. Using these bridges, we can deploy portlets developed using any non-standard technologies into Liferay. Some popular bridges are – <i>alloy</i> , <i>struts</i> , <i>bsf</i> , <i>groovy</i> , <i>javascript</i> , <i>jsf</i> , <i>jsp</i> , <i>php</i> , <i>python</i> and <i>ruby</i> .
util-java	Contains some of the most commonly used general purpose utility classes, that are used extensively across the portal
util-taglib	Contains the tag handler files for the various convenient taglibs that comes bundled with Liferay. There are nine major taglibs in Liferay. They are – <i>aui</i> , <i>core</i> , <i>faces</i> , <i>portlet</i> , <i>portlettext</i> , <i>security</i> , <i>theme</i> , <i>ui</i> and <i>util</i> .

Here are some important URL's for you to bookmark and remember.

Liferay SVN Repository	http://svn.liferay.com/repos/public/portal/tags/6.1.1/
Alternatively, you can access Liferay source code from Liferay's SVN repository. It will ask you to enter username and password to access this repository. Enter guest / guest in order to access the SVN repository online. You are highly encouraged to browse through the various files inside the folders and make yourselves comfortable.	
Liferay Releases	http://releases.liferay.com/portal/
You can also browse through the various releases of Liferay with this link. If you want to try an upcoming version of Liferay or a nightly build of Liferay you can do so by downloading the appropriate file from this location.	
Liferay Javadocs	http://docs.liferay.com
You can find complete javadocs for various Liferay products here. The various products of Liferay are – Liferay Portal and Liferay Social Office.	
Liferay GitHub	https://github.com/liferay
Liferay's source code is now at GitHub, the most powerful and widely used distributed version control system. If you're already aware of Git, you can "fork" the brewing Liferay source code directly from here.	
Liferay Issues	http://support.liferay.com
Found an issue with Liferay? Report the issue here. This is based on JIRA issue tracking system.	

2.6.3 Portal Source as Eclipse Project

Before we move on to the next section, let's import the portal source as a project in Eclipse. This will help us to quickly navigate through the portal source files and get a better understanding. Right click on the left panel in Eclipse IDE, **New → Project... → Java Project → Next > → Give project name as “liferay-portal-src-6.1.1-ce-ga2” and click “Finish”**. It will take some time for the new project to get properly configured within the current workspace. . Once this is completed, you 'will see the new project appearing on the left  liferay-portal-src-6.1.1-ce-ga2 panel.

You can right click on this project and click “**Close Project**” for the time being. We can open this project only when required to avoid un-necessary burden on Eclipse.

2.6.4 Plugins SDK

We'll now get the latest version of Liferay's plugins SDK (Software Development Kit). This SDK helps to quickly develop various kinds of Liferay plugins, viz – *Ext, Portlet, Hook, Theme, Layout* and *Web*. This SDK is based on the popular Apache Ant tool that we already installed in Section [2.2 Install Ant 1.8](#) of this book.

As with the earlier steps, download the plugins-sdk zip file from the Liferay downloads page. From the list of available downloads, select “**Plugins SDK**” and click “Download”. Save the downloaded file inside “**c:\liferay-downloads**”. Extract the contents of this file into “**c:\liferay-workspace**”. You will see a new folder

“**liferay-plugins-sdk-6.1.1**”. If you really want, you can rename this folder to a much simpler name. But better to leave it as it is.

At the end of this section the folder “**c:\liferay-downloads**” will contain the following three sub-folders:

- **liferay-portal-6.1.1-ce-ga2** (*Liferay tomcat bundle*)
- **liferay-plugins-sdk-6.1.1** (*Liferay plugins SDK*)
- **liferay-portal-src-6.1.1-ce-ga2** (*Liferay portal source code*)

If you really wish you can rename these folders with shorter names. But I prefer to leave them like this and move forward. In the rest of this book we’ll see how to develop a complete Portlet plugin. As mentioned in the section, [What is not covered in this book?](#), we’ll not be covering the details of developing other kinds of plugins – *Ext, Hook, Theme, Layout and Web*. Richard Sezov covers them in great detail in the Liferay’s official documentation and also in his book “**Liferay in Action**”, published by Manning Publications.

The location of Liferay’s official development documentation is <http://bit.ly/14HgAWN>. You’re encouraged to browse through the topics in this online document to reinforce your learning gained so far through this book.

Summary

In this important and initial chapter we have seen how to install various software's and tools that are required to get our Liferay development rolling. We started this chapter with the steps for installing the latest version of Java JDK. At the end of the first section, I've also provided some good pointers for people who are new to Java itself. They should get a betterhang of Java language before starting to move ahead with other chapters in this book. After installing Java, we have seen the steps to install Apache Ant and MySQL database. In both these sections, we have seen the variations of using Maven instead of Ant and using any other database other than MySQL. For the purpose of working out all examples in this book, I strongly recommend to use MySQL. Once you become comfortable, you can use any other database of your choice or of your customer's.

We moved on to install the latest version of Eclipse, Juno. After the eclipse installation we have installed two plugins for Eclipse – one is for Liferay and the other one is for connecting to a SVN repository. We have also seen the difference between Subclipse and Subversive – two popularly used Eclipse plugins for SVN. We continued the chapter with the installation of other required tools for making our development process very smooth, enjoyable and effective.

From here, we moved to download various Liferay artifacts starting with Liferay tomcat bundle. We have also downloaded the complete source code of Liferay and had a quick glimpse at its various folders and how they are organized. After the discussion on portal source, we have installed the Liferay plugins SDK. With this basic infrastructure in place and we are all set to begin our Liferay Portlet development journey.

3. Setting the stage with Liferay IDE

This chapter covers

- New Liferay Server
 - Basic Configuration
 - Pointing Liferay to MySQL Database
 - New Liferay SDK
 - New Liferay Project
-

This chapter will cover the configuration steps involved in setting up the development Liferay Server and the Plugins SDK within Eclipse. Let's do one after the other starting with setting up **New Liferay Server**. We'll also set the stage for our development activities that is going to start from the next chapter onwards. This chapter is more like a bridge that will connect the infrastructure that we setup in the previous chapter and the serious development we are going to undertake from the next chapter onwards. Once you complete reading this chapter, it is highly recommended to take a detour and learn some Liferay portal administration concepts.

Unfortunately, we have not covered the Portal Administration topics in this book. You should read these topics from Liferay's official portal Administration guide or by reading some other books on Liferay. I can suggest two books that cover these topics in great length.

1. “**Liferay Portal 6 Enterprise Intranets**” by Jonas X.Yuan, published by PACKT.
2. “**Practical Liferay**” by Poornachandra Sarang, published by APress.

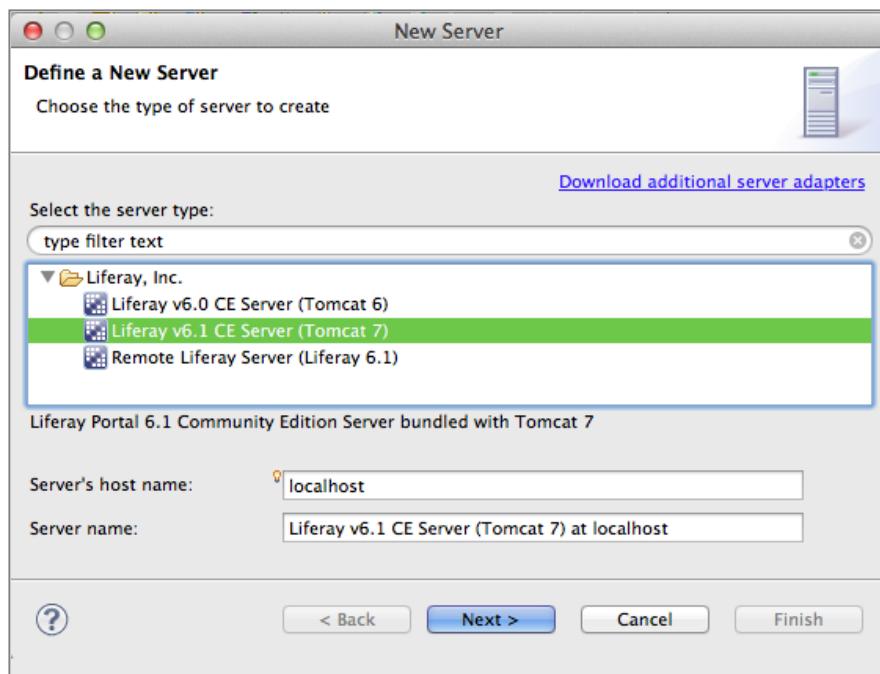
Let us now get started with the actual chapter, starting with setting up a new Liferay Server followed by other important sections.

3.1 New Liferay Server

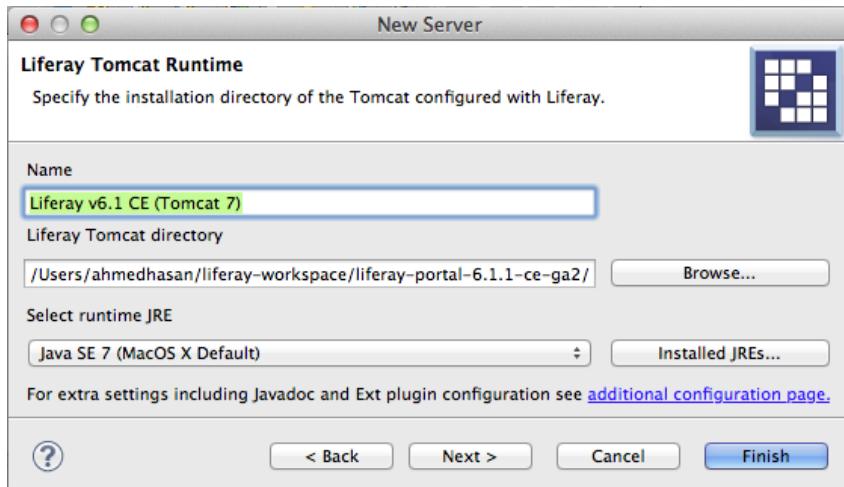
I am sure; you have kept the Eclipse window open. From the set of Liferay icons that appear on top, select the one which is at the end. Out of the two options available, click “New Liferay Server”.



In the dialog that opens, select the ‘Liferay v6.1 CE Server (Tomcat 7) option highlighted below and click “Next >”.



In the next screen (*as shown in next figure*) browse and select the location of the tomcat server that comes bundled with Liferay. In our case, it will be “**c:\liferay-workspace\liferay-portal-6.1.1-ce-ga2\tomcat-7.0.27**”. After selecting server, you can choose to either continue with “Next >” or click “Finish”. Since the other options can always be set later, I prefer you to click “Finish” and complete the New Server setup. You can give any name for the server as per your wish. But most of the times, you can go with the default values.

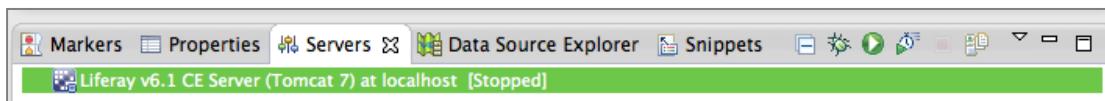


You have successfully setup the server. The next setup gives us the ability to start and stop the tomcat server within Eclipse itself. First, you have to confirm the server has been set properly by seeing a project by name “Servers” appearing on the “Project Explorer”

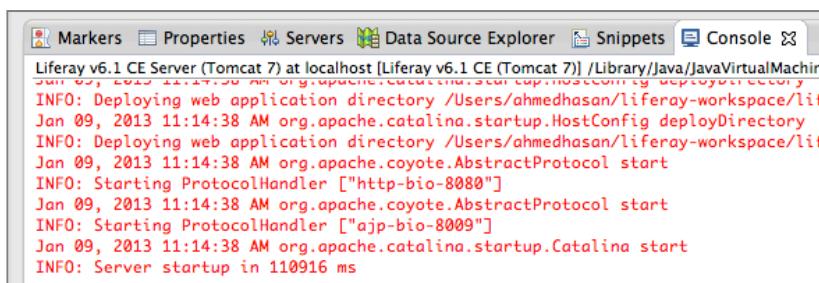
panel. Next, you Select **Eclipse Menu → Window → Show View → Servers**. The new server that just got added would appear inside the server pane as shown below. (*If the “Servers” option does not*



appear directly under “Show View”, and then find it by clicking “Other... ”).



We are about to start the server from inside Eclipse. Before starting the server you have to do one more thing. Remove folders “**resources-importer-web**” and “**welcome-theme**” from the “**webapps**” folder of Tomcat. The two apps are basically for creating a default sample website when Liferay starts for the first time. Once these folders are deleted directly from the explorer window, click the Green Arrow (*play*) icon to **start** the server. It takes a few minutes for the server to get completely started. Once the server has started you will see the following message on the server console “Server startup in **XXXX ms**”, where **XXXX** is the time taken.



The server has started successfully. One final change you have to do is to uncheck the options “Show Console when Standard Out Changes” and “Show Console when Standard Error Changes”.


Un-check the two icons above. They are selected by default.

One thing that should be observed in the console is the connection information to the default hypersonic (HSQLDB) database. As it states clearly, this embedded database is meant only for development and demoing purposes as seen in the figure below. In the subsequent section we'll see how to point to a different database.

```
... Determine dialect for HSQL Database Engine 2
11:13:08,509 WARN [pool-2-thread-1][DialectDetector:86] Liferay is
configured to use Hypersonic as its database. Do NOT use Hypersonic
in production. Hypersonic is an embedded database useful for
development and demo'ing purposes. The database settings can be
changed in portal-ext.properties.
11:13:08,681 INFO [pool-2-thread-1][DialectDetector:136] Found
dialect org.hibernate.dialect.HSQLDialect
```

3.2 Basic Configuration

Open your favorite browser, preferably Mozilla Firefox or Chrome and go to <http://localhost:8080> to see the portal LIVE for the first time. You will be taken to the portal's "**Basic Configuration**" page. From this page, you can set the default values for the server administrator and optionally point to a different database.

The screenshot shows the Liferay "Basic Configuration" interface. At the top, there's a logo and the word "LIFERAY". Below it, a title bar says "Basic Configuration". The interface is divided into several sections:

- Portal**:
 - Portal Name: Liferay
 - Default Language: English (United States)
 - Note: This database is useful for development and demo'ing purposes, but it is not recommended for production use. ([Change](#))
- Administrator User**:
 - First Name: Test
 - Last Name: Test
 - Email (Required): test@liferay.com
- Database**:
 - Default Database (Hypersonic)
 - Note: This database is useful for development and demo'ing purposes, but it is not recommended for production use. ([Change](#))

At the bottom left is a "Finish Configuration" button.

You can give the values according to your choice. Before clicking on "Finish Configuration" let's change the database settings. We'll make Liferay to point to a production ready database like MySQL that we have installed earlier as per the explanation given in Section [2.3 MySQL Database](#).

3.3 Pointing Liferay to MySQL Database

Let's create an empty database inside our MySQL by name “**Iportal**”. The new database can have name but for the time being let us go with the default name. Go to MySQL prompt and issue the command:

Confirm that the new database has been successfully created with the next command:

This command will show the list of databases including the one that just got created. At the moment there are no tables inside this new database and it is empty.

Now let's go back to the “**Basic Configuration**” screen in the browser and click on the “**Change**” hyperlink under Database section. This will open another form on the same screen with the settings for new database connection. Give the required inputs as shown in figure below and click “**Finish Configuration**”. Liferay will take few minutes to complete the configuration and show the next screen that will lead you to portal's default home page.

Database Type
MySQL

JDBC URL (Required)
jdbc:mysql://localhost/Iportal?useUnicode=true&characterEncoding=UTF-8&useFastDateParsing=false

JDBC Driver Class Name (Required)
com.mysql.jdbc.Driver

User Name
root

Password
....



After completion of configuration, you will get the above screen from where you can go to the portal's home page. Your configuration settings will be saved in a file “**{LIFERAY_HOME}\portal-setup-wizard.properties**”. You can open this file and check its contents. The entries will be like the ones shown here.

```
admin.email.from.name=Test Test
liferay.home=c:\liferay-workspace\liferay-portal-6.1.1-ce-ga2
admin.email.from.address=test@liferay.com

jdbc.default.driverClassName=com.mysql.jdbc.Driver
```

```
jdbc.default.username=root  
jdbc.default.url=jdbc:mysql://localhost/lportal?useUnicode=true&characterEncoding=UTF-8&useFastDateParsing=false  
setup.wizard.enabled=false
```

Go back to the browser and click the button “**Go to My Portal**”. This will take you to Portal’s default landing page. Since it is the first time landing, Liferay will prompt you to do the following things before taking to the actual default-landing page.

- Agree to the terms of use
- Set the default password for the admin user (*give “test1”*)
- Set the password reminder question and answer

Omni User: At this juncture I would like to say a word about the user you are currently logged in as. He is called as the Super User. Liferay calls it the Omni User, who has the control over the entire portal. One installation of Liferay can have more than one user to have this role (Administrator or Omni User). The “Server Administration” section inside the Control Panel will appear only for an Omni User.



Coming back to our original discussion, there is one final step left before we go ahead with setting up of SDK plugin that will help us create portlets. This step is to verify the tables of the new database “**Iportal**”. Once again get onto the mysql prompt and type the below commands:

```
mysql> use lportal;  
mysql> show tables;
```

You will see Liferay has newly created 180+ tables. Just have a closer look into these tables. You will find two distinct types of tables.

- **System (Portal) tables** – the tables that are used by the portal. They usually end with an underscore (“_”). But there are exemptions for this rule. Liferay will not function properly if any of these tables are dropped from the database. Examples of these tables are “*user_*”, “*role_*”, etc.
- **Portlet tables** – the tables that are used by the various portlets that come bundled with Liferay. Usually these tables start with the portlet name as the prefix. If you don’t want that portlet to be available in your portal, you can comfortably delete these tables from the database. Examples of these tables are “*shoppingcart*”, “*shoppingcategory*”, etc that are used by the shopping cart portlet of Liferay.

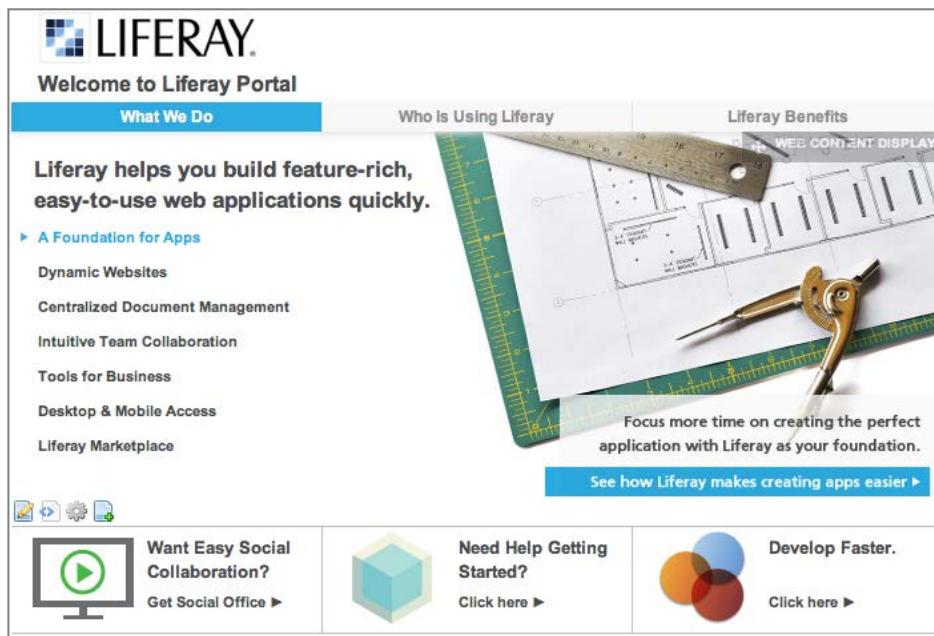
Do it yourself

1. List down all the tables for Blogs Portlet
2. List down all the tables for Wiki Portlet
3. List down some association tables – e.g. Users and Roles, Users and Groups, etc.
4. Create a new user account in Liferay and see what table’s data get inserted.

3.3.1 Initial Cleanup

(This step is required ONLY if you forgot to delete the two apps before starting the server for the first time. The apps are “welcome-theme” and “resources-importer-web”. If you’re using Liferay 6.2.X, you can completely ignore this step as this version of Liferay does not have the default sample website.)

After verifying the database tables, go back to the portal-landing page. If you have not deleted the two apps before starting Liferay for the first time, you will get the screen as shown in the next page.



We have to follow these **seven steps** to get rid of the default “Welcome to Liferay Portal” screen and get the normal Sign-in Page.

1. Delete the pages “Who is Using Liferay” and “Liferay Benefits” by clicking the “X” mark when you hover mouse on that page title.
2. Double Click on the page title “What We Do” and rename the page to “Welcome” and click the **tick** mark to save the change.
3. Click on “Manage → Page” from the top pane bar that will open a popup window. Click the “Public Pages” on the left navigation and change the “Look and Feel” to “Classic” from the current “Welcome” theme. Save your change by clicking on the “Save” button and close the popup window by clicking the “X” button on top right corner. You will now see the “Classic” theme applied to this page.
4. Remove the Web Content Display Portlet appearing on the Welcome page, by hovering on the top right corner of this portlet and clicking the “X” mark as shown WEB CONTENT DISPLAY
5. Go to “Add → More...” from the top pane bar and drag and drop the following Portlets on to the page – “Sign In” and “Hello World”.

-
6. Go to “**Manage → Page Layout**” and select the “**2 Columns (30/70)**” layout. Save the change and click “**X**” to close the popup. Rearrange the Portlets on the page by drag-and-drop so that the Sign In Portlet is on the left column (30) and the Hello world Portlet is on the right column (70).
 7. You can now click on the “**Look and Feel**” option of both these Portlets to make the Portlet border visible by changing the setting in “**Portlet Configuration**” tab.

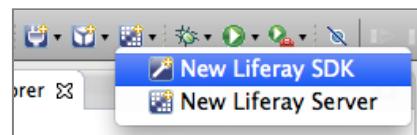
Reload the page and you will see the default-landing page appearing like below.



3.4 New Liferay SDK

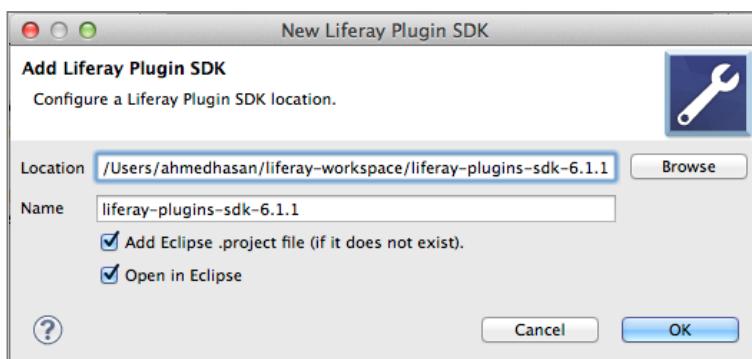
In this section, we will see how to configure the Liferay Plugin SDK with Eclipse which will help us create the different kinds of plugins – *Ext, Portlet, Theme, Hook, Layout and Webs*.

From the set of three Liferay icons that appear on the top, select the one which is at the end. From the drop-down menu, click “New Liferay SDK” as indicated here. In the resulting dialog that opens,

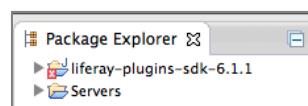


Browse for the location of the “**liferay-plugins-sdk-6.1.1**” folder and click “OK”. In your case, the location of this folder is “**c:\liferay-workspace\liferay-plugins-sdk-6.1.1**”. This is the location where we extracted the plugins SDK zip file as per section [2.6.4 Plugins SDK](#). In the same dialog check the option “Open in Eclipse”, so that it automatically opens as a java project inside Eclipse.

After you click “OK” in this dialog, you can confirm that the plugins SDK has been configured properly by going to “**Eclipse Menu → Window → Preferences → Liferay → Installed Plugin SDKs**” (*the “Open in Eclipse” is not working though*). For some reason if the Java project is not created automatically, you can manually create it by right clicking on “**Project Explorer**” pane → **New → Project... → Java Project** → giving the project name as “**liferay-plugins-sdk-6.1.1**” and clicking “**Finish**”.



After this you will see the new project appearing in the “**Project Explorer**” pane as shown here. You will also observe that this new



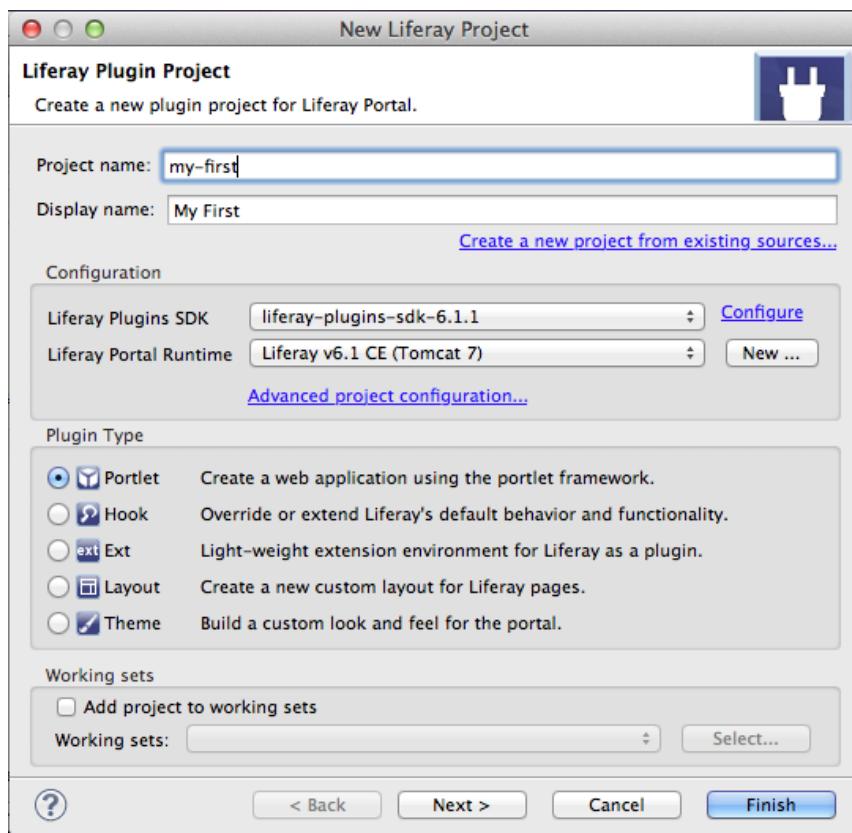
Configuration would have created a new file by name “**build.<your-computer-user-name>.properties**” inside “**liferay-plugins-sdk-6.1.1**” folder. The contents of this file will look like:

```
#Managed by Liferay IDE (remove this comment to prevent future
updates)
#Thu Jan 10 16:30:09 GMT+05:30 2013
app.server.portal.dir=c:/liferay-workspace/liferay-portal-6.1.1-ce-
ga2/tomcat-7.0.27/webapps/ROOT
app.server.lib.global.dir=c:/liferay-workspace/liferay-portal-6.1.1-
ce-ga2/tomcat-7.0.27/lib/ext
app.server.deploy.dir=c:/liferay-workspace/liferay-portal-6.1.1-ce-
ga2/tomcat-7.0.27/webapps
app.server.type=tomcat
app.server.dir=c:/liferay-workspace/liferay-portal-6.1.1-ce-
ga2/tomcat-7.0.27
```

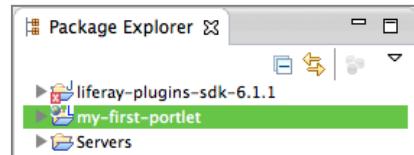
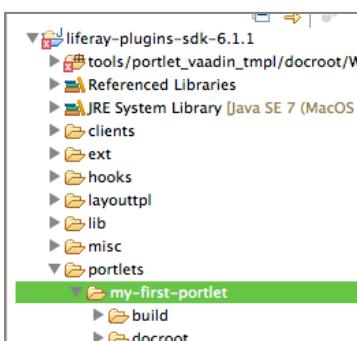
3.5 New Liferay Project

Now that the plugins SDK is properly installed and configured within Eclipse, we’ll quickly create a sample plugin and deploy the same to Liferay to check if everything is in order with what we have done so far.

Click on the first Liferay Icon in the Eclipse menu and Click “**New Liferay Project**” as indicated here. In the dialog that opens, enter the project name as “**my-first**” and click “Finish” leaving all other options un-changed as shown on the next page.



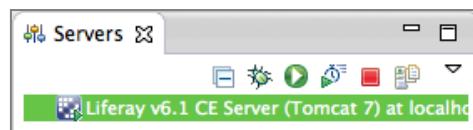
After this you will see the new project appearing on the “Project Explorer” Pane with the name “**my-first-portlet**” as shown here.



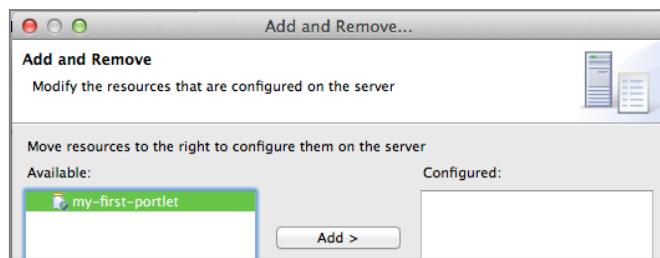
You will see a new folder appearing inside “**portlets**” folder of your “**liferay-plugins-sdk-6.1.1**” java project. In the following sections we’ll see how to deploy this Portlet to our server and also get to know the anatomy of a simple Portlet that will form the basis for our learning through-out this book.

3.5.1 Deploying “my-first-portlet” to Server

In order to deploy the portlet to the server for the first time, you have to do a quick setting. Right click on the “Server” in our “Servers” panel and select “**Add and Remove...**”.



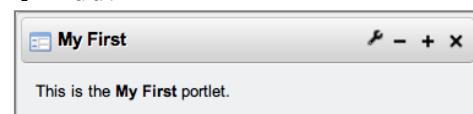
In the resulting dialog, select the “my-first-portlet” and move it to the other side by clicking “**Add >**”. Click “**Finish**” and this will instantly deploy your first Portlet to the tomcat server. You will see the message in the server console.



```
11:49:04,284 INFO [pool-2-thread-2][HotDeployImpl:178] Deploying my-first-portlet from queue
11:49:04,285 INFO [pool-2-thread-2][PluginPackageUtil:1033] Reading plugin package for my-first-portlet
```

Let’s go to portal and add this Portlet to our welcome page, by going to the **Top pane bar** → **Add** → **More...** → **Sample** → **My First** → **Add**.

You will see the Portlet successfully deployed to “Welcome” page. As with earlier Portlet, you can set the border for this Portlet as well. The Portlet will appear in the page as shown



Trouble-Shooting Tip:

Sometimes, you may encounter the Portlet not getting deployed properly. During these situations you can manually deploy the Portlet by Right clicking on the Portlet project in the “**Project Explorer**” panel → **Liferay** → **SDK** → **deploy**. This will directly invoke the underlying “ant” target in order to make a WAR file of the Portlet and copy to the “**deploy**” folder inside “**liferay-portal-6.1.1-ce-ga2**”. Liferay will

pick up this WAR file and deploy onto Tomcat. Once the deployment is successful, you will notice this Portlet appearing as a folder “**my-first-portlet**” inside the “**webapps**” of Tomcat. Even under normal situations, you are encouraged to have a look inside the “**webapps**” folder and confirm that you see “**my-first-portlet**” folder.

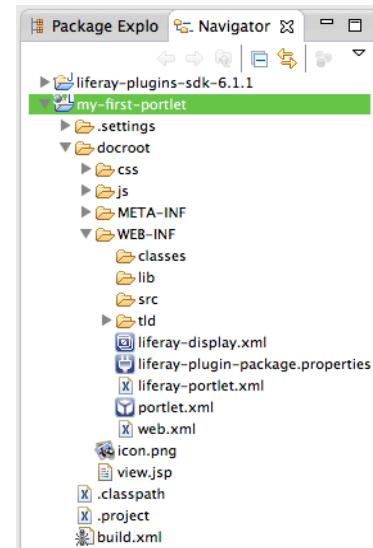
3.5.2 Anatomy of “my-first” Portlet

Now let’s start dissecting the various parts of our first Portlet to know the building blocks that make it work as per the JSR-286 specification explained in chapter one. It is very important to understand the various components of a Portlet. We’ll dedicate this sub-section only for this purpose. From your **Eclipse Menu → Window → Show View → Navigator**, open the navigator view panel and expand the “**my-first-portlet**” project on the panel as shown in the beginning of next page.

Configuration Files:

You will find all configuration files under “**docroot/WEB-INF**”.

1. “**portlet.xml**” – This is the central Portlet deployment descriptor that conveys the information about the Portlet to the Portlet container where it is getting executed (*runtime*). This file is per the requirement of JSR-286.
2. “**liferay-portlet.xml**” – This file contains the information about additional Portlet features that Liferay has built on top of JSR-286 Portlet specification. You can see the complete explanation of all the entries of this file in its corresponding DTD inside “**definitions**” folder of the portal source. You’re highly recommended to have a look into this DTD file.
3. “**liferay-display.xml**” – This file contains the information about placing of the Portlet in a particular hierarchy when it is displayed in the “**Add → More...**” dialog. You can also have a sub category within any given category.
4. “**liferay-plugin-package.properties**” – This file contains some information that helps during the packaging of the portlet as a deployable WAR file. We will see the usage of this file as we move forward to subsequent sections. This file also has the information that helps in displaying the portlet in a plugin repository.
5. “**web.xml**” – Since all the portlets are deployed as a web application, we need this deployment descriptor for this portlet as well.



Other Files:

1. “**icon.png**” is the icon displayed for this portlet that appears on the portlet title bar. This feature is not available in JSR-286, but an extension of Liferay. You will find the entry for this portlet icon in “**liferay-portlet.xml**”.

-
2. “**view.jsp**” is the default view of the portlet that appears when the portlet gets rendered on the portal page. You will find the entry for this default portlet view in “**portlet.xml**”, set as an init param “`view-template`”.
 3. Folder “**css**” optionally contains all the style sheet (css) files used by this portlet. You will see a “**main.css**” in this folder and its corresponding entry will be in “**liferay-portlet.xml**” as `<header-portlet-css>`.
 4. Folder “**js**” optionally contains all the javascript files used by this portlet. You will see a “**main.js**” and its corresponding entry will be in “**liferay-portlet.xml**” as `<footer-portlet-javascript>`.
 5. Folder “**src**” will house all the source code (.java files) that we will write for this portlet. For “**my-first**” portlet we have not used any java file including the portlet class. The `<portlet-class>` entry in “**portlet.xml**” refers to the class, “`com.liferay.util.bridges.mvc.MVCPortlet`” which indirectly extends from “`javax.portlet.GenericPortlet`”.
 6. Folder “**lib**” will house all the jar files that will be used by the java files present in this portlet under “**src**”. In case we want this portlet to make use of some jar file that is present in the portal level, then we have to give an entry in “**liferay-plugin-package.properties**”. We will see how to do this in Section [6.2 Referring to Portal’s TLDs and JARs](#).

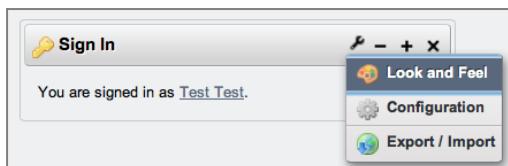
Plugin Packaging: Here I would like to mention something about packaging of your plugins. One “plugin” project can have more than one Portlet packaged as one single WAR file. It can also host themes, service layer, hooks and layout templates. It is up to you how you want to package your plugin project and deploy it as one single WAR into the server. Usually a plugin project will contain logically related entities. You should see a plugin as a highly re-usable, inter-operable component. It is good practice to package all the hooks in your application as one single plugin project, all themes and layouts as one single plugin project, all portlets or logically related portlets as one single plugin project. Eventually these plugins will be packaged and deployed as individual and independent WAR files into the server.

3.5.3 Visual Elements of a Portlet

In the previous section, we have seen the anatomy of a Portlet from the source code point of view. At the bare minimum, a Portlet is made up of these building blocks:

- **Configuration files** – “`portlet.xml`”, “`liferay-portlet.xml`”, etc.
- **Source files** – Portlet class and other class files that have the functionality
- **Views** – JSP files and xhtml files that define the view of the Portlet
- **JavaScript and CSS** – the associated JavaScript required for the Portlet and the style sheet (css) files.

It is equally important to know your Portlet from the visual point of view. Let’s take our “Sign In” Portlet on the welcome page and see what are its components that decide the physical appearance of the Portlet when it is rendered on a portal page, also called as a layout. The following figure identifies the various visual elements.



“Sign In” is the Portlet title which sits on Portlet title Bar. The key before Portlet title is the Portlet Icon. A Portlet has **header**, **body** and **footer** sections. Portlet’s actual content appears in its body section.

The four icons on the right of the title bar are called as the “Edit Controls”; the first one is used to set the options. Through “Look and Feel”, we can override the default appearance of the Portlet set at the theme level. Through “Configuration”, we can set the preferences for the Portlet. With the help of “Export / Import” we can export the Portlet data or its settings in the form of a LAR file and import it back in another instance of the same Portlet. We’ll continue to see some important concepts.

Portlet Modes: A Portlet mode indicates the function a Portlet is performing. Normally, portlets perform different tasks and create different content depending on the function they are currently performing. When invoking a Portlet, the Portlet container provides the current Portlet mode to the Portlet.

Portlets can programmatically change their Portlet mode when processing an action request. According to JSR-286 specification a Portlet can have three modes – *View* (default), *Edit* and *Help*. Liferay has added six more modes – *About*, *Config*, *Edit Defaults*, *Edit Guest*, *Preview* and *Print*.

Window States: The `WindowState` class of a Portlet represents the possible window states that a portlet window can assume. The most basic window states are:

- The MAXIMIZED window state is an indication that a Portlet may be the only Portlet being rendered in the portal page, or that the Portlet has more space compared to other portlets in the portal page
- When a Portlet is in MINIMIZED window state, the Portlet should only render minimal output or no output at all
- The NORMAL window state indicates that a Portlet may be sharing the page with other portlets

On top of the three window states defined by JSR-286, Liferay has extended couple of more window states – EXCLUSIVE and POP_UP.

Portlet Lifecycle: The Portlet 1.0 standard defines two lifecycle phases for the execution of a Portlet that a compliant Portlet container must support:

1. `javax.portlet.PortletRequest.RENDER_PHASE`, in which the Portlet container asks each Portlet to render itself as a fragment of HTML.
2. `javax.portlet.PortletRequest.ACTION_PHASE`, in which the Portlet container invokes actions related to HTML form submission. When the portal receives an HTTP GET request for a portal page, the Portlet container executes the Portlet lifecycle and each of the portlets on the page undergoes the `RENDER_PHASE`. When the portal receives an HTTP POST request, the Portlet container executes the Portlet lifecycle and the Portlet associated with the HTML form submission will first undergo the `ACTION_PHASE` before the `RENDER_PHASE` is invoked for all of the portlets on the page.

The Portlet 2.0 standard adds two more lifecycle phases that define the execution of a Portlet:

1. `javax.portlet.PortletRequest.EVENT_PHASE`, in which the Portlet container broadcasts events that are the result of an HTML form submission. During this phase, the Portlet container asks each Portlet to process events that they are interested in. The typical use case for the `EVENT_PHASE` is to achieve Inter-Portlet Communication (IPC), whereby two or more portlets on a portal page share data in some way.
2. `javax.portlet.PortletRequest.RESOURCE_PHASE`, in which the Portlet container asks a specific Portlet to perform resource-related processing. One typical use case for the `RESOURCE_PHASE` is for an individual Portlet to process Ajax requests. Another typical use case for the `RESOURCE_PHASE` is for an individual Portlet to generate non-HTML content (for download purposes) such as a PDF or spreadsheet document.

Download the complete JSR-286, Portlet 2.0 specification from <http://bit.ly/11ua0oT> to know more about these concepts.

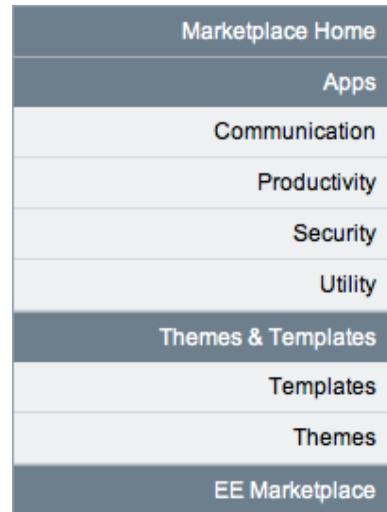
3.5.4 Liferay Marketplace

I would like to introduce you to Liferay Marketplace, a repository where Liferay has created an ecosystem for people to use (*consume*) and share (*contribute*) various useful plugins. Let's see how you can also benefit out of Liferay Marketplace both as a Consumer and as a Contributor. In other words, it is an online store of Liferay compatible apps (plugins). These include Portlets, Themes, Layouts and Hooks.

As a Consumer (User)

If you want to download some plugins from the Liferay Marketplace and inject them into your local Liferay installation follow these seven steps:

1. Login as Super admin (Liferay calls this as Omni Admin)
2. From the top bar pane, hover on “Go To” and click “Control Panel”
3. From the last section “Server”, click “Plugins Installation”
4. You will see the list of plugins that are currently installed inside the portal server
5. Now click on the “Install More Portlets” button from this screen
6. If this is your first time you will be prompted to enter your user id and password of your Liferay.com account. If you do not have an account, you can create one at this point.
7. After you enter your proper Liferay.com account credentials, you will be redirected to the Liferay marketplace home from where you can install additional plugins into your portal server.



Some of the plugins in the Marketplace work only with the Enterprise Edition of Liferay. This is indicated in the plugin itself. Liferay's official documentation covers the details of how you can benefit out of this marketplace as a Consumer (user). You can check the details from <http://bit.ly/WV5vvW>. The “**Purchased**” link under “**Marketplace**” section of the Control Panel gives the list of all apps that are purchased by you. From this list, download and install any particular plugin into your portal server on the fly.

As a Contributor (Developer)

As a developer if you have developed good and useful plugins you can make them available in the marketplace either for free or for some cost. . In order to publish your apps (plugins) to the Liferay's marketplace you have to fulfill certain basic requirements. The first and foremost is to register yourself or your company as a marketplace developer through this link <http://www.liferay.com/marketplace/become-a-developer>.

You can either create an Individual Account or a Company Account. This application goes through an approval process by Liferay's team; after approval you get enlisted as a registered Marketplace developer. This gives you the liberty to publish your apps in the marketplace. The apps (plugins) that you publish should also comply with certain norms set by Liferay to ensure your plugin downloaded by other users does not have any malicious code that will break the target system. You can get the complete details of this process from <http://bit.ly/11uaf30>.

Apart from the apps (*plugins*) that get officially listed in the Liferay market place, Liferay has a whole lot of sample plugins developed for the benefit of all of us. You can access them through this link, <http://svn.liferay.com/repos/public/plugins/trunk/>. You can also download any of these plugins into your own “plugins-sdk” and start re-building them. Alternatively, you can access the hot-deployable WAR files for these plugins (apps) from <http://releases.liferay.com/plugins/6.1.0-rc1/>.

Summary

Congratulations! You ‘have completed the first part of our book. Liferay is up and running. We have also made ourselves comfortable with the development environment. We have pointed Liferay to a production ready database. We have seen various concepts around Portlet 2.0 specifications like Portlet Modes, Window States, Portlet Requests, etc. Finally, we have taken a look at Liferay’s marketplace and how one can get benefit out of the marketplace both as a developer and as a consumer.

As I mentioned at the beginning of this chapter, before you move on to the next chapter, it is strongly recommended to play around with the various portal features as a portal administrator. You will get of the required guidance from Liferay’s official documentation for portal administration and also from the books that I mentioned in the beginning of this chapter. If you are running low on time and you aren’t able to browse through the portal administration contents, that’s OK as well. You can continue with the rest of the book. I am sure you will get a lot of inputs from this book itself that will help you to effectively administer a Liferay portal.

4. Library Management System

This chapter covers

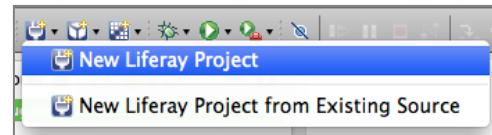
- A New Liferay Project
 - Establishing Basic Page flow
 - Anatomy of a Portlet URL
 - Creating a Simple Form
 - HTML form to AUI form
 - Injecting 3rd Party Libraries
-

We are now getting on to the real and serious Portlet development. In the last three chapters we have set the stage to enable a smooth development phase. In chapter one, we covered the general definitions of portals and portlets, comparison between Portlets and Servlet, Portlet specification JSR-286, the Liferay difference, etc. In chapter two, we learnt how to do all the required installations in order to get Liferay up and running – Java (JDK), Ant, MySQL, Eclipse and supporting tools. In chapter three, we setup Liferay IDE and made all other configurations required before and after getting the portal up and running on the browser. At the end of chapter three, we have discussed plugin packaging and how to make it a single WAR file.

In this chapter we are going to start developing our hypothetical “**Library Management System**”. Starting from this chapter, all the chapters will introduce some new concept pertaining to Portlet development. Hence it is very important for you to keep the sequence of your reading intact till you complete the whole book. Once you complete reading the whole book in sequence, you are free to refer to any chapter in random. Just a word of caution, this hypothetical application will be no way close to a real world Library Management System. But you can make it a full-fledged application and take it to the market if you really wish so.

4.1 New Liferay Project

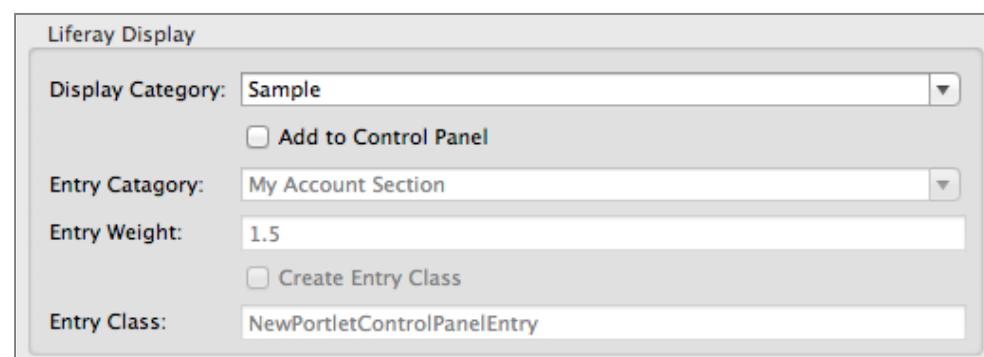
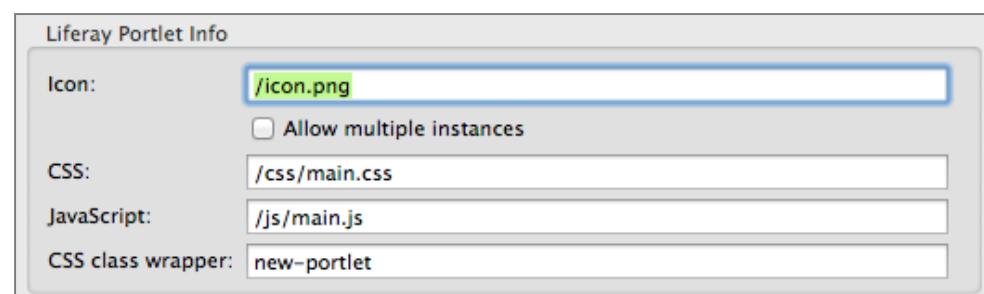
Let's create a fresh Liferay Project from the eclipse menu options. We'll keep this as a separate project and not merge with the one that we have created earlier – “**my-first**”.



You name this new project simply as “**library**”. The project name is purely at your discretion. The rest of the steps are very similar to the ones explained in section [3.5 New Liferay Project](#), except for one deviation – instead of clicking “**Finish**” in the “New Liferay Project” dialog, click “**Next >**”. In the next screen, leave the default option of “Liferay MVC”, check the option “**Create custom portlet class**” and click “**Next >**”. In the resulting screen, give the options as given below and click “**Next >**”.

Portlet Class	LibraryPortlet
Java package	com.library
Super class	com.liferay.util.bridges.mvc.MVCPortlet

There are four sections in the next screen – Portlet Info, Portlet Modes (JSR-286), Liferay Portlet Modes (Liferay’s extensions) and Resources. Leave everything as they are except checking the option “**Create resource bundle file**” and click “**Next >**”. The next screen prompts you to enter additional information for “**liferay-portlet.xml**” and “**liferay-display.xml**” files as shown in the following images. You can leave it with the default values and click “**Finish**”. This will create “**library-portlet**” project with the default structure, except for one variation – “src” folder under “WEB-INF” will contain the custom Portlet class “**LibraryPortlet.java**” and resource bundle file “**Language.properties**”.



As usual deploy the new “**library-portlet**” to the server as explained in Section [3.5.1 Deploying “my-first” to Server](#). Once it is successfully deployed, add this new Portlet on to a new page “My Library” by logging in as Omni Administrator. In the “30-70” layout of “My Library” page add this Portlet to the right column and set its borders by clicking “Look and Feel” from Portlet options. There is a reason for adding this to the right side that will be explained soon. The Portlet now appears majestically in the new page. Congratulations!

4.1.1 Types of portlets with Liferay IDE

Here, I want to quickly touch upon the types of Portlets that can be created with the help of Liferay IDE for Eclipse. Apart from the default “Liferay MVC” Portlet, we can quickly create and deploy the following types of Portlets either to the local server or to the remote server, by providing the required details.

JSF Standard	Standard UI components provided by the JSF runtime
Liferay Faces Alloy	Components that utilize Liferay’s Alloy UI technology based on YUI3
ICEFaces	Components based in part on YUI and jQuery with automatic Ajax and Ajax Push support
PrimeFaces	Lightweight, zero-configuration JSF UI framework built on jQuery
RichFaces	Next-generation JSF component framework by JBoss
Vaadin	A Java framework for building modern web applications which can be run as Portlets inside of Liferay portal

Going into the details of each of these types is not in the scope of this book; up to you to explore the other frameworks once you complete reading this book. All these frameworks provide excellent capabilities and libraries to develop JSR-286 Portlets and Liferay officially support some of them.

4.1.2 SVN Repository

As mentioned in the “Note” of section [2.4.2 Install Subclipse for subversion](#), I’ve checked in the code for our library Portlet in the Google code subversion repository for your ready reference. I will also mention the version number so that you can always know the exact code / changes that went inside the Portlet after every step / chapter. The base Portlet that just got created is checked in with the version number 4. You’re free to refer to this code during any time of your real time project development as many of the developers in mPower Global do.

Code Changes @ <http://code.google.com/p/lr-book/source/detail?r=4>

4.2 Establishing Basic Page flow

This section will explain how to establish a basic page flow (*navigation*) using Liferay MVC framework. This will be accomplished in next four steps.

Step 1: Let us start by creating a new JSP file “**update.jsp**” inside folder “**docroot/html/library**”. You can do this by Right clicking on this folder → New → File → specifying a new file name and clicking “**Finish**”. Put this line here and save.

```
<h1>Add / Edit Form</h1>
```

Step 2: Open “view.jsp” and replace the line given here with the block given next.

```
This is the <b>Library Portlet</b> portlet in View mode.
```

```
<portlet:renderURL var="updateBookURL">
    <portlet:param name="jspPage" value="/html/library/update.jsp" />
</portlet:renderURL>
<br/><a href="<% updateBookURL %>">Add new Book &raquo;</a>
```

We have just defined a “**renderURL**” using the Portlet tag (*declarative way*) and created an anchor tag to invoke the same. Go to “My Library” page in the browser and verify that the link “Add new Book” is taking you to the **Add / Edit Form**.

Step 3: In this step we’ll create a link to navigate back to the default view of the Portlet. Append this line to “**update.jsp**”.

```
<br/><a href=">&laquo; Go Back</a>
```

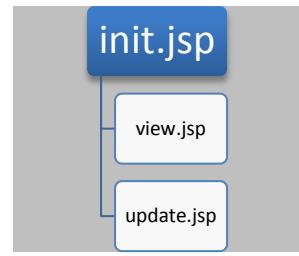
Check the Portlet now and you will see a “« Go Back” link. Unfortunately clicking on this link causes the portal to throw a “**Not Found**” page for the simple reason that the tag “**<portlet:renderURL/>**” is not visible inside “**update.jsp**”. We can fix this issue in the next step by defining a common “**init.jsp**”.

Step 4: Create a new file “**init.jsp**” under “**docroot/html/library**”. Move the below two lines from “**view.jsp**” to “**init.jsp**”.

```
<%@taglib uri="http://java.sun.com/portlet_2_0" prefix="portlet" %>
<portlet:defineObjects />
```

Insert “**<%@include file="/html/library/init.jsp" %>**” as a first line in other JSP files – “**view.jsp**” and “**update.jsp**”. Save your changes and check Portlet in the browser. The back link should be working properly now. The thumb rule here is to place all the commonly used taglibs and import statements in “**init.jsp**”. This useful feature of Liferay has been explained in the next section.

Know this Pattern: There will be some Java code in your JSPs that may require you to import some classes and tag libraries. A feature that Liferay uses to make this easier is to throw all imports, tag library declarations, and variable initializations in one file called “**init.jsp**”. Every other JSP that is created imports this “**init.jsp**” so it can take advantage of those declarations.



Congratulations, you have successfully established a basic page flow. It is now time for us to review a couple of issues.

1. We have used the “http://java.sun.com/portlet_2_0” taglib to define a Portlet URL that is quite different from a normal URL. We will see the anatomy of this URL very soon.
2. In the “**update.jsp**”, we have used the self closing tag, “`<portlet:renderURL/>`” to return to the default view of the Portlet without mentioning any specific JSP page to forward to.

Do it yourself: Find out all the tags of Portlet_2_0 taglib. (*Tip – To do this, open “view.jsp”, go to the last line and enter “<p”*. This feature of Eclipse is called as “Auto Suggestion” which is a great boon for Portlet development using Liferay IDE).

4.3 Anatomy of a Portlet URL

We need to spend some time in understanding the Portlet URL and what it contains. Technically, the JSR-286 specification has defined an interface to represent such an URL to the portal server or Portlet container, “`javax.portlet.PortletURL`”. Liferay in turn defined classes that inherit this interface. Those classes are – ActionURL, RenderURL and ResourceURL. We will come back to these different URL implementations later. For the time being, we will see the parts of a render URL.

Go to the Portlet and click “Add new Book ». Observe the URL in the browser’s location bar. This is how it will look like.

```
http://localhost:8080/web/guest/my-
library?p_p_id=library_WAR_libraryportlet&p_p_lifecycle=0&p_p_state=normal&p_
p_mode=view&p_p_col_id=column-
2&p_p_col_count=1& library_WAR_libraryportlet_jspPage=%2Fhtml%2Flibrary%
2Fupdate.jsp
```

I am sure you are unsettled by the complexity of this URL. Don’t worry. Let us break down this URL into smaller parts and know the purpose of each.

- <http://localhost:8080/web/guest/my-library> - The base URL of the page where the library Portlet is deployed. On real server it will not be “**localhost:8080**”.
- The remaining part is the query string separated by “&”. Let us further dissect these parts

Parameter	Value	What is conveyed to Portlet container?
p_p_id	library_WAR_libraryportlet	The Portlet ID as identified by the Portlet container. Every Portlet will have a unique ID. It is also known as the Portlet namespace.
p_p_lifecycle	0	Phase of the Portlet lifecycle has to be invoked <ul style="list-style-type: none"> • “0” - RENDER_PHASE • “1” – ACTION_PHASE
p_p_state	normal	Resultant window state of the Portlet – normal, maximized or minimized.
p_p_mode	view	Resultant Portlet mode – view, edit or help as per JSR-286
p_p_col_id	column-2	Column (cell) of the layout this Portlet should get rendered after the request is served
p_p_col_count	1	The index of this cell out of all the cells in the layout. In the current 30-70 layout there are two cells. 1 is the index of the second cell

The last attribute “_library_WAR_libraryportlet_jspPage” conveys to the Portlet container the next page (JSP) of the Portlet the request has to be redirected to. If the default view of the Portlet has to be invoked this parameter will not be there as part of the URL. Section [3.5.3 Visual Elements of a Portlet](#) explain the concepts of Portlet Lifecycle, Portlet Modes and Window states at greater length. In the later chapters we'll see how to convert this complex un-friendly URL into a simpler form called as “Friendly URL” hiding all the details from the user and making it Search Engine Optimization(SEO) friendly. To know more about SEO visit Wikipedia article <http://bit.ly/13TcRIZ>.

4.3.1 Liferay and Search Engine Optimization

Liferay provides lot of support for Search Engine Optimization (SEO). Liferay is often used to build public websites, and for this reason it provides a wide range of features to help make such sites show up at the top of the search results. Liferay implements the Sitemap Protocol to notify Google or Yahoo of the sitemap of a web site and enhance the chances of getting listed in the search results of these search engines. For a deeper understanding of Liferay's support for SEO, I recommend, you spend some time going through this Liferay Wiki on the subject [<http://bit.ly/YS2bpr>].

4.4 Creating a Simple Form

After establishing the basic navigation flow in the last section, in this section, we'll introduce a simple HTML form to add a book to our library. Insert the below block of code into “**update.jsp**” before “Go Back” link.

```
<%>
    PortletURL updateBookURL = renderResponse.createActionURL();
    updateBookURL.setParameter(
        ActionRequest.ACTION_NAME, "updateBook");
%>
<form name="fm" method="POST"
      action="<%= updateBookURL.toString() %>">
    Book Title:
    <input type="text" name="bookTitle" />
    <br/>Author:
    <input type="text" name="author" /> <br/>
    <input type="submit" value="Save" />
</form>
```

The first part of the above code is JSP scriptlet where we define a PortletURL programmatically followed by a simple HTML form. The interfaces “PortletURL” and “ActionRequest” will report errors. To get rid of these errors add the following imports in your “**init.jsp**” between the initial two lines that were there already.

```
<%@page import="javax.portlet.PortletURL"%>
<%@page import="javax.portlet.ActionRequest"%>
```

When you click the link “[Add new Book](#) »” in the Portlet, you will see a simple form. Though it is not formatted, we will leave it as it is for the time being. Click “Save” and you will see this error message thrown by the portal server.

 Portlet is temporarily unavailable.

Add / Edit Form

Book Title:	<input type="text"/>
Author:	<input type="text"/>
<input type="button" value="Save"/>	

Let's go to the server console and find out the root cause of this problem. It is obvious from the message that the portal server is not able to find the method “**updateBook**” which is given as part of our ActionURL parameter “**ACTION_NAME**”.

```
SEVERE: Servlet.service() for servlet library Servlet threw exception
java.lang.NoSuchMethodException:
com.library.LibraryPortlet.updateBook(javax.portlet.ActionRequest,
javax.portlet.ActionResponse)
at java.lang.Class.getMethod(Class.java:1622)
```

Where do we define this method? The answer is “**LibraryPortlet.java**” – our Portlet class. Let us open this class and create the method inside it.

```
public void updateBook(ActionRequest actionRequest,
                      ActionResponse actionResponse)
                      throws IOException, PortletException {
    String bookTitle =
        ParamUtil.getString(actionRequest, "bookTitle");
```

```

String author =
        ParamUtil.getString(actionRequest, "author");
System.out.println(
        "Your inputs ==> " + bookTitle + ", " + author);
}

```

The utility class “**ParamUtil**” will complain of compilation issues. Place the cursor above this text and press **Ctrl+Space** to import the corresponding dependency into this file – “com.liferay.portal.kernel.util.ParamUtil”.

How to leverage auto-suggestion feature?

The best way to define any new method inside this class is to do the following:

1. Keep the cursor between the main curly braces “{“ “}” and press **Ctrl+Space** to invoke the editor’s “Auto Suggestion” feature.
2. When the “Auto Suggestion” lists all the methods of this class, select “*processAction*” to insert the corresponding code automatically.
3. Remove the method annotation “@Override”, rename “*processAction*” to “*updateBook*” and remove the line that invokes the super class method, “**super**.processAction(actionRequest, actionResponse);”
4. Then insert whatever code you want to insert inside the method body

Refresh the browser, go to “My Library”, click “[Add new Book »](#)”, enter proper values for a book and click “Save”. You will see the message “Your inputs ==> Liferay Portlet Development, Ahmed Hasan” appearing on the server console and the Portlet now rendering the default view with the below confirmation message,



4.4.1 Avoid Hard Coding

As a good programmer you should avoid hard coding of literals inside your code. Try to externalize them as much as possible. This has multiple benefits. The first and foremost of them is the readability and maintainability of the code. Let’s quickly do some changes to our current code in order to move the literal “*updateBook*” and replacing it with a constant declared in another file “**LibraryConstants.java**”. Let’s do this in the following three steps.

Step 1: Create an interface “**LibraryConstants.java**” inside package “**com/library**” inside “**WEB-INF/src**” and declare a constant as below.

```
static final String ACTION_UPDATE_BOOK = "updateBook";
```

Step 2: Make the corresponding import in your “**init.jsp**” after the existing ones,

```
<%@page import="com.library.LibraryConstants" %>
```

Step 3: Open “**update.jsp**” and replace the string “*updateBook*” (*including quotes*) with, `LibraryConstants.ACTION_UPDATE_BOOK`.

Confirm that everything works fine as before. Liferay has used this kind of constants file throughout its code base. Let's do a quick exercise to find out how many constants file are there in Liferay portal source. In fact this exercise will make you comfortable with the search feature of Eclipse.

1. Open the Portal Source project that we closed earlier – “**liferay-portal-src-6.1.1-ce-ga2**”.
2. In the Eclipse window, Press **Ctrl+Shift+R** to open the search dialog. In the Search box enter “*Constants.java” to show all Constants files.

There are around 80 constant files excluding the one we have just written. You can open few of them and check their contents.

Do it yourself: One quick task for you. Move “`/html/library/update.jsp`” out of “`view.jsp`” and replace it with a constant defined in “**LibraryConstants.java**”,

```
static final String PAGE_UPDATE = "/html/library/update.jsp";
```

Congratulations, you have successfully implemented a new action inside our Portlet class and invoked it. There are some important lessons to be derived from this section. They are presented in the following sub-sections. Before moving forward, I've checked in all the changes so far to our SVN repository.

Code Changes @ <http://code.google.com/p/lr-book/source/detail?r=5>

4.4.2 RenderRequest Vs. ActionRequest

The following table gives the primary difference between these two significant types of Portlet requests.

RenderRequest	ActionRequest
Made by invoking a “renderURL”	Made by invoking an “actionURL”
Directly takes the Portlet through its RENDER_PHASE	Takes the Portlet through its ACTION_PHASE followed by RENDER_PHASE
Invokes the “ <code>render</code> ” method of the Portlet class.	Calls the “ <code>processAction</code> ” method of the Portlet class followed by invocation of “ <code>render</code> ” method.
Similar to HTTP GET method	Similar to HTTP POST method

Fundamental differences between “GET” and “POST”:

The HTML specifications technically define the difference between "GET" and "POST" requests. The former means that form data is to be encoded (*by a browser*) into a URL while the latter means that the form data is to appear within a message body. But the specifications also give the usage recommendation that the "GET" method should be used when the form processing is "idempotent" (*In computing, an idempotent operation is one that has no additional effect if it is called more than once with the same input parameters.*), and in those cases only. As a simplification, we

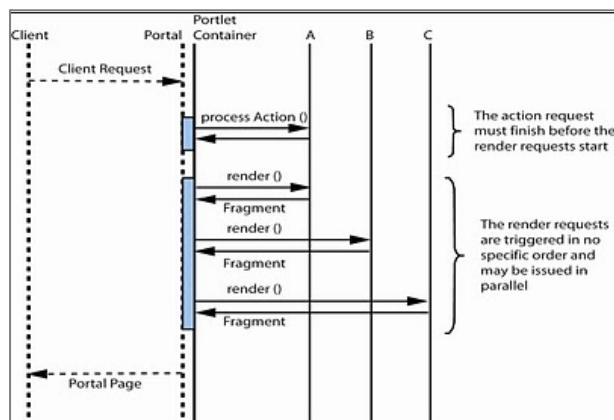
might say that "GET" is basically for just getting (*retrieving*) data whereas "POST" may involve actions like storing or updating data, or ordering a product, or sending e-mail.

Note on ResourceRequest:

Portlet 2.0 specification defines a third type of request called "ResourceRequest" (`javax.portlet.ResourceRequest`). This interface represents the request sent to the Portlet for rendering resources. It extends the interface `ClientDataRequest` [<http://bit.ly/XDbO7O>] and provides resource request information to portlets. The Portlet container creates a `ResourceRequest` object and passes it as argument to the Portlet's "serveResource" method. `ResourceRequest` is usually used to retrieve some resource from the server through the Portlet, e.g. a PDF document or an image. We'll see this usage of `ResourceRequest` in one of our upcoming chapters.

4.4.3 URL Formation

It is quite important to understand the difference between a normal HTTP URL and a Portlet URL. A HTTP URL makes a request that goes and hits the "Web Server" where as a Portlet URL gives rise to a `PortletRequest` that goes and hits the Portal Server / Portlet container. The Portlet container then forwards the request to the appropriate Portlet to which the request is destined. The following figure visually depicts the Portlet request sequence.



A Portlet URL can be formed either declaratively (*using the tags*) or programmatically. We have seen both these approaches in previous sections. Liferay also provides some utility classes for programmatically generating a Portlet URL dynamically inside the java code. Check with the class and its methods [<http://bit.ly/ZoCgVu>] – `PortletURLFactoryUtil`.

4.4.4 Usage of tag, <portlet:namespace/>

Name spacing ensures that the given name is uniquely associated with this Portlet and avoids name conflicts with other elements on the portal page or with other portlets on the same page. It is a best practice to always name space all HTML controls used in our Portlets and also the JavaScript functions to avoid any kind of potential conflicts. In the examples we have seen above, we have name spaced every html form element – `bookTitle` and `author`. In some situations, you may not able to use the taglib `<portlet:namespace/>`. In Such occasions, you can make use of the "getNamespace" method available inside "renderResponse" object.

```
String portletNS = renderResponse.getNamespace();
```

Let's see a practical scenario that mandates the need for name spacing JavaScript functions within the Portlets. Developer A develops Portlet A and developer B develops Portlet B and both have kept their Portlet on the same page. Both of these portlets have a submit button. There is always a high possibility that both the portlets have the same JavaScript function name for submit button; and when you hit submit button of Portlet A, You can never say which JavaScript submit function will be called since both the portlets have exactly same function name.

To overcome this issue, JSR 168 introduced Portlet namespace that differentiates two or more JavaScript functions with the same name. By prefixing `<portlet:namespace/>` to a JavaScript method, it makes it unique.

```
function <portlet:namespace/>submit() {  
    // some code here...  
}
```

4.4.5 Methods of ParamUtil class

We have already used this class in our Portlet class. This class provides very convenient wrapper classes to read different types of values from different types of objects –

- Portlet request (`javax.portlet.PortletRequest`)
- Service Context (`com.liferay.portal.service.ServiceContext`) and
- Servlet Request (`javax.servlet.http.HttpServletRequest`)

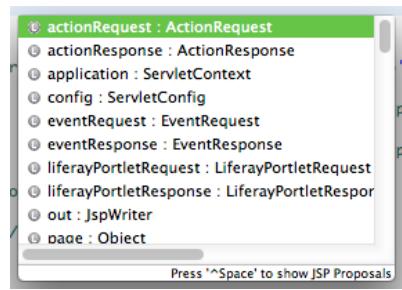
The values can be directly read as String, integer, float, Boolean, long, Double, Date, etc. You can now check all the methods of this class by opening the file “**LibraryPortlet.java**”, put an extra “.” immediately after this class and activating the auto-suggestion feature of Eclipse. Liferay provides many such utility classes. We'll see some of them as we move forward.

4.4.6 Implicit Objects

The tag “`<portlet:defineObjects/>`” you have used inside the “**init.jsp**” injects some very useful objects into our JSP. This tag establishes three objects; `renderRequest`, `renderResponse` and `portletConfig` for use in included Portlet JSP pages as per JSR-168 specification. JSR-286 has enhanced the inclusion of more implicit objects in the JSP.

- `renderRequest` and `renderResponse`
- `resourceRequest` and `resourceResponse`
- `actionRequest` and `actionResponse`
- `eventRequest` and `eventResponse`
- `portletSession`
- `portletPreferences`

You can do a quick check by opening your “**update.jsp**” file; place the cursor in a new line inside the JSP scriptlet where we defined “`updateBookURL`” and hitting **Ctrl+Space**. This command will list all the available implicit objects. Apart from the implicit objects defined by the Portlet 2.0 specification, Liferay has defined more implicit objects that we will see in the later sections.



4.5 HTML form to AUI form

Having seen the basic page flow and a simple HTML form to input a new book, we will now focus on building a form that is much more robust, HTML5 + CSS3 compliant and based on an extremely powerful set of tags provided by Alloy UI. Alloy is a UI meta-framework that provides a consistent and simple API for building web applications across all three levels of the browser: structure, style and behavior. AlloyUI, abbreviated as AUI, is another amazing product of Liferay built completely on top of Yahoo UI framework. You can check the complete details of this Liferay product from its home page, <http://alloyui.com>.

In order to use AUI tags in your JSP pages, you first have to include the corresponding taglib URI in your “**init.jsp**”.

```
<%@taglib uri="http://liferay.com/tld/aui" prefix="aui" %>
```

Open “**update.jsp**” and replace the form composed of HTML tags (`<form>`) with the one using AUI tags (`<aui:form>`).

```
<aui:form name="fm" method="POST"
    action="<% updateBookURL.toString() %>">
    <aui:input name="bookTitle" label="Book Title" />
    <aui:input name="author" />
    <aui:button type="submit" value="Save" />
</aui:form>
```

See how the complexity of the code has been reduced . By using the `<aui>` tags, we are no longer required to explicitly use “`<portlet:namespace/>`”. Refresh the browser and confirm everything is working fine as usual. Here are, four simple tasks to help you explore more features:

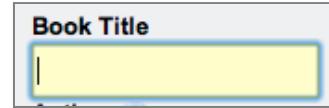
1. Open any of your JSP files, place the cursor in a new line, type characters “`<`” and “`a`” and list down all the “**aui**” tags that are available for use.
2. Place your cursor inside “`<aui:input name="author" />`”, just before the closing tag and list down all the “attributes” of this tag.
3. What happens if you remove the label attribute from “`bookTitle`” input field?
4. Add the “`helpMessage`” attribute for “`author`” input field and observe what impact you see in the Portlet UI.

4.5.1 Setting focus on first field

Go back to the browser and have a look at the current form. Wouldn't it be better if “`bookTitle`” was focused on when the page is loaded? Liferay has solution for this. Append the below code at the end of “**update.jsp**”.

```
<aui:script>
    Liferay.Util.focusFormField(
        document.<portlet:namespace/>fm.<portlet:namespace/>bookTitle);
</aui:script>
```

Check the Add Book form and now the focus is on the first field. The objective of this exercise was to illustrate the power of Liferay's own set of JavaScript libraries.



The libraries are grouped as modules. You can natively call these JavaScript API's within your Portlet functionality. You can find these modules under Liferay's portal source – “**portal-web/docroot/html/js/liferay**”. The Util module is inside “**util.js**”. We will see more examples of the usage of these Liferay's JavaScript API's in the later chapters of this book.

4.5.2 Liferay's Custom JavaScript

Liferay includes a lot of custom JavaScript and many of them are dependent on AlloyUI. These libraries help you to create a dynamic, modern web site. You will find that a lot of common functions you want, are already covered by Liferay's included JavaScript functions. These API's are all properly namespace, so even if you don't wish to use Liferay's implementations of certain pieces of functionality, you are free to write your own. If you are a Java developer you will appreciate the fact that many of the tag libraries you will use, implement their functionality using Alloy UI's JavaScript. The exhaustive list of JavaScript functionality available can be found in Liferay's source under “**portal-web/html/js/liferay**”. Here is a list of some common functionality you as a web developer, may want to use:

- `Liferay.AutoFields` – Contains functions for forms which have fields that repeat. We will use this feature to implement “Bulk Add” feature for books into our Library in one of the upcoming chapters.
- `Liferay.ColorPicker` – Creates an inline dialog that allows users to pick a color already being displayed and have that color's hexadecimal value inserted into a field.
- `Liferay.Language` – Allows users 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.
- `Liferay.Notice` – Controls the notification area that appears at the top of the screen. This notification appears if you have let your session time out. Liferay displays a countdown, and when it reaches zero, you're automatically logged out. You can use this area for your own notifications. As Liferay uses this notification area, you should avoid using it a lot, or you may run into instances where you unintentionally cover up a notification that Liferay is trying to display to the user.

-
- `Liferay.Panel` – 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 (*that is, it can be collapsed and certain events can be sent to it*).
 - `Liferay.Upload` – The widget used by the Document Library portlet to upload files. We can use this widget in our Portlet as well.
 - `Liferay.Util` – Contains many utility methods for working with document elements and form elements. For example, there are methods for dealing with check boxes (*such as `checkAll()`*), for enabling and disabling elements, for escaping HTML so it can be inserted into a database, for focusing fields, and much more. We have already seen a usage of this in the previous sub-section.

Let's look at some code that uses JSP to call Liferay JavaScript objects. One option is,

```
<aui:script use="liferay-autofields,liferay-notice,liferay-upload">
    new Liferay.Upload();
    new Liferay.AutoFields();
    new Liferay.Notice();
</aui:script>
```

Note here that, similar to a Java **import** statement, you have to declare what you're going to use first. This is because the modules aren't initialized right away; they're lazy-loaded. This speeds up the rendering of the page.

The same can also be done via JavaScript. This code uses JavaScript to call Liferay JavaScript objects. This gives us the second option.

```
AUI().use(
    'liferay-autofields', 'liferay-notice', 'liferay-upload',
    function(A){
        new Liferay.Upload();
        new Liferay.AutoFields();
        new Liferay.Notice();
    }
);
```

It is done in a similar way. The `AUI()` object is the global Alloy UI object, and it's used to do the initialization. The one exception to this initialization process is the `Liferay.Util` class. You don't have to explicitly initialize that class in order to use it; it's available by default on every page. We'll see many examples of AUI / Liferay JavaScript usage scattered all over the book as we move forward.

Note: *Liferay Version 6.1.1 uses Alloy UI 1.0.1 version. At the time of writing this book, Liferay has released another new version of AlloyUI that is 2.0. This latest version introduces many more advanced features including Diagram Builder, Scheduler, Form Builder, Image Cropper, Carousel, Char Counter, Data Table, TabView, TreeView, etc. When I tried to use this new version into our Library Portlet, I came across some compatibility issues. Hopefully this new version of AlloyUI will get fully integrated with Liferay from the next version of Liferay onwards.*

4.5.3 Form Validation

There is one problem in the current form. It allows the user to submit the form without valid inputs, though this is not a desired behavior. It is always important to put some basic validations on the client (browser) side using JavaScript to avoid unwanted and irrelevant inputs coming from the users. Why it is important to do validate fields “*Validation ensures that the user has provided sufficient information for a transaction to be completed successfully. Validation checks the integrity of the information provided*”. By having proper client side validation you can avoid unwanted traffic flowing into the server and eating the network bandwidth.

Alloy UI tags provide the validation feature by default without writing much of the code. Let’s apply validation rules for one of our input fields and see the behavior now. After adding validation to “`bookTitle`”, the code looks like:

```
<aui:input name="bookTitle" label="Book Title" >
    <aui:validator name="required"/>
</aui:input>
```

Go to the browser and try to submit the form without giving any value for “`bookTitle`”. You will not be able to submit. This is the simplest of validation you can add using “`<aui:validator/>`” tag. You can change the default error message by explicitly supplying a value for the “`errorMessage`” attribute. The “`name`” attribute can take one of the following values:

- **alpha** – the field will accept only characters and not digits
- **alphanum** – the field will accept both characters and numbers
- **date** – the field will accept only dates in the format mentioned within the validator tag
- **digits** – the field will allow only digits to be entered (positive numbers including zero)
- **email** – the field will allow only valid email addresses
- **equalTo** – checks if the value entered in this field is equal to the value in some other field of the same form. A good example is entering the password twice and checking both values are same
- **number** – same as “`digits`” but accepts both positive and negative values.
- **acceptFiles** – this is used for field whose type is ‘file’. This field can accept a comma separated list of file formats
- **min** – the minimum value the field can accept
- **minLength** – the minimum length of the field in terms of characters
- **max** – the max value this field can take
- **maxLength** – the max length of the field in characters
- **range** – the value of the field should be within the given range
- **rangeLength** – the length of the field in the given range in characters
- **required** – input for the field is mandatory
- **url** – the input should be a valid URL
- **custom** – we can write custom JavaScript validation for such a field.

Here is an example of a field, whose validator name is “`custom`”,

```

<aui:validator name="custom">
    function() {
        // validate an expression
        // return true or false based on the expression
    }
</aui:validator>

```

Using custom JavaScript you can even have dependency checks between the fields. For example the if the input for “**field2**” is dependent on the input for “**field1**”, the validation rules for “**field2**” will be different based on the input the user has given for “**field1**”.

A field can have more than one validator as you see in the below example:

```

<aui:input name="age">
    <!-- Make the field required. -->
    <aui:validator name="required" />

    <!-- Only allow digits in the field -->
    <aui:validator name="digits" />

    <!-- Make sure field value is between 1 and 99 -->
    <aui:validator name="range" >[1,99]</aui:validator >
</aui:input>

```

The following list shows the default error messages for the various validator types:

- **acceptFiles**: 'Please enter a value with a valid extension ({0}).',
- **alpha**: 'Please enter only alpha characters.',
- **alphanum**: 'Please enter only alphanumeric characters.',
- **date**: 'Please enter a valid date.',
- **digits**: 'Please enter only digits.',
- **email**: 'Please enter a valid email address.',
- **equalTo**: 'Please enter the same value again.',
- **max**: 'Please enter a value less than or equal to {0}.',
- **maxLength**: 'Please enter no more than {0} characters.',
- **min**: 'Please enter a value greater than or equal to {0}.',
- **minLength**: 'Please enter at least {0} characters.',
- **number**: 'Please enter a valid number.',
- **range**: 'Please enter a value between {0} and {1}.',
- **rangeLength**: 'Please enter a value between {0} and {1} characters long.',
- **required**: 'This field is required.',
- **url**: 'Please enter a valid URL.'

In this section we have seen some of the basic usage of **<aui>** tags and the validators. We will see more usages of AUI tags in the subsequent chapters.

4.6 Injecting 3rd Party Libraries

Liferay has a rich set of JavaScript library in-built with various modules. You will find all these packages, under “**portal-web/docroot/html/js/liferay/**” of your portal source code base. Apart from using these libraries and packages, you’re free to use any other third party JavaScript library in your Portlet(s). Liferay provides the following tags to inject a JavaScript library into our Portlet via “**liferay-portlet.xml**” file.

- `header-portal-javascript` – Path of JavaScript that will be referenced in the page's header relative to the portal's context path. You can include any JavaScript from “**portal-web/docroot/html/js**” to be available within your Portlet.
- `header-portlet-javascript` – Path of the JavaScript which is local to this Portlet. The path could be link to the local JS file or an external JavaScript URL.
- `footer-portal-javascript` & `footer-portlet-javascript` – Same as above, except that the JavaScript will get loaded after rest of the Portlet contents gets loaded.

Through these tags, you can inject any JavaScript library which is either available locally or remotely. Let us see a quick example of injecting jQuery library into our Library Portlet.

4.6.1 Injecting jQuery library

The popular jQuery [<http://jquery.com>] is not included in Liferay by default. If we want the jQuery functionality in our Portlet we will have to manually inject it through the Liferay Portlet configuration file. Let us do the following two steps.

Step 1: Insert below lines into the “**liferay-portlet.xml**” that will inject the new library to our Portlet. These lines have to be inserted immediately after, “`<header-portlet-css>`” line.

```
<header-portlet-javascript>
    http://code.jquery.com/jquery-latest.min.js
</header-portlet-javascript>
```

Step 2: Next, append this code in your “**view.jsp**” to test our library inclusion.

```
<hr/>
<a href="javascript:void();" id="jqueryText">jQuery in action</a>

<div id="sayHelloDiv" style="display:none">jQuery is working</div>

<script type="text/javascript">
    $(document).ready(function(){
        $('#jqueryText').on('click',function(){
            $('#sayHelloDiv').toggle();
        });
    });
</script>
```

We have added four elements –

- A horizontal ruler `<hr/>` to separate this feature, visually, from the rest of our functionality
- An anchor tag on which we have defined the click event
- A “div” that is hidden by default and will be shown or hidden based on clicking the anchor tag
- The JavaScript code where we’ have written the actual logic

Check the Portlet in the browser and keep clicking the link “[jQuery in action](#)”. You will see the magic of the “div” getting toggled.

4.6.2 Injecting jQueryUI library and its CSS

Let us try out another example on similar lines to improve our understanding. This time we shall inject another JavaScript library called jQueryUI [<http://jqueryui.com>] along with the related CSS. Let’s add some tabs to our Portlet with the help of this UI library by following these two steps.

Step 1: Insert the following CSS and JavaScript in appropriate location of “**liferay-portlet.xml**” file, using the tags “`<header-portlet-css>`” and “`<header-portlet-javascript>`”, respectively.

1	<code>http://code.jquery.com/ui/1.10.0/themes/base/jquery-ui.css</code>
2	<code>http://code.jquery.com/ui/1.10.0/jquery-ui.min.js</code>

One thing you have to ensure is to include the new JavaScript after the line for the base jQuery library as the former is dependent on the latter.

Step 2: Append the code below in your “**view.jsp**” to get the tabs support of jQuery UI library.

Save your changes and check the Portlet in the browser. You can even add the below line to the document ready function we have written earlier which will negate the need for a new `<script>` block.

```
$( "#tabs" ).tabs();
```

```
<hr/>
<div id="tabs">
  <ul>
    <li><a href="#tabs-1">Tab One</a></li>
    <li><a href="#tabs-2">Tab Two</a></li>
    <li><a href="#tabs-3">Tab Three</a></li>
  </ul>
  <div id="tabs-1">Tab 1 content</div>
  <div id="tabs-2">Tab 2 content</div>
  <div id="tabs-3">Tab 3 content</div>
</div>

<script type="text/javascript">
  $(function() {
    $("#tabs").tabs();
  });
</script>
```

“`$(function())`” is a short-hand notation for “`$(document).ready(function())`”.

Whenever you want to override the CSS of these libraries, you can do so by putting those CSS classes inside your Portlet's “**main.css**” and re-defining those classes that you want to override.



4.6.3 Common JavaScript libraries

There are many JavaScript libraries you can inject into your Portlets using the method described above. Whenever you include these libraries, it is always better to pull them from their respective **CDN (Content Delivery Network)** hosts. This will not only improve the performance of loading but will also put minimal load on the server on which Liferay is running. Google hosts many of those common libraries freely available for us to use in our applications. The URL is <http://bit.ly/YQqD8S>.

4.6.4 Making a library available for all portlets

There will be situations where you want to make a particular JavaScript library available for more than one Portlets of your portal. In this case, it does not make sense to give individual entries in the “**liferay-portlet.xml**” of each and every Portlet. We need a better approach. Liferay provides two ways of doing that.

- **Including a library at the theme level** – you can include all the libraries that you want to make available for your Portlets at the theme level, usually in the file “**portal_normal.vm**”. But the problem with this approach is that whenever you change the theme in the portal, these libraries will be no longer available and the portlets that are dependent on these libraries will cease to function properly. You go with this approach ONLY if you're sure that your portal will make use of only one theme and it is not going to change forever.
- **Injecting through a hook** – this is the best approach to make certain JavaScript libraries available for all Portlets that are deployed into various pages of the portal. You can override the JSP file “**/html/common/themes/top_js-ext.jspf**” through a simple JSP hook and add the entries for the JavaScript libraries that you want to inject. For example, the following line needs to be inserted into this file and deployed as a hook, if we want to inject jQuery in this manner:

```
<script src="http://code.jquery.com/jquery-latest.min.js"></script>
```

There are some benefits of loading the JavaScript at the bottom of the HTML page. But we are not going to get into the details of that, as it is not within the scope of this book. If you're an UI expert and want to know more about this you can always refer to many resources that are freely available on the Internet.

Code Changes @ <http://code.google.com/p/lr-book/source/detail?r=6>

Summary

We started this chapter by creating a new Liferay project in Eclipse. We have seen the various kinds of Portlets that can be developed with the help of the latest version of Liferay plugin for Eclipse. Soon after creating our first project we have checked in the code to our googlecode SVN repository that will help us to refer to the code at any time we want. Then we moved on to establishing the basic flow between the various pages of the same Portlet. This was to illustrate how easy it is to set up the Portlet navigational system with the help of forming renderURL. We then dissected the anatomy of a Portlet URL and understood its components.

Then we moved on to create a simple HTML form that will accept the book information and submit to the backend server. We have seen the difference between a **renderRequest** and an **actionRequest**, formation of an URL using programmatic way and declarative way, usage of **<portlet:namespace/>** tag and its significance in the portal ecosystem. In the same section the various methods and API's of ParamUtil class, a powerful utility class provided by Liferay have been mentioned. Finally we have seen the list of various implicit objects that are available by default within a JSP file of any given Portlet.

Following that we have seen how to convert the simple HTML form to a rich AUI form using the AlloyUI tags provided by Liferay. We were able to appreciate the value of using the AlloyUI tags. Connected to this discussion, we have also seen how to use some Liferay JavaScript to set focus on a particular form field. Another important subject we covered in detail is the browser-based validations using AlloyUI validator tags. For most of the forms we should try to include as many validations with the browser itself to avoid un-necessary round trips between the browser and the server.

In the last section of this chapter, we have seen how to include and use other third party JavaScript or CSS libraries within our own Portlet. We have seen a list of commonly used third party JavaScript Libraries and also seen how to make one particular library available to all portlets instead of making it available only for one Portlet. Overall this chapter was the curtain raiser for the Portlet development that we are going to cover in the rest of the book. If you're comfortable with all the aspects that we've seen in this chapter, the rest of this book is going to be a breeze.

In the next chapter, we are going to embark on a very interesting and powerful topic – Service Builder and Service Layer. You should take a good break before getting into the next chapter, as it demands complete focus and attention. Wish you all the very best to master the concepts that we are going to cover very soon.

5. Service Layer and Service Builder

This chapter covers

- Service Builder Tools
 - Generating our first Service Layer
 - Invoking the Service Layer API
 - Pulling Data – Use another API
 - Avoiding multiple submits
 - Separating the Business Logic
 - Service Layer of Portal Source
 - Sharing a custom service layer
 - Caching to improve performance
 - Explanation for “**service.xml**” DTD
-

In the last chapter we have seen many powerful features of the Portlet framework. We have successfully created our Library Portlet and seen many of its important aspects. We have left the Portlet with a simple form where the user can supply some inputs and submit it. But what are we going to do with these inputs? Currently we are just printing them on the server console. Is this sufficient? No. We have more to do with these inputs. In a real application the data has to be stored somewhere, usually in a relational database system or sent to some other application through its exposed web services.

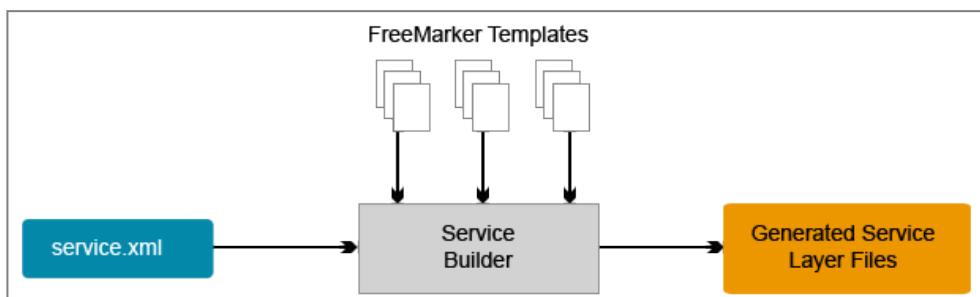
Let us assume that we want to persist the data to one of our underlying database tables that captures the information about all the books in our library. Conventionally we'll write some JDBC code to do this job. Imagine your application is going to be really big with hundreds of tables and many columns in each of them. Writing normal JDBC code is obviously not the right approach. In order to mitigate the limitations of JDBC API, modern java based web applications started using advanced API's called ORM (*Object Relational Mapping*) to indirectly talk to the database in the form of simple API's. Some of these popular ORM's are Hibernate (<http://www.hibernate.org>) and MyBatis (<http://blog.mybatis.org>). These frameworks are based on the JPA (Java Persistence API) specifications.

The Java Persistence API, sometimes referred to as JPA, is a Java programming language framework managing relational data in applications using Java Platform, Standard Edition and Java Platform, Enterprise Edition. You can get more details from http://en.wikipedia.org/wiki/Java_Persistence_API.

5.1 Service Builder Explained

Liferay has moved one step forward and built another layer generically called as “**Service Layer**” that provides lots of “services” – one of them is the persistence service. Liferay comes bundled with an extremely powerful tool called “Service Builder” which auto-generates the Service Layer with just one click. This tool takes a “**service.xml**” as input and generates whole lot of files that constitute the Service Layer. The generated files include classes, interfaces and configuration files based on Hibernate and Spring middleware. In other words the Service Builder is a “Model-Driven” code generator.

You should also know, technically, “Service Builder” is nothing but a java class, very much part of the portal source code. You can browse through this class by opening it inside Eclipse. Press **Ctrl+Shift+R** and type “ServiceBuilder”. In the result list, select the file and click “Open” button. You will find this file under “com.liferay.portal.tools.servicebuilder” package of “**portal-impl**” folder. It is a big file, running into more than 5000 lines of code. The files that are generated by Service Builder are kept as predefined “FreeMarker” templates (.ftl). You can find them under “**dependencies**” folder of the same package. FreeMarker (<http://freemarker.sourceforge.net>) is an open source powerful template engine, a generic tool to generate text output.



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, the data caching and the query logic stay inside the classes you never have to touch. You add code in a class that extends the generated class and if you add something that changes the interface / implementation contract, those changes are propagated up the chain so the contract is never broken. For those who are new to Spring (<http://www.springsource.org>), it is the most popular application development framework for the enterprise Java. Millions of developers use Spring to create high performing, easily testable, reusable codes without any vendor lock-in. Liferay extensively uses Spring as its middleware framework.

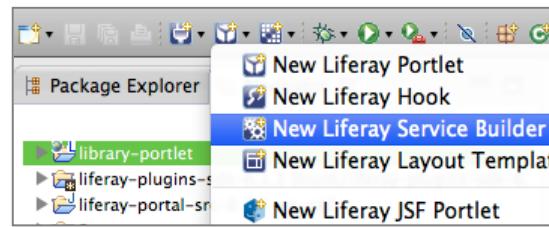
The contents of “**service.xml**” file will be in accordance to the semantics of its corresponding DTD file “**liferay-service-builder_6_1_0.dtd**”. You can find this file inside the “**definitions**” folder of portal source. We’ll see the explanation for the entries of this file in the later part of this chapter. Liferay IDE for Eclipse has the necessary wizard to visually create the “**service.xml**” file. But it can also be written by hand.

5.2 Generating our first Service Layer

Coming to one of the exciting stages of our book, we will now create our first Service Layer using Liferay's Service Builder tool. Using the API's generated by this layer, we are going to persist the book information into a table in the underlying database. After running this tool we'll come back and discuss the various files that got generated on the ground.

5.2.1 New Liferay Service Builder

While the “**library-portlet**” project is selected in Eclipse, click the second Liferay icon and select “**New Liferay Service Builder**” as shown in this figure. Alternatively, you can right click on the project → Hover on New → Liferay Service Builder.



In the dialog that opens, specify the values for Package path as “**com.slayer**”, Namespace as you wish (*e.g. LMS*), Author as your name and click “**Finish**”. This will create a dummy “**service.xml**” file with a sample entity and show the “Overview” of this file. The “Diagram” view displays the entities of this file and their relationship in a pictorial way. As a seasoned programmer, you can directly go to the “Source” view and start changing its contents. Replace the dummy entity “**Foo**” with a real entity **LMSBook** that has all the attributes of a real book in our library.

```
<entity name="LMSBook" local-service="true" remote-service="false">
    <!-- PK fields -->
    <column name="bookId" type="long" primary="true" />

    <!-- UI fields -->
    <column name="bookTitle" type="String" />
    <column name="author" type="String" />

    <!-- Audit fields -->
    <column name="createDate" type="Date" />
</entity>
```

Note that we have kept “**local-service**” as **true** and “**remote-service**” as **false**. This is the simplest entity you can ever think of with just four columns - one primary key column (**bookId**), two UI fields (**bookTitle** & **author**) and one audit field (**createDate**). Now it is time to actually generate the Service Layer. This can be done in two ways:

1. Pressing “Build Services” icon on the top right when you open the “Overview” mode of “**service.xml**”
2. Right click on “**library-portlet**” → Liferay → Build Services.

One of the above actions will start generating the service layer. At the end of this generation you will see the message “**BUILD SUCCESSFUL**” in your console output.

But isn't necessarily enough. Always scroll up the console window and make sure that there are no errors while generating the service layer. If there are any syntactic or semantic errors in the input “**service.xml**” file, the generation will not take place properly, but still you will see success message. Right click the “**library-portlet**” project and “Refresh” it to load all those newly generated files into the workspace. You are done!

5.2.2 Files Generated by Service Builder

The following files (*classes / interfaces*) are generated under “**WEB-INF/service**”.

Model	/com/slayer/model/LMSBook.java (<i>interface</i>) /com/slayer/model/LMSBookClp.java /com/slayer/model/LMSBookModel.java (<i>interface</i>) /com/slayer/model/LMSBookWrapper.java /com/slayer/model/LMSBookSoap.java
Persistence	/com/slayer/service/persistence/LMSBookPersistence.java (<i>interface</i>) /com/slayer/service/persistence/LMSBookUtil.java
Service	/com/slayer/service/ClpSerializer.java /com/slayer/service/LMSBookLocalService.java (<i>interface</i>) /com/slayer/service/LMSBookLocalServiceUtil.java /com/slayer/service/LMSBookLocalServiceClp.java /com/slayer/service/LMSBookLocalServiceWrapper.java
Messaging	/com/slayer/service/messaging/ClpMessageListener.java
Exception	/com/slayer/NoSuchBookException.java

The following files, all classes, are generated under “**WEB-INF/src**”.

Model	/com/slayer/model/impl/LMSBookModelImpl.java /com/slayer/model/impl/LMSBookBaseImpl.java /com/slayer/model/impl/LMSBookImpl.java /com/slayer/model/impl/LMSBookCacheModel.java
Service	/com/slayer/service/persistence/LMSBookPersistenceImpl.java /com/slayer/service/impl/LMSBookLocalServiceImpl.java /com/slayer/service/base/LMSBookLocalServiceBaseImpl.java* /com/slayer/service/base/LMSBookLocalServiceClpInvoker.java

* An abstract class

Spring and Hibernate related configuration files go under “**/src/META-INF**”. There are around ten of them in total. There is one “**service.properties**” file generated directly inside “**src**” which contains some settings for the service layer itself. Folder “**WEB-INF/sql**” contains the DDL statements to be run against the database to create the tables, indices and sequences for the entities that are defined in the “**service.xml**” file. The entire contents of “**WEB-INF/service**” get packaged as a JAR file, “**library-portlet-service.jar**” and gets into the “**WEB-INF/lib**” folder. You are strongly encouraged to open each of the java classes and interfaces and see how they are organized. We hardly modify them, except for one file (*DTO class*) “**LMSBookLocalServiceImpl**” under “**com.slayer.service.impl**” package.

The model classes are nothing but pure java objects with accessor methods (*getter* and *setters*) for that entity. You should also note, for one entity “LMSBook” you have a total of 22 (*there is one more file in the case of Liferay 6.2.X*) java files generated. For every entity you define another same set of files will get generated with the file names starting with the name of the entity. See the power of Service Builder tool. Without you writing one single line of code, the entire code base gets generated which takes care of the complete middleware and database access. Out of the generated files, one file is common for all entities – “**ClpSerializer.java**”, Class Loader Proxy.

5.2.3 Files to be checked in to SVN

After the generation of service layer the only two files that I am going to check in to the SVN repository are “**service.xml**” and “**LMSBookLocalServiceImpl.java**” (*DTO class*), because these are ones that will undergo changes. The other files can always get generated afresh. Hence it is discouraged to check them into the version control. I am going to defer this process till we do the next exercise of using one of the API’s to persist the book information.

Before we move on, let’s quickly check the database either using the MySQL prompt or MySQL query browser. You will notice a new table “**LMS_LMSBook**” got automatically created by Liferay when our service layer got deployed to the server. You don’t even have to create the tables manually. Thanks to Liferay everything is done for you. The table name is prefixed with the “Namespace” that you gave in “**service.xml**” followed by the actual entity name; in our case it is “**LMSBook**”. You can optionally disable this auto-prefix by specifying “**auto-namespace-tables="false"**” attribute for “**<service-builder>**” tag of the “**service.xml**”, immediately after “**package-path**” attribute.

5.3 Invoking the Service Layer API

With the successful generation of service layer, we’ll use one of the generated API’s to insert our book to the table when the form gets submitted. Open “**LibraryPortlet.java**” and insert the below code in “**updateBook**” method after the existing “**System.out.println**” line.

```
// 1. Instantiate an empty object of type LMSBookImpl
LMSBook lmsBook = new LMSBookImpl();

// 2. Generate a unique primary key to be set
long bookId = 01;
try {
    bookId = CounterLocalServiceUtil.increment();
} catch (SystemException e) {
    e.printStackTrace();
}

// 3. Set the fields for this object
lmsBook.setBookId(bookId);
lmsBook.setBookTitle(bookTitle);
lmsBook.setAuthor(author);
lmsBook.setCreateDate(new Date());
```

```
// 4. Call the Service Layer API to persist the object
try {
    lmsBook = LMSBookLocalServiceUtil.addLMSBook(lmsBook);
} catch (SystemException e) {
    e.printStackTrace();
}
```

As you can see, there are four distinct steps in this code:

1. Instantiate an empty object of type “**LMSBookImpl**”
2. Generate a unique primary key to be set for that object
3. Set the rest of fields for this object
4. Call the Service Layer API to persist the object to the database

Save your changes and check the Portlet in the browser. Try to add a book with some valid inputs. Confirm that a new row gets inserted into the “**LMS_LMSBook**” table every time you submit the Add Book form.

Congratulations!! You have successfully integrated the service layer API with our library Portlet. This is just the tip of the iceberg. You can do more wonders with the service layer. Infact the list of wonders is limitless. We’ll see some of them as we move forward.

5.3.1 Making our code as a new method

Don’t you think it will be better if we put our new code as a separate method in the same class instead of making “`updateBook`” method look very bulky? It is always good to break down the functionality of our class into smaller methods and this is what Object Oriented Programming is all about. Let’s make use of the Eclipse feature to do this refactoring instead of we doing.

Select the block of code that you want to make as a separate method. Then click “**Refactor → Extract Method...**” from the Eclipse Menu. In the dialog that opens, give the new method name as “`insertBook`” and click “OK” keeping all other options intact. You will see a new private method getting created and the original block replaced by one line of code, “`insertBook(bookTitle, author);`”. You can remove the print line statement. In later sections we’ll see how to use the logging feature. It is not a good practice to use “`System.out.println`” inside your code.

```
System.out.println("Your inputs ==> " + bookTitle + ", " + author);
```

Let’s move ahead and do some more changes in the subsequent sub-sections.

5.3.2 Auto-generating the primary key

You must have observed that for setting a unique primary key (`bookId`) we have relied on a service that is provided by Liferay for this purpose – CounterService. The “`increment()`” method of “`CounterLocalServiceUtil`” always returns a unique number from the “Counter” table. You can also have a custom counter for your entity. Using counter service of Liferay is highly preferred because your code becomes

completely agnostic of the underlying database and the application layer sets the primary key. Some databases have the capability of natively auto-generating a primary key for a record before it gets inserted into the table. MySQL is one such database. Let's see how to leverage this feature of MySQL to set the primary key. Do this only if you're sure that your application will not point to any other database in the production environment. This change requires three steps.

Step 1: Update the table definition to auto-generate the primary key by executing this command either from the MySQL prompt or the MySQL query browser.

```
ALTER TABLE LMS_LMSBook CHANGE COLUMN bookId bookId  
BIGINT(20) NOT NULL AUTO_INCREMENT;
```

To confirm the changes, Right click the table in MySQL Query browser and select

"Alter Table...". You will see the table's columns in this format with the option "AI" checked for "bookId" column.

Column	Datatype	PK	NN	UQ	BIN	UN	ZF	AI	Default
bookId	BIGINT(20)	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>					
bookTitle	VARCHAR(75)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	NULL
author	VARCHAR(75)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	NULL
createDate	DATETIME	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	NULL

Step 2: Open "**LibraryPortlet.java**" and comment out the code that generates and sets the primary key for "lmsBook" object. Just comment out and don't remove this code as we might require it back in later chapters. As a result of this change, the entire block "2. Generate a unique primary key to be set" and the first line of next block "lmsBook.setBookId(bookId);" will get commented out.

Open the browser and try adding a book to our library and confirm the insertion is happening properly. Also notice the primary key (*bookId*) of the newly inserted record. *New bookId = (previous bookId + 1)*. Though the functionality has worked perfectly fine, you have to explicitly inform the service layer that you're making use of the auto-increment feature of the underlying database. This is our next step.

Step 3: Open "**service.xml**", add new attribute "**id-type**" with value "*increment*" for column "*bookId*". Save file and re-generate service layer. The line will look like:

```
<column name="bookId" type="long" primary="true"  
id-type="increment"/>
```

Having completed this exercise, let us look at two major benefits of using this feature:

1. *Increased performance* – as the application is no longer has to set the primary key by invoking the relevant API's.
2. *Reduced code* – in our own example we have knocked down around eight lines of code, including the import statement for "**CounterLocalServiceUtil**".

Eclipse Tip: The imports in a class that are never used inside the code are underlined with yellow line. Press **Ctrl+Shift+O** to keep organizing the imports in a file.

5.4 Pulling Data – Use another API

The previous section covered pushing data to the database through the auto-generated persistence service of Liferay. In this section we will see how to pull data from the database through the same persistence layer. Let us demonstrate this with a simple exercise involving two steps. In this exercise, we'll start with simple HTML table and in the upcoming sections; we'll convert this table into a properly formatted grid with the help of appropriate tags provided by Liferay.

Step 1: Create a new file “list.jsp” inside “html/library” and put the below code.

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

<h1>List of books in our Library</h1>

<%
    List<LMSBook> books =
        LMSBookLocalServiceUtil.getLMSBooks(0, -1);
%>

<table border="1" width="80%">
    <tr>
        <th>Book Title</th>
        <th>Author</th>
        <th>Date Added</th>
    </tr>
    <%
        for (LMSBook book : books) {
            %
            <tr>
                <td><%= book.getBookTitle() %></td>
                <td><%= book.getAuthor() %></td>
                <td><%= book.getCreateDate() %></td>
            </tr>
            %
        }
        %
    </table>
<br/><a href=">">&laquo; Go Back</a>
```

Insert the required imports in “init.jsp”.

```
<%@page import="java.util.List"%>
<%@page import="com.slayer.model.LMSBook"%>
<%@page import="com.slayer.service.LMSBookLocalServiceUtil"%>
```

Step 2: Create a link in “view.jsp” to open “list.jsp” when clicked. You can insert this code immediately after the code for Add new Book URL.

```
<%
    PortletURL listBooksURL = renderResponse.createRenderURL();
    listBooksURL.setParameter("jspPage", "/html/library/list.jsp");
%>
  |  ;
<a href="<%= listBooksURL.toString() %>">Show All Books &raquo;</a>
```

Open the Portlet in the browser and you will see “[Show All Books](#) ». Click this link to display the list of books currently added to our library.

List of books in our Library

Book Title	Author	Date Added
Liferay in Action	Rich Sezov	Wed Jan 23 13:39:43 GMT 2013
Auto Primary Key	Hasan	Thu Jan 24 05:44:28 GMT 2013

[« Go Back](#)

You have successfully retrieved the records from the database table through the persistence API “`getLMSBooks(0, -1)`” and displayed them using a simple HTML table. Now find out all the methods that are available inside “`LMSBookLocalServiceUtil`” class. Keep this information ready as we are going to do some more interesting things. As a bonus you can add an entry for “`/html/library/list.jsp`” in “**LibraryConstants.java**”.

Code Changes @ <http://code.google.com/p/lr-book/source/detail?r=10>

Cleaning up some code: Before we move on to the next section, let’s remove the code in “`view.jsp`” we have written for explaining the concept of injecting 3rd party JavaScript libraries. This will not only visually clean the Portlet but also avoid the extra loading time. Don’t forget to even remove the entries we have made in “`liferay-portlet.xml`” for this purpose. If you want this code anytime in the future, you can always refer to the SVN repository. The changed two files are checked in.

Code Changes @ <http://code.google.com/p/lr-book/source/detail?r=11>

5.5 Avoiding Multiple Submits

Let’s do an important testing on the Portlet we have developed so far. If we don’t do this unit testing, our QA leads are going to catch one big bug. Go to Add book form and add a book. After the successful additions, press “F5” to refresh / reload the current page. Do this couple of times and then click “[Show All Books](#) ». In the list you will be surprised to see more than one occurrence of the same record that was lastly added. This is a classical problem in many web applications that needs attention even during the early stages of development. How to get rid of this problem? There are two ways to prevent it – declarative and programmatic. Let’s see both these options in the next two sub-sections. Please remember to implement them in your real applications / portlets.

5.5.1 Declarative way

Open “`liferay-portlet.xml`” and insert this directive immediately after “`<icon>`” tag.

```
<action-url-redirect>true</action-url-redirect>
```

The following is the explanation given as per this file’s DTD.

Set the action-url-redirect value to true if an action URL for this Portlet should cause an auto redirect. This helps prevent double submits. The default value is false.

Save the file, check the Portlet and verify that this new entry has addressed our original problem. The above approach seems to create some side effect when implementing some other Portlet feature. Hence, we may have to revert back this setting in “**liferay-portlet.xml**” file. The easiest is to just change the value to `false` for this directive, before we try our other alternative.

5.5.2 Programmatic way

This method involves two changes – one in JSP file and the other in the Portlet class.

Change 1: Create a default renderURL in “**update.jsp**” and set it as a hidden form element with name “`redirectURL`”.

```
<aui:input name="redirectURL" type="hidden"
           value="<%=> renderResponse.createRenderURL().toString() %>" />
```

Change 2: In “**LibraryPortlet.java**” insert these lines at the end of “`updateBook`” method for the proper redirection to happen after the action is performed.

```
// redirect after insert
String redirectURL =
    ParamUtil.getString(actionRequest, "redirectURL");
actionResponse.sendRedirect(redirectURL);
```

Check the Portlet and confirm even this approach has fixed our double submit issue.

5.5.3 Redirecting to any other page

Following the same approach, you can redirect to any other page of your choice after an action is executed. Let’s create “**success.jsp**” inside the same “**html/library**” folder and put this code there.

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

<h1>New book inserted !!</h1>

<br/><a href=">&laquo; Go Back</a>
```

Replace the two lines of redirection code in “**LibraryPortlet.java**” with the below code that creates a renderURL with the help of `PortletURLFactoryUtil` [<http://bit.ly/ZoCgVu>] programmatically.

```
ThemeDisplay themeDisplay =
    (ThemeDisplay) actionRequest.getAttribute(WebKeys.THEME_DISPLAY);

PortletConfig portletConfig =
    (PortletConfig) actionRequest.getAttribute("javax.portlet.config");

String portletName = portletConfig.getPortletName();

PortletURL successPageURL = PortletURLFactoryUtil.create(
    actionRequest,
    portletName + "_WAR_" + portletName + "portlet",
```

```

        themeDisplay.getPlid(),
        PortletRequest.RENDER_PHASE);

successPageURL.setParameter("jspPage",
        LibraryConstants.PAGE_SUCCESS);
actionResponse.sendRedirect(successPageURL.toString());

```

Import new classes and interfaces required by this class and also define a new constant, `LibraryConstants.PAGE_SUCCESS`. Save changes and check the Portlet to confirm that you're redirected to the success page that says, “**New book inserted!!**” after adding a book to our library. The third parameter to the “*create*” method of “`PortletURLFactoryUtil`” actually resolves to the Portlet ID during runtime. Its value in our case will be “**library_WAR_libraryportlet**”. `WebKeys` is an interface that contains the definitions for commonly used keys across the portal code base. The complete path is “`com.liferay.portal.kernel.util`”.

We can remove the hidden variable that you have defined earlier in “**update.jsp**” as it is no longer required. Five files are modified for this change and they are checked in to SVN with revision number “13”.

Code Changes @ <http://code.google.com/p/lr-book/source/detail?r=13>

Do it Yourself: Insert a new link “[Add Another Book »](#)” next to “[« Go Back](#)” in “**success.jsp**”. Use “`<aui:a href="">`” instead of a normal anchor tag to create the hyper link and find out the difference.

5.5.4 Attributes of `actionRequest`

It is important to know the various attributes of “`actionRequest`” object that is available in any action method of the Portlet class. We are already aware of the fact that `ActionRequest` is an interface that extends `javax.portlet.PortletRequest`. Each of these attributes provides lot of information about the Portlet itself and the context in which it is running, the Portlet Container. Apart from the parameters set by the HTML form, the following attributes and parameters can be retrieved from the “`actionRequest`” object.

Parameter/Attribute	Returns
<code>formDate (parameter)</code>	The current date stamp
<code>javax.portlet.action (parameter)</code>	Action method name
<code>THEME_DISPLAY</code>	ThemeDisplay object
<code>PORTLET_ID</code>	Current Portlet ID
<code>INVOKER_FILTER_URI</code>	URI of Invoker Filter
<code>WINDOW_STATE</code>	WindowState object
<code>javax.servlet.include.request_uri</code>	Request URI
<code>javax.servlet.include.context_path</code>	Context Path
<code>javax.servlet.include.servlet_path</code>	Servlet Path
<code>com.liferay.portal.kernel.servlet.PortletServletFilterChain</code>	PortletServletFilterChain
<code>com.liferay.portal.kernel.servlet.PortletServletResponse</code>	PortletServletResponse
<code>com.liferay.portal.kernel.servlet.PortletServletRequest</code>	PortletServletRequest
<code>javax.portlet.config</code>	PortletConfig object

javax.portlet.response	Portlet Response object
javax.portlet.portlet	Portlet object
com.liferay.portal.kernel.servlet.PortletServletConfig	PortletServletConfig
javax.portlet.lifecycle_phase	Portlet lifecycle
javax.portlet.request	Portlet Request object

Do it yourself: List down the methods of “themeDisplay” and “portletConfig” objects.

5.6 Separating the Business Logic

In this section we'll cover one of the most essential topics – separating our business logic as clearly isolated API's and writing them in a specific class, so that these API's are available for our Portlet class and JSP pages. The Class where we are going to write these API's is originally auto-generated by the service builder. The Class is “<Entity>LocalServiceImpl.java” where <Entity> is the name of the actual entity. In our case it is “LMSBook”. You will find this class inside the package “com.slayer.service.impl” of “src” folder. Open this class and have a look.

This class extends “LMSBookLocalServiceBaseImpl” and initially there are no methods (API's), but with one comment.

```
/*
 * NOTE FOR DEVELOPERS:
 *
 * Never reference this interface directly. Always use {@link
 * com.slayer.service.LMSBookLocalServiceUtil} to access the lms
 * book local service.
 */
```

Let's accomplish this change with three steps.

Step 1: Shift the “insertBook” method originally inside the Portlet Class “LibraryPortlet.java” into our DTO class.

```
public LMSBook insertBook(String bookTitle, String author) {
    // 1. Instantiate an empty object of type LMSBookImpl
    LMSBook lmsBook = new LMSBookImpl();

    // 2. Set the fields for this object
    lmsBook.setBookTitle(bookTitle);
    lmsBook.setAuthor(author);
    lmsBook.setCreateDate(new Date());

    // 3. Call the Service Layer API to persist the object
    try {
        lmsBook = addLMSBook(lmsBook);
    } catch (SystemException e) {
        e.printStackTrace();
    }

    return lmsBook;
}
```

In the process of moving this method to “**LMSBookLocalServiceImpl.java**” we have to make the following five changes.

1. Change the access modifier of the class from “**private**” to “**public**”, so that this method can be accessed from anywhere “locally” throughout the application
2. Make the return type of the method from “**void**” to “**LMSBook**”, so that wherever this method is invoked, it returns the fresh object back to the caller
3. Remove the code that sets the *bookId* which we commented out earlier after leveraging the auto-increment feature of the underlying database
4. “**LMSBookLocalServiceUtil.addLMSBook(lmsBook)**” has now become simply “**addLMSBook(lmsBook)**” as this method is now available within this Class by virtue of extending “**LMSBookLocalServiceBaseImpl**”.
5. Make all the necessary import statements that are required for this new method

Step 2: Save the file and now it is time to re-generate the service layer. Remember, we will always re-generate the service layer during the following three situations.

1. When any changes take place in “**service.xml**” which is the input for Service Builder tool.
2. When a new method (API) is added or an existing method deleted from either “**<Entity>LocalServiceImpl.java**” or “**<Entity>ServiceImpl.java**”. (*We'll see the second one in Chapter 7*)
3. When the signature of any of the methods (API's) change in the above two classes

This also implies that when there are actual code changes within any given method, there is no need to re-generate the service layer.

Once the service layer is re-generated, we move on to the next step.

Step 3: Go back to “**LibraryPortlet.java**” and you see the call to “**insertBook**” is complaining, as the method is no longer available in this class. Replace that line with,

```
LMSBookLocalServiceUtil.insertBook(bookTitle, author);
```

Remove the imports that are no longer required by pressing **Ctrl+Shift+O**. One thing you notice is we have used “**LMSBookLocalServiceUtil**” to invoke this method and not the original “**LocalServiceImpl**” class where we have written the method.

Save your changes and check the Portlet to check the “Add Book” functionality is working well as before and the new record is getting inserted into the database table through the new API we have written inside “**LMSBookLocalServiceImpl.java**”.

Code Changes @ <http://code.google.com/p/lr-book/source/detail?r=14>

5.6.1 Never write business logic in Portlet Class

One of the greatest things to know from this section is, never write your business logic code inside the Portlet Class. They are just controller classes that control the

flow of the Portlet functionality. Writing the business logic API's in the entity's “*ServiceImpl” classes has some big benefits. Some of them are listed here.

1. Complete separation of the business logic.
2. Re-usability of invocation from multiple places.
3. Auto-injection of dependency classes through Spring's dependency injection.
4. Automatic handling of transaction.
5. The API's can be wrapped by a remote service and exposed outside of the application.
6. Easy permission checking and security checks before performing an operation.

5.6.2 Objects injected by default

Inside any method of the “*ServiceImpl.java” Class, you will have the following objects injected by default by the Spring container.

counterLocalService	resourceService	userService
lmsBookLocalService	resourceLocalService	userLocalService
lmsBookPersistence	resourcePersistence	userPersistence

Additional objects can be injected with the help of “<reference>” tag in “service.xml”. We'll see a quick example of this now and it's usage later. e.g. you want “Image” service as part of your Portlet's service layer. All you have to do is to define this in your “service.xml” file. Insert this line at the end of “LMSBook” entity definition:

```
<reference package-path="com.liferay.portal" entity="Image" />
```

Re-generate the service layer and verify the availability of freshly injected objects inside “insertBook” method by pressing **Ctrl+Space**.

◇ imageLocalService : ImageLocalService – LM!
◇ imagePersistence : ImagePersistence – LMSB
◇ imageService : ImageService – LMSBookLoca

You can see the power of dependency injection with inclusion of just one tag .

5.7 Service Layer of Portal Source

Have you ever wondered where is the service layer and it's corresponding “service.xml” of Liferay's portal source code? To quench your thirst to know this information, quickly have a look into the “service.xml” under portal source location “/portal-impl/src/com/liferay/portal”. This is the biggest of all “service.xml” files in the entire portal source code with more than 2500 lines of code and definitions for almost seventy portal level entities.

Apart from this one the portal source has “service.xml” files for various other portlets that come pre-bundled with Liferay. There are around 23 of them as of Liferay version 6.1.1 GA2. You can press **Ctrl+Shift+R** to filter all “service.xml” files residing inside various folders. All the generated interfaces and utility classes for these service layers reside inside “portal-service” folder of the portal source. The entire contents of this folder are compiled and packaged as “portal-service.jar” file and put inside the global “lib/ext” folder of the tomcat installation or any other

web/application server, so that these services are available for all plugins (*portlets*) that will get deployed into the server with different contexts.

Portal Service Overview: Inside “**portal-service/src**” you will find an html file, “**portal-service-overview.html**”. Open this file in your browser and you will get some insight about the portal services that are available by default as part of the portal source code.

5.8 Sharing a custom service layer

While developing real time applications you may come across situations where the application will be developed and deployed as multiple plugins (WARs). Each plugin may have to access the same set of tables in the underlying database. It is not a good practice to create one service layer each for every plugin with definitions for the same set of tables replicated more than once. The service layer that is inside one plugin will not be accessible from the other plugin. Then what is the solution? There are two ways you can address this problem.

Solution 1: The service layer is originally part of plugin “A”. Copy the generated “**service.jar**” file from its “**WEB-INF/lib**” and paste into the equivalent “**lib**” folders of other plugins that want to use it.

Solution 2: Create a dummy Portlet and create the service layer inside it. Once the service layer is generated, copy the jar file and paste into the global “**lib/ext**” of the application server (*tomcat*). In this approach you should also make sure that there is no duplicate copy of this jar file still sitting inside “**WEB-INF/lib**” of any other plugin. If it is there, you will get strange exceptions like “`ClassCastException` Cannot cast MyObjectServiceUtil to MyObjectServiceUtil”.

Restart the server and now the API’s will be available for all the plugins that will get deployed into the server.

What a Service Layer API can return?

It is very important to note what object a service layer API can return. Any method that is defined in the DTO class “`<Entity>*ServiceImpl.java`” can ONLY return an object belonging to the following types:

- An instance of the same entity itself that the “`service.xml`” has defined.
- An list of entities of the above type
- A primitive value or it’s Class equivalent, eg. “`int`”, “`long`”, “`boolean`”, etc.

5.9 Caching to improve performance

I thought this is an apt juncture to touch on the caching mechanisms that Liferay adapts to improve the performance of the overall deployed portal. Liferay is known to provide very good performances and its caching system is a key contributor towards

achieving this great performance. You can refer to the Liferay Portal performance whitepaper from <http://bit.ly/12yecDi>.

The caching system of Liferay spans across all the three layers – front-end (*portlet*), middleware (*service layer*) and back-end (*persistence*). The following table gives the caching methods followed in all three layers.

Frontend	1. Page HTML Cache (CacheFilter) 2. Portlet Caching (UI and Resources) 3. CSS & JS Cache (Minified & Bundled)
Middleware	4. Multi VM Pool (Cluster aware) 5. Single VM Pool
Backend	6. ServiceBuilder Finder Cache & Entity Cache 7. Hibernate Caching (L1 & L2)

In this section, we'll just see the caching provided by the backend – persistence layer. At this layer, Liferay relies on Hibernate to do most of its database access. Hibernate has two cache layers called Level 1 (L1) and Level 2 (L2).

Level 1 – It is used to cache objects retrieved from the database within the current database session. In the case of Liferay, a session is tied to an invocation to a service layer. When the frontend layer (*or a web service*) invokes a service a database session is opened and re-used until the service method returns. All operations performed until that point would share the L1 cache so the same object won't be retrieved twice from the database.

Level 2 – It is able to span across database session and stores database objects (*Entity Cache*) and results of queries (*Query Cache*). For example, if any logic retrieves from the database all users that belong to a certain organization, the result will be stored in the cache. Note that what is really stored is the list of "references" to the actual objects which are stored in the Entity Cache. This is done to ensure that there aren't several copies of the same object in the cache.

Besides using Hibernate, Liferay's code also performs some complex database queries directly, reusing the database connection. For these queries Liferay has its own Entity and Query cache. One important aspect is that all of these caching use an underlying cache provider to manage the objects in memory. By default Liferay uses **ehcache** (<http://ehcache.org>) to do that, but it is also possible to change the caching provider through "**portal.properties**". The configuration is done within "**hibernate-clustered.xml**" that defines and configures several cache areas.

5.10 “service.xml” DTD Explained

We'll conclude this chapter with this very important section where we'll see the various directives that can be supplied to Service Builder tool through “**service.xml**” to generate a service layer. These directives / tags are defined / explained in the corresponding DTD file present inside the “**definitions**” folder of Liferay portal source. The file is “**liferay-service-builder_6_1_0.dtd**” as per the latest version of Liferay. Open this file and go through its contents. This will surely reinforce our

understanding on service layer we gained so far. We'll see how to use many of these directives in the subsequent chapters. After browsing through the contents answer the following ten questions.

- How many attributes are there for “entity” tag?
- How do you specify if the entity name is different from the table name in the underlying database?
- How do you disable caching for an entity? Is it enabled or disabled by default?
- What are the four implementations of “id-type” attribute?
- How to extend the default persistence class and specify a new class?
- How relationships between entities are expressed?
 - a. Give example of 1-1 relationship (*Shopping Portlet*)
 - b. Give example of 1-n relationship (*Shopping Portlet*)
 - c. Give example of n-n relationship (*Software Catalog Portlet*)
- How to specify if the entity column name is different than the actual table column name?
- What is the purpose of “data-source” attribute of an entity?
- What is the purpose of “finder” tag?
- How do you control the order of the entities while fetching from database?

Hint for question no. 6, you have to refer to the “**service.xml**” of the portlets mentioned in the sub list.

Congratulations, you have successfully completed one of the most important chapters of this book. I've seen many people who resort to pure JDBC or Hibernate just because they don't know about Liferay's Service Builder tool that does the entire job for us. As I mentioned earlier in this chapter, what we have seen so far is just the tip of the iceberg. We'll see more interesting things as we move along.

Summary

We started this chapter with a lucid explanation for Service Layer and the tool that generates this layer, Service Builder. This was a very high level overview, but soon we got into action and seen generating an actual service layer by writing our first “**service.xml**” file for the Library Portlet. We have analysed all the files that got generated by the Service Builder both inside the “**service**” folder and inside “**src**” folder. We have seen the interfaces, classes, abstract classes, XML files and SQL scripts that got generated by the Service Builder.

In the next section, we have seen how to write an API in the generated DTO class and re-generate the service layer by re-running the service builder. In the rest of the book, I will be using these two terms synonymously. Both are the same and you should not get confused. We have persisted the book into our Library by making use of the service layer API. In the following section, we have used another API to pull all the books from the Library and display them to the user. At the time of inserting the book, we have also seen how to make use of auto-increment feature of the underlying database. But you should not use this feature, if you’re making your Portlet very generic and planning to put it in the Liferay marketplace.

We moved on to see how to avoid multiple submits which is a classical problem in all web applications. We have seen how to deal with this problem both programmatically and declaratively by defining a tag in “**liferay-portlet.xml**” file. Followed by this, we have seen how to forward the control to a different page after an action is performed with the help of a redirectURL. In one of the upcoming chapters we are going to see another holistic approach for this re-direction. At the end of the same section, we have seen all the attributes and parameters of the actionRequest object.

In the next section 5.6, we have seen the principles of writing the business logic inside the DTO class and never writing them either in the Portlet class or inside the JSP files. We have seen the list of objects that are injected by default. In the subsequent three sections we have seen some details about the service layer of portal source, sharing a custom service layer between more than two plugins and the caching that is happening at the service layer level to improve performance of the data retrieval.

We concluded this chapter with some more explanation on the DTD file that provides the syntax of a “**service.xml**” file. I left this section with almost ten questions for you. I am sure you have found answers for all of them. Chapters 7 and 8 are going to take us deep inside the Service Layer forest where we are going to find so many other valuable treasures. It is just a matter of time; we have to do little more improvisation to our list of books in the next chapter.

6. Improving the Book List

This chapter covers

- HTML table to Search Container
 - Referring to Portal’s TLDs and JARs
 - Adding link to delete book
 - Adding “Actions” on a book
 - Editing a book
 - Viewing book details
 - Showing details as a popup
 - Sortable Columns
 - Performing an action on a set of items
-

While chapter 5 covered the “Service Layer” and “Service Builder” in great depth, we’ll take a little detour in this chapter and see how to improve our list of books in the library Portlet with various tags that Liferay provides by default and decorate the list in accordance with Liferay standards. In the process we are going to learn some very important aspects of building the core part of any Portlet functionality. We’ll begin this chapter by replacing our conventional HTML list with a powerful tag that Liferay provides through its `<liferay-ui>` taglib.

6.1 HTML table to Search Container

Earlier in section [5.4 Pulling Data – Use another API](#), we have created a simple list that displays all the books in our library. In the corresponding JSP file, “**list.jsp**”, we have used simple HTML table to render the list of books. Using `<table>` tag is no longer encouraged as it creates problems while rendering the Portlet in any other device apart from the web browser, especially in hand-held devices and smart phones. Liferay provides a very good UI alternative for displaying a grid in our Portlet. Though you can use any other JavaScript based grid component like Flexigrid (<http://flexigrid.info>) or jqGrid (<http://jqgrid.com>), learning this one has many advantages as we are going to see in the rest of this chapter. Liferay has used this component in almost all the places where there is a grid.

Without wasting much of our time, let’s quickly convert our HTML table to a proper grid component based on Liferay’s taglib called as “Search Container”. We’ll achieve this in just two steps.

Step 1: As we are going to make use of Liferay’s UI taglib, give the reference to its corresponding taglib URI in “**init.jsp**”.

```
<%@taglib uri="http://liferay.com/tld/ui" prefix="liferay-ui"%>
```

Step 2: Open “**list.jsp**” and replace the code for displaying the list with this one.

```
<liferay-ui:search-container delta="4"
    emptyResultsMessage="Sorry. There are no items to display.">
    <liferay-ui:search-container-results
        total="<% = books.size() %>"
        results="<% = ListUtil.subList(books,
            searchContainer.getStart(),
            searchContainer.getEnd()) %>" />

    <liferay-ui:search-container-row modelVar="book"
        className="LMSBook">
        <liferay-ui:search-container-column-text
            name="Book Title" property="bookTitle" />
        <liferay-ui:search-container-column-text name="Author"
            property="author" />
        <liferay-ui:search-container-column-text
            name="Date Added" property="createDate" />
    </liferay-ui:search-container-row>
    <liferay-ui:search-iterator
        searchContainer="<% = searchContainer %>" />
</liferay-ui:search-container>
```

Make the import for “**ListUtil**” within “**list.jsp**” itself. Later on, you should find out all the methods of this utility class.

```
<%@page import="com.liferay.portal.kernel.util.ListUtil"%>
```

Save both files “**init.jsp**” and “**list.jsp**”, go to browser and click “[Show All Books »](#)”. Now you see the old normal looking list got replaced with a beautiful grid with proper title for every column and a nice pagination control to scroll through the list. There is

also a hover effect when you mouse over on a particular row of the grid. If you're not seeing the pagination control that means you have lesser records than the "delta" value that we have set. Add more books to our library and you should see it.

Book Title	Author	Date Added
Liferay in Action	Rich Sezov	2013-01-23 13:39:43.0
Auto Primary Key	Hasan	2013-01-24 05:44:28.0
ttt	ttt	2013-01-24 10:13:38.0
ttt	ttt	2013-01-24 10:13:46.0

Showing 1 - 4 of 25 results. Items per Page Page of 7 [First](#) | [Previous](#) | [Next](#) | [Last](#)

6.1.1 The “search-container” tags

In this exercise, we got introduced to the following five “liferay-ui” tags.

1. `search-container` – This is the main container. It performs a lot of set up work behind the scenes like instantiating the “`searchContainer`” object.
2. `search-container-results` – This has two main attributes.
 - a. `total` – This is where you input the total number of items in your list.
 - b. `results` – This is where you input the results. “`results`” should be of type `List`. The important part is to make sure that your method supports some way to search from a beginning index to an end index in order to provide good performance pagination. As mentioned above, the “`searchContainer`” object is available because it has been instantiated already. Methods of this object you can use are:
 - i. `searchContainer.getStart()` – gets starting index of current results page.
 - ii. `searchContainer.getResultsEnd()` – gets ending index of current results page or index of last result (i.e. will return 3 if delta is 5 but there are only 3 results).
 - iii. `searchContainer.getEnd()` – gets last index of current results page regardless of size of actually results (i.e. will return 5 if delta is 5 even if there is only 3 results. Will throw out of bounds errors).
 - iv. `searchContainer.getCur()` – gets number of current results page.
 - v. `searchContainer.setTotal()` – must be set for this tag so that `getResultsEnd()` knows when to stop.
3. `search-container-row` – Represents one row of the list. It has three main attributes. This is equivalent to HTML “`<tr>`” tag of “`<table>`”.
 - a. `className` - The type of Object in your List. In this case, we have a List of objects of type “`LMSBook`”.
 - b. `keyProperty` - Primary Key of the record. This is optional.

-
- c. `modelVar` - The name of the variable to represent your model. In our case the model is `"book"` object.
4. `search-container-column-text` – The item that represents a column in the row. This is equivalent to HTML “`<td>`” tag of “`<tr>`”. This tag has the following main attributes.
- a. `name` - Name of the column
 - b. `value` - Value of the column
 - c. `href` - the text in this coulmn will be a link the this URL
 - d. `orderable` - allows the user to order the list of items by this column
 - e. `title` – title of the column
 - f. `property` – the attribute of the `modelVar` this column should render. This will automatically look in the `book` object for the `"bookTitle"` property. It's basically the same as calling `book.getBookTitle()`.
5. `search-iterator` – This is what actually iterates through and displays the List.

Do it yourself: Here are few tasks to familiarize you with the search container taglib.

- List down all attributes of “`<liferay-ui:search-container`” tag.
- List down all the methods of “`ListUtil`” utility class.

6.1.2 Fixing the Pagination issue

Click on Next link in the pagination control. Unfortunately, instead of taking you to the next set of books in the list, it takes you to the default view of the Portlet. Do you think this is an abnormal behavior? In this sub-section we'll see the fix for this issue by adding one more attribute to the “`search-container`” tag.

Insert attribute for “`search-container`”, `iteratorURL="<%= iteratorURL %>"`. Declare a new variable inside JSP scriptlet after “`List<LMSBook> books`” line.

```
PortletURL iteratorURL = renderResponse.createRenderURL();
iteratorURL.setParameter("jspPage", LibraryConstants.PAGE_LIST);
```

This will require you to declare a new constant in “`LibraryConstants.java`”.

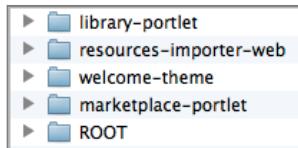
```
static final String PAGE_LIST = "/html/library/list.jsp";
```

Save changes and check the list now and ensure the pagination is working fine.

Code Changes @ <http://code.google.com/p/lr-book/source/detail?r=15>

6.2 Referring to Portal's TLDs and JARs

Let's have a quick look into the “**webapps**” folder of our tomcat installation. Some of the apps that appear here may not be available if you use a different version of Liferay. Anyways, let's come to the main discussion.



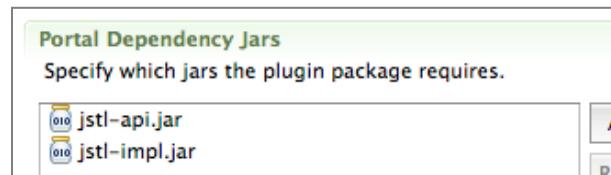
The main context, “**ROOT**” where Liferay Portal is running has got so many third party jars and taglib definitions (**tld**) available inside it. Say, if you want to inject some of them into our Portlet that is running on a different context, how do you achieve it? Here comes “**liferay-plugin-package.properties**” very handy to do this kind of injection during the time of packaging the Portlet plugin in the form of a hot deployable WAR file. You will find this file inside the “**WEB-INF**” folder of the Portlet’s “**docroot**”. Let's demonstrate this feature with a quick example before we move on to discuss the various other settings of this properties file.

Navigate to this file and double click on it. The file will open in its “Properties” mode. Click on the “Source” mode to see its default contents looking something like this. At the later part of this section I will give a link where you can see the explanation for each of these entries in detail. Click again on the “Properties” mode to start adding the required JAR files and TLD files.

```
name=Library  
module-group-id=liferay  
module-incremental-version=1  
tags=  
short-description=  
change-log=  
page-url=http://www.liferay.com  
author=Liferay, Inc.  
licenses=LGPL  
liferay-versions=6.1.1
```

Before doing this, let's very clearly state what we are going to achieve out of this exercise. We are going to inject some default JSTL taglib that will help us to format the “**createDate**” column for the list. Currently it is in the raw format. We'll format using the “**fmt**” tag of JSTL taglib and make it much more visually appealing. Let's accomplish this in the following four steps.

Step 1: When you open “**liferay-plugin-package.properties**” file in its “Properties” mode, click “Add...” under Portal Dependency Jars and add the two dependency jar files as shown.



Step 2: In the similar fashion, click “Add...” under Portal Dependency Tlds and add “**fmt.tld**” through the dialog that opens. Save changes and switch to “Source” to confirm the new entries have got added.

```
portal-dependency-jars=\  
jstl-api.jar,\  
jstl-impl.jar  
portal-dependency-tlds=fmt.tld
```

Let's also confirm that the plugin has got packaged with these dependencies at the time of deployment. Expand “**library-portlet**” webapp under tomcat's “**webapps**” folder. Inside “**WEB-INF/tld**”, you will see the newly injected “**fmt.tld**” file and inside “**WEB-INF/lib**” you will see the two newly injected JAR files.

Step 3: It is time to make changes to the Portlet code to make use of these dependencies. Open “**init.jsp**” and insert the taglib URI definition for JSTL taglib.

```
<%@taglib uri="http://java.sun.com/jsp/jstl/fmt" prefix="fmt"%>
```

Step 4: Open “**list.jsp**” and replace the line for “createDate” with the one below.

```
<liferay-ui:search-container-column-text name="Date Added">
    <fmt:formatDate value="<% book.getCreateDate() %>" 
        pattern="dd/MMM/yyyy" />
</liferay-ui:search-container-column-text>
```

Save your changes, open the Portlet in browser and click “[Show All Books](#)” to see the new date format.

6.2.1 Other benefits of “liferay-plugin-package.properties”

In this section so far, we have just seen one usage of “**liferay-plugin-package.properties**” which contains many packaging information about the plugin itself. You can refer to this link <http://bit.ly/UWjlmG> to know all the possible entries of this file. We may see few of them in our later chapters. The main sections of this file are given below. They are mainly used to make a plugin qualified for listing in Liferay’s marketplace.

General	Security Manager	Class Loader Security
Expando Bridge Security	File Security	Bean Security
JNDI Security	Message Bus Security	Search Engine Security
Portal Service Security	Portlet Service Security	Socket Security
SQL Security	Thread Security	

6.3 Adding link to delete book

In this section, let’s add one more link in our list to delete a book from the library. This exercise will help us to learn how to add a new Portlet functionality by introducing a new action class and also help us to know how to add a new link in the search container grid component. This exercise has three steps.

Step 1: Introduce “**deleteBookURL**” immediately after “**iteratorURL**” in “**list.jsp**”.

```
PortletURL deleteBookURL = renderResponse.createActionURL();
deleteBookURL.setParameter(ActionRequest.ACTION_NAME,
    LibraryConstants.ACTION_DELETE_BOOK);
deleteBookURL.setParameter("redirectURL", iteratorURL.toString());
```

In the process, declare a new constant in “**LibraryConstants.java**”.

```
static final String ACTION_DELETE_BOOK = "deleteBook";
```

In the third line of the above code, we have set a new parameter “`redirectURL`” to redirect the control back to the list page after performing delete action.

Step 2: The next change is again in the same “`list.jsp`”. Insert a new column that will have a link to delete a book represented as a row in the grid.

```
<% deleteBookURL.setParameter(  
    "bookId", Long.toString(book.getBookId())); %>  
<liferay-ui:search-container-column-text name="Delete"  
    href="<% deleteBookURL.toString() %>" value="delete"/>
```

In line one; we are setting an additional parameter “`bookId`” to “`deleteBookURL`”. Save the changes and try clicking on “delete »” link. The Portlet throws an error and when you check the server console, you will soon realize what is the root cause of the problem. In the next step we’ll add a new action method in our Portlet class to actually delete a book when the link is clicked and get rid of this error.

Step 3: Introduce this new method to our Portlet class “`LibraryPortlet.java`”.

```
public void deleteBook(ActionRequest actionRequest,  
    ActionResponse actionResponse)  
    throws IOException, PortletException {  
  
    long bookId = ParamUtil.getLong(actionRequest, "bookId");  
    if (bookId > 0) { // valid bookId  
        try {  
            LMSBookLocalServiceUtil.deleteLMSBook(bookId);  
        } catch (PortalException | SystemException e) {  
            e.printStackTrace();  
        }  
    }  
    // gracefully redirecting to the list view  
    String redirectURL =  
        ParamUtil.getString(actionRequest, "redirectURL");  
    actionResponse.sendRedirect(redirectURL);  
}
```

Note: The multiple catch syntax “`(PortalException | SystemException e)`” used above may not work in some earlier versions of java. In such cases, use individual catch for every of exception that need to be caught.

Save changes and confirm the delete functionality is working fine now. Congratulations! But there is still one small problem. If you delete an item from page 2, the Portlet should come back to the same page. But unfortunately it goes back to the first page of the list. Any idea how to fix this problem? Another test case is, if a list has three pages and only one item in the last page. If you delete that item, the Portlet should come back to the previous page. I am sure with some little analysis you should be able to implement these fixes.

Code Changes @ <http://code.google.com/p/lr-book/source/detail?r=16>

6.4 Adding “Actions” on a Book

In the previous section we have seen how to add just ONE action to every row in the list – “delete”. In this section we’ll see how to add a set of “actions” embedded within another JSP page exclusively made for this purpose. In the process we’ll also see some more “`<liferay-ui>`” tag, “`<liferay-ui:icon-menu>`”. This set of actions will form the basis for our learning in later chapter that talks about “Security and Permissions”. There are two steps to introduce an “Actions” button to every row.

Step 1: Create a new file “`actions.jsp`” and put the below contents.

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

<%@page import="com.liferay.portal.kernel.util.WebKeys"%>
<%@page import="com.liferay.portal.kernel.dao.search.ResultRow"%>

<%
    ResultRow row = (ResultRow)
        request.getAttribute(WebKeys.SEARCH_CONTAINER_RESULT_ROW);

    LMSBook book = (LMSBook) row.getObject();
    PortletURL editBookURL = renderResponse.createRenderURL();
    PortletURL viewBookURL = renderResponse.createRenderURL();
%>

<liferay-ui:icon-menu>
    <liferay-ui:icon image="edit" message="Edit Book"
        url="<%=\editBookURL.toString()%>" />
    <liferay-ui:icon image="view" message="View Details"
        url="<%=\viewBookURL.toString()%>" />
</liferay-ui:icon-menu>
```

Step 2: Open “`list.jsp`” and append a new column inside the row.

```
<liferay-ui:search-container-column-jsp name="Actions"
    path="<%=\LibraryConstants.PAGE_ACTIONS%>" />
```

The JSP will complain till you declare a new constant in “**LibraryConstants.java**”.

```
static final String PAGE_ACTIONS = "/html/library/actions.jsp";
```

Save the files and check the book list how it appears now. When you click on “Actions” all the items “`<liferay-ui:icon>`” that we have inside the menu appears nicely. Try clicking these options. It doesn’t take you anywhere except taking you to the default Portlet view. In the next sections we’ll perform some serious functionality to be performed when these options are clicked. You should also note that using “`search-container-column-jsp`” we could embed the contents of a whole JSP page within a given column of the search container grid.



Do it yourself: Here is one quick exercise for you before we move on with the next section. Add the “Delete” action that we wrote earlier also as part of this list of actions. When you click on the “Delete” option it should actually delete the particular item and take you back to the same list.

6.5 Editing a book

We’ll implement the “Edit Book” feature for our library Portlet in this section. In the process, we’ll re-use the same “**update.jsp**” for the edit as well. We’ll quickly create the edit book UI by making some minor updates to both “**actions.jsp**” and “**update.jsp**”.

Change 1 (actions.jsp): Add additional parameters to “editBookURL”.

```
editBookURL.setParameter("bookId", Long.toString(book.getBookId()));  
editBookURL.setParameter("jspPage", LibraryConstants.PAGE_UPDATE);
```

Change 2 (update.jsp):

- a) Add the following code to the existing scriptlet block.

```
LMSBook lmsBook = new LMSBookImpl();  
long bookId = ParamUtil.getLong(request, "bookId");  
  
if (bookId > 0) {  
    lmsBook = LMSBookLocalServiceUtil.fetchLMSBook(bookId);  
}
```

- b) Make the necessary imports either within the same JSP or “**init.jsp**”.

```
<%@page import="com.liferay.portal.kernel.util.ParamUtil"%>  
<%@page import="com.slayer.model.impl.LMSBookImpl"%>
```

- c) Add “**value**” attribute for “**bookTitle**” and “**author**” input fields as below,

1. `<%= lmsBook.getBookTitle() %>`
2. `<%= lmsBook.getAuthor() %>`

Save changes and check the book list. When you click on “Edit Book” for a book row, it should take you to the same form but with the “**Book Title**” and “**Author**” fields pre-populated with proper values as shown here. Click on “Save” button and you will notice that a new record has got inserted into the table. This is not the desired behavior that we want. We want the same record to get updated. But this is not going to happen unless and until we program for it. In the next sub-section we are going to write the update functionality that will take care of this form update.



The screenshot shows a simple form with two input fields. The first field is labeled "Book Title" and contains the value "Liferay in Action". The second field is labeled "Author" and contains the value "Rich Sezov". Both fields have a small blue circular icon with a question mark next to them, likely indicating they are required fields.

Note: On a side note, you should also check the “Add Book” functionality and confirm that it is working fine as before and we have not broken the existing functionality.

6.5.1 Modifying the book information

We'll fix the problem we found earlier of new record insertion when the edit form is saved. Let's address this by following these two steps.

Step 1: Introduce a new hidden variable as part of the “`<aui:form>`” in “**update.jsp**”.

```
<aui:input name="bookId" type="hidden"
           value="<% lmsBook.getBookId() %>" />
```

Step 2: Write some additional logic in “`updateBook`” method of “**LibraryPortlet.java**” based on the presence of this parameter, “`bookId`”.

Replace first line below with the subsequent new block of code.

```
LMSBookLocalServiceUtil.insertBook(bookTitle, author);
```

```
long bookId = ParamUtil.getLong(actionRequest, "bookId");
if (bookId > 0) {
    modifyBook(bookId, bookTitle, author);
} else {
    LMSBookLocalServiceUtil.insertBook(bookTitle, author);
}
```

The new “`modifyBook`” method is still not created. Click on the  icon that appears on that line and click “Create method”. This will create an empty body for a new private method, “`modifyBook`”. Scroll to that method and inside its body; write this code to update the book.

```
private void modifyBook(
    long bookId, String bookTitle, String author) {

    LMSBook lmsBook = null;
    try {
        lmsBook = LMSBookLocalServiceUtil.fetchLMSBook(bookId);
    } catch (SystemException e) {
        e.printStackTrace();
    }

    if (Validator.isNotNull(lmsBook)) {
        lmsBook.setBookTitle(bookTitle);
        lmsBook.setAuthor(author);
        try {
            LMSBookLocalServiceUtil.updateLMSBook(lmsBook);
        } catch (SystemException e) {
            e.printStackTrace();
        }
    }
}
```

Make necessary imports and save changes. Check the “Edit Book” behavior now. This time, the book information should properly get updated in the table. Also re-confirm that Add Book functionality is not broken with this change.

Code Changes @ <http://code.google.com/p/lr-book/source/detail?r=18>

The upcoming three sub sections are something you're going to do by yourself. I will give you the basic guidance. Doing these exercises will not only add more functionality to our Portlet but will also reinforce your understanding so far. We'll begin with fixing the page flow problem after a book has been updated.

6.5.2 Redirect to another page after update

Perform updating a book from the list. After the update action ideally it should have taken you to the list view again instead of the success page that is the behavior when a new book is added to the library. Now the task for you is to redirect to “**list.jsp**” after updating a book.

Solution Hint: Replace line in “**LibraryPortlet.java**”,

```
successPageURL.setParameter("jspPage", LibraryConstants.PAGE_SUCCESS);
```

With below line and save the file

```
successPageURL.setParameter("jspPage", (bookId > 0) ?  
    LibraryConstants.PAGE_LIST : LibraryConstants.PAGE_SUCCESS);
```

6.5.3 Moving “modifyBook” to LocalServiceImpl Class

We have seen in the last chapter that it is always advisable to write all our business logic API's inside “*ServiceImpl.java” Classes. In the same lines shift the method “**modifyBook**” from the Portlet Class to “**LMSBookLocalServiceImpl.java**”. Immediately after you do this change you have re generate the service layer.

Solution Hint: The new API will look like,

```
public LMSBook modifyBook(  
    long bookId, String bookTitle, String author) {  
  
    LMSBook lmsBook = null;  
    try {  
        lmsBook = fetchLMSBook(bookId);  
    } catch (SystemException e) {  
        e.printStackTrace();  
    }  
  
    if (Validator.isNotNull(lmsBook)) {  
        lmsBook.setBookTitle(bookTitle);  
        lmsBook.setAuthor(author);  
  
        try {  
            lmsBook = updateLMSBook(lmsBook);  
        } catch (SystemException e) {  
            e.printStackTrace();  
        }  
    }  
    return lmsBook;
```

```
}
```

Also make the respective change in “**LibraryPortlet.java**” to invoke the new API.

6.5.4 Introducing a new audit field – “modifiedDate”

Add a new audit field to LMSBook entity – “`modifiedDate`” of type `java.util.Date`. Set value for this field when a book is updated. This task will show you, with the help of Service Builder, how easy it is to add and remove columns to our underlying table and making changes to the application code. There are only minimal changes to be made.

Solution Hints:

- a) Update “**service.xml**” LMSBook entity with the new field and run Service Builder.

```
<column name="modifiedDate" type="Date" />
```

- b) [Before running this script, confirm that the new column is not automatically added to the table. If it is there, then you can skip this step.] Alter table to append this new column. Run this script against “**Iportal**” database.

```
ALTER TABLE lms_lmsbook ADD COLUMN modifiedDate DATETIME;
```

- c) Update “`modifyBook`” method of “**LMSBookLocalServiceImpl.java**” to set this field whenever a book gets modified, with the statement:

```
lmsBook.setModifiedDate(new Date());
```

Code Changes @ <http://code.google.com/p/lr-book/source/detail?r=19>

6.6 Viewing book details

From the list of books, why not give a link for every book to view it's details in a separate page? Let's do this exercise in this section. The exercise involves three steps.

Step 1: Create new page “**detail.jsp**” in “**/html/library**” with following contents.

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

<h1>Book Details</h1>

<%
    LMSBook lmsBook = null;
    long bookId = ParamUtil.getLong(request, "bookId");
    if (bookId > 0L) {
        lmsBook = LMSBookLocalServiceUtil.fetchLMSBook(bookId);
    }
%>

<c:if test="<% Validator.isNotNull(lmsBook) %>">
    <table border="1">
        <tr>
            <td>Book Title</td>
            <td><%= lmsBook.getBookTitle() %></td>
        </tr>
        <tr>
            <td>Author</td>
            <td><%= lmsBook.getAuthor() %></td>
        </tr>
        <tr>
            <td>Date Added</td>
            <td><%= lmsBook.getCreateDate() %></td>
        </tr>
        <tr>
            <td>Last Modified</td>
            <td><%= lmsBook.getModifiedDate() %></td>
        </tr>
    </table>
</c:if>
```

Step 2: Immediately after step 1, make the necessary imports in “**init.jsp**”.

```
<%@page import="com.liferay.portal.kernel.util.ParamUtil"%>
<%@page import="com.liferay.portal.kernel.util.Validator"%>
```

Add “**c.tld**” to “Portal Dependency Tlds” in “**liferay-plugin-package.properties**” and include its URI in “**init.jsp**”.

```
<%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
```

Step 3: Open “**list.jsp**” and you have to do four changes in order to give a link to the new details page.

a) Define a new constant inside “**LibraryConstants.java**” that we are going to refer.

```
static final String PAGE_DETAILS = "/html/library/detail.jsp";
```

b) Define a new Portlet URL “bookDetailsURL” inside the scriptlet block.

```
PortletURL bookDetailsURL = renderResponse.createRenderURL();
bookDetailsURL.setParameter("jspPage",
    LibraryConstants.PAGE_DETAILS);
```

c) Modify “`bookTitle`” column of search container to include “`href`” attribute. The value of this attribute will be, “`<%= bookDetailsURL.toString() %>`”.

d) Just before “`search-container-column-text`” tag, set “`bookId`” parameter for the new Portlet URL by inserting this scriptlet.

```
<% bookDetailsURL.setParameter(
    "bookId", Long.toString(book.getBookId())); %>
```

Save all your changes and check the Portlet in browser. Every book title in the list should have a hyperlink. Clicking this link should take you to the book’s details page.

6.6.1 Back URL to list view

Being in the details page of a book, how nice will it be to have a link to come back to the list of books, so that the user can view the details of the next book in the list? Liferay provides another convenient mechanism to have this feature. Let’s see what it is. This exercise involves following three steps.

Step 1: In “`detail.jsp`” remove existing header line “`<h1>Book Details</h1>`” and append this line to the existing scriptlet:

```
String backURL = ParamUtil.getString(request, "backURL");
```

Insert this code immediately after the scriptlet and before “`<c:if>`” tag.

```
<liferay-ui:header
    backLabel="“Back to List”
    title="Book Details" backURL="<%=" backURL %>" />
```

Step 2: Open “`list.jsp`” and set an additional parameter for “`bookDetailsURL`” within the existing scriptlet itself.

```
bookDetailsURL.setParameter("backURL", themeDisplay.getURLCurrent());
```

Step 3: The implicit object “`themeDisplay`” is not available and it is showing an error. Let’s inject this into our JSP. Open “`init.jsp`” and insert these two lines in their appropriate places.

```
<%@taglib uri="http://liferay.com/tld/theme" prefix="liferay-theme" %>
<liferay-theme:defineObjects/>
```

Save all the above changes and check the Portlet in the browser. It should have a proper link in the details page to get back to the list page. You will also notice that this link will take you back to the right page from where you originated.

Do it yourself:

- List down all the tags of “`liferay-theme`” taglib?
- List down the methods of “`themeDisplay`” that return a proper URL, e.g. the ones that start with “`getURL`”. Just think of how you can use them inside your JSP pages.

Code Changes @ <http://code.google.com/p/lr-book/source/detail?r=20>

6.6.2 Objects injected by “`liferay-theme`” tag

There are some very interesting and useful objects that get injected by making use of the tag, “`<liferay-theme:defineObjects/>`”. All those objects are listed out here for your reference. You may use them very frequently in your real time applications. I’ve gone one step further to list down all the objects that are default available in any JSP and the ones that are injected by “`<portlet:defineObjects/>`”.

Defaultly Available	Injected by <code><portlet:defineObjects/></code>	Injected by <code><liferay-theme:defineObjects/></code>
application	actionRequest	account
config	actionResponse	colorScheme
out	eventRequest	company
page	eventResponse	contact
pageContext	liferayPortletRequest	layout
request	liferayPortletResponse	layouts
response	portletConfig	layoutTypePortlet
session	portletName	locale
	portletPreferences	permissionChecker
	portletPreferencesValues	plid
	portletSession	portletDisplay
	portletSessionScope	portletGroupId
	renderRequest	realUser
	renderResponse	scopeGroupId
	resourceRequest	theme
	resourceResponse	themeDisplay
		timeZone
		user

There are a total of **eight** objects available by default, **sixteen** that get injected by “`<portlet:defineObjects/>`” and **eighteen** objects made available through “`<liferay-theme:defineObjects/>`”. I would recommend you to familiarize yourself with these objects and the methods (*API’s*) they provide.

6.7 Showing details as a popup

In modern web applications, the usage of popup windows is an attractive way of showing some information when the user clicks on a particular link or a button. Popups will prevent the user from navigating out of the current page/frame and going

to a new page. They help the user remain focused in the same page and when the details are viewed he/she can just click the “Close” icon in the popup window to get back to the parent window that opened the popup. Popups also play a very important role in building Rich Internet Applications (RIA) based on the concepts of Web 3.0 and SPA (Single Page Application). See some of the links below to know more about these very important concepts. They are taken from Wikipedia.

RIA	http://en.wikipedia.org/wiki/Rich_Internet_application
Web 3.0	http://en.wikipedia.org/wiki/Web_2.0#Web_3.0
SPA	http://en.wikipedia.org/wiki/Single-page_application

6.7.1 Implementing the popup

Liferay has two types of popups – Window popup and Floating Div popup. Before going ahead and discussing the details about these two types, let's quickly implement a window popup to show the book details when action tab is  clicked for any book in the list. This will be done in three steps. All these changes are going to be in “**actions.jsp**” file only.

Step 1: Set additional attributes / parameters for “viewBookURL” object.

```
viewBookURL.setWindowState(LiferayWindowState.POP_UP);
viewBookURL.setParameter("jspPage", LibraryConstants.PAGE_DETAILS);
viewBookURL.setParameter("bookId", Long.toString(book.getBookId()));
```

Note, we have set theWindowState as “LiferayWindowState.POP_UP”. Make the necessary import in the same JSP.

```
<%@page
import="com.liferay.portal.kernel.portlet.LiferayWindowState"%>
```

Step 2: Define a new method inside “`<aui:script>`” for popup window.

```
<aui:script>
    function popup(url){
        AUI().use('aui-dialog', function(A) {
            var dialog = new A.Dialog({
                title: 'Book Details',
                centered: true,
                modal: true,
                width: 500,
                height: 400,
            }).plug(A.Plugin.IO, {uri: url}).render();
        });
    }
</aui:script>
```

Step 3: Write the invocation for this method. Change value for “`url`” attribute of “`<liferay-ui:icon>`” that shows “View Details” link to “`<%= popup %>`”. Declare a String variable “`popup`” within the JSP scriptlet.

```
String popup = "javascript:popup('" + viewBookURL.toString() + "')";
```

You can check the details of two types of Liferay popups from <http://bit.ly/WT4Jm7>.

6.7.2 Removing header from popup

There is one minor problem in our popup. If the details page is opened as a popup, then we don't require an explicit header as the popup window has a title already. How to remove the header only when it is opened as popup? In normal mode the header should appear. Just follow these two steps.

Step 1: Set one more parameter for “viewBookURL” in “**actions.jsp**”.

```
viewBookURL.setParameter("showHeader", Boolean.toString(false));
```

Step 2: In “**detail.jsp**”, surround the tag “`<liferay-ui:header>`” with the below “`<c:if>`” condition ended with corresponding “`</c:if>`”.

```
<c:if test="<=% showHeader %>"> Inside scriptlet declare a boolean,
```

```
boolean showHeader = ParamUtil.getBoolean(request, "showHeader", true);
```

Save changes and check the popup now. The header is not showing up anymore.

6.8 Sortable Columns

In any grid, usually the columns are sortable. In our book list we are not seeing this feature. In this section we'll actually make our list sortable by two columns – `bookTitle` and `author`. First and foremost, you have to add these attributes for the fields that you want to make sortable in “**list.jsp**”. This is how it is done for the column “`bookTitle`”. Do the same thing for “`author`” as well.

```
orderable="true" orderableProperty="bookTitle"
```

Save this change and check the list. You should see both “Book Title” and “Author” column titles have become links. Click on them and observe the list behavior. No actual sorting is happening. For this to happen we have to do some little coding. Let's quickly copy+paste the URL that appears in the browser's location bar when we click one of these column titles.

```
http://localhost:8080/web/guest/my-library?p_p_id=library_WAR_libraryportlet&p_p_lifecycle=0&p_p_state=normal&p_p_mode=view&p_p_col_id=column-2&p_p_col_count=1&_library_WAR_libraryportlet_jspPage=%2Fhtml%2Flibrary%2Flist.jsp&_library_WAR_libraryportlet_cur=1&_library_WAR_libraryportlet_delta=4&_library_WAR_libraryportlet_keywords=&_library_WAR_libraryportlet_advancedSearch=false&_library_WAR_libraryportlet_andOperator=true&_library_WAR_libraryportlet_orderByCol=bookTitle&_library_WAR_libraryportlet_orderByType=null
```

At the end of this URL, there are two parameters “`orderByCol`” and “`orderByType`” which play an important role in sorting the list. We are going to achieve sorting by

manipulating the values of these parameters. Let's start with overriding "render" method in our Portlet Class. Inside the over-ridden method we are setting the parameter that is used to sort the list.

```
public void render(RenderRequest request, RenderResponse response)
    throws PortletException, IOException {
    setSortParams(request);
    super.render(request, response);
}
```

The "setSortParams" method that we have introduced newly is still not defined. Let's write this method in the same class. Every Portlet finally goes through the render phase and this method will get executed based on "jspPage" attribute.

```
private void setSortParams(RenderRequest request) {
    String jspPage = ParamUtil.getString(request, "jspPage");

    if (jspPage.equalsIgnoreCase(LibraryConstants.PAGE_LIST)) {
        String orderByCol = ParamUtil.getString(
            request, "orderByCol", "bookTitle");
        request.setAttribute("orderByCol", orderByCol);

        String orderByType = ParamUtil.getString(
            request, "orderByType", "asc");
        request.setAttribute("orderByType", orderByType);
    }
}
```

Having done the changes in the Portlet Class, let's come back to "list.jsp" to do three more changes.

Change 1: Open "liferay-plugin-package.properties" to include two new files under Portal Dependency Jars – **commons-beanutils.jar** and **commons-collections.jar**. These JARs contain the API's that does the actual sorting.

Change 2: Open "list.jsp" and insert the following block of code inside the scriptlet immediately after the books object is declared (*line 1 of scriptlet*).

```
// additional code for sorting the list
String orderByCol = (String) request.getAttribute("orderByCol");
String orderByType = (String) request.getAttribute("orderByType");

BeanComparator comp = new BeanComparator(orderByCol);
Collections.sort(books, comp);

if (orderByType.equalsIgnoreCase("desc")) {
    Collections.reverse(books);
}
```

Make the necessary imports in the same JSP itself.

```
<%@page import="java.util.Collections"%>
<%@page import="org.apache.commons.beanutils.BeanComparator"%>
```

Change 3: In the same "list.jsp" add the additional two attributes for "<liferay-ui:search-container>" tag.

```
orderByCol="<%= orderByCol %>" orderByType="<%= orderByType %>"
```

Save all changes and check the list now. The new list should be sortable by the first two columns. You should also see the up and down arrows (*icons*) to indicate the current column the list has been sorted. Congratulations! You have successfully implemented this powerful feature of sorting the data list based on a few columns.

There is one small problem. From outside, our changes seem to work fine. Now try editing or deleting a book. The Portlet may throw an error. If you dig deeper into the server console to know the real cause of the error, you will observe this.

```
Caused by: java.lang.UnsupportedOperationException: Please make a copy of this read-only list before modifying it.
```

How to fix this issue? The Section [7.2.1 Adding New Method to Search](#) has the answer. So, let's put this issue in cold storage for some time.

6.9 Performing an action on a set of items

Imagine a situation where we want to implement something that allows the user to delete more than one books from the library at the same time instead of deleting one book at a time. In the current situation he/she can delete only one book at a time. Let's begin with adding a new attribute to “`<liferay-ui:search-container>`” in “`list.jsp`”.

```
rowChecker="<%= new RowChecker(renderResponse) %>"
```

Make the import required for “`RowChecker`”.

```
<%@page import="com.liferay.portal.kernel.dao.search.RowChecker" %>
```

Save and check the list now. You see the first column of the grid with a checkbox. If you click on a row, then the whole row is highlighted in a different color. If you select the checkbox on the top of the grid, then all the rows get highlighted. All this you're able to do without writing one single line of code. This is the beauty of using these Liferay's taglibs. Now we'll actually build a group action button.

Step 1: Place a “Delete” button just above the grid. Insert this code before “`<liferay-ui:search-container>`”.

```
<c:if test="<%= !books.isEmpty() %>">
    <% String functionName =
        renderResponse.getNamespace() + "submitFormForAction();"; %>
    <aui:button-row>
        <aui:button value="delete" cssClass="delete-books-button"
            onClick="<%= functionName %>" />
    </aui:button-row>
</c:if>
```

Correspondingly define the script block with this method at the end of “`list.jsp`”.

```

<aui:script use="aui-base">
    <portlet:namespace>submitFormForAction=function() {
        alert('you're inside submitFormForAction');
    }
</aui:script>

```

Save changes and confirm the alert message is displayed when you click “Delete” button from the list of books page. Right now defer asking me the purpose of `use="aui-base"` as part of `<aui:script>` tag. This is something to do with the Liferay JavaScript libraries and modules. I will be very soon dedicating one section exclusively on this topic. Until then, we’ll be making use of these JavaScript libraries as part of our Library Portlet.

Step 2: Insert this script immediately after “`<aui:script use="aui-base">`”.

```

var deleteBooksBtn = A.one('.delete-books-button');

if (deleteBooksBtn != 'undefined') {
    var toggleDisabled = function(disabled) {
        deleteBooksBtn.one(':button').attr('disabled', disabled);
        deleteBooksBtn.toggleClass('aui-button-disabled', disabled);
    };

    var resultsGrid = A.one('.results-grid');

    if (resultsGrid) {
        resultsGrid.delegate(
            'click',
            function(event) {
                var disabled = (resultsGrid.one(':checked') == null);
                toggleDisabled(disabled);
            },
            ':checkbox'
        );
    }

    toggleDisabled(true);
}

```

Save changes and check the list now. The “Delete” button should enable and disable based on the rows selected (*checked*).

Step 3: Let’s define a form with a hidden field that will capture all the bookIds that need to be deleted when the “Delete” button is clicked. Insert this block immediately before the opening “`<liferay-ui:search-container>`” tag.

```

<portlet:actionURL
    name="<% LibraryConstants.ACTION_DELETE_BOOKS %>"
    var="deleteBooksURL">
    <portlet:param name="redirectURL"
        value="<% iteratorURL.toString() %>" />
</portlet:actionURL>

<aui:form action="<% deleteBooksURL.toString() %>">
    <aui:input name="bookIdsForDelete" type="hidden" />
</aui:form>

```

Declare a new constant in “**LibraryConstants.java**” for this new action.

```
static final String ACTION_DELETE_BOOKS = "deleteBooks";
```

Close the form with “</aui:form>” after the closing “</liferay-ui:search-container>” tag and make necessary indentation for the file to get formatted properly, something you can NEVER get away with. You may also use **Ctrl+Shift+F** but I always prefer manual formatting to this automated Eclipse feature.

Step 4: Write code to submit the form when “Delete” button is clicked. Replace the existing “submitFormForAction” function with this code.

```
Liferay.provide(
    window,
    '<portlet:namespace/>submitFormForAction',
    function() {
        var accepted = confirm('<%='
            UnicodeLanguageUtil.get(pageContext,
            "are-you-sure-you-want-to-delete-selected-books") %>');
        if (accepted) {
            var frm = document.<portlet:namespace/>fm;
            var hiddenField =
                frm.<portlet:namespace/>bookIdsForDelete;
            hiddenField.value =
                Liferay.Util.listCheckedExcept(
                    frm, "<portlet:namespace/>allRowIds");
            submitForm(frm);
        }
    },
    ['liferay-util-list-fields']
);
```

I think we are almost done. Just couple of things left before we get this 100% working. Before we move on insert the import statement for “**UnicodeLanguageUtil**” in “**init.jsp**”.

Step 5: Create a new action method in “**LibraryPortlet.java**” as below,

```
public void deleteBooks(ActionRequest actionRequest,
    ActionResponse actionResponse)
        throws IOException, PortletException {

    String bookIdsForDelete =
        ParamUtil.getString(actionRequest, "bookIdsForDelete");

    // convert this into JSON format.
    bookIdsForDelete = "[" + bookIdsForDelete + "]";

    // The presence of ":" in the string
    // creates problem while parsing.
    // replace all occurrence of ":" with
    // some other unique string, eg. "-"
    bookIdsForDelete = bookIdsForDelete.replaceAll(":", "-");
```

```

// parse and get a JSON array of objects
JSONArray jsonArray = null;
try {
    jsonArray =
        JSONFactoryUtil.createJSONArray(bookIdsForDelete);
} catch (JSONException e) {
    e.printStackTrace();
}

// process the jsonArray
if (Validator.isNotNull(jsonArray)) {
    for (int i=0; i<jsonArray.length(); i++) {
        JSONObject jsonObject = jsonArray.getJSONObject(i);

        long bookId = jsonObject.getLong("bookId");
        try {
            LMSBookLocalServiceUtil.deleteLMSBook(bookId);
        } catch (PortalException e) {
            e.printStackTrace();
        } catch (SystemException e) {
            e.printStackTrace();
        }
    }
}

// redirect to the list page again.
actionResponse.setRenderParameter(
    "jspPage", LibraryConstants.PAGE_LIST);
}

```

Save all changes and test the functionality now. Everything should work perfectly fine as long as you have followed the above steps correctly. In this method, we have converted the input string into a JSON array of objects. Before the conversion we have replaced all occurrence of “:” with “-” as there is some bug and the conversion is not happening if the string contains the character “:”. Once the JSON array is available, we are looping through the array to get the every book that is checked in the UI and delete that book from the database by calling the appropriate API.

Step 6: How about showing a proper message in the JavaScript confirm box when “Delete” is clicked. Currently it is in raw format.

Open “**Language.properties**” under “**WEB-INF/src/content**” and add the first entry for “**are-you-sure-you-want-to-delete-selected-books**”. (*You will see this file only if you have checked the appropriate option at the time of creating the Portlet’sing Eclipse IDE. If it is not there you can manually create now and make a <resource-bundle>content/Language</resource-bundle> entry in portlet.xml*). Save the file and check the Portlet now. The utility “**UnicodeLanguageUtil**” will fetch the right value from the language resource bundle files based on the language set in the “pageContext”.

Why externalization is Important?

The main purpose of this “**UnicodeLanguageUtil**” [<http://bit.ly/10rzE8F>] class is to internationalize our Portlet. We’ll see this concept in detail later in this book. Even if you’re not planning to internationalize your Portlet, it is always a good practice to

externalize all contents (string literals) and messages outside of the JSP and JAVA code, so that it will be easier to change them anytime we want, without touching the code. So, make it a habit while you develop real world applications. We'll discuss more about externalization in many other chapters of this book.

Do it yourself:

Let's conclude this chapter by doing a quick comparison of our list with one of the lists that Liferay has implemented by default. Login as Omni Administrator and click "Control Panel". From the "Portal Section" click "Web Content". You will see a grid with three records by default. Compare and contrast the features of this grid with that our list of books in the Library Portlet. The screenshot is given in the next page.

ID	Title	Status	Modified Date	Display Date	Author	Action
<input checked="" type="checkbox"/>	LIFERAY-BENEFITS	Liferay Benefits	Approved	1/9/13 12:56 AM	1/31/10 1:00 PM	Actions
<input type="checkbox"/>	WHAT-WE-DO	What We Do	Approved	1/9/13 12:56 AM	1/31/10 1:00 PM	Actions
<input type="checkbox"/>	WHO-IS-USING-LIFERAY	Who Is Using Liferay	Approved	1/9/13 12:56 AM	1/31/10 1:00 PM	Actions

Showing 3 results.

Here are some questions for you after you observe this grid.

1. What actions are there for every item in this list?
2. In the action "View" there is one icon after the text. What does it mean? How to implement the same in our Portlet?
3. On what columns the above list can get sorted?
4. What are the group actions available for this list?
5. Why the list paginator is not displayed here?

Summary

We started this chapter by converting the HTML table that we used in Chapter 4 to show the list of books into a well-formatted and neat grid constructed with the help of Liferay's search-container tag. We have seen the various sub-elements of this tag and their attributes. We have fixed the pagination issue by introducing the iteratorURL for the search-container tag. In the next section, we have further improvised the list by properly formatting the date column. In the process we have seen how to refer to some jar files and tld files that are part of the portal. The file "**liferay-plugin-package.properties**" came very handy to us during this process.

In the next section, we have created a link to delete one book from the library by writing a new action method in the Portlet class. We continued improving the list by introducing a set of actions with the help of a new file "**actions.jsp**". Inside this file we have used some new tags to display the actions as a list of icons. Each icon was mapped to either a renderRequest or an actionRequest. We merged even the delete link as part of this action list. We tested all our functionality and made sure everything worked as usual.

We continued to build more functionality for our Library Portlet. This time, we implemented the Edit Book feature to update the book information. This actually comprised of two parts – first to fetch the book for editing and writing an action to actually update the book with the new set of details. At the end of this exercise, we have seen how to redirect the flow to another page after an update action. In the process we have moved the modifyBook method from the Portlet class to the DTO class. After this we have introduced the book details page and URL to come back to the list page, viewing the book details as a popup.

In the last couple of sections of this chapter, we have seen how to make the columns of the grid sortable. We presented the whole code that you can use in your real time applications. In the last section, we saw how to perform an action on a set of items. We got ourselves accustomed with lot of new JavaScript code and handling the multiple items in the Portlet class after the form gets submitted. We concluded this chapter mentioning about the importance of externalization and finally with some "Do it yourself" task. I am sure you're able to do that task successfully.

Congratulations! In this chapter you have seen many features around the search container and how write the actual Portlet functionality. In the next chapter we are going to bring our discussion back to our Service Layer and Service Builder in order to see some more advanced features and facilities.

7. Data Retrieval Methods

This chapter covers

- Order By Clause
 - Finder Tags
 - Dynamic Query
 - Custom SQL Statements
 - Some Real world use-cases
 - Which mechanism is Best?
-

This chapter will talk more about the various data retrieval mechanisms that Liferay provides with the help of service layer. In the last two chapters, we have seen how to retrieve the list of books and display the same with the help of Search Container. Now we'll see more specifically on filtering the data based on some actual search criteria. We'll begin this chapter with the most basic “order by” support that the service layer provides out-of-the-box.

7.1 Order By Clause

If you want the data returned by the service layer API to be of some pre-defined order, then you just have to define it in “**service.xml**”. Imagine a scenario where you want the books to be ordered by “`modifiedDate`” with the most recently updated book appearing on the top of the list. Open the “**service.xml**” file and you have to just give this directive for “LMSBook” entity and re generate the service layer. Now the results will always be sorted. You can specify multiple “order by” clause just the same way you do in a SQL query. For a detailed understanding of this feature refer to corresponding section in DTD file.

```
<order by="desc">
<order-column name="modifiedDate" />
</order>
```

7.1.1 API Level support for Sorting

The information that we are going to see here will be applicable for the next section as well. Since, this section is all about sorting and order by I thought it is worth covering it here instead of procrastinating till the next section. The order by tag that we have just seen is a feature available in the service builder. Even if you have not leveraged this feature, you can still achieve the same thing with the help of passing a while invoking any available service layer API’s including the finder methods that we are going to discuss in the next section. Here is a sample:

```
lmsBookPersistence.findAll(start, end, orderByComparator);
```

Here the “`orderByComparator`” is an object of type “`OrderByComparator`” which is an abstract class. You can get more information from <http://bit.ly/15NSM3Z>. To create an “`orderByComparator`” you can take one of the two approaches.

Approach 1: Create a new class that extends the base abstract class, “`OrderByComparator`”. You can see many examples of this type of classes in the portal source. There are close to 80 such classes.

Approach 2: This approach is to get an instance of “`OrderByComparator`” from a factory meant for this purpose. The code will look something like this.

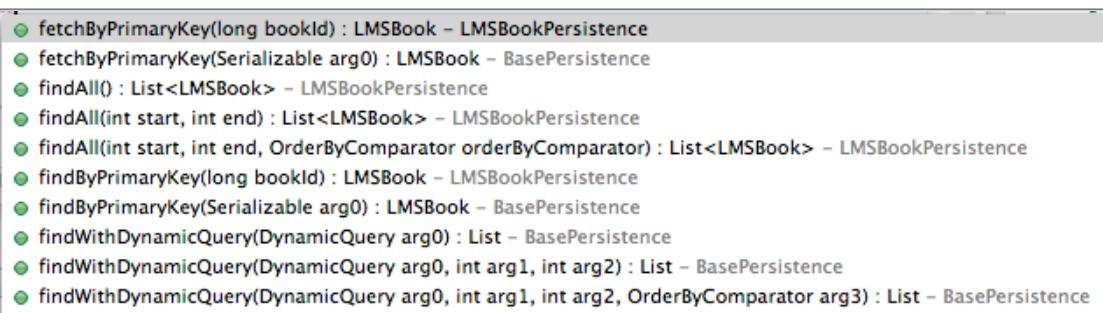
```
OrderByComparator orderByComparator =
    OrderByComparatorFactoryUtil.create(
        "lms_lmsBook", "bookTitle ASC", "author DESC");
```

You need to pass “ASC” or “DESC” as part of the sort columns. You can pass any number of search columns but it should be an even number.

7.2 Finder Tags

From this section onwards we'll actually discuss the various filter options for data that are available out there. The first and foremost of them is defining “`<finder>`” tags in the “`service.xml`” for that entity. A finder tag automatically generates the finder methods in the persistence class that we can make use of in our service implementation classes. Before we go ahead and define anything new, let's see what are the existing finder methods in our persistence object – “`lmsBookPersistence`” which is generated.

Open “`LMSBookLocalServiceImpl.java`” and define an object within one of the two existing methods and immediately put a “.” after it. Autosuggestion feature will list out all those methods.



The screenshot shows a list of generated finder methods for the `LMSBookPersistence` class. The methods are:

- `fetchByPrimaryKey(long bookId) : LMSBook` – `LMSBookPersistence`
- `fetchByPrimaryKey(Serializable arg0) : LMSBook` – `BasePersistence`
- `findAll() : List<LMSBook>` – `LMSBookPersistence`
- `findAll(int start, int end) : List<LMSBook>` – `LMSBookPersistence`
- `findAll(int start, int end, OrderByComparator orderByComparator) : List<LMSBook>` – `LMSBookPersistence`
- `findByIdPrimaryKey(long bookId) : LMSBook` – `LMSBookPersistence`
- `findByIdPrimaryKey(Serializable arg0) : LMSBook` – `BasePersistence`
- `findWithDynamicQuery(DynamicQuery arg0) : List` – `BasePersistence`
- `findWithDynamicQuery(DynamicQuery arg0, int arg1, int arg2) : List` – `BasePersistence`
- `findWithDynamicQuery(DynamicQuery arg0, int arg1, int arg2, OrderByComparator arg3) : List` – `BasePersistence`

The following is the breakup, (*The numbers may slightly vary on 6.2.X version*)

1. Two “`fetchBy`” methods – we have used one of them in earlier chapters
2. Three “`findAll`” methods – we have used one of them in earlier chapters
3. Two “`findBy`” methods – what is the difference between this and corresponding “`fetchBy`” method? We'll see this
4. Three “`findWith`” methods – basically used while creating a dynamic query

Now let's actually go to our “`service.xml`” define the following two finder tags and re generate the service layer to check the new API's of the persistence Class. You can have any number of finder tags defined in this file. But at the same time please make sure you don't put finders for every small thing and make it very cluttered, then it'll become very difficult to manage our finder methods and effectively use them.

```
<finder name="BookTitle" return-type="Collection">
    <finder-column name="bookTitle"/>
</finder>

<finder name="BookTitle_Author" return-type="LMSBook">
    <finder-column name="bookTitle"/>
    <finder-column name="author"/>
</finder>
```

The first finder tag returns a “`Collection`” of objects (a list). The second one returns only one object of type “`LMSBook`”. The following table shows the additional attributes for both “`finder`” and “`finder-column`” tags. For details, you're strongly encouraged to refer to the DTD file.

finder	finder-column
--------	---------------

db-index	arrayable-operator
unique	case-sensitive
where	Comparator

Once the service layer is re-generated, let's introspect the new methods of the persistence class. There are four “**fetchBy**” methods and seven “**findBy**” methods newly added.

```
• fetchByBookTitle_Author(String bookTitle, String author) : LMSBook – LMSBookPersistence
• fetchByBookTitle_Author(String bookTitle, String author, boolean retrieveFromCache) : LMSBook – LMSBookPersistence
• fetchByBookTitle_First(String bookTitle, OrderByComparator orderByComparator) : LMSBook – LMSBookPersistence
• fetchByBookTitle_Last(String bookTitle, OrderByComparator orderByComparator) : LMSBook – LMSBookPersistence
```

```
• findByBookTitle(String bookTitle) : List<LMSBook> – LMSBookPersistence
• findByBookTitle(String bookTitle, int start, int end) : List<LMSBook> – LMSBookPersistence
• findByBookTitle(String bookTitle, int start, int end, OrderByComparator orderByComparator) : List<LMSBook> – LMSBookPersistence
• findByBookTitle_Author(String bookTitle, String author) : LMSBook – LMSBookPersistence
• findByBookTitle_First(String bookTitle, OrderByComparator orderByComparator) : LMSBook – LMSBookPersistence
• findByBookTitle_Last(String bookTitle, OrderByComparator orderByComparator) : LMSBook – LMSBookPersistence
• findByBookTitle_PrevAndNext(long bookId, String bookTitle, OrderByComparator orderByComparator) : LMSBook[] – LMSBookPersistence
```

If you observe closely, there is only one “**findBy**” method for the finder that returns a single object, “**LMSBook**”. All “**fetchBy**” methods always return one single object. Actually in your application code (***ServiceImpl.java**), you can invoke both these kind of API’s. Let’s do a quick exercise to see the difference between “**findBy**” and “**fetchBy**”. It is always recommended to use “**fetchBy**” when you want to retrieve a single object through your persistence layer.

Put this dummy code, inside one method of “**LMSBookLocalServiceImpl.java**”.

```
try {
    LMSBook lmsBook1 =
        lmsBookPersistence.fetchByBookTitle_Author(bookTitle, author);
} catch (SystemException e) {
    e.printStackTrace();
}

try {
    LMSBook lmsBook2 =
        lmsBookPersistence.findByBookTitle_Author(bookTitle, author);
} catch (NoSuchBookException | SystemException e) {
    e.printStackTrace();
}
```

From this code, you can clearly see “**fetchBy**” method throws a “**SystemException**” if the record is not found, where as “**findBy**” method additionally throws “**NoSuchBookException**”. You can remove the dummy code in this file once you’re comfortable with the concepts.

7.2.1 Adding New Method to Search

In this section we'll make use of one of these finder methods auto-created in the previous section in order to build custom search functionality. We'll implement the new feature by taking a layered approach. This approach you can keep as a generic methodology (technic) for any of your real time application development and this is not limited to this feature alone. If you start seeing your whole application from the layered perspective and break down the functionality at various layers, you will be able to nicely and subtly build complex applications. This is an art in itself, which I am sure you will soon master. It is just that it requires a little practice.

Consider our application to have three distinct layers. This is nothing but our classical MVC pattern.

Layer 1	Layer 2	Layer 3
Portlet UI (JSP)	Portlet Class (Controller)	*ServiceImpl.java

It is always better to first get the back-end API's ready. We also follow the same rule.

Step 1: Define a new API for custom search inside “**LMSBookLocalServiceImpl**”.

```
public List<LMSBook> searchBooks(String bookTitle)
    throws SystemException {
    return lmsBookPersistence.findByBookTitle(bookTitle);
}
```

Re-generate the service layer for the changes to take effect.

Note: There is a wrapper utility class for any persistence object. In our case it will be “LMBookUtil”, the persistence utility for the LMSBook service. This utility wraps LMSBookPersistenceImpl and provides direct access to the database for CRUD operations. The service layer should only use this utility, as it must operate within a transaction. Never access this utility in a JSP, controller, model, or other front-end class. You can invoke a finder method through this utility also.

Step 2: Now let's get the UI ready. Append this code in “**view.jsp**”.

```
<hr />
<portlet:actionURL var="searchBooksURL"
    name="<% LibraryConstants.ACTION_SEARCH_BOOKS %>" />

<aui:form action="<% searchBooksURL.toString() %>">
    <aui:input name="searchTerm" label="Enter Title to search" />
    <aui:button type="submit" value="Search" />
</aui:form>
```

We have defined a new “**actionURL**” in a declarative way and also a new constant.

```
static final String ACTION_SEARCH_BOOKS = "searchBooks";
```

Save changes and check the UI. You will see a form. Click “Search” and it will throw error, as we have still not defined the action in our Portlet class.

Step 3: Open “**LibraryPortlet.java**” and define a new “processAction” method named `searchBooks`. This is where we are going to literally glue the UI and the back-end layers.

```
public void searchBooks(ActionRequest actionRequest,
    ActionResponse actionResponse)
        throws IOException, PortletException {
    String searchTerm =
        ParamUtil.getString(actionRequest, "searchTerm");

    if (Validator.isNull(searchTerm)) return;

    try {
        List<LMSBook> lmsBooks =
            LMSBookLocalServiceUtil.searchBooks(searchTerm);

        actionRequest.setAttribute("SEARCH_RESULT", lmsBooks);
        actionResponse.setRenderParameter("jspPage",
            LibraryConstants.PAGE_LIST);
    } catch (SystemException e) {
        e.printStackTrace();
    }
}
```

Attention: It is important to note that the below line is sufficient for redirecting the request to a new JSP page after the action is performed. We don’t have to explicitly call, “`actionResponse.sendRedirect(redirectURL)`;” as we did earlier.

```
actionResponse.setRenderParameter(
    "jspPage", LibraryConstants.PAGE_LIST);
```

Save changes and check the Portlet. When you click “Search” after entering a book title, it is currently taking you to the list page. But unfortunately it is showing all the books. We wanted to see only the book that was entered in the box. How to change this behavior? Step 4 has the answer.

Step 4: Let’s revisit our UI layer, but this time “**list.jsp**”. Inside the scriptlet, replace the first line given first with the block given next.

```
List<LMSBook> books = LMSBookLocalServiceUtil.getLMSBooks(0, -1);

List<LMSBook> books = (List<LMSBook>)
    request.getAttribute("SEARCH_RESULT");
if (Validator.isNull(books)) {
    books = LMSBookLocalServiceUtil.getLMSBooks(0, -1);
}
```

In the above code, we are trying to retrieve the attribute “`SEARCH_RESULT`” earlier set by “`searchBooks`” method of our Portlet class. Save changes and try to search our library for a proper (valid) book title. After clicking “Search” it is taking you to the list page, but there is an error. If you check the server console, the error says,

```
Caused by: java.lang.UnsupportedOperationException: Please make a
copy of this read-only list before modifying it.
```

This is exactly the same problem we face at the end of last chapter. We keep it aside. Let's take this out and fix permanently. The reason for this error is; Liferay has made all objects of type “java.util.List” as un-modifiable. While developing Portlets, you should take care of this. Let's modify the original block.

```
List<LMSBook> booksTemp = (List<LMSBook>)
    request.getAttribute("SEARCH_RESULT");

List<LMSBook> books = Validator.isNotNull(booksTemp)?
    ListUtil.copy(booksTemp) :
    LMSBookLocalServiceUtil.getLMSBooks(0, -1);
```

Save changes and the search should work properly now. Test this by giving the title of a book that is actually in our library and that of one not there.

Important: Even after fixing the issue we may still get the same error thrown by the line to sort the list, “Collections.sort(books, comp);”. As before, we need to assign the list to a temporary list, do the sorting and reassign back to the original list. Surround both `sort` and `reverse` functions on the list with a call to `copy`.

```
booksTemp = ListUtil.copy(books);
Collections.sort(booksTemp, comp);

if (orderByType.equalsIgnoreCase("desc")) {
    Collections.reverse(booksTemp);
}
books = ListUtil.copy(booksTemp);
```

7.2.2 Pros and Cons of Finder tags

There are both advantages and limitations of using finder tags. First let's see the advantages.

- For every finder tag this is defined in service.xml Liferay automatically generates the corresponding index and it gets executed against the database at the time of deployment. Because of this natural index created, the fetching of the data from the database is really very fast. You can open “**indexes.sql**” inside “**WEB-INF/sql**” folder to check these SQL statements.

```
create index IX_7896CF6B on LMS_LMSBook (bookTitle);
create index IX_6001C2AA on LMS_LMSBook (bookTitle, author);
```

- For every finder tag, all the corresponding API's are auto-created and are well defined. The same can be quickly accessed / used in our implementation classes. The data retrieved are also cached in the server's VM. We have already explained about caching in chapter 5.

Now let's see the limitations of using finder tags. I can list down four of them here.

1. We have to know them in advance at the time of developing our application itself.

-
- 2. Every time a finder is defined, the service layer has to be re-generated and the Portlet deployed.
 - 3. Finder methods do not take wild card characters. For e.g. we cannot find all books that contain the word “java” using a finder method.
 - 4. We cannot perform complex queries on our tables – the ones involving joins and sub queries.

7.3 Dynamic Query

While developing applications (Portlet) in Liferay you may come across situations where the capabilities provided by the basic finder methods are not at all sufficient. For example .. there is a search for where the user can fill in either the book title or the author name or both. If the user has only input book title, then we need to query the database only on this field. If the user has input only author, then the query criteria is going to be different. In short, the nature of the query itself is going to change during run time based on the user input or any other factors.

In order to simplify our efforts Liferay provides several ways to define complex queries used in retrieving database data. Each service Entity typically defines several 'finder' methods that form the basis for the default views used throughout the portal. There are several use-cases that force us to see beyond those existing “finder” queries:

- 1. The level of complexity allowed by the service generation is not sufficient
- 2. Queries that implement aggregate SQL operations such as max, min, avg, etc.
- 3. Return composite objects or tuples rather than the mapped object types
- 4. Access data in a way not originally conceived for whatever reason
- 5. Query optimization, complex data access, like reporting

In this section we are going to see how to use Liferay dynamicQuery API to search books in our library that satisfy a particular criteria. We are just going to make changes to the implementation of our underlying “searchBooks” written inside “**LMSBookLocalServiceImpl.java**”. Replace the earlier return statement with,

```
DynamicQuery dynamicQuery =
    DynamicQueryFactoryUtil.forClass(LMSBook.class);

Property bookTitleProperty = PropertyFactoryUtil.forName("bookTitle");
dynamicQuery.add(bookTitleProperty.like("%" + bookTitle + "%"));

return dynamicQuery(dynamicQuery);
```

Don’t forget to make the proper import for “PropertyFactoryUtil”. Save the changes. One good news; you need not have to re-generate the service layer and only the method implementation has been changed and nothing else. Confirm the search for book title is working fine. Now it should allow you to find books in our library that “contain” some key words. It was not possible with the earlier search implementation.

The above is the simplest of dynamic queries that you can ever think of. In real time they are going to be very complex with,

-
- Setting of criteria by using one of these utility classes, `PropertyFactoryUtil` [<http://bit.ly/16tYVDW>] or `RestrictionsFactoryUtil` [<http://bit.ly/XxSzIX>].
 - Setting a sub query as a criteria itself
 - Using projections and aggregate functions
 - Using various kinds of join operations on the tables

For effectively using dynamic queries you have to be well versed with Hibernate SQL API's. Liferay has just build wrappers around these Hibernate API's.

Other layers un-touched: One thing you have to appreciate and realize here is the fact that when we wanted to change the search implementation from normal finder to dynamic query, the only place we touched was the “searchBooks” API of “**LMSBookLocalServiceImpl**” and the other layers have not been touched at all. This is the freedom and flexibility you get if you have implemented our business logic into this separate layer.

7.3.1 Scenarios while using DynamicQuery

There are a couple of scenarios that you have to take care while using dynamic query API of Liferay.

Scenario 1: By default, the dynamic query API uses the current thread's class loader rather than the portal class loader. Because the Impl classes can only be found in Liferay's class loader, when you try to utilize the Dynamic Query API for plugins that are located in a different class loader, Hibernate silently returns nothing instead. The solution for this is to pass the portal class loader when you're initializing your dynamic query object, and Hibernate will know to use the portal class loader when looking for classes.

For e.g. you have to query the “User” entity of the Portal through dynamic query. This is how you will initialize the `dynamicQuery` object. We'll see a real example for this type of usage in [Chapter 10](#) of this book.

```
DynamicQuery dynamicQuery =  
    DynamicQueryFactoryUtil.forClass(  
        User.class, PortalClassLoaderUtil.getClassLoader());
```

Scenario 2: While using projections in your query you have to use the Liferay's version of that `Projection` class. The projections are usually the stuff that you put in your “SELECT” clause of a SQL query. Using projections you can retrieve only few columns of the given entity, or row count, or a max value for a particular column. To set a projection on a dynamic query, you have to use a statement like this.

```
dynamicQuery.setProjection(ProjectionFactoryUtil.rowCount());
```

7.3.2 Example of Dynamic Query API usage

There is one classical place where Liferay has used a complex dynamic query API to query layout and portletpreferences tables. Open “**LayoutLocalServiceImpl.java**”. This file is inside “`com.liferay.portal.service.impl`” of “**portal-impl**”. The query is inside the method “`updateScopedPortletNames`” of this class. This code uses almost all concepts around dynamic query API. Please try to understand this code, as it will give some very good insight about the proper usage of dynamic query. I’ve put some excerpts from the original method for your ready reference.

```
DynamicQuery portletPreferencesDynamicQuery =
    DynamicQueryFactoryUtil.forClass(
        PortletPreferences.class,
        PortletPreferencesImpl.TABLE_NAME,
        PACLClassLoaderUtil.getPortalClassLoader()));

Property plidProperty = PropertyFactoryUtil.forName("plid");

DynamicQuery layoutDynamicQuery =
    DynamicQueryFactoryUtil.forClass(
        Layout.class, LayoutImpl.TABLE_NAME,
        PACLClassLoaderUtil.getPortalClassLoader());

Projection plidProjection = ProjectionFactoryUtil.property("plid");

layoutDynamicQuery.setProjection(plidProjection);

Property groupIdProperty = PropertyFactoryUtil.forName("groupId");

layoutDynamicQuery.add(groupIdProperty.eq(groupId));

Property privateLayoutProperty =
    PropertyFactoryUtil.forName("privateLayout");

layoutDynamicQuery.add(privateLayoutProperty.eq(privateLayout));

portletPreferencesDynamicQuery.add(
    plidProperty.in(layoutDynamicQuery));

Junction junction = RestrictionsFactoryUtil.disjunction();
```

With the incredible power of dynamic query API, sadly there are limitations too . The biggest limitation is if you want to change the query, you have to browse through all those lines of code and make changes to the java file. Remember, every change in the code requires the Portlet to be packaged and redeployed again. The other disadvantage is the developer has to be well versed with the API syntax in order to effectively make use of it. A wrong usage will lead to lot of time wasted in debugging the query. Just to ensure the correctness of the query, the functionality has to be run multiple times.

You have to use dynamic query API very judiciously while developing real time applications as it is like a double-sided sword. Only when you’re sure that the code that you’re going to write using dynamic query is not going to create any issues, should you attempt it. I bet, if you master this API, then you will start loving it and will not go for any other methods except using this.

7.4 Custom SQL Statements

In this section we are going to cover the third method of querying the database. This method is combines the API approach of the finder tags and flexibility of dynamic query. The greatest advantage of this method is the separation of the actual query from the code, hence making the query more maintainable and modifiable. Another attraction of this method is the ability to write our queries in standard SQL format in a XML file that comes very natural for any application developer who has got decent knowledge about writing SQL statements. These SQL queries are transformed into HQL (Hibernate Query Language) during run time and executed against the Hibernate ORM layer in order to fetch the results from the underlying table(s).

Let's see one example of making use of custom SQL statements. You need to follow some simple steps. (*This approach is also called as **custom finders**.*)

Step 1: Create “**default.xml**” under “**WEB-INF/src/custom-sql**” (*create this folder first with the same name*) and put this content.

```
<?xml version="1.0" encoding="UTF-8"?>
<custom-sql>
    <sql id="com.slayer.service.persistence.LMSBookFinderImpl.findBooks">
        <![CDATA[
            SELECT *
            FROM lms_lmsbook
            WHERE (bookTitle like ?)
        ]]>
    </sql>
</custom-sql>
```

This file will have all the queries in simple SQL format in the form of key-value pairs. The key is the `id` that can be anything. The value is the actual query embedded within `<![CDATA[` and `]]>`. If you look the value it is nothing but a simple SQL select statement. We are done with the configuration step.

Step 2: Create a new finder implementation class “**LMSBookFinderImpl**” inside “**WEB-INF/src/com/slaver/service/persistence**”. While creating this class make it implement “**LMSBookFinder**” interface and extend “**BasePersistenceImpl**”. Immediately after creating the file with Eclipse, it will look like this.

```
package com.slayer.service.persistence;

import
com.liferay.portal.service.persistence.impl.BasePersistenceImpl;
import com.slayer.model.LMSBook;

public class LMSBookFinderImpl
    extends BasePersistenceImpl<LMSBook>
    implements LMSBookFinder {
    // empty
}
```

Unfortunately the compiler is not able to find “`LMSBookFinder`”. To get rid of this problem, re-generate the service layer after saving the new file. After generation, you will see new files inside “**WEB-INF/service/com/slayer/service/persistence**”.

- `LMSBookFinder` – interface which we are implementing
- `LMSBookFinderUtil` – The wrapper utility class around our just created implementation class

Step 3: Introduce a new custom finder API inside the class created in the above step. This API will in turn use our custom SQL query.

```
static String FIND_BOOKS =
    LMSBookFinderImpl.class.getName() + ".findBooks";

@SuppressWarnings("unchecked")
public List<LMSBook> findBooks(String bookTitle)
    throws SystemException {

    // 1. Open an ORM session
    Session session = openSession();

    // 2. Get SQL statement from XML file with its name
    String sql = CustomSQLUtil.get(FIND_BOOKS);

    // 3. Transform the normal query to HQL query
    SQLQuery query = session.createSQLQuery(sql);

    // 4. Add the actual entity to be searched
    query.addEntity("LMSBook", LMSBookImpl.class);

    // 5. Replace positional parameters in the query
    QueryPos qPos = QueryPos.getInstance(query);
    qPos.add(bookTitle);

    // 6. Execute query and return results.
    return (List<LMSBook>) query.list();
}
```

Make all the necessary imports required by the new code.

```
import com.liferay.portal.kernel.dao.orm.QueryPos;
import com.liferay.portal.kernel.dao.orm.SQLQuery;
import com.liferay.portal.kernel.dao.orm.Session;
import com.liferay.portal.kernel.exception.SystemException;
import com.liferay.util.dao.orm.CustomSQLUtil;
import com.slayer.model.impl.LMSBookImpl;
```

In writing any such a custom finder API, there are six distinct steps.

1. Open an ORM session (Hibernate)
2. Get an SQL statement from XML file with its name
3. Transform the normal query to HQL query
4. Add the actual entity to be searched
5. Replace positional parameters in the query
6. Execute query and return results

As we have written a new finder API, re-generate the service layer.

Step 4: Update “searchBooks” in “**LMSBookLocalServiceImpl**” to invoke our new custom finder. Comment out the existing code and replace it with this one line.

```
return LMSBookFinderUtil.findBooks("%" + bookTitle + "%");
```

Make the import for `LMSBookFinderUtil`. Remove unwanted imports by pressing **Ctrl+Shift+O**.

Save changes and confirm the search feature is working as normal as before. Congratulations! You have successfully implemented all the three major methods of querying and retrieving data from the underlying database through the service layer.

Code Changes @ <http://code.google.com/p/lr-book/source/detail?r=24>

Note: Whenever you encounter the following error anytime during your development work, just restart the server and continue your development activities. Sometime the shutdown does not happen properly when you do it from Eclipse. In those cases, you have to manually kill the process and start the server afresh.

```
Exception in thread
"ContainerBackgroundProcessor[StandardEngine[Catalina]]"
java.lang.OutOfMemoryError: PermGen space
Exception in thread "http-bio-8080-exec-549"
java.lang.OutOfMemoryError: PermGen space
```

7.4.1 Liferay's use of custom SQL

Liferay has extensively used custom SQL as part of its portal source code. You can have a look into “**portal-impl/src/custom-sql**” folder of Liferay portal source. Apart from the custom SQL written at the portal level through “**portal.xml**” there are custom SQL defined for the various portlets that come bundled with Liferay.

Announcements	Dynamic Data Lists	Blogs	Social
Bookmarks	Document Library	Journal	Calendar
Asset	Dynamic Data Mapping	Polls	Shopping
Message Boards	Mobile Device Rules	Ratings	Wiki

You're strongly encouraged to open each of the xml files present inside this folder and have a look at the various custom SQL statements that are defined. Most of them are quite direct and some of them are complex. Many of the SQL statements also have INNER and OUTER joins. Some queries have replacement strings like - - [\$CLASS_NAME_ID_COM.LIFERAY.PORTLET.MESSAGEBOARDS.MODEL.MBMESSAGE\$], which get replaced with actual values during runtime.

7.5 Some Real world use-cases

This section covers some very important real world use-cases related with the service layer. Some of them may not be Liferay standards, but can be considered as best

practices and design patterns worth adapting in our real time development of complex business and web applications. I am presenting them here as a knowledge sharing exercise and also to equip you with these advanced topics to face the challenges of real world applications.

7.5.1 Expressing Entity relationship through “service.xml”

We have touched on this topic at the end of chapter 5 as part of “Do It Yourself” exercise in 5.10. Like how we can model relationships between the tables of a given databases, we can replicate exactly the same in the service layer with the help of “**service.xml**”. All the three types of relationships can be seamlessly expressed – **one-to-one**, **one-to-many** and **many-to-many** between any two entities.

You must have seen the examples of these relationships in the “**service.xml**” files of Shopping and Software Catalog Portlets. In this section, we’ll create one more entity in our service layer for the Library Portlet and establish a one-to-many relationship between our “**LMSBook**” and this new entity. Let’s name this as “**LMSBorrowing**” which is a transactional table with the following fields.

borrowingId	bookId	memberId	dateBorrowed	dateReturned
(long - PKey)	(long - FKey)	(long - FKey)	(Date)	(Date)

Any library book can have multiple records of this entity (one-to-many). Whenever a member of the library borrows a book, one record gets inserted into this table with even the memberId of the borrower being stored. The table has a primary key and two foreign keys – one to the LMSBook table itself (parent) and another to the User table where member information are stored. If you want to express this relationship at the database level, you have all liberty to do that. But honestly it is not going to help much if your Portlet is going to get deployed onto an environment where you do not have control on the database. So, it is preferred to keep your stuff as database independent as possible. This is the whole mantra behind using an ORM.

Step 1: Define this entity as part of our “**service.xml**” and re-generate the service layer for the corresponding table to get created in the underlying database.

```
<entity name="LMSBorrowing" local-service="true"
        remote-service="false">
    <!-- PK fields -->
    <column name="borrowingId" type="long" primary="true"
            id-type="increment"/>
    <!-- FK fields -->
    <column name="bookId" type="long" />
    <column name="memberId" type="long" />

    <!-- Audit fields -->
    <column name="dateBorrowed" type="Date" />
    <column name="dateReturned" type="Date" />
</entity>
```

Step 2: After successful deployment, you see the new table “LMS_LMSBorrowing” created in our database. Run the following script to make some initial alterations to the table, mainly for setting the primary key as auto-increment.

```
ALTER TABLE lportal.LMS_LMSBorrowing
CHANGE COLUMN borrowingId borrowingId BIGINT(20)
    NOT NULL AUTO_INCREMENT,
CHANGE COLUMN bookId bookId BIGINT(20) NOT NULL,
CHANGE COLUMN memberId memberId BIGINT(20) NOT NULL,
CHANGE COLUMN dateBorrowed dateBorrowed DATETIME NOT NULL;
```

Next is setting the foreign key at the database level. For some reason Liferay has not done this for all its tables. But it is always a good practice to set these relationships in order to maintain the referential integrity and accidental delete of a parent when children exist. It is also good from the maintainability point of view so that what table has got what relationship with other tables. Run the following scripts to establish the relationship of our new table with “LMS_LMSBook” and Liferay’s “User_” table.

```
alter table LMS_LMSBorrowing add foreign key (bookId)
    references LMS_LMSBook (bookId);
alter table LMS_LMSBorrowing add foreign key (memberId)
    references User_ (userId);
```

Step 3: It is time for us to express this one-to-many relationship at the service layer. Go back to “**service.xml**” and introduce a new column for “LMS_LMSBook” as shown here. You can insert this column before the audit field section.

```
<!-- Relationship -->
<column name="lmsBorrowings" type="Collection"
entity="LMSBorrowing" mapping-key="bookId" />
```

Unlike a normal column, a column that denotes a relationship has two more additional attributes – **entity** that maps to its child and **mapping-key** the key of the child entity through which the relationship is established. In one-to-many relationship the column defined in the parent entity returns a list of objects (**Collection**), hence the column name usually given in plural (**lmsBorrowings**). In one-to-one relationship the return type will be the entity type of the other entity and returns only one object. Re-generate the service layer for the changes to take effect and you will see new getter methods (`getLMSBorrowings`) available in “`lmsBookPersistence`” and its corresponding wrapper “`lmsBookUtil`”.

Unfortunately the model object does not contain an attribute for “`lmsBorrowings`” and its corresponding getter and setter methods. One of my colleagues at mPower Global has modified the service builder through an EXT plugin to generate these methods for the referential attributes of any entity. In the next step, let’s see how to manually create a new getter method in the “`ImsBook`” model class.

Step 4: This step covers the details of overriding a generated model class, something very similar to how we override the “**LMSBookLocalServiceImpl**” with new set of API’s. Assume we have a method “`getBorrowings()`” that will return all “`LMSBorrowing`” children of an “`lmsBook`” object.

-
- a) Create a new API in “**LMSBookLocalServiceImpl**” and regenerate service layer. This method will call “`getLMSBorrowings`” of low-level persistence layer.

```
public List<LMSBorrowing> getBorrowings(long bookId)
    throws SystemException {
    return lmsBookPersistence.getLMSBorrowings(bookId);
}
```

- b) Open “**LMSBookImpl.java**” inside “**src/com/slayer/model/impl**”. This class originally extends “**LMSBookBaseImpl**”. Define a new method in this class and generate service layer again.

```
public List<LMSBorrowing> getLMSBorrowings()
    throws SystemException {
    return LMSBookLocalServiceUtil.getBorrowings(getBookId());
}
```

You can confirm the availability of a new method for the “`lmsBook`” object.

Alternate Approach: The alternate way of doing the above exercise is to define a finder tag for “`LMSBorrowing`” entity in “`service.xml`” like this.

```
<finder return-type="Collection" name="BookId">
    <finder-column name="bookId" />
</finder>
```

Once this is put and service layer regenerated, you can modify “`getBorrowings`” API, replacing the old line with this new line.

```
return LMSBorrowingUtil.findByBookId(bookId);
```

Code Changes @ <http://code.google.com/p/lr-book/source/detail?r=26>

Overriding model class, a Liferay example

A very good example of overriding the generated model class from Liferay’s source code is that of “`ContactImpl`”. Search for “`ContactImpl`” in Eclipse and open the file. You will see how a new method is made available in this class that extends “`ContactBaseImpl`”.

```
public String getFullName() {
    FullNameGenerator fullNameGenerator =
        FullNameGeneratorFactory.getInstance();

    return fullNameGenerator.getFullName(
        getFirstName(), getMiddleName(), getLastName());
}
```

7.5.2 Modeling Database View into service layer

Imagine a situation that demands you to display in the home page of our Library Management Portal a Portlet with a grid looking like the one below. It should display the most recent 5 items.

Most Recent Borrowings:

Book1	Borrower1
Book2	Borrower2
Book3	Borrower3

How will you combine the three tables – LMSBook, LMSBorrowing and User_ to fetch this information and show in the Portlet? The UI looks pretty simple but to fetch the data you need to do some real gymnastics. Ok, just think of a solution. You may encounter many such situations while developing real applications. After doing lots of permutations and combinations we finally arrived at the final query to fetch this information. *One strong disclaimer – this query may not be the most optimized query. I should admit; I am not a database programmer myself.*

```
SELECT
    rand() as dummyId, a.bookId, a.bookTitle, b.userId as memberId,
    concat(b.lastName, ' ', b.firstName) as memberName,
    c.dateBorrowed
FROM
    lms_lmsBook a, User_ b, lms_lmsBorrowing c
WHERE
    a.bookId = c.bookId and
    b.userId = c.memberId and
    c.dateReturned is null
ORDER BY c.dateBorrowed desc;
```

Any guess? What could be the solution? Let me spill the beans. I can think of the following three solutions.

Solution 1: Completely programmatic – create an API in “LocalServiceImpl” class with multiple loops (iterations) to arrive at the data what we want and finally return it back to the UI. Definitely this is not an optimal solution as it is going to badly hit the application performance. It may even bring the server down if the data is going to exponentially increase.

Solution 2: Create an entity called “**LMSRecentBorrowing**” through “**service.xml**” with only the columns required by our new Portlet. Once the entity is in place, you can either make use of dynamic query API OR use Custom SQL in order to fetch the data from the back-end and populate the fields of this entity. This solution is far better than the previous one as we’ll be directly invoking the query. Let’s now actually create this new logical entity in “**service.xml**”.

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

```

<column name="dummyId" type="long" primary="true" />

<column name="bookId" type="long" />
<column name="bookTitle" type="String" />

<column name="memberId" type="long" />
<column name="memberName" type="String" />

<column name="dateBorrowed" type="Date" />
</entity>
```

Remember, this is a logical entity so we don't want a table to get automatically created in the database. To disable this, create a new file “**service-ext.properties**” under “**WEB-INF/src**” and insert “`build.auto.upgrade=false`” before generating the service layer. Be cautious while doing this, as it might flush the previous data stored in our tables. Also note, we have kept a dummy column as primary key for this entity as service builder does not accept any entity without a valid primary key.

Solution 3: I am presenting another holistic solution to meeting this use-case. But be informed, this solution is applicable only for applications where the underlying database supports the concept of “views”. You should be aware a view is a virtual table and data cannot be inserted, updated or deleted there. It is just a logical view that may pull data from either a single table or from multiple tables using join and some complex selection criteria.

In this solution, we'll create a view in our underlying database with the “create view” command and point our new logical entity in “**service.xml**” to this view with the “`table`” as if we are pointing to a real table. Give the name of the view as the value of this attribute. Generate the service layer and just one call will get you the data you want to be displayed on the UI home page.

```
LMSRecentBorrowingLocalServiceUtil.getLMSRecentBorrowings(0, -1);
```

7.5.3 Service Layer Dummy Entity

In some scenarios, you might want a dummy service layer entity where you can just define the back-end API's for various reasons. In such situations, you can define an entity in “**service.xml**” without any columns and re-generate the service layer. Let's put a self-closing entity definition as below, re-generate the service layer and check.

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

Notice new implementation class generated under “`com.slayer.service.impl`” package. You can start writing generic API's in this class that can actually make use of the persistence and service layer of other entities through the appropriate utility wrappers, e.g. “`LMSBookUtil`”.

7.5.4 Connecting to different Data Sources

Another typical use-case in the real world situations is having multiple database schemas. The Liferay application may have to get connected and interact with many different databases. By default Liferay connects to the default data source defined in the “**portal-ext.properties**”. Unlike simple applications, connecting to multiple databases usually happens in the case of large enterprise applications. Liferay provides a mechanism by which we can achieve this enterprise grade capability.

The other possibility is with enterprises that are already having dozens of applications that are already up and running, each of them meeting some specific need of the organization. Now we are building just a portal either for the employee or customers or suppliers of the organization. The CIO of the organization tells very clearly not to disturb any of the existing legacy applications and databases they connect to. At the same time, our new Liferay portal has to fetch information from these different databases originally used by the legacy system.

Connecting to a different data-source can be accomplished in four steps. In the example below I am showing you how to connect to one datasource. But you will follow the same steps to connect to any number of datasources from where you want to push and pull data.

Step 1: Make the entries for the new database inside “**portal-ext.properties**” file of an EXT plugin that has to be deployed into the server. Just to avoid a new plugin, we can quickly put these entries inside “**portal-setup-wizard.properties**” under **PORTAL_HOME** and restart the server.

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

This is for the additional data-source that is going to be accessed by the service layer. In our case, create another database with name “**lportal_custom**” in MySQL.

Step 2: Inside the “**library-portlet**”, create a new file “**ext-spring.xml**” inside “**WEB-INF/src/META-INF**” and copy the contents from the location,
<http://lr-book.googlecode.com/files/ext-spring.xml>

Step 3: Define a new entity in “**service.xml**”, mapped to another data-source and not the default one. The additional three attributes **data-source**, **tx-manager** and **session-factory** are mandatory. Save file and re-generate service layer.

```
<entity name="Employee"  
       local-service="true" remote-service="false"  
       data-source="customDataSource"  
       tx-manager="customTransactionManager"  
       session-factory="customSessionFactory">  
  
    <column name="employeeId" type="long" primary="true" />  
    <column name="employeeName" type="String" />  
</entity>
```

Step 4: Unlike the entities that are mapped to the default data-source, the table for this entity will not be created automatically. We need to create it manually (LMS_Employee). Earlier you would have already created the database with the name “**Iportal_custom**”. You’re done with all the changes. Now invoke this entity like any other entity in your application code.

```
EmployeeLocalServiceUtil.addEmployee(employee);
```

Confirm that the record gets inserted into the new table of “**Iportal_custom**” database. You can even write the application code to pull data from this table and show up inside the Portlet or server console.

7.5.5 Splitting a large “service.xml” file

In real life, the “**service.xml**” file will be much bigger if your application has loads of entities representing various database tables. Hence, it is strongly recommended to split a large file into smaller files and have different package structure for each different service.

By making use of `<service-builder-import>` inside parent “**service.xml**” we can break a big file into multiple smaller ones. You can actually try this by creating another “**service.xml**” file inside a different folder under “**src**”. The DTD gives the following description.

The service-builder-import allows you to split up a large Service Builder file into smaller files by aggregating the smaller Service Builder into one file. Note that there can be at most one author element among all the files. There can also only be one and only one namespace element among all the files. The attribute “file” is interpreted as relative to the file that is importing it.

To test this, create another “**service.xml**” inside “**src/META-INF**” with this content.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE service-builder PUBLIC
  "-//Liferay//DTD Service Builder 6.1.0//EN"
  "http://www.liferay.com/dtd/liferay-service-builder_6_1_0.dtd">

<service-builder package-path="com.slayer1">
  <entity name="Employee1" data-source="customDataSource"
    local-service="true" remote-service="false">

    <column name="employeeId" type="long" primary="true" />
    <column name="employeeName" type="String" />
  </entity>
</service-builder>
```

Notice that the package is different here (`com.slayer1`) and there is NO “author” and “namespace” tags as they can be only in the parent “**service.xml**” and has to be common for all the children. In the parent file add this block and re-generate the service layer to confirm the new files are generated properly.

```
<service-builder-import file="src/META-INF/service.xml" />
```

7.5.6 Invoking a Stored Procedure via Service Layer

At the end of 2012, a Middle Eastern company invited me to architect a solution based on Liferay to build a portal for their Government. I was surprised to see them not ready to use the powerful Service Builder component of Liferay in that application for the simple reason that it does not support “Stored Procedures” – a compact way of writing database queries as routines in the underlying database itself. I had a tough time trying to convince them that calling a stored procedure from the generated service layer is very much possible.

First of all, using stored procedures or triggers in an enterprise application is not a good idea, as they break the fundamental principle of making an application truly agnostic of the underlying database. The moment an application starts having these artifacts, then it becomes very much tied to that database which makes the interoperability to any other database almost impossible. Having said this, in some situations it becomes inevitable to break certain golden rules for various reasons, the biggest being the perception of people.

After this episode I decided to have one section in this book that explains how to invoke a stored procedure through service layer and wrap the results in the form of generated POJO objects. We have to be flexible. Liferay is a platform that gives that flexibility to make the application adapt to any situations. In the rest of this section we’ll see how to create a stored procedure at the database level and how to invoke it through an API call.

Step 1: Create a very simple parameterized stored procedure in your database by running this script. What it does is self-explanatory, although we are going to execute it soon. If you are new to stored procedure, it is worth reading this Wikipedia article, http://en.wikipedia.org/wiki/Stored_procedure.

```
DELIMITER $$

CREATE PROCEDURE GetBooks (in_bookTitle VARCHAR(75))
BEGIN
    SELECT * FROM lms_lmsBook
    WHERE bookTitle like concat('%', in_bookTitle, '%');
END $$

DELIMITER ;
```

Test the stored procedure by executing the command “`CALL GetBooks('java');`” from the MySQL command prompt. This will return all books from the table whose title contains the word “java”. This confirms that our stored procedure is up and running.

Step 2: In this step I wanted to extend the generated persistence class and introduce a new API in the extend class in order to do the job by making use of “`persistence-class`” attribute of “`LMSBook`” entity of “`service.xml`”. As this was throwing a `ClassCastException` during runtime at the time of invoking the method during

runtime, I resorted to our “**LMSBookFinderImpl**” class that we have introduced in an earlier section of this chapter while introducing Custom SQL finders. Create a new method in this class that calls our stored procedure.

```
@SuppressWarnings("unchecked")
public List<LMSBook> findBooksThroughSP(String bookTitle)
    throws SystemException {

    Session session = openSession();

    SQLQuery query =
        session.createSQLQuery("CALL GetBooks(:bookTitle)");

    query.addEntity("LMSBook", LMSBookImpl.class);
    query.setString("bookTitle", bookTitle);

    return (List<LMSBook>) query.list();
}
```

Re-run the Service Builder as we have introduced a new API. One thing I would like to highlight here. We can even invoke any other queries (INSERT, UPDATE or DELETE) from this class other than SELECT queries.

Step 3: The new API is ready to be invoked from any other “*” class. This API will in turn invoke our underlying stored procedure.

```
LMSBookFinderUtil.findBooksThroughSP(bookTitle);
```

7.5.7 Generating Exception classes

Before concluding this chapter I would like to highlight one more useful feature of Service Builder tool. In real world applications, one might require lots of custom exceptions that can be thrown when some specific errors happen during runtime program execution. The same exceptions can be cascaded to the Portlet layer and then to the UI to inform the end-user about what exactly went wrong. With the help of Service Builder, we can auto-generate as many exception classes that we want instead of writing them manually, which is a waste of time and effort.

By default for every entity defined in the “**service.xml**” file there is “**NoSuch<EntityName>Exception.java**” generated. If we need more custom exception classes, all we have to do is to define them inside “**<exceptions>**” tag of “**service.xml**” like how it is shown here. Execute the Service Builder and check the new exception classes that are generated inside “**WEB-INF/service/com/slayer**”. This block should come immediately after the closing “**</entity>**” tag.

```
<exceptions>
    <exception>BookNotAvailableForLending</exception>
    <exception>BookOutOfStock</exception>
</exceptions>
```

7.6 Which mechanism is Best?

After knowing about the four different ways of pulling data from the back-end database through the service layer, a question naturally comes to every one's mind – which one is the best? But honestly this question does not have an answer. The usage of a particular mechanism is purely based on the situation at hand and many other factors one will come to know only through experience and practice. I did a quick experiment just for comparing these four methods from the “performance” point of view. Finder methods emerged as the clear winner.

I am presenting the results as it may be of some use to you while you develop mission critical applications where performance is the key success factor. I used the Apache Commons, “org.apache.commons.lang3.time.StopWatch” to put timers and record my findings. The columns A to F are runs. The values are in milliseconds.

Mechanism	A	B	C	D	E	F	Avg
Finder Method	10	0	0	0	0	0	2
Dynamic Query	24	7	2	4	2	3	7
Stored Procedure	27	10	7	5	4	5	10
Custom SQL	56	11	8	6	5	6	15



We already have seen the advantages and limitations of each of these mechanisms in the respective sections where we discussed them.

The performance of the methods, especially custom SQL and Stored Procedures (worst case) is heavily dependent on the nature of the queries that one writes. The queries have to be well optimized and tested with lots of data. Usually they don't show up their real form during development, as the data will be less. Only in the real production environment, poorly written queries start giving lots of performance issues for the overall application. Hence you need to be very careful while designing the queries, especially the ones with multiple joins and sub-queries. If you were new to SQL, then I would recommend you spend some time around this subject once you complete reading this book.

The best place to start is, <http://www.w3schools.com/sql/default.asp>. If you're planning to use MySQL as the database base for any of your real time applications, then referring to the official MySQL manual is also a good idea to learn the art of writing good queries that are optimal and highly performing. The manual's location is <http://dev.mysql.com/doc/refman/5.6/en/tutorial.html>.

Apart from the above there are a lot of resources available in the Internet which you can always Google and find out for yourself.

Summary

Let's conclude this chapter by summarizing things that we learnt here. We started with "Order by" clause then moved on to discussing the various ways data can be fetched from the back-end through the service layer – finder tags, dynamic query API and custom SQL statements. We continued the chapter by discussing various real world scenarios pertaining to the service layer and how to address them. The topics covered are;

1. Expressing Entity relationship through "**service.xml**"
2. Modeling Database View into service layer
3. Service Layer Dummy Entity
4. Connecting to different Data Sources
5. Splitting a large "**service.xml**" file
6. Invoking a Stored Procedure thru Service Layer
7. Generating exception classes

The next chapter will be the last chapter where we are going to still hover over the service layer itself. The reason is very obvious; its very important role in the whole of Liferay architecture itself. In this whole book, we have dedicated three chapters exclusively on the service layer (*chapter 5, chapter 7 and upcoming chapter 8*).

8. Remote Services and Beyond

This chapter covers

- SOAP / RPC based Web Services
 - JSON Web Service
 - RESTful Web Services
 - The “Beyond” Part
 - Calling Portal’s JSON Web Service
 - Service Context and its significance
-

Before starting this chapter, let me give good news to you. You have all reasons to rejoice. This is going to be the last chapter on a very important framework of Liferay – Service Builder and the layer it generates – Service Layer. I am sure by this time, with the help of chapters 5 and 7, you would have already got friendlier with this layer. This chapter will strengthen this bond and make you much closer with this layer. This is going to be another interesting topic if you’re ready for this true friendship.

We’ll start this chapter with how Liferay has been built from ground up based on Service Oriented Architecture (SOA). Anything and everything in Liferay is exposed as a Web Service. That means the API’s of the portal can be literally accessed from any other application of the world, as long as the other application can access the endpoint URL of the exposed Web Service. Before we dive deeper into the subject, let’s see how W3C (<http://www.w3.org>) defines a Web Service. This is how it goes:

A web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.

There are various ways / methods / protocols this kind of interaction can happen between diverse applications. They are SOAP, RPC, JSON, REST, etc. Getting into the details of each one of them is not the objective of this book. But definitely we’ll see how Liferay supports these different protocols in the first part of this chapter. For a detailed understanding of Web Services and various protocols you can have look here, http://en.wikipedia.org/wiki/List_of_web_service_protocols.

8.1 SOAP / RPC based Web Services

In general, web services are resources that may be called over the HTTP protocol to return data. They are platform independent, allowing communication between applications on different operating systems and application servers. When Service Builder generates the database entries, web services can be generated as well based on Apache Axis. As a result, nearly all of the backend API calls can be made as equivalent web services.

8.1.1 Web Services Exposed by Liferay

First things first, let's begin with seeing the API's of Liferay that are exposed by default. Hit the following URL from your browser, <http://localhost:8080/api/axis>. You must be seeing all the API's that are currently exposed by Liferay under the heading "**And now... Some Services**". Let's take one example from this big list of API's. There are close to a hundred of them.

- Portal_LayoutBranchService ([wsdl](#))
 - addLayoutBranch
 - deleteLayoutBranch
 - updateLayoutBranch

At the top is the Service Group itself. If the prefix is "Portal" then the Portal itself offers the service. If it is "Portlet" then a Portlet that comes bundled with Liferay offers the service. The second part (e.g. LayoutBranchService) is the name of the service itself. Clicking on the wsdl hits the end-point URL of the service and displays the contents of the respective WSDL (Web Services Definition Language) file. The items below the service are the distinct API's that are part of the service group. Each API has its own set of parameters and the details one can find inside the WSDL file. The end-point URL of a service will look something like the one below, http://localhost:8080/api/axis/Portal_LayoutBranchService?wsdl.

8.1.2 Invoking (Consuming) Existing Web Service

Here is an exercise that demonstrates how to invoke an existing Web Service of Liferay either at Portal or Portlet level. The objective of this exercise is to find all organizations a Liferay user belongs to. The API for this operation is under the service, "**Portal_OrganizationService**". This exercise comprises of five steps. We'll start with creating a stand-alone java project that is nothing to do with our portal.

Step 1: Create a simple Java project in Eclipse by right clicking on the left panel and take the necessary actions. **New → Project... → Java Project → Next >**. Give the name of the project as "**ws-client**" and press "Finish". Now you will see a new project with the same name appearing on the left panel.

Step 2: In this step we'll make the stubs required for the Web Services available for the client application. We'll make use of the Eclipse feature to generate these stubs on

the fly by the help of WSDL file. Note down the end-point URL for the service “**Portal_OrganizationService**”. The URL is,

http://localhost:8080/api/axis/Portal_OrganizationService?wsdl

Now right click on the project “ws-client”, **New → Others... → Web Services → Web Services Client** and click “Next >”. In the dialog that opens, give the above URL in the Service definition field and click “Finish”. This task will generate all the required stubs for our client program. Once these files are generated you can check-in them to the repository and there is no need to do this exercise ever again unless and until the API signatures are not getting changed.

Step 3: In this step we’ll actually write the client program. Right click on “src” and create a new Java class by name “**WSTest1**”. Put the following contents in the body of this class. After inserting the code make necessary formatting. As mentioned earlier, for formatting the code use “**Ctrl+Shift+F**”.

```
public static void main(String[] args) {
    System.out.println("Inside the client program!!");

    try {
        OrganizationServiceSoapServiceLocator locator =
            new OrganizationServiceSoapServiceLocator();

        OrganizationServiceSoap soap =
            locator.getPortal_OrganizationService(
                _getURL("Portal_OrganizationService"));

        OrganizationSoap[] organizations =
            soap.getUserOrganizations(101951);

        for (int i = 0; i < organizations.length; i++) {
            OrganizationSoap organization = organizations[i];
            System.out.println(organization.getName());
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
}

private static URL _getURL(String serviceName)
throws Exception {

    String remoteUser = "10195";
    String password = "test1";

    String url = "http://" + remoteUser + ":" + password +
        "@localhost:8080/api/secure/axis/" + serviceName;
    return new URL(url);
}
```

* Change values for “remoteUser” and “password” as per your setup.
Make the necessary imports.

```
java.net.URL;
com.liferay.portal.model.OrganizationSoap;
com.liferay.portal.service.http.OrganizationServiceSoap;
com.liferay.portal.service.http.OrganizationServiceSoapServiceLocator;
```

Change the value for the actual user Id of the user for whom you want to find the organization that he/she belongs. You can get this information from the “User_” table of the underlying database.

Right click on the program and run the program by selecting “**Run As → 1 Java Application**”. You should see the following error in the console. Let’s fix this.

```
WARNING: Unable to find required classes  
(javax.activation.DataHandler and javax.mail.internet.MimeMultipart).  
Attachment support is disabled.  
(401)Unauthorized  
AxisFault  
faultCode: {http://xml.apache.org/axis/}HTTP
```

You can safely ignore the ‘WARNING’. But the other error says that Liferay is not authorizing the credentials we are sending from the client program to invoke the web service. We’ll fix this issue in the next step.

Step 4: Stop the server, append the below entry in your “**portal-setup-wizard.properties**” and restart the server. This is because the current version of Liferay is not taking email Address as part of the credentials while invoking a Web Service. After the server restart, run the client program again. This time the program would have run successfully but without showing any output.

```
company.security.auth.type=userId
```

Step 5: In this step, let’s quickly create few organizations and assign the Omni Admin user to these organizations. To login to the portal now, you have to provide the userId of the Omni Admin for login now. It will no longer take the email address as we have changed the “auth.type” mechanism to “userId” before starting the server.

After login in, go to **Control Panel → Users and Organization** under “Portal” section. Add two regular organizations “**Org1**” and “**Org2**” just for the sake of testing our functionality. In the same page, go to “**Users Without an Organization**” and click “Edit” action for the user “Test Test”. In the edit screen, select “Organizations” and add this user to both the organizations “**Org1**” and “**Org2**” with the “[Select](#)” option.

Save the changes and run the “**WSTest1**” client program again. This time it should properly print the names of the organizations to which this user (being passed in the API, `getUserOrganizations`) belongs.

Congratulations! You have successfully invoked a Web Service exposed by Liferay. Following the same example you can invoke any other Web Service from any of your other applications. In the next section we’ll see how to expose and consume a Web Service for the Library Portlet that that we develop.

8.1.3 Few Important notes around Web Services

There is couple of things that I would like to mention here around this topic. They are detailed below.

Non-Eclipse Environments:

With God's Grace and thanks to Eclipse, we were able to quickly generate the stubs just with the help of the end-point URL for the WSDL corresponding to the Web Service we want to invoke. But imagine situations where this kind of liberty is not there. Don't panic. Liferay has an answer.

Copy the file “**portal-client.jar**” from “**TOMCAT/webapps/ROOT/WEB-INF/lib**” and paste to “**lib**” folder of “**ws-client**”. This jar should ultimately in the class path of the client environment. This jar file contains all the stubs and interfaces required to make the Web Service call. Apart from this file, you need to have the following eight jar files in the class path (“**lib**” folder). You can manually download these files from the Internet or copy them either from your “**PLUGINS-SDK/lib**” or “**TOMCAT/webapps/ROOT/WEB-INF/lib**” and put inside “**lib**” folder of “**ws-client**”.

- activation.jar
- axis.jar
- commons-discovery.jar
- commons-logging.jar
- jaxrpc.jar
- mail.jar
- wsdl4j.jar
- xercesImpl.jar

Securing your Web Service:

By default all Web Services that are exposed by Liferay can be accessed by anyone in the world as long as they are able to supply the proper credentials (if that Web Service needs it) and use the right URL. The “localhost” in the end-point URL has to be replaced with the actual IP address or the domain name of the Web Service host. Liferay also has the ability of selectively accepting Web Service requests ONLY from known sources by configuring the following settings through “**portal-ext.properties**”. In the comma separated value you have to give the IP address of actual sources (servers) that can make the Web Service calls to our server.

```
# See the properties "main.servlet.hosts.allowed" and  
# "main.servlet.https.required" on how to protect this servlet.  
#  
axis.servlet.hosts.allowed=127.0.0.1,SERVER_IP  
axis.servlet.https.required=false
```

In the example above we have see the format of a secure URL for making a Web Service request. Just refer to “**_getURL**” private method of “**WebTest1.java**”. In the URL itself we are encoding the credentials before making the Web Service call. We can also make some Web Services “un-secured” without the need for sending the authorization parameters. An un-secured URL will be of this format. “**serviceName**” has to be replaced with the actual name of the service to be invoked.

```
http://<HOST_ADDRESS>/api/axis/<serviceName>
```

Do it Yourself: Write a client program to insert an user into the Liferay database by making user of the appropriate API present inside “**Portal.UserService**”. Follow exactly the five steps that we have seen just now.

Before moving on to the next step, let’s revert back the changes we made to “**portal-setup-wizard.properties**” and restart the server.

8.1.4 Exposing Web Service for a Custom Plugin

In the previous sections, we have seen how to consume a Web Service that is already exposed by Liferay. In this section we are going to see how to expose a new custom Web Service out of a Portlet that we are developing. For example we want to expose the “*insertBook*” API of our service layer to the outside world, so that anyone in the world can (securely) insert a book into our Library by invoking this API. In order to expose any thing out of Liferay all it takes is just three steps. Can we give a try and see? Here it goes...

Step 1: Identify what you want to expose. As decided before let’s expose “*insertBook*” of “**LMSBook**” entity. First and foremost we have to enable the “*remote-service*” (set to “*true*”) for this entity and run the Service Builder. Immediately after running the Service Builder you will see a new file “*LMSBookServiceImpl*” got generated inside “*com.slayer.service.impl*”. You will also notice a new file “**service.js**” generated inside “**docroot/js**”.

Step 2: Inside the newly generated class “*LMSBookServiceImpl*” that extends “*LMSBookServiceBaseImpl*”, define a new method (API) what we are going to expose. Give same name as its equivalent method in “*LMSBookLocalServiceImpl*”.

```
public LMSBook insertBook(String bookTitle, String author) {  
    return lmsBookLocalService.insertBook(bookTitle, author);  
}
```

All we are doing here is inside this method we are just invoking the method of the corresponding Local Service. Hence, a “Local Service” is one that is available only locally. A “Remote Service” is a service can be obtained globally (from outside of the application). This is the literal difference between these two distinct types of service offered by the service layer.

Step 3: The last step is to actually expose this new service to the outside world. We’ll do this by invoking both “Build Service” and “Build WSDD” targets on the Service Builder. First run “Build Service” and next either click icon next to “Build Services” or from the menu select the project, go to **Liferay → Build WSDD**. What are the effects of running this? Let’s see.

- A new file “**server-config.wsdd**” got generated inside “**WEB-INF**” and deployed to the server. This is the Web Services deployment descriptor that contains information about the Web Services that we are exposing.
- Like before, hit the URL, <http://localhost:8080/library-portlet/api/axis>. You will be amazed to see our new API exposed to the outside world. In this URL

“library-portlet” is the context underwhich our Portlet is running inside the TOMCAT server. The WSDL end-point URL of this service is,
http://HOST_IP/library-portlet/api/axis/Plugin_LMS_LMSBookService?wsdl

Congratulations! You have successfully exposed our custom Web Service. In the next section we'll see how to consume / invoke it from a different application.

8.1.5 Consuming (Invoking) custom Web Service

In this section we are going to see how to consume the Web Service that has just got exposed by our Library plugin. The steps are pretty simple and similar to the ones we have seen in section [8.1.2 Invoking \(Consuming\) Existing Web Service](#). As long as we know the end-point URL there are no problems. Let's start with generating the stubs through Eclipse using the WSDL URL of the Web Service that we want to invoke and call one of its API remotely through RPC (Remote Procedure Call).

Step 1: Generate the stubs and interfaces required for the remote call with the help of WSDL URL through Eclipse. Right click on “ws-client” project → **New → Others...** → **Web Services → Web Service Client**. In the resulting dialog, give the path of our WSDL file in the Service Definition box and click “**Finish**”. Once it is done, you will see the generated files under “src/com/slayer” of “ws-client” project. The stubs are ready. Now it is time to write the actual client program.

Step 2: Create a new class “**WSTest2**” under “src” of “ws-client” project and copy paste the following contents.

```
public static void main(String[] args) {

    LMSBookServiceSoapServiceLocator locator =
        new LMSBookServiceSoapServiceLocator();
    URL portAddress = null;
    try {
        portAddress = _getURL("Plugin_LMS_LMSBookService");
    } catch (Exception e) {
        e.printStackTrace();
    }

    LMSBookServiceSoap soap = null;
    try {
        soap = locator.getPlugin_LMS_LMSBookService(portAddress);
    } catch (ServiceException e) {
        e.printStackTrace();
    }

    if (soap != null) {
        String bookTitle = "Web Services in action";
        String author = "Ahmed Hasan";

        LMSBookSoap lmsBook = null;
        try {
            lmsBook = soap.insertBook(bookTitle, author);
            System.out.println("book Id ==> " +
                lmsBook.getBookId());
        } catch (RemoteException e) {
    }
}
```

```

        e.printStackTrace();
    }
}

private static URL _getURL(String serviceName) throws Exception {
    // Unauthenticated url
    String url =
        "http://localhost:8080/library-portlet/api/axis/" + serviceName;

    return new URL(url);
}

```

Make the necessary imports.

```

java.net.URL;
java.rmi.RemoteException;
javax.xml.rpc.ServiceException;
com.slayer.model.LMSBookSoap;
com.slayer.service.http.LMSBookServiceSoap;
com.slayer.service.http.LMSBookServiceSoapServiceLocator;

```

Run the program. It should successfully run and print the output of the book that got inserted into the “remote” server through a remote API. Confirm this by going to the portal’s database and digging into the “LMS_LMSBook” table and verify the new records has indeed got created.

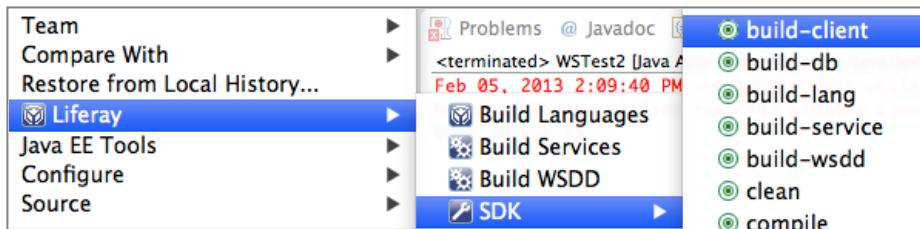
It’s really amazing to see how simple it is to expose and consume our API’s with the help of remote services. Here I would like to highlight couple of differences between this program (WSTest2) and the one we wrote earlier (WSTest1).

- Check the URL pattern. It is an un-authenticated URL where the credentials are not passed. Ideally you want your API’s to become public properties, so that any one and every one can start inserting books into our Library. There is a way to protect your Web Service and securing it, so that only applications / people with proper credentials can invoke it. How to do that? This is done through “Permission Checker”. We’ll cover this in our chapter on “Liferay Security and Permissions”. Until then, keep reading.
- “**WSTest1**” had just one “try-catch” block. This is not the right way of doing things. In “**WSTest2**” we have surrounded every line with “try-catch” that can potentially throw some exception. This practice greatly enhances the debugging and maintainability of the application. You will come to know what exactly had gone wrong if case of any exception or failure to complete an execution.

8.1.6 Generating “library-service-client.jar”

In situations where you don’t have a feature to automatically generate the stubs as you have in Eclipse just with the end-point URL, Liferay has a work-around. Run the “Ant” target “build-client” by right clicking on the “library-portlet” project and selecting the option as shown here. Once this is run, you will see the jar file inside “WEB-INF/client” of our plugin project. Copy this file and paste into the “lib” folder

of the “**ws-client**” project and subsequently add other dependency jars as mentioned in section 8.1.3.



All the changes so far have been checked in and the revision is 29. You can get the source code for the two sample client programs from <http://bit.ly/Xr7n0E> as they are not part of the core Library Portlet.

Code Changes @ <http://code.google.com/p/lr-book/source/detail?r=29>

Do it Yourself: Here is an interesting task for you. Write a client program to list all the available books in our library. I am sure this exercise is going to be a cakewalk for you as you have already got well accustomed with the subject.

8.2 JSON Web Service

In the last section, we have seen the SOAP / RPC based Web Service feature. There are no doubts; SOAP is an extremely powerful way of making the Web Services to work. But with all its uses, there are limitations too. Some of them are listed here. Getting into details is out of scope in this book.

1. It is a bit heavy as lot of marshaling and un-marshalling happens both in the client and the server end.
2. The client should always aware of the interfaces and they have to be made available to them.
3. SOAP is more “old school”, and tends to be the favored approach by Java developers

The idea behind JSON Web Services is to provide portal service methods as JSON API. This makes services methods easily accessible using HTTP request not only from portals javascript, but also from any JSON-speaking client out there. Just to give a very little background about JSON, it is a lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate. JSON is a text format that is completely language independent but uses conventions that are familiar to programmers. These properties make JSON an ideal data-interchange language. To know more, visit <http://www.json.org>.

8.2.1 JSON Services Exposed by Liferay

Liferay by default exposes all the methods that are defined inside the various “ServiceImpl” classes as JSON based Web Service that can accept a JSON request and return back a JSON response. With this feature, our API’s can invariably

accessed from any client. First we'll check the JSON Web Services that are exposed by Liferay by default. Hit this URL from the browser. It will list out all those JSON Web Services. <http://localhost:8080/api/jsonws>

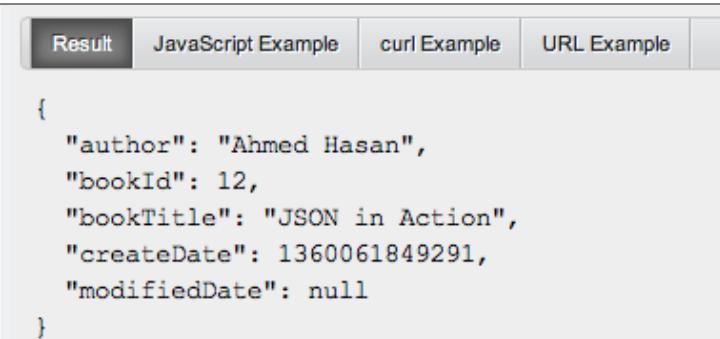
This will show all the services. If you click any of the service, it will show its Parameters, Return Type, Exception and a form to invoke if it is a POST kind of API. It also displays whether the HTTP request type of a particular API is POST or GET.

8.2.2 JSON Services Exposed by Our Plugin

Having seen this, now let's look into the JSON API's that are generated for our Library Portlet by virtue of having a method in the "ServiceImpl" class. As seen earlier, invoke the below URL in the browser.

<http://localhost:8080/library-portlet/api/jsonws>

Kudos! You see our "[insert-book](#)" API being exposed as a JSON Web Service. Now click on it. You will see the details of this service same as before. Things do not end here. The guys at Liferay are very generous to even create a form to test our web service without even writing one single line of code. See, how much Liferay has simplified your life. Now give some inputs for the "bookTitle" and "author" field and press "Invoke" and see what happens. It shows the result of our invocation in the first tab as shown here in this screen capture.



```
{  
    "author": "Ahmed Hasan",  
    "bookId": 12,  
    "bookTitle": "JSON in Action",  
    "createDate": 1360061849291,  
    "modifiedDate": null  
}
```

Let's first confirm that this new record actually got inserted into the "LMS_LMSBook" table as a result of this invocation. I am sure it is there, because only then it returns the object back as response that we are seeing above. The result what you see here is also in the JSON format. Now click on the next three tabs. They give examples of how the same JSON web service can be invoked through various diverse ways, from any kind of device that can comprehend JSON.

JavaScript Example: Code snippet showing how this web service can be invoked from JavaScript running inside a browser.

```
Liferay.Service(  
    '/library-portlet#lmsbook/insert-book',  
    {  
        bookTitle: 'JSON in Action',  
        author: 'Ahmed Hasan'  
    },  
    function(obj) {
```

```
        console.log(obj);
    }
);
```

Curl Example: An example showing how the API can be invoked through curl. curl is a command line tool usually used in Linux for transferring data with URL syntax

```
curl http://localhost:8080/library-
portlet/api/secure/jsonws/lmsbook/insert-book \
-u test@liferay.com:test \
-d bookTitle='JSON in Action' \
-d author='Ahmed Hasan'
```

URL Example: This example shows how to invoke the API directly through the browser URL or a RESTful service. This example forms the basis of the RESTful API that we are going to discuss subsequently in this chapter. In the example below the actual values are URL escaped.

```
http://localhost:8080/library-
portlet/api/secure/jsonws/lmsbook/insert-book/book-
title/JSON%20in%20Action/author/Ahmed%20Hasan
```

Now coming back to the question, why there are only two fields in this form instead of all the fields of the “LMSBook” entity? The answer is very simple. The form is generated based on the number of attributes in the method signature of the underlying service layer.

8.2.3 Consuming the JSON Service – Legacy Way

Invoking a JSON service within a JavaScript helps in building RIA’s (Rich Internet Applications) and SPA’s (Single Page Applications). If you’re really passionate about building applications of these types then you have to become well versed with JSON web service and their invocation. This section we’ll see how to consume a JSON Web Service in a Legacy fashion inside our JavaScript. First let’s create a story for implementing this exercise. We’ll see how to insert a book into our Library by invoking the JSON web service through an AJAX call directly from the JavaScript. This exercise has four steps.

Step 1: Lets inject the generated “service.js” into our Portlet by making the appropriate entry in “**liferay-portlet.xml**”. Remember this file has been created under “**docroot/js**” after defining a method inside “**LMSBookServiceImpl.java**” and running the Service Builder. This file is Liferay’s own way of marshelling the data and invoking the JSON web service API. The auto-generated contents of this file will look something like this.

```
Liferay.Service.register(
"Lifecycle.Service.LMS", "com.slayer.service", "library-portlet");

Liferay.Service.registerClass(
    Liferay.Service.LMS, "LMSBook",
{
    insertBook: true
```

```
    }  
);
```

Having seen this file and confirming that it does exist in its place, make the actual entry for this file as just mentioned, before “`<footer-portlet-javascript>`”.

```
<header-portlet-javascript>/js/service.js</header-portlet-javascript>
```

To confirm the injection of this JavaScript file has properly happened, just insert an alert statement “`alert('hi');`” as the file line of “**service.js**”, saving it and checking in browser by visiting “**my-library**” page.

Step 2: In this step, we’ll create our UI to invoke the JSON service from the front-end UI. We’ll create a new button which when “clicked” will do this job. Open your “**update.jsp**”. Yes, I can understand we are touching this file after a very long time . Insert this block of code immediately after line “`<aui:button type="submit" value="Save" />`” in the AUI form.

```
<% String functionName =  
    renderResponse.getNamespace() + "invokeJSONWS();"; %>  
  
<aui:button value="Save through JSON"  
    onClick="<%=" functionName %>" />
```

Infact I wanted to just have one line like the one given below, but unfortunately “`<aui:button>`” tag does not properly translate “`<portlet:namespace/>`”. For the same reason we have followed the above approach in one of the earlier examples also when we seen performing an action on multiple items of a list at one shot.

```
<aui:button value="Save thru JSON"  
    onClick="invokeJSONWS()" />
```

Define a new JavaScript function inside “`<aui:script>`” at the bottom of this JSP file with just an “alert” message as this is always the best way to start.

```
<portlet:namespace/>invokeJSONWS=function() {  
    alert('you're inside invokeJSONWS method');  
}
```

Save the changes and confirm you get the alert message when “Save thru JSON” button is clicked. Good, you have put the basic skleton in place.

Step 3: In this step we’ll write the actual code to invoke our JSON Web Service that inserts a book into our Library. Replace the line that makes the alert inside “`invokeJSONWS`” function with this block of code,

```
var frm = document.<portlet:namespace/>fm;  
  
// create the payload in JSON format  
var payload = {  
    bookTitle: frm.<portlet:namespace/>bookTitle.value,  
    author: frm.<portlet:namespace/>author.value  
};
```

```

// invoke the Web Service
Liferay.Service.LMS.LMSBook.insertBook(payload);

// clear values in the input fields
frm.<portlet:namespace/>bookTitle.value = '';
frm.<portlet:namespace/>author.value = '';

```

Save changes, go to browser and click “Save thru JSON” button after giving proper inputs for “bookTitle” and “author” fields. You will be amazed to see the magic happening. A new book gets inserted into the table without any noise or hype, without any page reload. See how beautiful it is. As I told earlier JSON Web Services are the main ingredients for RIA’s and SPA’s that are completed based on AJAX.

Tip: *The magic might have not worked if you’re not currently logged into the portal. Immediately login and try the magic again. It should work. In case, if you face any other problem making this work, just do a manual deployment of the Portlet thru Eclipse. Hopefully everything should work fine. I did the same thing (fresh deployment) when the data was not getting inserted for the first time when I invoked.*

Note: *The word “Payload” refer to the data that is being passed during a Web Service call. In air and spacecraft parlance it means “the carrying capacity of an aircraft or spacecraft”. In computing world it means “the cargo information within a data transmission”.*

Step 4: There is one more piece left in this puzzle. It is the “call back” that processes the response back from the server. Based on the callback, you can do whatever you want just by playing around with the JavaScript DOM object and manipulate any portion of the screen like showing or hiding a DIV, displaying a fading type popup, etc. In this step we’ll quickly introduce a “call back” that just alerts the id of the book that just got inserted to our Library. Update the one line invocation of JSON Web Service with this block.

```

Liferay.Service.LMS.LMSBook.insertBook(payload, function(obj) {
    alert('Id of the book just got created --> ' + obj.bookId);
});

```

Here “obj” is the return object of the original remote method written inside “**LMSBookServiceImpl**” class that is nothing but “**LMSBook**”. We can directly access the attributes of this object like `obj.bookId` and `obj.bookTitle`.

As a habit, whenever we do something in our Library Portlet, I immediately give a reference or similitude in the portal source. I am sure you acknowledge and appreciate this style of mine. Ok, now let me do this here as well. Open Eclipse and press **Ctrl+Space+R** to open “service.js” in the portal source. You should find this file inside “**portal-web/docroot/html/js/liferay**” of PORTAL_SRC. This file has close to thirteen hundred lines of code. Liferay has used these methods in various places. I am going to ask you to have a look at one such place.

Login to the portal as Omin Admin and go to Control Panel. From here, click “Users and Organizations”. Try adding an organization of type “Location”. In the form that opens, select a country in the “Country” drop-down, then all the Regions of that country gets listed out in the next drop-down. This happens by calling the appropriate JSON Web Service – “getRegions” under the service [Liferay.Service.Portal, “Region”]. The tables in the database are “country” and “region”.

Name (Required)	Test Location
Type	Location
Country	United States
Region	Texas

8.2.4 Consuming the JSON Service – Modern Way

From Liferay 6.1 onwards, Liferay has introduced a better way of invoking the JSON Web Service. We have already seen this in section 8.2.2 above. Now we are going to implement this and see.

Step 1: You no longer required injecting “service.js”. This approach is so independent that literally there is no new injection of any JavaScript except the default Liferay’s JavaScript that usually gets loaded at the time of portal loading in the browser. To remove the injection, remove the corresponding entry in “**liferay-portlet.xml**” for “**service.js**” and save the file. The modern way does not have any dependency on this file anymore.

Step 2: Replace the block of code corresponding to the original invocation of JSON Web Service “Liferay.Service.LMS.LMSBook.insertBook”, with this block, originally copied from the “Javascript Example” tab that appears after submitting the form using the web UI. After copying just replace the code with actual values for the inputs fields from “frm”.

```
Liferay.Service(
    '/library-portlet#lmsbook/insert-book',
    payload,
    function(obj) {
        console.log(obj);
    }
);
```

“function(obj)” is the callback function. Whatever you want to do once the response is got from the server; you can do here in this block. This is a much cleaner approach than the earlier one. Then what made us to even know the legacy way? Why this is important? Because, in majority of the portlets that Liferay ships with, this approach has been retained. May be in the future releases of Liferay it may get updated to the modern way.

Code Changes @ <http://code.google.com/p/lr-book/source/detail?r=31>

Do it Yourself: Define a new method “**getAllBooks**” in “**LMSBookServiceImpl**”. This method should return all the books in the library. In the “**view.jsp**” create a link that when clicked, should display the list of books in a tabular format by first

invoking the API through JSON Web Service and rendering the results list in the form of HTML table.

8.3 RESTful Web Services

There is yet another elegant way of communication between any two applications over the Internet that is extremely lightweight and completely de-coupled from one another. All it requires is one simple URL either HTTP or HTTPS for the communication to happen. As long as there is valid REST api any application or device can invoke that Web Service and the ONLY prerequisite is that application should comprehend HTTP (S) protocol.

One of REST's primary advantages is its ease of implementation on the client side of an application. In addition to this idea of ease of implementation, the results of a REST call are also easily human readable. Unlike SOAP (XML-RPC), REST does not have a lot of extra XML markup allowing any implementation of it to be comparatively lightweight. It only requires a HTTP stack to work, so there is an argument to be made that REST is more interoperable than SOAP; despite the fact that SOAP was designed with that exact thing in mind. With REST not only is the client and server implementation usually a more lightweight solution, but they also tend to use less bandwidth than their SOAP counterparts.

In this section we'll see how to make use of a REST API and we'll actually implement client program to programmatically invoke this through any client.

8.3.1 What's available by default?

The moment you make some API's remote with the help of Liferay's Service Builder and expose them as Web Services, REST API's are also exposed. We have seen the format of one of our API's in section - 8.2.2 JSON Services Exposed by Our Plugin. There we have seen an example as well as soon as we submitted the form in Web UI.

```
http://localhost:8080/library-
portlet/api/secure/jsonws/lmsbook/insert-book/book-
title/JSON%20in%20Action/author/Ahmed%20Hasan
```

All I am asking you to do is to just copy paste this URL into the browser and hit "enter" to see what happens. The browser will ask you to enter the credentials. Enter test@liferay.com and the password. As a result you will see a new record getting inserted into our Library. This is another magic. No single line of code, just one simple URL you're able to hit our server from anywhere in the world. This is the power of RESTful API's. You just have to secure them, so that that there is no unanimous access.

Now in the URL, you just remove the word "secure" and try hitting "enter". This time it will not ask you to enter the credentials. Another beauty in this URL is that the values for bookTitle and author are passed in the URL itself and they are properly escaped. You try hitting the same URL with different values and everytime a new record gets inserted into the table. The response of this request is in the JSON / TEXT format. Any application can understand this response object very easily.

```
{"author": "Ahmed Hasan", "bookId": 33, "bookTitle": "JSON in
Action", "createDate": 1360133886937, "modifiedDate": null}
```

8.3.2 Programmatically invoking a RESTful Service

In this section we are going to see how to invoke a RESTful service in a programmatic manner, so that any application can make use of it to make remote calls to our Liferay portal installation over HTTP protocol. Let's do this first in an un-secure way and then secure it with proper credentials.

Create a new java class inside "ws-client" project's "src" folder and name it as "**RESTTest**". Paste the following contents.

```
public static void main(String[] args) {
    String uri = "http://localhost:8080/library-portlet/" +
    "api/jsonws/lmsbook/insert-book";
    HttpClient client = new HttpClient();

    PostMethod method = new PostMethod(uri);

    method.addParameter("bookTitle", "REST in action");
    method.addParameter("author", "Ahmed Hasan");

    int returnCode = client.executeMethod(method);

    if (returnCode == HttpStatus.SC_OK) {
```

```

        System.out.println("Status Code is: " + returnCode);
    } else {
        System.out.println("There is a problem: " + returnCode);
    }
}

```

After copy-paste, you see problems with `HttpClient`, `PostMethod` and `HttpStatus`. As we are making use of an Apache library called “`HTTPClient`” the program expects this jar file to be in the classpath. Right click on “`ws-client`” project and add the jar “**commons-httpclient.jar**” into the classpath by going to “Java Build Path” inside “Properties”. After this change, you should be able to successfully import the required classes.

```

import org.apache.commons.httpclient.HttpClient;
import org.apache.commons.httpclient.HttpStatus;
import org.apache.commons.httpclient.methods.PostMethod;

```

But still the line containing “`client.executeMethod(method)`” is giving some problem. Click on the red icon on that line and surround that line with proper “try-catch” block like this.

```

int returnCode = 0;
try {
    returnCode = client.executeMethod(method);
} catch (HttpException e) {
    e.printStackTrace();
} catch (IOException e) {
    e.printStackTrace();
}

```

Try running the program with right click and selecting “**Run As → Java Application**”. There is still a problem that demands you to make one more jar available in the classpath. This time it is “**commons-codec.jar**”. Include this exactly the same way we did for the other one. After the change when you try running it for the second time, you should see the console output “`Status Code is: 200`” and a record got inserted to our Library.

8.3.4 Passing credentials to a RESTful API

In this quick sub-section we’ll see how to pass credentials as part of the RESTful API invocation. Ideally a RESTful API will be secured so that no one can access it without proper authentication. Add the word “secure” as part of the URI [`api/secure/jsonws`] in the program and try running it. You should get an output like “`There is a problem: 401`”. The 401 error is always something to do the authentication and authorization.

This problem calls for a proper authentication credentials to be passed while invoking the RESTful API programmatically. Insert this block immediate after instantiating the “`client`” object of type “`HttpClient`” and make necessary imports. `HttpClient` is an Apache project used to exactly simulate the browser requests going to the server. You can know more details from the official website, <http://hc.apache.org/httpclient-3.x/>.

```
// setting the credentials
String userName = "test@liferay.com";
String password = "test1";

Credentials credentials =
    new UsernamePasswordCredentials(userName,password);
client.getState().setCredentials(AuthScope.ANY, credentials);
client.getParams().setParameter("http.useragent", "Test Client");
client.getParams().setAuthenticationPreemptive(true);
```

Run the program now and it should output “Status Code is: 200”, confirming the happening of proper authentication of the credentials we are passing. In case, if this is still giving problem, confirm that we have reverted back the changes we did earlier to “**portal-setup-wizard.properties**” – the entry for “**company.security.auth.type**” has to be either commented out or removed and server restarted.

8.3.5 Processing the Response of RESTful API

In this section, I am going to show you how to process the output / response after calling a RESTful API. Insert the below block inside the “if” condition for “**HttpStatus.SC_OK**”. This will show some compilation issues forcing you to add another jar file to the project, “**json-java.jar**” and make necessary import for “**JSONObject**”.

```
try {
    JSONObject object =
        new JSONObject(method.getResponseBodyAsString());
    System.out.println("ID of the book added ==> " +
        object.getLong("bookId"));
} catch (JSONException e) {
    e.printStackTrace();
} catch (IOException e) {
    e.printStackTrace();
} finally {
    method.releaseConnection();
}
```

When you save the changes and run the program it should print the id of the book that just got created like “**ID of the book added ==> 38**”.

Congratulations! You have successfully incorporated programmatic invocation of a RESTful API that Liferay has exposed through service layer. For your convenience, I’ve checked in this new file “**RESTTest.java**” to SVN along with other two client programs that we have written so far.

8.4 The “Beyond” Part

You have to bear with me for bringing this topic in this chapter where the main theme is around remote services. No doubt this topic is an “*odd-man-out*” in the current context. May be as an excuse for taking this deviation I’ve diplomatically titled this section as “Beyond Part”. Being a Liferay geek, I never intended to be a writer of this sort who can insert any topic anywhere in his book. But definitely there is couple of compelling reasons for me to do so with your kind consent.

- I would like to remind you about the promise I made at the beginning of this chapter. The promise was that this is going to be last and final chapter on Service Layer and Service Builder. Out of fear that I may not find a suitable chapter / occasion where I can take up this very important topic, I wanted to cover in this chapter itself.
- In the next section, I am going to take up a topic that is very much in line with this chapter’s central theme. But for doing that topic, the one that we are going to discuss in this section is essential and will serve as the bridge.

Hope I’ve reasonably justified my point and got your buying before moving ahead. Now coming to the next important question. What is the “Beyond Part”? Let me once again bring in a case study. At mPower Global we have developed a complete Travel Suite with integration with GDS solutions like Galileo and Amadeus. This product is targeted for the big travel agencies in the whole of Middle East. No doubt it was a massive application with hundreds of use cases.

But do you know how many custom tables we have created for this application? You will not believe it. It was just fifteen! The next question is then where in the world did we dump the rest of the application data? We used most of the Liferay tables to persist them without the need for creating duplicate tables and the headache of managing them. Ok, I am not going to get into the details of how we did that. The exercise we are going to do now is good enough to illustrate this point. Let’s begin as usual and come back to discuss what we learnt.

8.4.1 Saving the Address information

We are going to improvise our Library Portlet to capture the “Address Information” and “Contact Details” of the author at the time of adding a book to the Library so that our users can get in touch with them (book authors) directly whenever they want. And we are not going to create custom tables to store this additional data; instead we are going to make use of some of the default tables of Liferay and access them with the help of some “*LocalServiceUtil” class API that were already made available by Liferay. They also also called as “Local Service” API’s. Let’s start with the capture of “Address Information” and then proceed to capture Phone numbers. Steps? There are three of them.

Step 1: Create a new file “**contact-info.jspf**” (JSP Fragment) inside “**html/library**” with the below contents. We are going to make use of some new AUI and Liferay-UI tags here.

```

<%@page import="com.liferay.portal.model.Country" %>
<%@page import="com.liferay.portal.service.CountryServiceUtil" %>

<br />
<liferay-ui:panel-container extended="true">
    <liferay-ui:panel title="Author Address" collapsible="true"
        defaultState="collapsed"
        helpMessage="Optionally enter Author's address details">

        <aui:input name="saveAddress" type="checkbox"
            label="Save Author Address" />

        <aui:select name="countryId" />

        <aui:input name="city" />
        <aui:input name="street1" />
        <aui:input name="street2" />
        <aui:input name="zip" />

        <aui:input name="mobile-phone" label="Mobile Phone" />
        <aui:input name="business" label="Business Phone" />
    </liferay-ui:panel>
</liferay-ui:panel-container>

```

Open “**update.jsp**” and update the existing form to capture the address information by including the JSPF file with usual include directive. This line has to be inserted just above the line for “Save” button. This is a good way to modularize and fragment bigger JSP files instead of keeping them as a big chunk running in to several hundred lines of code.

```
<%@include file="/html/library/contact-info.jspf" %>
```

Save the changes and check the Add Book form. It should have a new form now to add the address information, but the country list is empty, as we have not populated it with any options.

Step 2: Append a `<aui:script>` block script at the bottom of “**contact-info.jspf**” and insert the following code that gets executed when the page loads. This script makes a JSON Web Service (modern) call to get list of countries from the back-end through AJAX and dynamically populate country drop-down list with proper values.

```

AUI().ready(function(A){
    var frm = document.<portlet:namespace/>fm;
    var countries = frm.<portlet:namespace/>countryId;

    Liferay.Service(
        '/country/get-countries',
        { active: true },
        function(obj) {
            for (var key in obj) {
                if (obj.hasOwnProperty(key)) {
                    var name = obj[key].name;
                    var countryId = obj[key].countryId;
                    countries.options[key] =
                        new Option(name, countryId);
                }
            }
        }
    );
}

```

```

        }
    );
}

```

If you check the “Add Book” form now, you should see the country list populated.

Step 3: In this step we are going to write the actual logic in our portlet class “**LibraryPortlet.java**” to persist author’s address. Replace the original block below in “updateBook” with next block of code.

```

if (bookId > 0) {
    LMSBookLocalServiceUtil.modifyBook(bookId, bookTitle, author);
} else {
    LMSBookLocalServiceUtil.insertBook(bookTitle, author);
}

```

```

LMSBook lmsBook = (bookId > 0)?
    LMSBookLocalServiceUtil.modifyBook(bookId, bookTitle, author) :
    LMSBookLocalServiceUtil.insertBook(bookTitle, author);
saveAddress(actionRequest, lmsBook);

```

a) Create the private method “`saveAddress`” like this with two parameters. This method will extract the actual parameters from the “`PortletRequest`” and call another variant of “`saveAddress`” with a different signature altogether. You may even name it differently if you wish as long as it is a meaningful name.

```

private void saveAddress(PortletRequest request, LMSBook lmsBook) {
    boolean saveAddress = ParamUtil.getBoolean(
        request, "saveAddress", false);
    if (!saveAddress) return;

    long addressId = ParamUtil.getLong(request, "addressId", 0);
    long countryId = ParamUtil.getLong(request, "countryId");
    String city = ParamUtil.getString(request, "city");
    String street1 = ParamUtil.getString(request, "street1");
    String street2 = ParamUtil.getString(request, "street2");
    String zip = ParamUtil.getString(request, "zip");

    long userId = PortalUtil.getUserId(request);
    long companyId = PortalUtil.getCompanyId(request);

    saveAddress(addressId, countryId, city, street1,
               street2, zip, userId, companyId,
               lmsBook.getBookId(), LMSBook.class.getName());
}

```

b) This is the code for the other “`saveAddress`” method in the same Portlet class that takes simple (mostly primitive) values as parameters. This is how you have to make your business logic API’s completely agnostic of any classes from “`javax`” packages or classes of any other third party libraries. In this case, the second “`saveAddress`” need not have to know anything about “`PortletRequest`” or “`LMSBook`” objects.

```

private void saveAddress(long addressId, long countryId,
    String city, String street1, String street2,
    String zip, long userId, long companyId,
    long parentId, String parentClassName) {

```

```

// 1. creating the address object (fresh or old)
Address address = getAddress(addressId);

// 2. setting the UI fields
address.setCountryId(countryId);
address.setCity(city);
address.setStreet1(street1);
address.setStreet2(street2);
address.setZip(zip);

// 3. set audit fields
address.setCreateDate(new Date());
address.setUserId(userId);
address.setCompanyId(companyId);

// 4. Set address type and if primary.
address.setTypeId(getTypeId("address", "personal"));
address.setPrimary(true);

// 5. most importantly set the parent details.
address.setClassPK(parentId);
address.setClassName(parentClassName);

// 6. finally update the object (save address)
try {
    AddressLocalServiceUtil.updateAddress(address);
} catch (SystemException e) {
    e.printStackTrace();
}
}

```

c) Let's define our third private method "getAddress" as below in the same class.

```

private Address getAddress(long addressId) {
    Address address = null;
    if (addressId > 0) {
        try {
            address =
                AddressLocalServiceUtil.fetchAddress(addressId);
        } catch (SystemException e) {
            e.printStackTrace();
        }
    } else {
        try {
            addressId = CounterLocalServiceUtil.increment();
        } catch (SystemException e) {
            e.printStackTrace();
        }
        address =
            AddressLocalServiceUtil.createAddress(addressId);
    }
    return address;
}

```

d) Create our fourth private method "getTypeId" as below.

```

private int getTypeId(String entity, String type) {
    int typeId = 0;
}

```

```

List<ListType> listTypes = null;
try {
    listTypes = ListTypeServiceUtil.getListTypes(
        Contact.class.getName() + "." + entity);
} catch (SystemException e) {
    e.printStackTrace();
}

for (ListType listType: listTypes) {
    if (listType.getName().equalsIgnoreCase(type)) {
        typeId = listType.getListTypeId();
        break;
    }
}
return typeId;
}

```

Make all necessary imports.

```

import java.util.Date;
import com.liferay.counter.service.CounterLocalServiceUtil;
import com.liferay.portal.model.Address;
import com.liferay.portal.model.Contact;
import com.liferay.portal.model.ListType;
import com.liferay.portal.service.AddressLocalServiceUtil;
import com.liferay.portal.service.ListTypeServiceUtil;

```

Take a deep breath. We have done so much of work. Without wasting much time, let's insert a book with author's address. Don't forget to check the option "Save Author Address"; only then the data gets saved. Did u check the "**address**" table in the database? Yes, you should see a record got inserted successfully. Let's go back and see the important things out of this exercise.

8.4.2 Points to Ponder

There are some important points to ponder from the above exercise. I've subtly introduced many new things / concepts. Let's take up each of them and quickly discuss. Understanding each of these points is extremely important.

Point 1: First and foremost is the usage of "PortalUtil" utility class. This is again a powerful class that Liferay provides with whose help we can get many inside information about the portal itself. In the above exercise, we have obtained the userId and companyId of the user who is currently adding a book to the library. You can have a quick look at the various methods of this class from <http://bit.ly/V3fx0q>. We'll use some of these methods in the subsequent chapters of this book.

Point 2: In the first "saveAddress" method we have just done the extraction of form values with the help of "ParamUtil" class. See the first line of this method that tries to get the value for the checkbox. If the value is not there, it defaults to "**false**".

Point 3: We have written our actual business logic in a separate method so that there is a clear demarcation. The second "saveAddress" takes simple arguments. This method has six important steps. We'll discuss each of them in subsequent points of this bulleted list.

Point 4: By calling “`getAddress(addressId)`” we are getting an “address” object. The method returns a fresh object if `addressId` is “0” else it returns the actual object based on a proper Id. Get into this method and you have to see how we are getting the reference of an object if the value of `addressId` is “0”. Unlike “LMSBook” we are not using the constructor to instantiate the object. You know why? The implementation for the address model is NOT in the current context; instead it is in the portal “ROOT” context packaged inside “**portal-impl.jar**”. Unfortunately from the context our “**library-portlet**” is running we do not have access to the actual implementation class. Hence, we are using a work around to get a reference to the object. “`AddressLocalServiceUtil.createAddress(addressId)`” actually creates an empty object and inserts a record into the underlying table.

Point 5: Next step “`2. setting the UI fields`” we are actually setting all the fields that user entered through the UI. They all now come as inputs to the second “`saveAddress`” method. These fields are set to the appropriate attributes of the “`address`” object.

Point 6: In step 3 we are setting all the audit fields required for this “`address`” object like `createDate`, `userId` and `companyId`. But there is one problem here. “`createDate`” will be set both during insertion and updation which is a logical error. In one of the tasks that I am going to give you shortly I will ask you to rectify this.

Point 7: In step 4 we are setting the type of address for this record and also marking this records as the “primary” address for this author. An entity like user, organization, and author can have multiple addresses, which is a kind of one-to-many relationships. Out of the multiple records, one has to be designated as the primary record for the parent entity. The address type is obtained by a call to “`getTypeId`” method that takes two arguments – entity and type. Based on these values the actual implementation of this method looks up the “`ListType`” table and fetches the correct type that we want to have for this new record. In one of the tasks that I am going to give you shortly, you have to use this same method to set the “`phone`” type while inserting / updating a phone entity using “**PhoneLocalServiceUtil**”.

Point 8: The next step (5) is the most important step. Without this step the newly inserted address recorded would have been orphaned. This is one of the beautiful ways Liferay has designed its database. May be for this reason there is no foreign key relationship between the default tables of Liferay. One address record can belong to any parent – a user, an organization, an author or anything else that is going to come in the future. Instead of re-inventing the wheel everything we need to store address information, we can make use of the existing table of Liferay for this purpose. You can take this step as the reason for brining up this whole section.

Now quickly login in the portal and add an address for yourself by going to the “My Account” section and for the “Org1” organization we have created earlier. See what is happening to the address table. Now you should see three records, one for the user (yourself), the organization (org1) and the author (of a library book). `setClassPK(parentId)` sets the “`classPK`” field and “`classNameId`” is set by the other setter method. If you see the actual record, “`classNameId`” points to a record in

another table “classname_” which has all the entities defined with and a unique id for every entity. The “classPK” refers to the primary key of the actual entity (e.g. User_).

Point 9: The final sixth step actually updates the object into the database with all required fields of the object populated with proper values.

Code Changes @ <http://code.google.com/p/lr-book/source/detail?r=34>

8.4.3 ListType and drop-downs

I think I'll not be doing justice if I don't quickly tell you about the “ListType” Service of Liferay. This is another nice API Liferay provides that helps in rendering various types of list based form elements in the UI. A list element can be a drop-down box (*single select or multiple select*) or a list of radio buttons or a list of check boxes or just a list of labels displayed on the user interface forms. Through the raw API, you can pull the values required from the “listtype” table and render them in whatever way you want. Here is a simple example of how you will render the list of contact phone numbers in the UI using an appropriate AlloyUI tag.

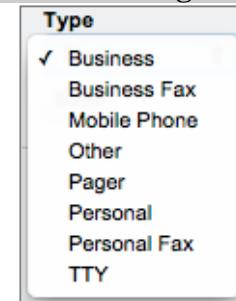
```
<aui:select label="type" name="phoneTypeId"
    listType="<%= Contact.class.getName() +
    ListTypeConstants.PHONE %>" />
```

This tag not only pulls the data from the “listtype” table but also takes care of translating the list items into other languages based on the keys provided in “**Language.properties**”. You can use this mechanism to even store values for other lists that you may use in your custom applications. The “`aui:select`” tag has got lot of other useful attributes. You should check all of them.

Value in the “listtype” table.

listTypeId	name	type
11006	business	com.liferay.portal.model.Contact.phone
11007	business-fax	com.liferay.portal.model.Contact.phone
11008	mobile-phone	com.liferay.portal.model.Contact.phone
11009	other	com.liferay.portal.model.Contact.phone
11010	pager	com.liferay.portal.model.Contact.phone
11011	personal	com.liferay.portal.model.Contact.phone
11012	personal-fax	com.liferay.portal.model.Contact.phone
11013	tty	com.liferay.portal.model.Contact.phone

UI rendering



8.4.4 Here are few challenges for you!

I am throwing four challenges for you to complete in this exercise before we move forward with the next section.

Challenge 1: You knew very well writing the business logic in the Portlet class is not a good idea. Move the methods “`saveAddress`” (not the first one), “`getTypeId`” and “`getAddress`” to “`LMSBookLocalServiceImpl`”. Run Service Builder and make change in the first “`saveAddress`” for things to work as usual.

Challenge 2: Inside “**LMSBookLocalServiceImpl**” class, instead of using “AddressLocalServiceUtil” I want you to make use of the low level persistence class “addressPersistence” or the corresponding “addressLocalService” to operate on my entity. How do you inject these objects into “**LMSBookLocalServiceImpl**” by making some changes in “**service.xml**”? (*We have already seen this before*).

Challenge 3: Make changes to “**contact-info.jspf**” and the API’s so that it can handle both addition and updation of an address entry. You need to take care of even setting “createDate” and “modifiedDate” fields appropriately based on the type of operation.

Challenge 4: Coming to the last challenge, in the UI you already have “Business Phone” and “Mobile Phone”. You have to update them as well into the “phone” table of your database by making use of the appropriate API’s. Run this query and check how many phone type are there.

```
SELECT * FROM lportal.listtype where type_ like '%Contact.phone' ;
```

Once you successfully complete these challenges and properly verify your changes, we can move on to the next important section that will be an addon to the current one.

8.5 Calling Portal's JSON Web Service

In section “8.2.3 Consuming the JSON Service – Legacy Way” we have seen how to consume a custom JSON Web Service once it is made available for invocation. In the same section we have also seen one example from the portal where the list of regions get populated based on the country the user selects. We are going to replicate the same feature in our form where we are adding / updating the authors address in our Library Portlet. I am going to do this using the legacy way and leave it to you to modify the code in accordance with the modern way. Let's start.

Step 1: Add more code to “**contact-info.jspf**” for the additional drop down to show all regions of a country. Replace line “`<aui:select name="countryId" />`” with,

```
<aui:select name="countryId"
            onChange="javascript:listRegions(this);"/>
<aui:select name="regionId"/>
```

We have just added one more “`select`” field and inserted `onChange` event for the existing one.

Step 2: Also add a new JavaScript method before closing “`</aui:script>`” tag.

```
function listRegions(country) {
    var frm = document.<portlet:namespace/>fm;
    var regions = frm.<portlet:namespace/>regionId;

    var _countryId = country.value;
    regions.disabled = (_countryId <= 0);

    if (_countryId <= 0) {
        regions.selectedIndex = 0;
    } else {
        // calling regions of this country
        var payload = {
            countryId: _countryId,
            active: true
        };

        Liferay.Service.Portal.Region.getRegions(payload,
            function(data) {
                regions.options.length = data.length;
                for (var i=0; i<(data.length-1); i++) {
                    var rgn = data[i];
                    regions.options[i] =
                        new Option(rgn.name, rgn.regionId);
                }
            });
    }
}
```

Save changes and check if it is working. It should not, as we have not injected the “**service.js**” of portal. This is required for the proper invocation of JSON Web Service. How to do it? The next step has the answer.

Step 3: Inject the “**service.js**” through “**liferay-portlet.xml**”. Open this XML file and insert this entry. We have used “`<header-portal-javascript>`” as the file is in the portal level and not within the folders of our Library Portlet.

```
<header-portal-javascript>
    /html/js/liferay/service.js
</header-portal-javascript>
```

Save the changes and check. Did it work? If you’re currently logged in then it should work. If not, then putting the line “`alert(data.toSource());`” inside the inner function will throw an exception in Firefox which says, “`((exception:"Please sign in to invoke this method"))`”. This means this JSON Service requires some authentication credentials to be passed at the time invocation. We’ll not get into the details now. I leave this to you to figure out by yourself.

Code Changes @ <http://code.google.com/p/lr-book/source/detail?r=35>

Now I am going to give two challenges for you.

Challenge 1: Change the legacy way of invocation with the modern way like this. This should take care of two things.

- Invalidate the need for putting an extra entry in “**liferay-portlet.xml**” for portal’s “**service.js**”
- Fix the sing in issue. This should work without user logged in to the portal.

```
Liferay.Service(
    '/region/get-regions',
    payload,
    function(data) {
        regions.options.length = data.length;
        for (var i=0; i<(data.length-1); i++) {
            regions.options[i] =
                new Option(data[i].name, data[i].regionId);
        }
    }
);
```

Challenge 2: Modify “**LibraryPortlet.java**” and corresponding API signature in “**LMSBookLocalServiceImpl**” to incorporate “`regionId`” and set it in “`address`” object before persisting to database. By the way, once you complete this task, just check for which countries, you get the list of regions. You will see only for few countries the Liferay’s region table has values. What about the rest? See next.

Here is a Gift for You!!

As a reward for successfully completing the above two challenges, I am going to give you a gift now. You must have observed not all countries have their regions in “region” table. I am going to provide this for you so that you can use this in your real time projects. This list is based on ISO standards. All you have to do is to download and run this sql script against your database. This is a MySQL dump. The location of the script is, <http://hoperay.googlecode.com/svn/trunk/plugins-sdk/sql/regions.sql>.

After importing this dump into the database, you may not see the changes reflecting in the Portlet immediately, unless and until you do one of the below as the earlier data would have been cached in the server. This is a good use case to understand caching.

1. Login to the portal as Omni Admin. Go to **Control Panel → Server → Server Administration**. Click “**Execute**” against “**Clear the database cache**” inside “**Resources**” tab.
2. Restart the tomcat server.

8.6 Service Context and its significance

Hmmm, I am sure you would have guessed what I am going to ask you for. “Excuse me once again!” This last chapter on Service Layer will not be complete until we discuss about this powerful framework of Liferay. You must have heard of various contexts through out your experience working with Java EE – Servlet Context, Spring Context, Portlet Context, etc. But what is Service Context? Sounds new, isn’t it? Don’t worry we are going to see it in detail in the last section of this great chapter. Liferay’s official documentation gives the following definition of a Service Context.

The ServiceContext class is a parameter class to be used in passing contextual information for a service. By using a parameter class, it is possible to consolidate many different methods with different sets of optional parameters into a single, easier to use method. The class also aggregates information necessary for transversal features such as permissioning, tagging, categorization, etc.

Technically Service Context is like any other class of Liferay source code under the package “com.liferay.portal.service” of “**portal-service**” folder. View Javadocs for this class at <http://bit.ly/VIo8D1>. Let’s get into the details. Not only here. In the whole of the book we are going to see many applications of this object. You will be amazed to see the benefits of this object. Not many have really used it to its full potential. More than just a class, it is a new pattern in itself. By using this pattern it is possible to consolidate many different methods with different sets of optional parameters into a single, easier to use method. It also aggregates information necessary for transversal features such as permissioning, tagging, categorization, etc. The next sub-section we are going to analyse its default set of methods and fields.

8.6.1 Service Context Fields

The ServiceContext class has many fields. We can group them under the following nine groups.

Actions <ul style="list-style-type: none">▪ _command▪ _workflowAction	Attributes <ul style="list-style-type: none">▪ _attributes▪ _expandoBridgeAttributes	Language <ul style="list-style-type: none">▪ languageId
Classification <ul style="list-style-type: none">▪ _assetCategoryIds▪ _assetTagNames	Miscellaneous <ul style="list-style-type: none">▪ _headers▪ _signedIn	IDs & Scope <ul style="list-style-type: none">▪ _companyId▪ _portletPreferencesIds▪ _plid & _uuid▪ _scopeGroupId▪ _userId
Permissions <ul style="list-style-type: none">◦ _addGroupPermissions◦ _addGuestPermissions◦ _deriveDefaultPermissions◦ _groupPermissions◦ _guestPermissions	Resources <ul style="list-style-type: none">◦ _assetEntryVisible◦ _assetLinkEntryIds◦ _createDate◦ _indexingEnabled◦ _modifiedDate	URL's and paths <ul style="list-style-type: none">◦ _currentURL◦ _layoutFullURL◦ _layoutURL◦ _pathMain◦ _portalURL◦ _remoteAddr◦ _remoteHost◦ _userDisplayURL

Are you wondering how these fields are set and who sets them? The next sub-section has the answer.

8.6.2 Creating and Populating a Service Context

The main purpose of Service Context is to share data between our front-end class and the back-end API. There are three different ways a ServiceContext object can be made available inside a Portlet class or a controller class.

- **Pure Instantiation:** This kind of instantiation is mainly to have a common object between the Portlet class and the “*ServiceImpl” API.

```
ServiceContext serviceContext = new ServiceContext();
```

- **Getting from Factory:** In this method we can get an instance that has already been instantiated and available inside “ServiceContextFactory” by one of the available `getInstance` methods as shown here (*there are totally four of them*).

```
try {
    ServiceContext serviceContext =
        ServiceContextFactory.getInstance(request);
} catch (PortalException e) {
    e.printStackTrace();
} catch (SystemException e) {
    e.printStackTrace();
}
```

8.6.3 Accessing Service Context data

In this sub-section I would like you to have a look into the code of “**BlogsEntryLocalServiceImpl.java**” from the Liferay portal source and understand how the data in the “serviceContext” object is being used for various purpose in its “addEntry” method (API).

```
public BlogsEntry addEntry(
    long userId, String title, String description, String content,
    int displayDateMonth, int displayDateDay, int displayDateYear,
    int displayDateHour, int displayDateMinute,
    boolean allowPingbacks, boolean allowTrackbacks,
    String[] trackbacks, boolean smallImage, String smallImageURL,
    String smallImageFileName, InputStream smallImageInputStream,
    ServiceContext serviceContext)
```

You will soon appreciate that the data retrieved from the service context has been used for various purpose – setting and getting the scopeGroupId, Message Board, Checking permissions, updating the workflow, updating the asset information, etc.

8.6.4 A Simple usage of Service Context – UUID

We’ll conclude this section with a simple illustration to understand the usage of Service Context object inside of the Portlet class. In this process we’ll also come to know another powerful feature that Liferay has – UUID. UUID stands for “Universally Unique Identifier”. If you quickly have a look into many of the entities (tables) of Liferay they have a column called “uuid_” which uniquely identifies that records “Universally” [<http://bit.ly/14Xixj1>]. I am not sure whether it is really unique globally (*if we go with the literal meaning of this word*) or unique within the given portal installation. May be you can figure that out once you master all the topics in this book. There are around 50 tables that have this column “uuid_” in our database. If you observe, most of them are main entities of the portal like “user_”, “blogsentry”, “calevent”, etc.

In the rest of this exercise, we’ll see how to enable “uuid” for our “LMSBook” entry and set it programmatically before adding a book into the Library.

Step 1: Update “**service.xml**” adding a new attribute `uuid="true"` for “LMSBook” entity and run the Service Builder for the changes to take effect. This is what the DTD file says about “uuid”.

If the `uuid` value is `true`, then the service will generate a `UUID` column for the service. This column will automatically be populated with a `UUID`. Developers will also be able to find and remove based on that `UUID`. The default value is `false`.

The moment you do this you should see “`setUuid`” method in the “`lmsBook`” object. Since our table is already there, we have to now manually add this column by running this command against the database “**Iportal**”.

```
ALTER TABLE lms_lmsbook ADD COLUMN uuid_ VARCHAR(75);
```

Step 2: Now we'll make few changes to our "insertBook" method of "LMSBookLocalServiceImpl".

- a) Add the third parameter to this method, "ServiceContext serviceContext", so that the method signature changes.
- b) Insert line "lmsBook.setUuid(serviceContext.getUuid());" in appropriate place of the method.
- c) The remote method in "LMSBookServiceImpl" has a reference to this person. Instantiate and pass a fresh Service Context object from there as there is no other way of getting an object from inside this class. Make necessary imports in both Impl classes and run the Service Builder.
- d) While making a call to this API from "LibraryPortlet.java" pass the addional parameter "serviceContext" which is obtained like this,

```
ServiceContext serviceContext = null;  
try {  
    serviceContext =  
        ServiceContextFactory.getInstance(actionRequest);  
} catch (PortalException e) {  
    e.printStackTrace();  
} catch (SystemException e) {  
    e.printStackTrace();  
}
```

Save all your changes and insert a book into our Library. Confirm the new record is created with the proper value for "uuid_" field. Try doing this by running either "WSTest2" or "RESTTest" programs to confirm even our remote API is working perfectly fine and setting a value for this field. I am sure you will get errors while running "WSTest2". Rectify the problem by re-generating the stubs using the WSDL URL and running the program again. This is one limitation of SOAP. Whenver the model or the signature of the API change we have to generate the client stubs. But it is not the case with REST. It worked perfectly fine as before.

Code Changes @ <http://code.google.com/p/lr-book/source/detail?r=37>

Congratulations! You have completed reading one of the finest chapters of this book and executing all the exercises.

Summary

I am sure you enjoyed browsing through this chapter and working out all the examples and exercise here. This is one of the most important chapters of this book where we have discussed the various kinds of API's that are provided by Liferay out-of-the-box. We started with the Remote service API's, then moved on to JSON Web Service API's, then to RESTful Web Service API.

After this I changed the gear to talk about the Local API's that are provided by Liferay, how to make use of Liferay's existing tables to persist the data of our custom applications. We learnt a good bit of AJAX and JavaScript invocation that will help us to build RIA's and SPA. Finally we concluded this chapter with a good discussion on Service Context which serves as the bridge between our front-end classes and back-end API's. Lastly we did one interesting exercise to make use of the Service Context to set UUID for our library book in order to make it unique in global sense during data interchange between diverse systems.

9. More Features and API's

This Chapter Covers

- Portlet Filters
 - Implementing Friendly URL's
 - Encrypting & Decrypting Portlet Data
 - Enabling Logger at Portlet Level
 - Portlet Internationalization (I18N)
 - Portlet and WCM Marriage
 - Making Portlet appear in Control Panel
 - Running Backend Jobs through Quartz
 - Firing Emails from Portlet
 - Getting Direct Access to Database
-

We'll try to keep this chapter as light as possible so that we can take a good break while going through its contents and also to get ourselves mentally prepared for covering more advanced and interesting topics in the subsequent chapters. The last two chapters has been quite intensive trying to introduce as many concepts as possible in order to put a very strong foundation on our Liferay learning process. In this chapter we are going to merge back with the topics that we left at Chapter [6. Improving the Book List](#).

We'll introduce many new concepts / techniques around the Portlet functionality itself. Some of them are part of the original JSR-286 (Portlet 2.0) specification and many of them are Liferay extensions (add ons). While going through these topics, it is upto you to indentiy the difference, though I may give some hints. The topics that we are going to cover may not be in any specific order.

9.1 Portlet Filters

As an honor and respect to the original JSR-286 (Portlet 2.0) specification let's begin this chapter with a topic / feature that is very much part of the specification itself. You must have heard of Servlet Filters if you're a Java EE developer. But what are Portlet Filters? They are not very different from those of the Servlets. While Servlet Filter intercepts an `HttpServletRequest` and `HttpServletResponse`, a Portlet Filter intercepts and processes a `PortletRequest` and `PortletResponse`. The only difference is that a servlet has only one request handling method “`service()`” and therefore there is only one type of the servlet filter. A portlet on the other hand has four types of request handling methods and hence has four different types of portlet filters.

The portlet API defines the following interfaces for creating portlet filters:

1. `javax.portlet.filter.ActionFilter` - For `processAction` method
2. `javax.portlet.filter.EventFilter` - For `processEvent` method
3. `javax.portlet.filter.RenderFilter` - For `render` method
4. `javax.portlet.filter.ResourceFilter` - For `serveResource` method

Each filter interface extends `javax.portlet.filter.PortletFilter`, a common base interface with two methods: `init(javax.portlet.filter.FilterConfig filterConfig)` and `destroy()`. The `init()` method makes sure that every filter has access to a “`FilterConfig`” object from which it can obtain its initialization parameters, a reference to the “`PortletContext`” which it can use, for example, to load resources needed for filtering tasks. The `destroy()` method signifies the end of service of the filter. These two methods of a portlet filter are called only once during their lifetime.

A single filter class can provide filter functionality for more than one life cycle method. Also, a single filter can provide filter functionality for more than one portlet. Multiple filters can be associated with one life cycle method of a portlet. The `doFilter()` method of a portlet filter might create customized request and response objects by using the `*RequestWrapper` and `*ResponseWrapper` classes and by passing these wrappers to the `doFilter()` method of the `FilterChain` object.

Enough of theory. Let's get to business. We'll write a portlet filter following these two steps. As usual the first step is write the filter itself and second is to properly configure it.

Step 1: Create a filter class as below inside “`src/com/library/filter`”. You name it as “`HitCountFilter`” implementing “`javax.portlet.filter.RenderFilter`”.

```
public class HitCountFilter implements RenderFilter {  
  
    private static int count = 0;  
    public void init(FilterConfig filterConfig)  
        throws PortletException {  
    }  
  
    public void doFilter(RenderRequest renderRequest,  
        RenderResponse renderResponse,
```

```

        FilterChain filterChain)
        throws IOException, PortletException {
    System.out.println("Hit count is..." + (++count));
    filterChain.doFilter(renderRequest, renderResponse);
}

public void destroy() {
}
}

```

Keep “init” and “destroy” methods empty. Make the necessary imports.

```

import java.io.IOException;

import javax.portlet.PortletException;
import javax.portlet.RenderRequest;
import javax.portlet.RenderResponse;
import javax.portlet.filter.FilterChain;
import javax.portlet.filter.FilterConfig;
import javax.portlet.filter.RenderFilter;

```

Step 2: Open our Library Portlet’s “**portlet.xml**” file and define a filter with the help of the appropriate tags. Append the new block after the closing “`</portlet>`” tag.

```

<filter>
    <filter-name>HitCountFilter</filter-name>
    <filter-class>com.library.filter.HitCountFilter</filter-class>
    <lifecycle>RENDER_PHASE</lifecycle>
</filter>
<filter-mapping>
    <filter-name>HitCountFilter</filter-name>
    <portlet-name>library</portlet-name>
</filter-mapping>

```

As you see above. One filter can be mapped to any number of portlets with the help of “`<filter-mapping>`” tag. Save the changes and try hitting your Portlet. Every time you refresh the page, the counter has to increase. This means our filter is working properly. This is a very basic example. On top of this you can write any kind of code that you want to write as per your application requirement.

Code Changes @ <http://code.google.com/p/lr-book/source/detail?r=38>

Do it Yourself: Write a filter implementing `javax.portlet.RenderFilter`. Make appropriate entries in the “**portlet.xml**” to make the new Portlet work and get invoked as and when an action is invoked on the Portlet.

9.2 Implementing Friendly URL's

Imagine a situation where you would like the details of any book in our library accessible through a simple, human-readable, meaningful URL that can be shared through various social networking sites like facebook, twitter, etc. and can be bookmarked with sites like del.icio.us and reddit.com. I am sure; you will definitely hate sharing an URL that is currently of the below format. Even those social networking sites will start blocking your site. Even, some how, if we are able to share one such link, any one who sees this kind of link will surely be surprised.

```
http://localhost:8080/web/guest/my-
library?p_p_id=library_WAR_libraryportlet&p_p_lifecycle=0&p_p_state=normal&p_p_mod
e=view&p_p_col_id=column-
2&p_p_col_count=1&library_WAR_libraryportlet_jspPage=%2Fhtml%2Flibrary%2Fdetail.
jsp&library_WAR_libraryportlet_backURL=%2Fweb%2Fguest%2Fmy-
library%3Fp_p_id%3Dlibrary_WAR_libraryportlet%26p_p_lifecycle%3D0%26p_p_state%3
Dnormal%26p_p_mode%3Dview%26p_p_col_id%3Dcolumn-
2%26p_p_col_count%3D1%26library_WAR_libraryportlet_jspPage%3D%252Fhtml%252F
library%252Flist.jsp&library_WAR_libraryportlet_bookId=32
```

The above URL is also difficult for search engines to crawl and index your website. Then what is the solution? Don't worry. As usual Liferay has an answer. Before getting into the nettigritties of implementing it, let's visit liferay.com and see one example of this to get a practical understanding. Navigate to Community → Blogs → Click on the first blog entry from "Recent Bloggers" on the left hand side of the page layout. Observe the browser's location bar and you will see an URL like this.

```
http://www.liferay.com/web/olaf.kock/blog/-/blogs/displaying-portlet-monitoring-
information
```

You must appreciate there is a huge difference between the above two URL's, the first one shows the details of a book in our Library and the other shows a blog entry. The later one is very small, human-readable and bookmarkable when compared to the former. In the rest of this section we are going to see how to make the book details URL of this nature too. This involves four steps – definition, configuration and improvements.

Step 1: Create a new file "routes.xml" inside "**WEB-INF/src/META-INF**". This file will contain the definition for various friendly URLs of our Portlet. The contents for our simple entry will look like this.

```
<?xml version="1.0"?>
<!DOCTYPE routes PUBLIC
"-//Liferay//DTD Friendly URL Routes 6.1.0//EN"
"http://www.liferay.com/dtd/liferay-friendly-url-routes_6_1_0.dtd">
<routes>
    <route>
        <pattern>/detail</pattern>
        <implicit-parameter name="jspPage">
            /html/library/detail.jsp
        </implicit-parameter>
    </route>
</routes>
```

```
</route>
</routes>
```

Each “route” will have details of one particular friendly URL that maps a “pattern” with the actual URL. While specifying a “route” you can mention different types of parameters depending on how we want to customize the URL and make it look like. The full details you can get from the corresponding DTD file inside “**definitions**” of portal source. We’ll take a quick look here.

Parameter	What does it specify?
Ignored	A parameter that should be ignored and not included in generated URLs. Ignored parameters do not affect URL recognition.
Implicit	A parameter that is not present in the route pattern.
Overridden	A parameter that should be set to a certain value when a URL is recognized. This override value will be set regardless of any preexisting value, including one from a implicit-parameter or one extracted from the URL.
Generated	The pattern of a parameter that will be generated from other parameters when a URL is recognized. When a URL is built, these virtual parameters will be parsed from the generated parameter and made available in the route pattern for constructing the URL.

Step 2: This step is about configuring our Portlet to make use of the “**routes.xml**” we just defined in the previous step. Open your “liferay-portlet.xml” and insert the below lines in the appropriate location.

```
<friendly-url-mapper-class>
    com.liferay.portal.kernel.portlet.DefaultFriendlyURLMapper
</friendly-url-mapper-class>
<friendly-url-mapping>library</friendly-url-mapping>
<friendly-url-routes>META-INF/routes.xml</friendly-url-routes>
```

Save the changes and check the functionality by clicking on the book details of any book from the list of books. I think the changes are not reflecting. There is one problem still that could be a bug in Liferay. In the “**routes.xml**” put any parameter entry as one single line as shown below, instead of breaking into multiple lines.

```
<implicit-parameter name="jspPage">/html/library/detail.jsp
</implicit-parameter>
```

After saving the changes, you will see the URL to be of this format. There is an improvement but still not convincing.

```
http://localhost:8080/web/guest/my-library/
/library/detail?_library_WAR_libraryportlet_backURL=%2Fweb%2Fguest%2Fmy-
library%3Fp_p_id%3Dlibrary_WAR_libraryportlet%26p_p_lifecycle%3D0%26p_p_
state%3Dnormal%26p_p_mode%3Dview%26p_p_col_id%3Dcolumn-
2%26p_p_col_count%3D1%26_library_WAR_libraryportlet_jspPage%3D%252Fhtm
1%252Flibrary%252Flist.jsp&_library_WAR_libraryportlet_bookId=32
```

In the new URL, you no longer see the usual parameters that are originally part of any Portlet URL. They are *p_p_id*, *p_p_col_id*, *p_p_col_pos*, *p_p_col_count*,

p_p_lifecycle, *p_p_state* and *p_p_mode*. They all have become implicit by default after implementing the friendly URL.

Step 3: We'll do some improvement to the above URL, as it is still not “friendly”. In this step we are going to “ignore” the backURL parameter, as it is not required for anyone who is directly viewing the book details. Introduce a new tag in “**routes.xml**”, `<ignored-parameter name="backURL" />`. Save and check the changes now. The link would have got much simplified like this.

```
http://localhost:8080/web/guest/my-library/-/library/detail?_library_WAR_libraryportlet_bookId=32
```

Why not replace “`_library_WAR_libraryportlet_bookId`” with just the bookId? This is a good scope for further improvement. Let's do this in our next step.

Step 4: Change the pattern format to “`<pattern>/detail/{bookId}</pattern>`” in “**routes.xml**”. Now check the Portlet. You will be amazed to see the new URL. Now compare the original URL with this one. What a massive change?

```
http://localhost:8080/web/guest/my-library/-/library/detail/32
```

Congratulations! You have successfully implemented friendly URL for just one feature (book details) of the Portlet. I am going to give you two challenges now.

Challenge 1: Create a friendly URL route for “Add / Edit” URL.

Challenge 2: Implement your own class for Friendly URL mapping that will extend `com.liferay.portal.kernel.portlet.DefaultFriendlyURLMapper`. What are the API's of this class?

For a more detailed understanding of friendly URLs and its advanced options you can refer to the Liferay's Wiki page @ <http://bit.ly/14Prk5s>. Friendly URLs are a great way of making your site social and SEO friendly. You should have a look into “**routes.xml**” files for various portlets that ship with Liferay. Press **Ctrl+Shift+R** to filter all files with the pattern “***routes.xml**”. There are more than a dozen of them.

```
Code Changes @ http://code.google.com/p/lr-book/source/detail?r=39
```

9.2.1 “**portlet.xml**” verses “**liferay-portlet.xml**”

In the previous section (Portlet Filters), we have made the new entries in “**portlet.xml**” and in this section we have made new entries to “**liferay-portlet.xml**”. Any idea why? The answer is quite simple. Any feature that is part of the original JSR-286 (Portlet 2.0) specification it goes into “**portlet.xml**”. Any additional feature that Liferay has introduced finds its place inside “**liferay-portlet.xml**”. Both of them are deployment descriptors for any given Portlet.

9.3 Encrypting & Decrypting Portlet Data

There will be situations where you have to deal with some of your application data that are very sensitive in nature. You have to handle them and store them differently than the rest of the application data. For example, in our Library Management System, we are capturing the Credit Card information of every member so that we can charge a monthly recurring subscription fee for membership. Immediately after capturing the data you have to store them securely in an encrypted format so that no one will ever be able to decipher the actual data in its original format. Let's see how to encrypt and decrypt data with the help of the API's that Liferay provides.

Honestly I don't want to build a new UI for this purpose for capturing member's credit card information. Let's try this out with one of our existing fields and revert back our changes once we understand the subject. I am taking the "author" field to get saved in the encrypted format and retrieved accordingly. Let's carry out both the steps – one to encrypt and the other to decrypt.

9.3.1 Encrypting the Data

Step 1: Let's write a general-purpose utility class that will have all commonly used API's. Name it as "**LMSUtil.java**" created inside "**src/com/util**" package with one **static** method defined.

```
public static String encrypt(String data, long companyId) {  
  
    String encryptedData = null;  
  
    Company company = null;  
    try {  
        company = CompanyLocalServiceUtil.fetchCompany(companyId);  
    } catch (SystemException e) {  
        e.printStackTrace();  
    }  
  
    if (company != null) {  
        Key keyObj = company.getKeyObj();  
        try {  
            encryptedData = Encryptor.encrypt(keyObj, data);  
        } catch (EncryptorException e) {  
            e.printStackTrace();  
        }  
    }  
  
    return encryptedData;  
}
```

Make all the necessary imports in this file.

```
import java.security.Key;  
import com.liferay.portal.kernel.exception.SystemException;  
import com.liferay.portal.model.Company;  
import com.liferay.portal.service.CompanyLocalServiceUtil;  
import com.liferay.util.Encryptor;  
import com.liferay.util.EncryptorException;
```

Step 2: Update the line that sets the author field in “insertBook” method of “**LMSBookLocalServiceImpl**” as below. Assume that we have already changed the method signature to accept the “serviceContext” as well in the last chapter.

```
lmsBook.setAuthor(  
    LMSUtil.encrypt(author, serviceContext.getCompanyId()));
```

Save the changes and add a new book into our Library. You should see the author name getting saved in the encrypted format. I’d like to give you some insight about how the encryption works. Encryption is based on two settings that are inside “**portal.properties**” and the same can be over-ridden via “**portal-ext.properties**”. You can have a quick look at the original properties file to know more about them.

company.encryption.algorithm	company.encryption.key.size
------------------------------	-----------------------------

After saving the book, go to the list page to see the list of books. The value in the author column for the book just got inserted displays in the encrypted format. In the next section we’ll see how to decrypt and show this information.

9.3.2 Decrypting the Data

As before, let’s introduce another utility method inside “**LMSUtil**” for decrypting. Instead of repeating the same set of code in both “encrypt” and “decrypt” methods, I tend to write a new private method and call it from original methods. Let’s call this method as “transform”.

```
private static String transform(String data,  
    long companyId, boolean encrypt) {  
  
    String transformedData = null;  
  
    Company company = null;  
    try {  
        company = CompanyLocalServiceUtil.fetchCompany(companyId);  
    } catch (SystemException e) {  
        e.printStackTrace();  
    }  
  
    if (company != null) {  
        Key keyObj = company.getKeyObj();  
        try {  
            transformedData = encrypt?  
                Encryptor.encrypt(keyObj, data):  
                Encryptor.decrypt(keyObj, data);  
        } catch (EncryptorException e) {  
            e.printStackTrace();  
        }  
    }  
  
    return transformedData;  
}
```

The original method will now look like below.

```

public static String encrypt(String data, long companyId) {
    return transform(data, companyId, true);
}

```

In the case of the new “decrypt” method, the third parameter for “transform” will be “**false**”. Done, our improved utility API’s are ready now. What to do next? Open “**detail.jsp**” and change the line where we are displaying “author” field. The “company” object is an implied object within the JSP itself. Making the below change will also require you to make the import for “**LMSUtil**” within the same JSP file.

```
LMSUtil.decrypt(lmsBook.getAuthor(), company.getCompanyId())
```

After this change check, the detail page of a book and you should see the author’s name getting displayed in it’s original format.

Code Changes @ <http://code.google.com/p/lr-book/source/detail?r=40>

9.3.3 Modifying at the Model Level – Audited Model

Yes, we have made the necessary change in one JSP file. What about making the similar change in “**list.jsp**” where author’s name appears? There could be many other places in the whole application where this data is displayed. Does is really make sense to invoke “**LMSUtil.decrypt**” from each and every place where author’s information is displayed? Definitely this is not a good idea. We need one place where we can do the decryption of the encrypted data before pushing it back to the UI for display purpose. Any idea where exactly are we going to do this? It is none other than our model Impl class “**LMSBookImpl.java**”. We already came across this class when we discussed about entity relationship in section 7.5.1 *Expressing Entity relationship through “service.xml”*.

Open this file and let’s override “**getAuthor**” method like this. Soon you will realize that you cannot make an “**LMSUtil.decrypt**” call, as it requires **companyId** to be passed, which unfortunately we don’t have.

```

@JSON
public String getAuthor() {
    String data = super.getAuthor();
    return data;
}

```

How to make “**companyId**” available inside the generated model class? This is possible by making a normal model class to implement the interface “**com.liferay.portal.model.AuditedModel**”. Another pre-condition for a model to implement this interface is also to have the following fields as it has to mandatorily implement some accessor methods.

companyId	userId	createDate
	username	modifiedDate

Let’s convert our “**LMSBook**” model to an “**AuditedModel**” by implementing that interface. Adding these fields to that entity defined in “**service.xml**” usually takes care of this.

Step 1: Open “service.xml” and insert the new columns that are not there before. Save and run the Service Builder.

```
<column name="companyId" type="long" />
<column name="userId" type="long" />
<column name="userName" type="String" />
```

By virtue of having these fields as part of the entity definition, the Service Builder automatically makes this model of kind “AuditedModel”. You can open the generated “LMSBookModel” interface and confirm this fact.

```
public interface LMSBookModel extends AuditedModel, BaseModel<LMSBook>
```

Step 2: You will also find the new methods now available inside “LMSBookImpl”. Now it should allow us to make “LMSUtil.decrypt” call. Modify **return** statement as “**return** LMSUtil.decrypt(data, getCompanyId());”.

Great! You have nicely over-ridden “getAuthor” method. But when you go and check the Portlet now, it should throw this error, as the actual table is not altered with the new set of columns.

```
Caused by: com.mysql.jdbc.exceptions.MySQLSyntaxErrorException:
Unknown column 'lmsbookimp0_.companyId' in 'field list'
```

Step 3: Run the commands against “lportal” database for appending new columns.

```
ALTER TABLE lms_lmsbook ADD COLUMN companyId BIGINT(20);
ALTER TABLE lms_lmsbook ADD COLUMN userId BIGINT(20);
ALTER TABLE lms_lmsbook ADD COLUMN userName VARCHAR(75);
```

After this step, the Portlet should work fine, but still the data is not getting displayed properly for “author”.

Step 4: Set the additional fields for the new fields at the time of inserting a new book into our Library. This will be done in “**LMSBookLocalServiceImpl.java**”.

```
lmsBook.setCompanyId(serviceContext.getCompanyId());
lmsBook.setUserId(serviceContext.getUserId());
```

Try adding a new book to the Library and confirm that all its information gets properly displayed. Ufff, there seems to be a problem still. The new record is getting saved unencrypted. When we try to access the list of books, there is another error.

```
05:21:43,341 ERROR [http-bio-8080-exec-218][render_portlet_jsp:154]
java.lang.StringIndexOutOfBoundsException: String index out of range: 7
```

Step 5: Even though we are encrypting the data before calling “setAuthor” in our Impl class it is getting decrypted before actually getting saved to the database. The only option now is to over ride the “setAuthor” inside “LMSBookImpl” model class and invoke the encrypt call there.

```
public void setAuthor(String author) {
```

```
    super.setAuthor(LMSUtil.encrypt(author, getCompanyId()));
}
```

Also remove the calls to “`LMSUtil.encrypt`” and “`LMSUtil.decrypt`” from “**LMSBookLocalServiceImpl.java**”. Make sure that the call “`setCompanyId`” comes before “`setAuthor`”. Inspite of doing the problem still persists. Anyhow this exercise, gave us some good insight about how to override the model class. Let’s park this subject for the time being and come back to the same issue at a later point. Comment out or remove both “`getAuthor`” and “`setAuthor`” methods from the model class “`LMSBookImpl.java`”, run the Service Builder and confirm everything is working fine as usual. This means we are no longer encrypting or decrypting the “author” field.

Before we move on, I’d suggest you to have a look at the API docs of `AuditedModel` interface from <http://bit.ly/Ud3TRn>.

Code Changes @ <http://code.google.com/p/lr-book/source/detail?r=41>

9.3.4 `toString()` and `toXMLString()` methods

Any generated model has two interesting methods for quick debugging purposes. They are “`toString()`” and “`toXMLString()`”. Let’s quickly put some print statements inside “**detail.jsp**” and see what is being output.

```
System.out.println(lmsBook.toString());
System.out.println(lmsBook.toXmlString());
```

For a sample book object the first statement should print something like this.

```
{uuid=0e812d73-b92f-4b1b-b88e-46f6f0f1ac8c, bookId=69, bookTitle=eee,
author=eee, createDate=2013-02-13 06:55:16.0, modifiedDate=null,
companyId=10153, userId=10157, userName=}
```

Do it Yourself: Move the methods “`getTypeId`” and “`getAddress`” from `LibraryPortlet.java` to the utility class `LMSUtil.java` so that they can be accessed from anywhere. Correspondingly make changes in the places where they are originally invoked.

9.4 Enabling Logger at Portlet Level

Logging is a very important aspect of any application. They are mainly used for debugging purposes and enhance the maintainability of an application both during development and during production. At the time of development developers usually tend to put lot of “`System.out.println`” within the code to check the control flow of the code they are writing. We are no exception for this and have been doing the knowingly or unknowingly while developing the various examples of this book so far. This is driven by (bad) habit. But usage of “`System.out.println`” has got some serious problems and therefore should be seldom used.

The major concern is the degradation of performance as all “SOP” calls are synchronous and consumes CPU cycles for writing the output to “Standard out”. To mitigate all the limitations of “SOP” Apache has come up with a very powerful logging component called Log4J (<http://logging.apache.org/log4j/2.x/>). Liferay has extensively used Log4J through out its code base. Usine Log4J has many advantages that the conventional “SOP” cannot bring to the table. Some of them are:

- **Flexibility:** It provides different levels for logging. We can segregate the log message accordingly.
- **Configurability:** In just one parameter change we can switch off all the logging statements.
- **Maintainability:** Imagine if we have hundreds of SOPs littered all through the application, it would be difficult to maintain the program over a period.
- **Granularity:** In an application, every single class can have a different logger and controlled accordingly.
- **Utility:** Option for redirecting the message is limited in `System.out`, but in case of a logger you have appenders that provides numerous options.

We should start using Logger in the custom plugins that we are developing to leverage all these benefits. In the upcoming sub sections we'll see how to implement proper logging and how to control it through Liferay's administrative interface.

9.4.1 Integrating with Log4J Logger

Open the file that you want the logger feature and declare a global private static variable. In this case, let's declare a variable inside “`LMSBookLocalServiceImpl`”. Make the additional imports as seen in the next block.

```
private static final Log _log =  
    LogFactoryUtil.getLog(LMSBookLocalServiceImpl.class);  
  
import com.liferay.portal.kernel.log.Log;  
import com.liferay.portal.kernel.log.LogFactoryUtil;
```

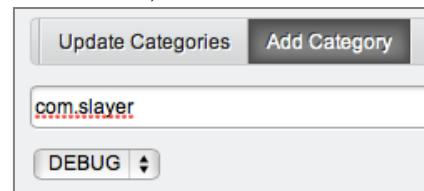
The parameter to the “`getLog()`” method will be the class itself for which we are initializing the logger. Having declared this, let's use this variable inside any of our existing API's. Insert these lines immediately after a book gets inserted in “`insertBook`” method just before “`return lmsBook;`”.

```
_log.debug("Book just got added - debug");
_log.info("Book just got added - info");
_log.warn("Book just got added - warn");
_log.error("Book just got added - error");
_log.fatal("Book just got added - fatal");
```

Try adding a book to our Library and you should see these log messages appear in the server console, except “`_log.debug`” as the default log level starts from “info” in this cascading log levels. But during the time of development you need to put lots of “`_log.debug`” so that when the application goes to production they don’t appear on the console or un-necessary building the size of “**catalina.out**” the file where the output of the console is generally redirected (logged). In the next exercise, we are going to see how to enable “debug” level for this particular instance of the logger, so that all those debug messages start appearing.

9.4.2 Controlling Logging Levels

Liferay has a very powerful interface for controlling the log levels of the various components that are deployed into the application server. This controlling can be done on the fly without the need for writing any code or bringing down the server. Login to portal as Omin Admin. From dockbar, go to **Control Panel → Server → Server Administration → Log Levels**. When you’re here in this screen, first search for a package (component) starting with “**com.slayer**”. You will not find it. Click “Add Category” to create it anew with default log level as “**DEBUG**” and click “Save” as shown here. Go back to our Portlet and try adding a book to our Library. You should see the “debug” messages start appearing now.



```
08:26:12,057 DEBUG [http-bio-8080-exec-151][LMSBookLocalServiceImpl:78] Book just got added - debug
```

This exercise has also illustrated you how to control the log levels of various components (categories) that are already deployed in the portal through the admin interface itself. One thing you have to note is, a category includes all the classes and sub-packages that are inside it. Say for e.g. if we specify a category as “**com.slayer**”, then this applies to all the classes inside this package and the classes that are inside the other sub-packages of this main package.

There is one caveat in the above approach. All the changes made through the UI remain in-memory and just disappears after a server restart. In order to permanently define a default log level for a plugin that is deployed, you have to keep a “**log4j.properties**” file inside the “**src**” folder and deploy the plugin. You can actually copy this from the tomcat server to “**library-portlet**” project and change first line to “`log4j.rootLogger=DEBUG, CONSOLE`”.

9.5 Portlet Internationalization (I18N)

This is another interesting and hot topic. Any big organization wants its portal to be available in many different languages to make a global business presence. Liferay has got many tools in-built into the framework that will help to internationalize any portal and publish the contents in any language. The one aspect of this process is to provide the translation for portal contents and the other aspect is even the applications that are deployed should be language aware. In other words it is called as making our applications “**multi-lingual**”.

9.5.1 Illustration of Internationalization

Let's begin this topic looking into some of the tools and quickly applying a language on one of the existing Portlets of Liferay that comes by default.

Step 1: Login to the portal as administrator, go to “My Library” page and add “Language” Portlet to this page. It will initially appear like the one on left side. Change the configuration settings of the Portlet and change the “Display Style” to “Select Box” to make it appear like the one in right side.



Step 2: Add Blogs Portlet to the same page just below our Library Portlet. You change the language from English to any other language, the labels and buttons in the blog Portlet also got changed immediately. Also observe the browser URL now, it has become like <http://localhost:8080/fr/web/guest/my-library>, with “fr” in it.

At the time of developing the application one can enable or disable the list of languages that the portal can support. Over-riding an entry in “portal.properties” does this. The property is “`locales`”, a comma separated list of language codes. This change will impact globally across the portal. During runtime, the portal administrator of a particular portal instance can enable or disable languages through the interface, Control Panel → Portal → Portal Settings → Miscellaneous → Display Settings → Available Languages.

9.5.2 Internationalizing our Library Portlet

Now it is time to try this out in our Portlet. Unfortunately when you change the language, there is no impact to our Portlet, as we have still not injected the I18N support into it. First let's identify all the labels, messages and contents of the Portlet that we want to get displayed in many languages.

Step 1: In the first round let's identify these texts as candidates for I18N – “Add new Book”, “Show All Books”, “Enter Title to Search” and “Search”. Make entry for each of them in “**Language.properties**” under “**src/content**” as below. We have already created one entry in this file for a different exercise.

```

add-new-book=Add new Book
show-all-books>Show All Books
enter-title-to-search=Enter Title to Search
search=Search

```

Step 2: The next step is to generate the translation for these entries in various other languages that Liferay supports by default. To do this, right click on “library-portlet” project → Liferay → Build Languages has to be run. Do it now and see what happens. For the first time Eclipse will give some warning related to file encoding. Click Ok and try running Build Languages for the second time. After successful completion of this task, you will notice there are many new files generated inside “src/content”.

In the other part, pick up the language files that need a proper translation and give the translated entries. For example, open “**Language_fr.properties**” and modify the entries as below. The best place to get the translated text is through Google translation [<http://translate.google.com>]. Repeat the same exercise for other languages.

```

add-new-book=Commander nouvelle
are-you-sure-you-want-to-delete-selected-books=\n
    Êtes-vous sûr de vouloir supprimer les livres sélectionnés?
enter-title-to-search=Entrez le titre à la recherche
search=Rechercher
show-all-books=Voir Tous les livres

```

Step 3: Great! The tedious job is done! Now it is time to make changes in our application code. Let’s begin making changes to our “**view.jsp**” where these entries are found. Make the following changes to the original text and labels.

Show All Books	<liferay-ui:message key="show-all-books"/>
Add new Book	<liferay-ui:message key="add-new-book"/>
Enter Title to search	enter-title-to-search

The value of the submit button “**Search**” will get transformed like below. Add a new entry in “LibraryConstants”.

```
<%= LanguageUtil.get(locale, LibraryConstants.KEY_SEARCH) %>
```

There are four different ways to pull the translated text for a given key.

1. Using `<liferay-ui:message>` tag as in the case of the two links that appear on the top.
2. Directly using a key for specifying a label for any “aui” tag as in the case of “`Enter Title to search`”.
3. Using `LanguageUtil` API that Liferay has provided.
4. Using `UnicodeLanguageUtil` when the text has to be converted to equivalent Unicode.

Continue the same process for all other JSPs. At last your Portlet will be fully internationalized. Job Well Done!!

9.5.3 Accessing the entries from Portal Source

You can have a quick look at the language properties files (resource bundles) from Liferay's portal source "portal-impl/src/content". Each of the language files has close to 6400 entries. There are three interesting things that I would like to mention before we move on to the next section of this chapter.

a) If you want to make use of any of these entries in your own plugin (say Library Portlet), we need not have to do anything extra. Just use the key and the actual translated text will be supplied from the portal. You can try this out with a simple example. In the same "view.jsp" change some of the keys that we have entered with the key that you will find under the "**content**" folder of portal source.

b) A language can be either left to right (ltr) or right to left (rtl) like Arabic. You should also observe these entries in any of these properties files.

```
lang.dir=rtl  
lang.line.begin=right  
lang.line.end=left
```

c) You will see some entries in these files in the form of templates like,

```
activity-blogs-add-entry={1} wrote a new blog entry, {3}.
```

Searching the source code, you will find the entry for "activity-blogs-add-entry" in "BlogsActivityInterpreter.java". Digging little deeper, you will soon come to know how the template is used to generate the actual translated text with the help of the lines that look similar to the ones below.

```
String titlePattern = "activity-blogs-add-entry";  
  
String title =  
    themeDisplay.translate(titlePattern, titleArguments);
```

This conveys that we can also use the "translate" of "themeDisplay" object that is available in any JSP or Portlet class. There are three different "translate" methods available, all of them returning the translated text of the language that is currently set in the application context.

```
themeDisplay.translate(key);  
themeDisplay.translate(pattern, argument);  
themeDisplay.translate(pattern, arguments);
```

Note: While giving entries in the original "Language.properties" avoid breaking long lines with "\ " and also see to it that the "=" does not have any spaces on either sides. These are some hygiene factors that will help you to reduce any potential problems.

9.6 Portlet and WCM Marriage

The title itself is very different You must be thinking how suddenly we got into this interesting subject. Yes, it is going to be very interesting as we are going to see the various ways our Portlet and WCM can work together seamlessly in any given Portal page. WCM stands for Web Content Management that will help the portal users to create and publish the portal contents with the help of various tools meant for that purpose. The content creation is done on the fly without writing one single line of code. To know about WCM you should quickly go through Liferay's official documentation [<http://bit.ly/15e8ZQF>]. I am also going to give some insight about this subject in an appendix for this book. You should also go through the advanced WCM features of Liferay from <http://bit.ly/XD7uYK>.

9.6.1 Embedding Web Content in a Portlet

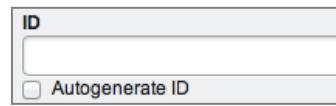
Imagine how nice it will be if you give the option for the Librarian who is managing our Library to put some nice contents in the front page of our Portlet. These contents talk something good about our Library and can change over time. Remember the creation and editing of this content has to be done by a non-programmer during runtime, on the fly. Liferay provides this feature with the help of another powerful tag that pulls data from the site's Content Management System (CMS). Let's try this out in our Portlet and make it much more intuitive for the Librarians to use it. We'll perform the following three steps.

Step 1: Login to the portal as administrator. Go to **Control Panel → Liferay (Site) → Web Content**. You will see the list of web contents already out there. Before we create a new Web Content (article) for the front page of our Library, we have to do an important thing. This is to append the below entries inside “**portal-setup-wizard.properties**” under PORTAL_HOME and restart the Tomcat server. We can carry on with other steps even without carrying out this one. But there are some compelling reasons to do this. I'll soon explain them to you.

```
journal.article.force.autogenerate.id=false  
journal.structure.force.autogenerate.id=false  
journal.template.force.autogenerate.id=false
```

By making these entries to **false**, we are telling the portal server not to auto-generate the IDs for an article, structure and template. If it is not going to get generated by itself, we have to enter these ID's manually which has a definite benefit. After the server restart check “Web Content” admin page from the Control Panel. Click the button **Add → Basic Web Content** to create a new article.

Now the form will have a field to manually enter the article ID and also a checkbox to auto-generate the ID. This was not the case before. Leave this box unchecked.



A screenshot of a web form titled "Basic Web Content". It contains a text input field labeled "ID" and a checkbox labeled "Autogenerate ID".

Step 2: Create a new content with an ID as “LIBRARY_WELCOME_MESSAGE” (*you can give any meaningful name*), a title and some contents as much as you want. You can decorate the contents will all the tools available with the WYSIWYG editor called [CKEditor](#) that Liferay uses as the default Rich Text Editor (RTE). All these

editors are configured for different applications of Liferay through the “**portal.properties**” file. You can find them under section “**Editors**” in that file. Once you have put the contents click “Publish” to save and publish the content so that it can be deployed on any page of our Portal or consumed by any Portlet.

Step 3: In this step we are actually going to consume this Web Content (article) inside our Portlet’s front page. Open “**view.jsp**” and insert the tag immediately after the **import** and **include** statements.

```
<liferay-ui:journal-article articleId="LIBRARY_WELCOME_MESSAGE"
    groupId="<% themeDisplay.getScopeGroupId() %>" />
```

Save the changes and check the front page of our Portlet now. The contents you have entered should appear in the top. Using “**<liferay-ui:journal-article**” tag, we can embed any article anywhere inside the Portlet and easily manage the contents without touching the Portlet code and re-deploying it. This tag takes other attributes as well. You can have a look at all those attributes and identify the purpose of each.

Now I am going to tell you why we enabled the option to manually enter the article Id instead of the portal auto-generating it. If you allow the portal to auto-generate the Id, then you will not have control over that value and every environment (development, staging and production) will have a different value for the Id. Hence, in every environment you need to change this value inside the JSP file before deploying the Portlet that will be a overkill. The other reason is for readability. Just by going through the JSP file now, you can very clearly make out which article you’re embedding, as the “**articleId**” is a meaningful name. If it were a real number, then it is going to be little difficult though not impossible.

There is only one thing that you have to take care while embedding the web contents inside a Portlet. Whenever a Portlet is deployed onto an environment, you have to either freshly create all the dependent web contents or import the LAR file which was earlier exported from a different environment. Say for example from the development environment to the production environment.

9.6.2 Embedding Portlet inside Web Content

We have just seen how to embed web content inside a Portlet. Why not try something opposite? How about the idea of embedding a Portlet inside web content, totally the other way round? Sounds crazy? Definitely not! It is very much possible with the capability Liferay provides. With this approach we can mitigate the limitations of the previous approach. We added header information there, injected via Web Content. If we want content managed footer then we need another Web Content. Not only this, but the Portlet code has to be modified and redeployed for the changes to take effect. In the new approach, with the Portlet embedded inside Web Content, we have complete freedom. You can change anything you want through CMS without impacting the actual Portlet that is embedded. Let’s give it a try.

Step 1: This time let’s not start from the Control Panel. Instead let’s create a new page by name “Embedded”. When the new page is ready, drag-and-drop a “Web Content

Display” Portlet onto the right side of the page by clicking Add pull-down menu from the dockbar. When the new instance of Web Content Display Portlet is ready on the page, it will show the message “**Select existing web content or add some web content to be displayed in this Portlet**”. You can optionally set the Portlet border by going to “Look and Feel” and checking that option. Now our container is ready to be the host.

Step 2: As per the message displayed in the Portlet, click the second icon at the bottom to add a new Web Content. In the resulting page, give the new article ID as “LIBRARY_PORTLET_HOST”, a good title and give some good contents. Before you click “Publish” to save the article, switch to the “Source” mode to insert the embedding line inside the bunch of HTML code.



Embed the below line and finally click “Publish”. You will be taken to the new page and you will get excited to see our Library portle nicely embedded inside the Web Content. You can do all operations on this Portlet as before and everything will not happen within the containing Web Content, “LIBRARY_PORTLET_HOST”.

```
<runtime-portlet name="library_WAR_libraryportlet"></runtime-portlet>
```

Here “`library_WAR_libraryportlet`” is the Id of the Portlet that we have to embed. You can get this id from Liferay’s “portlet” table in the database. Usually while doing this kind of embedding you will use advanced WCM instead of basic WCM. The advanced WCM will use Structures and Templates, so that you can individually manage each and every piece of the Web Content instead of putting everything as one single HTML as in this case. There is a nice blog at Liferay.com written by Barrie Selack that explains this approach. You should have a quick look into this blog once you have some spare time. The URL is <http://bit.ly/YfE7tT>.

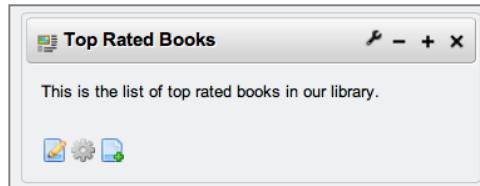
9.6.3 Accessing Service Layer API's from Web Content

Next is another powerful feature of Liferay. In the earlier two sub-sections we have seen how to embed Web Content inside a Portlet and Vice-versa. Think of a situation where you can have direct access to all the back-end API's of the Service Layer. Using these API's you can get the data into the HTML and it is completely up to you to decide how you want the data to appear in the front-end. Let's take our own Library Portlet. Very soon, we may make this Portlet to be used only by the Librarians to add and edit the book information. For the front-end users we are going to show the book information very nicely in a good design with a link to the books. Clicking on the link, the user will be taken to the book's detail page. Sounds very interesting ! Let's actually do it and see it happening with our own eyes.

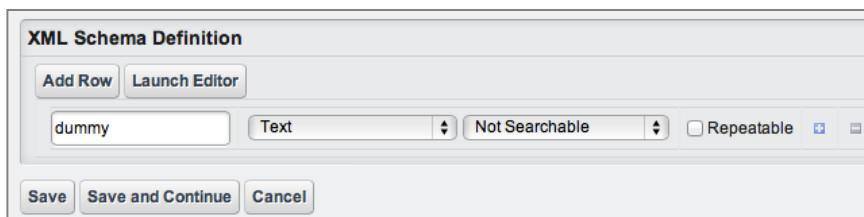
Step 1: If we have to access Service Layer API's directly from the velocity template files, we have to first enable this option by overriding an entry in “**portal.properties**”. Put this entry in “**portal-setup-wizard.properties**” removing “`serviceLocator`” from the list and restart the server.

```
# Input a comma delimited list of variables which are restricted from
# the context in Velocity based Journal templates.
journal.template.velocity.restricted.variables=
```

Step 2: After the server restart, login to the portal as administrator and add Web Content Display Portlet in the Welcome page and make it appear like the one below by editing the Portlet title.



We'll park this Web Content Display Portlet for a little while till we create a Structure and Template. You can do this directly by clicking the “edit” icon appearing in the bottom of this Portlet. But for getting more clarity for the first time, let's do this from the Control Panel. Go there, click “Web Content” and select “Structures” tab. Click “Add Structure” to add a new structure with Structure ID and title as “HOT_BOOKS_STRUCTURE” with one “dummy” field of type “text” and save.



In the same lines, create a new template with template ID and title as “HOT_BOOKS_TEMPLATE”. Select the structure that we just created and click “**Launch Editor**” button to replace its original contents (sample) with this block of velocity code and click “Update”.

```
#set($lmsBookLocalService = $serviceLocator.findService("library-
portlet" , "com.slayer.service.LMSBookLocalService"))

#set($lmsBooks = $lmsBookLocalService.getLMSBooks(0,-1))

#foreach ($lmsBook in $lmsBooks)
    <div>$lmsBook.getBookTitle()</div>
#end
```

Save the template and go back to our original Web Content Display Portlet titled “Top Rates Books” placed in the Welcome page. Click “edit” icon on the bottom. In the resulting page, click “select template” icon.

Do it Yourself: In the template file put more HTML to decorate your list. Once you have done this, you have to provide a link on each book to go to its details page. You should make use of the friendly URL that we have seen in [Section 9.2 Implementing Friendly URL's](#). The friendly URL for the details page is of the format “/web/guest/my-library/-/library/detail/**XX**” where **XX** has to be replaced by “`$lmsBook.getId()`” inside the “foreach” loop.

Here I would like to mention [couple of things](#):

1) If you have to access any of the portal's service from the velocity template files, that you will not use the first parameter for "findService", where we passed the name of the web application. Say for example you have to access the userService, and then the call will be like below. You should put this in one line to avoid any problems. For shortage of space, I've broken down into two lines.

```
$serviceLocator.findService(  
    "com.liferay.portal.service.UserLocalService"))
```

2) You will apply the same principle to access any of the Service Layer API's from the velocity files of a theme. Any theme will have the five velocity template files.

init_custom.vm	navigation.vm	portal_normal.vm
portal_pop_up.vm	portlet.vm	

9.6.4 Accessing Theme Resources from Our Portlet

Let's end this section with this quick topic that will show how to access the images from the underlying theme and make use of them inside our Portlet. Themes govern the overall Look and Feel of a Site in the Portal. To see this in action, open "view.jsp" and insert the below code just ahead of the code that shows up the "[Add new Book »](#)" link. You can optionally set additional attributes for the HTML "`img`" tag.

```
<% String addImagePath =  
    themeDisplay.getPathThemeImages() + "/common/add.png"; %>  

```

Save the changes and check the Portlet to confirm the add icon beautifully appearing before the actual link. These small icons are of great use while building your own applications and will greatly enhance the appearance of your Portlet. You need not have to purchase a new set of icons. Liferay has lots of these icons already made available for you absolutely free of cost. You will find all those icons under "**PORTAL-SOURCE/portal-web/docroot/html/themes/_unstyled/images**".

9.7 Making Our Portlet appear in Control Panel

In the beginning of Section [9.6.3 Accessing Service Layer API's from Web Content](#), I've given a small hint. The hint was to make our Portlet available only for the Librarians of various organizations to manage the books and not to be accessed by the members (end users) of the Library. Though we may not do this shift immediately, we'll put a base for this transition in this section by making our Portlet appear in the Control Panel.

When you login to the portal as Omni Admin and go to the Control Panel you see five sections there. The "Server" section is visible ONLY for Omni administrators of the Liferay server. The "Portal" section is displayed to a Portal Administrator. The "Web Site" section appears for a user who can manage a web site. The "User" section is for any logged in user. Marketplace section is to manage our applications that are obtained from Liferay Marketplace and deployed to the server.



We are not going to get into the details of what options appear inside each of these five sections. Leaving this subject matter for you to explore. What we are going to do now is to give a link to our Library Portlet under one of these sections. First we need to make a decision that is the appropriate section for any portlet to be placed in a real world scenario. The decision is purely based on how granular we would like our Portlet to be – Global level, Portal level, Web Site level or User level. In the case of our Library Portlet let's go with the assumption that each site (organization) that are in an enterprise or an institution will have it's own instance of Library and a set of members.

Say for example in a university there can be central Library where students from different departments can come and borrow books. Apart from this central Library, each department will have its Library where the department specific books are maintained. I am sure you'd have got a fair idea about how our Library Portlet can cater to the needs of the university's central Library and the individual Libraries of various departments of the University.



Let's start our activity. Open "**liferay-portlet.xml**" file for our Portlet. Insert the two lines at their appropriate location (*just before "`action-url-redirect`" entry*).

```
<control-panel-entry-category>content</control-panel-entry-category>
<control-panel-entry-weight>3.5</control-panel-entry-weight>
```

Save the file and check your Control Panel. Our Library Portlet should appear in the "content" section immediately after the Site related links.

9.7.1 More about Control Panel

There are three entries with respect to “`control-panel-entry`” - `category`, `class` and `weight`. Category can take one of these values – *my*, *content*, *portal* and *server*. This entry will make the Portlet to appear in the appropriate section in the Control Panel. The second entry is a class that should either implement [ControlPanelEntry](#) [<http://bit.ly/WJlsL2>] OR extends [DefaultControlPanelEntry](#) [<http://bit.ly/16u8L8B>] and is called by the Control Panel to decide whether the Portlet should be shown to a specific user in a specific context. The default value is set in “**portal.properties**”. In our case we have not supplied this value and hence it takes the default class. Set the **weight** value to a double number to control the position of the entry within its Control Panel category. Higher values mean that the entry will appear lower in the Control Panel menu.

You can look at the following entries in “**portal.properties**” that are related to the default settings of the Control Panel and its appearance. You can override these settings with the help of “**portal-ext.properties**”.

```
control.panel.layout.name=Control Panel  
control.panel.layout.friendly.url=/manage  
control.panel.layout.regular.theme.id=controlpanel  
control.panel.navigation.max.sites=100  
control.panel.default.entry.class=\  
    com.liferay.portlet.DefaultControlPanelEntry
```

It is worth taking a look at this Liferay Wiki [<http://bit.ly/15OfjOj>] to get a different perspective of what we have seen so far. Just to summarize, Liferay’s Control Panel is the central location where everything can be administered. If you will be administering a Liferay portal, you will spend most of your time in this very place. The Control Panel is well organized and easy to navigate.

9.7.2 Making our Library unique for every Department

Now we are coming to the most important stage in the development of our Library Management System. I took up this section primarily to drive home a very important concept. Before getting into that let’s create couple of new “Sites” one for the “Life Sciences” department and the other for the “Humanities” department. Create the new sites by going click in the heading “Sites” inside Portal section of the Control Panel. You can keep both these sites as of type “Open” so that any one can access them.

The moment the sites are created they should start appearing in the pull down menu of “Web Site” section of the same Control Panel. Now click on these sites and subsequently check the contents of their respective Libraries. You will be shown all the books that we have added so far. But ideally they should show ONLY the books that are unique to them.



The problem is quite natural, as we NEVER programmed our application to take care of this special requirement, thus far. How are we going to do this? The answer is by logically separating the data in the underlying table by another column called `groupId`.

This will ensure that each record in the table belongs to a particular site within a given portal instance (company). Let's see the actual implementation next.

9.7.3 Introducing “groupId” for LMSBook

If you have a quick look at all the main entities of Liferay portal, they have this column of “groupId” apart from their regular five audit fields. For your satisfaction you can check the columns of the tables “blogsentry”, “calevent”, “dlfileentry”, “journalarticle” and “user_”. Let's apply the same rule for our “LMSBook” as well.

Step 1: Open “service.xml” and add this new column of type “*long*”. Run the Service Builder.

```
<column name="groupId" type="long" />
```

Usually the changes do not get reflected automatically in the underlying database. We have to manually append this column to the underlying table by running the following command against the “lportal” database.

```
ALTER TABLE lms_lmsbook ADD COLUMN groupId BIGINT(20);
```

Step 2: Now our entity is ready, we have to do two code level changes that are explained in the next two steps. Set the “groupId” attribute while saving a book into the table. This is done with the help of “serviceContext” object available inside the “insertBook” API of our DTO class “**LMSBookLocalServiceImpl**”.

```
lmsBook.setGroupId(serviceContext.getScopeGroupId());
```

Save changes and try entering book directly from the Control panel of both sites – “Life Sciences” and “Humanities”. While doing this exercise, I got two records inserted into the table with values for “groupId” as 18002 and 18006. What are these numbers? Let's investigate further. They are nothing but the primary key of records in “group_” table where every site that we create in Liferay is stored. Infact the same table is used not just to store site information but also for another three entities of the Liferay Portal – an organization, a user group and a user itself. I think you should quickly create one entity from each of these types directly from the Control Panel and see how the “group_” table gets impacted.

Step 3: Both the schema changes and API changes are done. The book data is getting stored properly with logical separation at group level. We have to now write the logic to retrieve the books based on “companyId” and “groupId” filter and display in our Library Portlet. This will ensure the member of one department Library will not see the books of the other department or of a book in the university’s central Library. The best way to make this separation at the application level is to define a finder and use it in our application layer.

a) In “service.xml” define a finder for “**LMSBook**” entity and run Service Builder.

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

```
<finder-column name="groupId" />  
</finder>
```

- b) Define a new API in the DTO class “**LMSBookLocalServiceImpl**” that will return the books in the Library database based on these two fields. Run Service Builder once again as we have introduced a new API.

```
public List<LMSBook> getLibraryBooks(long companyId, long groupId)  
    throws SystemException {  
    return lmsBookPersistence.findByCompanyId_GroupId(  
        companyId, groupId);  
}
```

- c) Open “**list.jsp**” to replace “`LMSBookLocalServiceUtil.getLMSBooks(0, -1)`” that fetches all the books from the underlying table with this line, in order to get the books that are relevant to a particular department.

```
LMSBookLocalServiceUtil.getLibraryBooks(  
    company.getCompanyId(), themeDisplay.getScopeGroupId());
```

Save the changes and confirm when you hit “Library Portlet” from the Control Panel; it should list ONLY the books that are relevant to a particular department.

Do it Yourself: 90% of the work is done. Don’t you think we can even modify our search API’s to pull data relevant to a company and a group? This is a task for you to do. Our original search API “`searchBooks(String bookTitle)`” is no longer valid. Create a new API that will have the below format.

```
public List<LMSBook> searchBooks(String bookTitle,  
    long companyId, long groupId)
```

Make corresponding changes to both dynamic query and custom SQL to pass the additional two attributes while querying the table. Just to give you a hint the additional parameters to the dynamic query will be set like this.

```
dynamicQuery.add(RestrictionsFactoryUtil.eq("companyId", companyId));  
dynamicQuery.add(RestrictionsFactoryUtil.eq("groupId", groupId));
```

Similarly the “WHERE” clause in our custom SQL query will become like,

```
bookTitle like ? and companyId = ? and groupId = ?
```

The following two more positional parameters will be set for the query that is being called from the API “`findBooks`” of “**LMSBookFinderImpl**”.

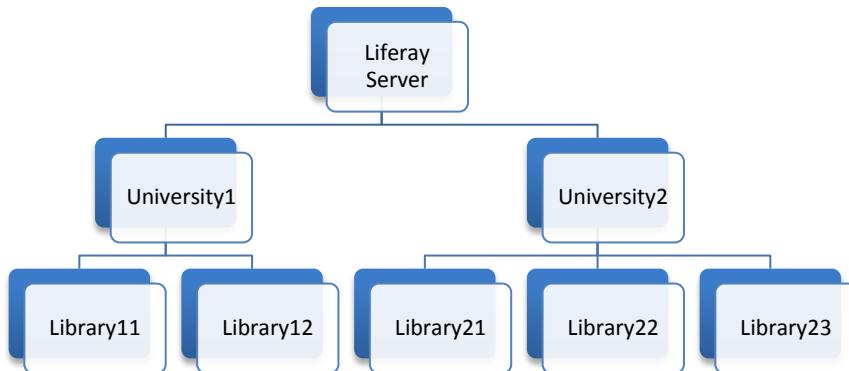
```
qPos.add(companyId);  
qPos.add(groupId);
```

The method signature in this class will become like,

```
public List<LMSBook> findBooks(String bookTitle,  
    long companyId, long groupId)
```

9.7.4 Liferay and Multitenancy

With the introduction of “companyId” and “groupId” we have put a strong foundation for one of the most striking feature of Liferay – Multitenancy. So far we have been thinking about one university and several departments under that university. But if we want to host our solutions as SaaS (Software as a Service) that can be used by many universities across the globe, will our Library Management System scale up to that extent? The answer is, YES! With the data being segregated both at the database level and the application level, our Library Management System can support as many universities as we want. This feature is generically called as Multitenancy.



We have seen Liferay to support 100% the following Wikipedia definition of Multitenancy. You can get the details from <http://en.wikipedia.org/wiki/Multitenancy>.

Multitenancy refers to a principle in software architecture where a single instance of the software runs on a server, serving multiple client organizations (tenants). Multitenancy is contrasted with a multi-instance architecture where separate software instances (or hardware systems) are set up for different client organizations. With a multitenant architecture, a software application is designed to virtually partition its data and configuration, and each client organization works with a customized virtual application instance.

Before we conclude this sub-section, I am going to quickly tell you how to create a new portal instance in your existing setup. Just follow these six steps in your local machine. These steps will vary in a real production environment.

1. Login to the portal as Omni administrator and go to **Control Panel → Server → Portal Instances**.
2. Create a new portal instance with “utlibrary.com” as the value for Web ID, Virtual Host and Mail Domain. Click Save. Confirm a new instance got created nicely. In this example we are creating an instance for University of Texas.
3. Open “**c:/Windows/System32/drivers/etc/hosts**” in a notepad and append “127.0.0.1 utlibrary.com” before saving this file. (*You can edit this file ONLY you're an administrator of the machine you're currently working. If you're Linux or Mac user, the path of this file will be slightly different.*)
4. Open the browser and type <http://utlibrary.com:8080>, you will be made to land in the new portal instance. <http://localhost:8080> will still land you in the original portal instance.

-
5. Login to the new instance with email as test@utlibrary.com and password “test”. Go to the Control Panel and you will not see the “Server” section now.
 6. Under the “Portal” section of the Control Panel, Click Portal Settings → Email Notifications. In the “Sender” tab, change the values for Name and Address fields as “UT Library Admin” and admin@utlibrary.com respectively and save.

As a side effect of adding a new portal instance the following five things happen in the underlying database. You should verify these changes yourself.

1. A new record gets inserted in “company” table that has a foreign key to “account_” table.
2. Let’s check the account table. You will find a new record there as well which contains the additional information about the portal instance that just got created.
3. There is a parallel entry in “virtualhost” table.
4. Two records get inserted in “user_” table one is default@utlibrary.com (default user) and the other is test@utlibrary.com (portal admin).
5. The “sixth” step that we did earlier has also inserted a record in “portalpreferences” table with the companyId in ownerId column and ownerType as “1”. Just see the contents of the preferenes column for this record, you will find all the settings for the company saved in the form of XML.

9.7.5 Check with Existing Applications

So far we have successfully made our same Library Portlet’sable at various levels of any enterprise / institution. At this juncture, I would like you to take a look into the way the other portlets of Liferay are implemented. They follow the exactly same kind of logical separation at the portal level and then the group level. This feature will help us to develop applications that can be seamlessly used by various stake holders at various levels in the organization hierarchy with the data being complete secure. Before we end this section I would like to bring another important feature to your kind attention and knowledge. This feature is setting the “scope” of the Portlet data. We need not have to do anything for this. Liferay automatically takes care of this feature, just by specifying the “`<scopeable>`” directive in “`liferay-portlet.xml`”.

Insert “`<scopeable>true</scopeable>`”, immediately after “`</instanceable>`” in this XML file and save the changes. Go back to our Library Portlet originally deployed inside “My Library” page. Go to its configuration settings and you will observe a new tab “Scope” along with the existing ones – Permissions and Sharing. Click on this tab and there will an option to select the scope of this Portlet and its data. There is a default scope, a global scope and page level scope. I feel it is worth taking a look at an old documentation of Liferay to get some more theoretical knowledge about scopes. The link is <http://bit.ly/1LYftX>.

Code Changes @ <http://code.google.com/p/lr-book/source/detail?r=45>

9.8 Running Backend Jobs through Quartz

Everything is fine so far. But soon the administration of the universities’ central library realized that the members (students) who are borrowing the books from the

Library are not returning the books on time for various reasons. They either forget the return date or quite occupied with other things. Because of the delay in returning the book the other students who are in need of a particular book get impacted. Off lately this has become a serious problem for the Library administration and they wanted to come up with a mechanism that will notify the students to return the book 2/3 days ahead of the actual return date. The library administration approached the IT team that is maintaining the Library solution built using Liferay. Both sat and discussed together and decided to notify the defaulting members through email and SMS.

The members of the IT team did some googling and soon figured out the fact that Liferay has an in-built feature to send this kinda of notifications through an offline process with the help of a powerful scheduling and triggering engine called Quartz (<http://quartz-scheduler.org>). The component is well integrated with Liferay so that any Portlet plugin can seamlessly execute any kind of back-end job in an asynchronous fashion without impacting the frontend usage of the application. The scheduler can be triggered based on either based on specific intervals or on CRON expressions. If you're new to CRON, you must have a look at <http://en.wikipedia.org/wiki/Cron> to know more about this great UNIX utility.

9.8.1 Implementing the actual Job

The developers of Library IT team plunged into action writing the scheduled job for notifying the members who are defaulting and not returning the books on time. The process involved just two steps.

Step 1: The first step was to write the job itself. Create a new class inside package, “**com.library.job**”. Name is as “NotifyDefaultingMembers” and make it to implement, “**com.liferay.portal.kernel.messaging.MessageListener**”. The empty method will look like below. Make the other necessary imports.

```
public class NotifyDefaultingMembers implements MessageListener {  
    public void receive(Message message)  
        throws MessageListenerException {  
            System.out.println("inside receive method...");  
    }  
}
```

For the purpose of initial testing, we have put a SOP statement inside this method. Before we start writing the actual logic, let's move to the next step to configure the new offline job and registering with the Quartz Scheduler that is running inside the portal server. Instead of implementing the **MessageListener** interface you can also extend our scheduler class from a base class, [**BaseMessageListener**](#) and override the “doReceive” method of this class.

Step 2: The second step was to configure this job through “**liferay-portlet.xml**”, using the appropriate tags. Before making the entry, the IT guys decided to run this job on a daily basis at 6.30 AM in the morning, so that members can get timely notification enabling them to come to the Library and return the book before proceeding to their classes. Insert these entries immediately after the “[**<icon>**](#)” tag and save the file.

```

<scheduler-entry>
    <scheduler-event-listener-class>
        com.library.job.NotifyDefaultingMembers
    </scheduler-event-listener-class>
    <trigger>
        <cron>
            <cron-trigger-value>
                0 30 06 ? * MON-FRI
            </cron-trigger-value>
        </cron>
    </trigger>
</scheduler-entry>

```

One of the best places where you can get some good examples of CRON expressions is from the official Quartz site itself, <http://bit.ly/15jLHZV>. The “`<trigger>`” tag in the above block can be of two types “`<cron>`” and “`<simple>`”. The simple element specifies an interval trigger for a scheduler with the help of “`<time-unit>`” sub tag that can take one of the values – *second*, *minute*, *hour*, *day* and *week*. The other sub-tag “`<simple-trigger-value>`” supplies the actual value of the interval. One beauty about both these types of triggers is, we can externalize the value of “`<cron-trigger-value>`” or “`<simple-trigger-value>`” with “`<property-key>`” which is an entry in “**portlet.properties**” file.

As of the above example, the developer of the IT team cannot wait till next day morning to verify the job is running properly. Hence, for the time being, he replaced cron trigger with a simple trigger that runs every **5 minutes**.

```

<simple>
    <simple-trigger-value>5</simple-trigger-value>
    <time-unit>minute</time-unit>
</simple>

```

The moment the changes are saved, you can also start seeing the job getting executed every five minutes with the message “inside receive method....” getting printed on the server console. There are many entries in “**portal.properties**” that you can override to change the default behavior of the quartz scheduler. You can find all of them under the `Quartz` section in that file.

What's next? You have to help the IT team developer to write the actual logic inside the schedule job class to pull all those defaulting members and send them the email notification. I am sure you will be very glad to do this help. This is the exercise for you now before we move on to the next topic.

9.8.2 Contents of “message” object

If you're quite an inquisitive person like me, I am sure you will write some code inside the “`receive`” method to know the default contents of “`message`” object.

```

Map<String, Object> map = message.getValues();
Iterator<String> itr = map.keySet().iterator();

```

```

while (itr.hasNext()) {
    System.out.println(itr.next());
}

```

The server console will show you the following items present inside this object including the “compayId”. You can actually make use of these values inside the Job class that is being executed.

principalPassword JOB_NAME EXCEPTIONS_MAX_SIZE CONTEXT_PATH	principalName GROUP_NAME DESTINATION_NAME PORTLET_ID	companyId JOB_STATE MESSAGE_LISTENER_CLASS_NAME RECEIVER_KEY MESSAGE_LISTENER_UUID
--	---	--

9.8.3 Winning of the Portlet Contest

As we are discussing about Quartz Scheduler, I just became reminiscent of one great thing that happened way back in the year 2007 and wanted to share the same with you. It was a time when Quartz was not embedded as a part of the Liferay stack. I developed a Portlet on top of Quartz using which the Portal administrator can seamlessly manage the various offline jobs that are deployed onto the server, change their schedule and view the complete history. This Portlet won the international Portlet contest joined organized by Liferay Inc. and [FireScope Inc](#). I got a cash prize of 1000 US Dollars after the award selection committee selected that component as the winner. It was a great milestone in my Liferay career where I could showcase something to the international audience.

Code Changes @ <http://code.google.com/p/lr-book/source/detail?r=46>

Note: An Offline Job can either implement the interface as we did above or extend another abstract class “BaseMessageListener” inside the same package as the interface, “com.liferay.portal.kernel.messaging”. In this case our Job class has to override the abstract method “doReceive”.

9.9 Firing Emails from Portlet

I am sure you have helped the guy in Library's IT team to write the query to find out all the members who have not returned the books even after the due date. In this process, you must have inserted another column to “*LMSBorrowing*” entity in “*service.xml*” and run Service Builder – `<column name="dateOfReturn" type="Date" />`. After pulling the list of records, we need to loop through them and fire email to each member. Assume that we have written the following API “*notifyMember*” to do that part in our Job class “*NotifyDefaultingMembers*”.

```
private void notifyMember(LMSBorrowing borrowing,
                         String subjectTemplate, String bodyTemplate) {

    User member = null;
    try {
        member =
            UserLocalServiceUtil.fetchUser(borrowing.getMemberId());
    } catch (SystemException e) {
        e.printStackTrace();
    }

    if (Validator.isNotNull(member)) return;

    long companyId = member.getCompanyId();

    InternetAddress fromAddress = getFromAddress(companyId);
    MailMessage mailMessage = new MailMessage();
    mailMessage.setFrom(fromAddress);
    mailMessage.setSubject(subjectTemplate);
    mailMessage.setBody(bodyTemplate);

    String toEmail = member.getEmailAddress();
    String toName = member.getFullName();
    try {
        mailMessage.setTo(
            new InternetAddress(toEmail, toName));
    } catch (UnsupportedEncodingException e) {
        e.printStackTrace();
    }

    MailServiceUtil.sendEmail(mailMessage);
}
```

This method has a dependency on another private method “*getFromAddress*” based on the *companyId* (*portal instance or university information*). Let's write this method as well.

```
private InternetAddress getFromAddress(long companyId) {
    PortletPreferences portletPreferences = null;
    try {
        portletPreferences =
            PortalPreferencesLocalServiceUtil.getPreferences(
                companyId, companyId, 1);
    } catch (SystemException e) {
        e.printStackTrace();
    }
}
```

```

String key1 = PropsKeys.ADMIN_EMAIL_FROM_ADDRESS;
String fromEmail = portletPreferences.getValue(key1,
    PropsUtil.get(key1));

String key2 = PropsKeys.ADMIN_EMAIL_FROM_NAME;
String fromName = portletPreferences.getValue(key2,
    PropsUtil.get(key2));

InternetAddress fromAddress = new InternetAddress();
fromAddress.setAddress(fromEmail);
try {
    fromAddress.setPersonal(fromName);
} catch (UnsupportedEncodingException e) {
    e.printStackTrace();
}

return fromAddress;
}

```

Make all the necessary imports that are required by these two methods, most importantly “`javax.portlet.PortletPreferences`”.

The above two methods have illustrated to many things. The most important of them is how we get the “`fromEmail`” and “`fromName`” from the settings of a particular company (university in our case), so that every member will get the email directly from his university. The second important thing is how we used the “`MailServiceUtil`” class to finally send the email with “`mailMessage`” object that has all the information about the email like `fromAddress`, `toAddress`, `mail subject` and `mail body`. In the next chapter we’ll see more about `PropsKeys`. There are still some important aspects that are left in this mail program. They are:

- Templatizing the mail subject and mail body
- Setting the SMTP configuration

We are going to park these topics for the time being and pick them back in our next chapter where we are going to talk in detail about the various configuration mechanisms.

Code Changes @ <http://code.google.com/p/lr-book/source/detail?r=47>

9.10 Getting Direct Access to Database

We have seen in length the various aspects of Service Layer and the API's it provides in earlier chapters of this book. Inspite of all the available features, you may encounter situations where you have to make direct calls to be Database layer through a connection that is obtained from the Liferay's connection pool. Liferay by default uses C3P0 for connecting pooling (<http://sourceforge.net/projects/c3p0/>). At the same time it can also make use of other connection pooling mechanisms. You can find these entries in “**portal.properties**” and these entries can be over-ridden with the help of “**portal-ext.properties**”.

9.10.1 Obtaining a Connection from Pool

Whenever you want a JDBC connection instance to be obtained from the pool to perform any low-level database operations, you will use the code something like this.

```
Connection connection = null;
try {
    connection = DataAccess.getConnection();
} catch (SQLException e) {
    e.printStackTrace();
} finally {
    DataAccess.cleanUp(connection);
}
```

Make the necessary imports.

```
import java.sql.Connection;
import java.sql.SQLException;
import com.liferay.portal.kernel.dao.jdbc.DataAccess;
```

Whenever you get a connection from the pool, it is extremely important to release the connection back to the pool once we are done with whatever we wanted to do. “`DataAccess.cleanUp(connection);`” will do this for you. Failing to do so will very soon make the connection pool run out of connections and eventually bring down the server. Hence, you're hereby adviced to make use of this API with lot of care and use it only if it is very critical.

9.10.2 Max Database Connections

In the event of running more than one instances of Liferay on the same server, each using different ports, you have to reset the value of the entry “`jdbc.default.maxPoolSize`” in “**portal.properties**” to a lesser value than 100. The reason is by default the database will open 100 connections to the application server. If both the instances of Liferay will try to acquire all these 100 connections from the database server, then there is going to be definitely a contention. This will also certainly lead to server getting collapsed.

Summary

Hi friend, we have successfully covered some very important topics in this chapter. I strongly feel, you should read the chapter once again before moving on to the next chapter, keeping the significance of the topics covered.

We started this chapter with Portlet Filter with the intention of giving due respect and honor to the original Portlet 2.0 specification. Then we moved on to cover the Friendly URL where we have seen the various aspects of making the Portlet URL friendly with the help of one “routes.xml” and some settings in “liferay-portlet.xml”. The chapter continued with the next topic of encrypting and decrypting the Portlet data that have to be kept secure in the database. Next we spoke about logging where we have seen how to use the Logger API of log4j to log the message of the Portlet. In the same section we have seen how to control the log levels from the admin UI.

The chapter continued to discuss how to make our Portlet multilingual, so that people all over the world can use it without the need for an interpreter. We have seen how to generate the various language files with the help of “Build Language” utility. The next section discussed the integration between our Portlet and Web Content Management – embedding web content inside a Portlet, embedding a Portlet inside web content and accessing the servier layer of the Portlet from the web content. Then we moved on to another important topic of making the Portlet to appear in the control panel so that only the administrators can use it.

In this topic, we have seen the various levels of achieving multitenancy using “groupId” and “companyId” columns. We explained this with the help of a nice example of universities and their departments. Next we moved on to running back-end jobs thru Quartz that comes bundled with Liferay by default. We have seen the various ways the trigger can be configured. Contining in the same lines, we have seen how to fire emails from our application with the help of MailServiceUtil. We parked some of the topics, as this chapter was getting quite lengthier.

We concluded the chapter with the last section on how to directly access the database with the help of getting a connection object from the connection pool using DataAccess class. Overall, hope you enjoyed reading this chapter. The subsequent chapters will build upon some of the topics that we have seen here.

10. Configuration & Communication

This chapter covers

- Static Portlet Preferences
 - Dynamic Portlet Preferences
 - Reading from “properties” files
 - Continuation of Firing Emails from Portlets
 - Portlet Data Handlers – Export / Import
 - Inter-Portlet Communication
 - Non-Standard ways of IPC
 - Portlet URL Invocation
 - Customizing Portlet Based on Query String
-

Welcome to Chapter 10. We have spent good amount of time in the last chapter learning so many good things. All the topics we cover in this book are very relevant to the real world requirements. They have been churned out of my experience working with more than two dozen major Liferay implementation in the last eight years. The topics have been hand picked to ensure that they benefit you to the maximum as a reader and an expert Liferay developer.

Having said this, I’m going to logically and broadly divide this chapter into two parts. The first part will cover the various ways the Portlet settings can be configured. We will see many use-cases around this subject including the email-firing program that we parked at the end of last chapter. The second part of this chapter is all about communication between two or more portlets in real time. In other words, how portlets are going to communicate between each other. This subject is called as *Inter-Portlet Communication (IPC)* or *Portlet to Portlet Communication (PPC)*. As I mentioned the two parts will be logically separated without any physical separation at least in this chapter. There are some reasons for keeping these two subjects underneath one chapter. The common denominator is data. The first part covers the data related to setting and the second part refers to the data related to behavior.

Let’s get started and have a nice journey through this fantastic chapter.

10.1 Static Portlet Preferences

Have you heard of a very famous acronym called WORA? It stands for “Write Once Run Anywhere”. I was surprised to see that there is a [Wikipedia definition](#) as well for this popular acronym in the JAVA world. How to make your Portlet apps follow the same WORA principle? This is going to be the crux of this section and few upcoming sections of this chapter. You can make your portlets completely configurable and “re-usable”, another key aspect of Object Oriented programming. This very feature makes your Portlet deployable on many portals where there is a need for a Library Management System.

In [Section 9.9 Firing Emails from Portlet](#) we have seen Portal Preferences and the corresponding table where various portal settings are persisted (*saved*). Based on these settings the runtime behavior of the portal can be modified and it is easy for the portal administrator to configure and change these settings on the fly. The same applies to each and every Portlet that is deployed inside Liferay Portal Server. The Portlet behavior and many of its attributes can be made configurable, insspite of they being hard coded inside the Portlet code. In this section we are going to discuss about the various options that are available in Liferay in order to make a Portlet highly configurable and customizable.

10.1.1 Specifying preferences via “portlet.xml”

Supplying the preferences (*settings*) for a Portlet during the time of Portlet initialization through “**portlet.xml**” is part of the original Portlet 2.0 specification. The specification has defined an interface exclusively for this purpose, [javax.portlet.PortletPreferences](#). This is something not new to us. We have already set some preferences for our Library Portlet in its deployment descriptor. A preference is supplied with the help of “`<init-param>`” tag.

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

By default Liferay keeps the following preferences in memory for every instance of the Portlet added to a page. You can have a look into [MVCPortlet.java](#) for details.

<code>about-template</code>	<code>config-template</code>
<code>edit-template</code>	<code>edit-default-template</code>
<code>edit-guest-template</code>	<code>help-template</code>
<code>preview-template</code>	<code>print-template</code>
<code>view-template</code>	<code>clear-request-parameters</code>
<code>copy-request-parameters</code>	<code>add-process-action-success-action</code>

“`copy-request-parameters`” – Used to copy the request parameter from the request to the response. This is useful in cases where you do validation on a form and wish to repopulate the form with the input value in case of errors. Default value is false.

“`add-process-action-success-action`” – set to false will disable Liferay status messages. We’ll do a quick exercise overriding this preference. Default value is true.

Apart from these default preferences; you can set any number of preferences in “`portlet.xml`” using “`<init-param>`” tag as per our requirement. Whatever is being set here can be retrieved in our Portlet class using “`getInitParameter(String name)`” method. There is also another method “`getInitParameterNames()`” that return an “`Enumeration<String>`” of all preferences that are available for a given Portlet. Technically when a Portlet gets loaded inside the Portlet container these preferences are set for the Portlet when the “`init()`” method is invoked.

10.1.2 Disabling “success” messages post an Action

In this sub-section, we are going to do a quick exercise of disabling the success message after every action. I am sure you would have already observed it. In our Portlet after every action the message “**Your request completed successfully.**” gets displayed. This happens by default if we use Liferay’s MVCPortlet. In certain applications this may not be a desired feature. How to disable this, so that we may put our custom message based on the action that is being performed. There are two ways to disable the default message.

Method 1: The first option is by explicitly adding the “`<init-param>`” in “`portlet.xml`” overriding the default value.

```
<init-param>
    <name>add-process-action-success-action</name>
    <value>false</value>
</init-param>
```

Method 2: The second option is to override the “`init()`” method in the Portlet class, `LibraryPortlet.java` and reset the value of “`addProcessActionSuccessMessage`” boolean attribute to false.

```
public void init() throws PortletException {
    super.init();
    addProcessActionSuccessMessage = false;
}
```

Do it Yourself: Add a new “`<init-param>`” as below and try to access the same inside your Portlet class.

```
<name>max-books-limit</name>
<value>2000</value>
```

Hint: Inside your Portlet you will access this preference like,

```
int maxBooksLimit = GetterUtil.getInteger(
    getInitParameter("max-books-limit"));
```

`GetterUtil` is another great utility of Liferay that helps us to retrieve any property or parameter in its primitive form. You can find out all other API’s of this class.

10.1.3 Accessing a preference inside JSP

If you have to access the same “preference” inside any of the portlet’s JSP files, how to do that? Here is the answer. In Section [6.6.2 Objects injected by “liferay-theme” tag](#) we have seen that Portlet 2.0 injects the object “portletConfig”. We can just get the preferences value from this object. Let’s put this code at the end of “view.jsp” to check whether it is returning the proper value.

```
<h1>Limit:<%=portletConfig.getInitParameter( "max-books-limit" )%></h1>
```

Note: After adding preferences to “portlet.xml” it may so happen that it is not getting immediately reflected either inside the Portlet class or inside the JSP files. The best way is to remove the Portlet from the page and add it afresh again.

10.2 Dynamic Portlet Preferences

In the static way of setting the Portlet preferences through “portlet.xml” there are limitations. The primary limitation is we can’t modify the preference during runtime. Say for example, each Library instance that is deployed for one particular department, we want to set some preferences. We’ll not be able to do it with the previous approach. Hence, we need a dynamic way of setting the preferences on every instance of the Portlet so that the behavior of that particular Portlet instance can be customized and configured by the portal administration himself.

Liferay has implemented the [PortletPreferences](#) interface inorder to make the settings dynamic in nature. Usually the preferences for a Portlet are set through the edit mode for that Portlet. The dynamic approach enables the portal administrators to change a preference during runtime on the fly. Sounds cool? Yes, it is! You’re going to see this in action very soon.

10.2.1 Dynamic Preferences through edit mode

Let’s do a quick exercise where by we’ll make “max-books-limit” modifiable per Portlet instance. This will involve four steps.

Step 1: The first step is to enable to edit mode for the Portlet. This can be done either at the time of creating the Portlet through Eclipse IDE or manually making the corresponding entries in “portlet.xml”. There are two changes to be made in this file.

1. Add a “`<init-param>`” param, “edit-template” with value “/html/library/edit.jsp”.
2. Insert “`<portlet-mode>edit</portlet-mode>`” inside “`<supports>`” tag.

The moment these two changes are done and the file is saved check the “Options” menu for the protlet. A new item “Preferences” will appear in the list as shown here.



Click this link, but unfortunately the Portlet shows an error, as we have still not created the “**edit.jsp**” that will get displayed during the edit mode of the Portlet. We’ll create this file in the next step.

Step 2: Create “**edit.jsp**” inside “**html/library**” with the following contents.

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

<h1>Portlet Settings</h1>

<portlet:actionURL var="setPreferencesURL" name="setPreferences" />

<aui:form action="<% setPreferencesURL.toString() %>">
    <aui:input name="maxBooksLimit" />
    <aui:button type="submit" value="Set Preferences" />
</aui:form>
```

When you save this file and check the “Preferences” link, the form will appear. The auto-label for the input field, “max-books-limit” should have a corresponding entry in “**Language.properties**” file. Just add an entry there and check the Portlet.

```
max-books-limit=Max Capacity of the Library
```

Submit the form and you will get an error as we have still not written the code for “**setPreferences**” method in our Portlet class – **LibraryPortlet.java**. Let’s do that in our next step.

Step 3: Define a new `processAction` method “`setPreferences`” in “**LibraryPortlet.java**” as below to save the preferences to the database.

```
public void setPreferences(ActionRequest actionRequest,
                           ActionResponse actionResponse)
                           throws IOException, PortletException {

    String maxBooksLimit =
        ParamUtil.getString(actionRequest, "maxBooksLimit");

    PortletPreferences preferences =
        actionRequest.getPreferences();
    preferences.setValue("maxBooksLimit", maxBooksLimit);
    preferences.store();
}
```

Make the necessary import.

```
import javax.portlet.PortletPreferences;
```

That’s it. Save the changes and go to Preferences page to set a value for “`maxBooksLimit`” through the UI. After submitting the form, go directly to the database to check the “**portletpreferences**” table. You will see the “preferences” column for the record corresponding to this Portlet getting updated with the value we just saved. Copy and paste contents of this column into an XML file in Eclipse and do formatting by pressing **Ctrl+Shift+F** (*this is purely for the purpose of testing*).

```

<portlet-preferences>
    <preference>
        <name>lfrWapInitialWindowState</name>
        <value>NORMAL</value>
    </preference>
    <preference>
        <name>portletSetupUseCustomTitle</name>
        <value>false</value>
    </preference>
    <preference>
        <name>portletSetupCss</name>
        <value>{&#034;wapData&#034;}.....</value>
    </preference>
    <preference>
        <name>portletSetupShowBorders</name>
        <value>true</value>
    </preference>
    <preference>
        <name>maxBooksLimit</name>
        <value>3000</value>
    </preference>
</portlet-preferences>

```

Step 4: In this step we are going to see how to retrieve this value back in our JSP file and make it appear inside the input field by default. Go back to “edit.jsp” and add “`<%= maxBooksLimit %>`” for the “`value`” attribute. This variable has to be defined earlier inside scriptlet of this JSP. Here “`portletPreferences`” is an object injected by “`<portlet:defineObjects/>`”, something very similar to “`portletConfig`” used to retrieve a static preference.

```

String maxBooksLimit =
    portletPreferences.getValue( "maxBooksLimit" , " " );

```

Save the changes and let’s check the Portlet now. The current value for this preference should appear in the input field of the form. Great! If you have done till this point, there is only one more optional thing to do. How about taking the Portlet back to “view” mode as soon as the setting is saved. Append this line at the end of “`setPreferences`” action method to do this for you.

```

actionResponse.setPortletMode( PortletMode.VIEW );

```

Make the required import.

```

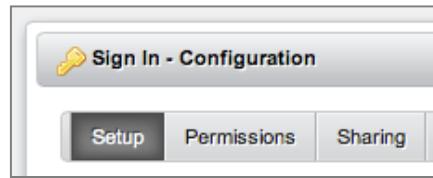
import javax.portlet.PortletMode;

```

Code Changes @ <http://code.google.com/p/lr-book/source/detail?r=48>

10.2.2 Setting preferences the Liferay Way

Have you ever observed in some of the Liferay's default portlets there is a "Setup" tab when you go to the portlet's configuration from "Options" menu? Take the example of our Sign-in Portlet on the Welcome page.

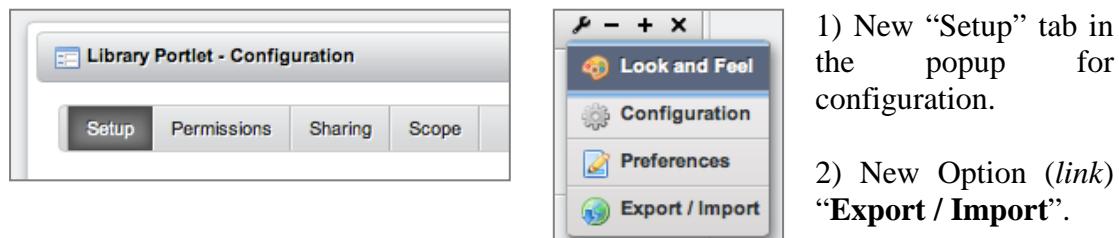


Under the Setup tab there are various options to set the runtime preferences for that Portlet. There is also an option "[Archive/Restore Setup](#)" that we are going to see very soon. How to get the Setup tab for our Library Portlet? This is going to be the subject matter for this sub-section. This time let's introduce a new preference, a custom name for the Library called "libraryName". Since this is a Liferay extension we'll start with changes in "**liferay-portlet.xml**".

Step 1: Insert the below line in "**liferay-portlet.xml**" between "`</icon>`" and "`<scheduler-entry>`" tags.

```
<configuration-action-class>
    com.liferay.portal.kernel.portlet.DefaultConfigurationAction
</configuration-action-class>
```

The entry for "`<configuration-action-class>`" should either implement the interface [ConfigurationAction](#) or extends the class [DefaultConfigurationAction](#). Since we are not writing a new class, let's use DefaultConfigurationAction itself as the value for this entry. The moment you put this entry and save this file, you will observe two changes in the Portlet. Both of them are visually shown below.



Step 2: Open "**portlet.xml**" and define a new "`<init-param>`" with "`<name>`" as "config-template" and "`<value>`" as "/html/library/config.jsp". Similar to what we did for enabling edit mode, let's enable config mode by appending "`<portlet-mode>config</portlet-mode>`" inside "`<supports>`" tag (*Seems like things work even without this entry. May be you can verify this fact*). After saving, let's physically create the new file, "**config.jsp**" under "**html/library**" and put the following contents.

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

<%@ taglib uri="http://liferay.com/tld/portlet"
           prefix="liferay-portlet" %>

<liferay-portlet:actionURL var="configurationURL"
                           portletConfiguration="true" />

<aui:form action="<%=" configurationURL.toString() %>">
```

```

<aui:input type="hidden" name="<%= Constants.CMD %>" value="<%= Constants.UPDATE %>" />
<aui:input name="preferences--libraryName--" />
<aui:button type="submit" value="Save Settings"/>
</aui:form>

```

In the above code, please note that we have used a special tag to create the configurationURL. If you want you can give the taglib URI entry inside “**init.jsp**”. Make an entry for key “**library-name**” in “**Language.properties**”. Actually, you can avoid changes to “**portlet.xml**” if you create a file by name “**configuration.jsp**” directly under “**docroot**” with the above contents. There are two important things that you have to note here.

- 1) Inside the form you have to define a hidden field. Without this the preference will not be saved to the portletPreferences table.
- 2) Any preference that you want to set should be prefixed by “**preferences--**” and suffixed by “**--**”. This will help the processAction method of the ConfigurationAction class to process only these fields and properly store them to the database.

Make the necessary import either within the same JSP or inside “**init.jsp**”.

```
<%@page import="com.liferay.portal.kernel.util.Constants" %>
```

Save the changes and check the configuration page now. When you enter a Library Name in the configuration page and click “Save Settings”, the value should properly go and sit inside the “**preferences**” column for the appropriate record in “**portletPreferences**” table. This is awesome!

Step 3: We need to show the Library Name appearing by default. Let’s try pulling this preference from “**portletPreferences**” as we did before and set this as the “**value**” attribute for the input field (*these changes are to be done in “config.jsp”*).

```
String libraryName = portletPreferences.getValue("libraryName", "");
```

To our surprise, this does not seem to work as Liferay treats the config settings quite differently. Then what else is the alternative way to pull this data? Let’s try this.

```
PortletPreferences preferences = renderRequest.getPreferences();
String libraryName = preferences.getValue("libraryName", "");
```

Unfortunately even this does not seem to work. Ok, let’s try with “**portletResource**” object. Have this code inside the scriptlet.

```
PortletPreferences preferences = renderRequest.getPreferences();
String portletResource = ParamUtil.getString(
    request, "portletResource");
if (Validator.isNotNull(portletResource)) {
    preferences = PortletPreferencesFactoryUtil.getPortletSetup(
        request, portletResource);
}
String libraryName = preferences.getValue("libraryName", "");
```

Note: Infact you can move this entire block, except the last line to “`init.jsp`” so that the proper “`preferences`” object is readily available in all JSP files than just in our “`config.jsp`”. In any case, you need the addional imports.

```
<%@page import="javax.portlet.PortletPreferences"%>
<%@page import="com.liferay.portlet.PortletPreferencesFactoryUtil"%>
```

This worked perfectly fine and previously saved value is showing up well. Congratulations! You have successfully implemented the Liferay’s way of setting the various configurations for a Portlet. Let’s do a quick experiment. Go to the portal and drag and drop an “Asset Publisher” Portlet on to the page. If it does not appear for the first time, just refresh the page. When the Portlet is there, click on the Configuration option. Do you see a complex form under the “Setup” tab? This is how the configurations for a real world Portlet will look like. You see there are six sections – Source, Filter, Custer User Attributes, Ordering and Grouping, Display Settings and RSS. While writing such complex configuration settings, we have to override `DefaultConfigurationAction` class and write our own.

[PortletPreferencesFactoryUtil](#) provides many convenience methods that help us to play around with both portal preferences and Portlet preferences. You can click on the link to open its apidocs.

Code Changes @ <http://code.google.com/p/lr-book/source/detail?r=49>

Do it Yourself: Here is a challenge for you. Drag and drop a “Web Content Display” Portlet on your page. The moment the Portlet gets added to the page, it is displayed like this with the message – *Select existing web content or add some web contents to be displayed in this Portlet.*



You select one of the existing web contents, this message goes off and the actual web content is displayed here. Can you figure out how this is done?

This is a very good exercise to know how Liferay has extensively used Configuration action. Document your findings as I am going to ask you more questions as we move forward.

10.2.3 Archival and Restoration of Preferences (settings)

Before we could end this section, I would like to bring your attention to another great feature of Liferay “Archival and Restoration”. This is something you’re going to try out by clicking the link “[Archive/Restore Setup](#)” when you’re in the “Setup” tab of Portlet Configuration. This is some kind of versioning for the Portlet setup.

Name	User	Modified Date
Setup1	Test Test	3/12/13 3:13 AM

Showing 1 result.

Archive Name for Current Setup

Save

10.2.4 Preferences Entries in Liferay Deployment Descriptor

Apart from the various ways of configuring the preferences of a Portlet, there are some entries in “**liferay-portlet.xml**” that help in achieving some advanced preferences settings for a particular Portlet. Those entries are explained below.

preferences-company-wide: Set this value to **true** if the preferences for the Portlet are across the entire company. Setting this value to **true** means the value for **preferences-unique-per-layout** and **preferences-owned-by-group** are not used. The default value is **false**. For example, an administrator could set the preferences to an Announcements portlet that would save a message in the portlet's preferences. This message would then be used across all pages for that company.

The portlet must not be instanceable because instanceable portlets have uniquely generated portlet ids. The default behavior of the bundled Announcements portlet sets the instanceable value to true so that normal users cannot create company wide messages. A future release would include permissions for the edit mode versus the view mode that would allow an administrator to set the message while users would just view the message.

preferences-owned-by-group: Set this value to **true** if the group owns the preferences for the portlet when the portlet is shown in a group page. If set to **false**, the preferences are owned by the user at all times. The default value is **true**. Suppose the Stocks portlet has **preferences-unique-per-layout** set to true and **preferences-owned-by-group** set to **false**. Users can set a different list of stocks for every personal page. Users can set a different list of stocks for every community page.

Suppose the Stocks portlet has **preferences-unique-per-layout** set to **false** and **preferences-owned-by-group** set to **false**. Users can set one list of stocks to be shared across all personal pages. Users can set one list of stocks to be shared across a community's set of pages. Suppose the Stocks portlet has **preferences-unique-per-layout** set to true and **preferences-owned-by-group** set to true. Users can set a different list of stocks for every personal page. Administrators set the portlet preferences for users in a community page. Administrators can set a different list of stocks for every community page that are then shared by all users within a community. Suppose the Stocks portlet has **preferences-unique-per-layout** set to **false** and **preferences-owned-by-group** set to **true**. Users can set one list of stocks to

be shared across all personal pages. Administrators set the portlet preferences for users in a community page. Administrators can set one list of stocks to be shared by all users across a community's set of pages.

preferences-unique-per-layout: Set this value to **true** if the preferences for the portlet are unique across all pages. If set to **false**, the preferences for the portlet are shared across all pages. The default value is **true**. The **preferences-unique-per-layout** element is used in combination with the **preferences-owned-by-group** element.

The following table explains the above facts in a subtle way.

Set	company-wide	owned-by-group	unique-per-layout	Portlet Behaviour
1	True	Not Used	Not Used	Preferences for the portlet are across the entire company. You set the preferences in one place and all the instances of the same Portlet get impacted.
2	False	True	False	The group owns the preferences for the portlet when the portlet is shown in a group page. Users can set one set of preferences to be shared across all personal pages. Administrators set the portlet preferences for users in a community page. Administrators can set one set of preferences to be shared by all users across a community's set of pages.
3	False	True	True	Users can set a different set of preferences for every personal page. Administrators set the portlet preferences for users in a community page. Administrators can set a different set of preferences for every community page that are then shared by all users within a community (site).
4	False	False	True	The user owns the preferences at all times. Users can set a different set of preferences for every personal page and for every community (site) page.
5	False	False	False	Users can set one set of preferences to be shared across all personal pages. Users can set one set of preferences to be shared across a community's set of pages (site).

There are four columns in the “portletpreferences” table that store the preferences differently based on the above settings. You can actually create a sample Portlet to play around with these settings in “**liferay-portlet.xml**” and check how the records in the table get impacted.

portletPreferencesId	ownerId	ownerType	plid	portletId	preferences
	NULL	NULL	NULL	NULL	NULL

10.3 Reading from “properties” files

We are going to keep this section short, as you’re already familiar with “**portal.properties**” and overriding the entries in this file through one of its extension variants listed here. As a side note, each portal instance can have its own overridden property file following the convention portal-companyWebId.properties. To enable this feature, set the “**company-id-properties**” system property to true in “**system-ext.properties**” file. This file contains all Java related settings and is an offshoot of “**system.properties**” which is a sibling of “**portal.properties**”.

<code>portal-bundle.properties</code>	<code>portal-ext.properties</code>
<code>portal-setup-wizard.properties</code>	<code>portal-{companyWebId}.properties</code>

A string constant inside an interface `com.liferay.portal.kernel.util.PropsKeys` denotes every property defined in “**portal.properties**” file, so that they can be uniformly used and accessed inside the java and JSP code. Now the real question is how to access the value for these keys inside of our Portlet. I think we have already seen one example of this in the previous chapter. Here is one example.

```
String blogsEmailFromName =  
    PropsUtil.get(PropsKeys.BLOGS_EMAIL_FROM_NAME);
```

This is how you will retrieve any value from “**portal*.properties**” inside your Portlet and perform the business logic accordingly. The utility class “**PropsUtil**” provides many convenient methods than just a simple “**get**” method. You can find out all other methods of this class with the help of autosuggestion feature.

Similar to the entries in “**portal.properties**”, for a plugin project, we usually keep a “**portlet.properties**” file under “**WEB-INF/src**” where all the configuration parameters required for the various portlets of that plugin are kept. The entries could literally be anything that usually will not change over different environments. An example of an entry could be a comma-separated values (CSV) of text used to render a select drop-down in one of the user entry forms in our application. Let’s quickly do this here following these three steps.

Step 1: Create “**portlet.properties**” under “**WEB-INF/src**” and make an entry there.

```
book.types=novel,science,fiction,spiritual
```

Step 2: Inside “**LibraryConstants.java**” create a static string referring to this key

```
static final String PROP_BOOK_TYPES = "book.types";
```

Step 3: Inside a Java class or JSP of your portlets within the plugin, use the following code to retrieve the value with the help of PortletProps class.

```
String[] bookTypes = PortletProps.getArray(  
    LibraryConstants.PROP_BOOK_TYPES);
```

10.4 Firing Emails from Portlet – Continued

Do you remember we have parked this topic for a little while at the end of last chapter? If you remember right it was Section [9.9 Firing Emails from Portlet](#). Two major reasons made us to park the topic there and taken it back here – 1) the previous chapter was growing really lengthy breaking our original plan of keeping it short and sweet, 2) I felt it is more appropriate to cover the topics in the context of the current chapter as it is all about externalizing the email contents. In other words, how to make the contents for the email configurable from outside instead of hard coding inside the java program itself? Sounds reasonable? It ought to be.

Let's take some excerpts from the original class “NotifyDefaultingMembers” and move forward accordingly.

10.4.1 Replacing the values for Subject Line

We have a parameter “String subjectTemplate” to “notifyMember” method. We are setting this as the subject of the email that is being sent to a defaulting member.

```
mailMessage.setSubject(subjectTemplate);
```

Ideally the subjectTemplate has to be pulled from the “**portlet.properties**” file inside this method and actual values replaced. Let's do this change in next couple of steps.

Step 1: Make an entry in “**portlet.properties**” file and a corresponding entry in “**LibraryConstants.java**” respectively as shown below.

```
defaulting.reminder.email.subject=\n    Book borrowed by you - [{BOOK_TITLE}], is overdue.  
  
static final String PROP_DEFAULTING_REMINDER_EMAIL SUBJECT =\n    "defaulting.reminder.email.subject";
```

Step 2: Inside the “notifyMember” method, read this entry from the properties file and replace the template with actual value for “[{BOOK_TITLE}]”. We are going to use another powerful String manipulation utility of Liferay for this purpose.

```
// getting the original book object and bookTitle.  
String bookTitle = null;  
try {  
    LMSBook lmsBook =  
        LMSBookLocalServiceUtil.fetchLMSBook(borrowing.getBookId());  
    bookTitle = lmsBook.getBookTitle();
```

```

} catch (SystemException e1) {
    e1.printStackTrace();
}

// forming the actual subject line for the email
String subjectTemplate = PortletProps.get(
    LibraryConstants.PROP_DEFAULTING_REMINDER_EMAIL_SUBJECT);
String subjectLine =
    StringUtil.replace(subjectTemplate,
        "[${BOOK_TITLE$}", bookTitle);
mailMessage.setSubject(subjectLine);

```

You're done. Using "StringUtil" we have set the actual subject line before firing the email. In the process of doing the above changes, I've actually changed the signature of the method as below as it is no longer required to pass the subject and body templates as parameters to this method.

```
private void notifyMember(LMSBorrowing borrowing)
```

Great! Subject line is fine. What about the mail body which again is a template. Is it a good idea to put this in "**portlet.properties**" as well? Let's see next.

10.4.2 Mail Body template – Use ContentUtil

We could have followed the similar approach for the email body as well; very similar to how we did for the subject line. But the only problem is the content is going to be a big chunk of HTML template with multiple replacements during runtime. It is going to be very cumbersome if we put this whole lot of contents as a key-value pair in properties file. Modifying the contents is also going to be challenging. To cater to such situations, Liferay comes for our rescue. It provides another powerful API that helps in reading a file directly from the file system. Let's store our properly formatted HTML email template inside "**WEB-INF/src/template**" and name it as "**reminder-email tmpl**", the extension denoting a template file. The content of this file will look like the below, but stored in HTML format. (*For readability the HTML tags are removed.*)

```

Hi [${MEMBER_NAME$} ,

You have borrowed a book on [${DATE_BORROWED$}].

Your actual return date is [[${RETURN_DATE$}]] which is overdue.

Request to return the book at the earliest to avoid penalty.

```

Inside the method, we'll read this content using ContentUtil API like below.

```

// preparing the email body
String messageBody =
    ContentUtil.get("/template/reminder-email tmpl");
String[] tokens =
    {[${MEMBER_NAME$}], [${DATE_BORROWED$}], [${RETURN_DATE$}]};

DateFormat dateFormat = new SimpleDateFormat("MM/dd/yyyy");

```

```

String dateBorrowed =
        dateFormat.format(borrowing.getDateBorrowed());
String dateOfReturn =
        dateFormat.format(borrowing.getDateOfReturn());
String[] values =
{member.getFullName(), dateBorrowed, dateOfReturn};

messageBody = StringUtil.replace(messageBody, tokens, values);
mailMessage.setBody(messageBody);

```

You can check this program and confirm that the template is being properly read from the file system and the tokens are replaced by the actual values before set as the body for “`mailMessage`”. `ContentUtil` is a great API. Using it, you can even pull some files / templates that are stored at the portal level. For this scenario you have to pass the portal class loader as the first parameter to its “`get`” method. The biggest limitation is in editing the content that is sitting in the file system. It is not going to be that easy especially if the application is already live. To mitigate the limitations of this mechanism, we at mPower Global use a very holistic method in some of our major Liferay implementations. .

10.4.3 Storing email template as a Web Content

How do the Librarians feel, if they are given the option to manage the various email templates and decorating them nicely on the fly without the intervention of any developer from our IT team? This approach is going to provide them the level of flexibility. This is something very similar to how you manage some of the email templates when you login to the portal as portal administrator and go to **Control Panel → Portal → Portal Settings → Email Notifications**. We’ll follow the same principle, but this time we are going to create Web Content with Id, “`RETURN_DATE_REMINDER_EMAIL_TEMPLATE`” with some nice and effective content.

Inside your “`notifyMember`” method, you will fetch the HTMT template by making use of the following code. Once the template is available, you can make the token replacements as usual using “`StringUtil`”.

```

private String getEmailTemplate(String articleId, User member) {

    String emailTemplate = StringPool.BLANK;
    DynamicQuery dynamicQuery =
        DynamicQueryFactoryUtil.forClass(JournalArticle.class,
            PortalClassLoaderUtil.getClassLoader());

    dynamicQuery.add(RestrictionsFactoryUtil.eq(
        "companyId", member.getCompanyId()));
    dynamicQuery.add(RestrictionsFactoryUtil.eq(
        "articleId", articleId));

    List<JournalArticle> articles = null;
    try {
        articles =
            JournalArticleLocalServiceUtil.dynamicQuery(dynamicQuery);
    } catch (SystemException e) {
        e.printStackTrace();
    }
}

```

```

        }

        JournalArticle journalArticle = null;
        if (Validator.isNotNull(articles) && !articles.isEmpty()) {
            journalArticle = articles.get(0);
        }
        if (Validator.isNull(journalArticle)) return emailTemplate;

        try {
            emailTemplate =
                JournalArticleLocalServiceUtil.getArticleContent(
                    journalArticle, null, null,
                    member.getLanguageId(), null);
        } catch (PortalException e) {
            e.printStackTrace();
        } catch (SystemException e) {
            e.printStackTrace();
        }

        return emailTemplate;
    }
}

```

I prefer to write this as another private method within the same class. If you give a closer look, the first of this method we are forming is a “dynamicQuery” to get the appropriate “journalArticle” based on “articleId”. The second part of the method we are getting the HTML article content using “getArticleContent” method. We have to use “dynamicQuery” as there is no API in “JournalArticleLocalServiceUtil” to directly get an article based on a valid “articleId”. We can actually inject this method by overriding this service using a “Service Hook” which unfortunately is not in the scope of this book.

10.4.4 SMTP settings for actual email

For real delivery of emails to the member’s inbox you need a real SMTP server. Usually in a development environment you will not have access to a real SMTP relay server. Since you have to really test the actual delivery of email, we have a work-around if you have an account in gmail. Open your “**portal-setup-wizard.properties**”, append these entries with proper values and restart the server.

```

mail.session.mail.smtp.host=smtp.gmail.com
mail.session.mail.smtp.password=<your password>
mail.session.mail.smtp.user=<your gmail address>
mail.session.mail.smtp.port=465
mail.session.mail.smtp.auth=true
mail.session.mail.smtp.starttls.enable=true
mail.session.mail.socketFactory.class=javax.net.ssl.SSLSocketFactory

```

To verify that the SMTP settings are working fine, you just create a new account in the portal with your own valid email Id and check if the email is getting delivered.

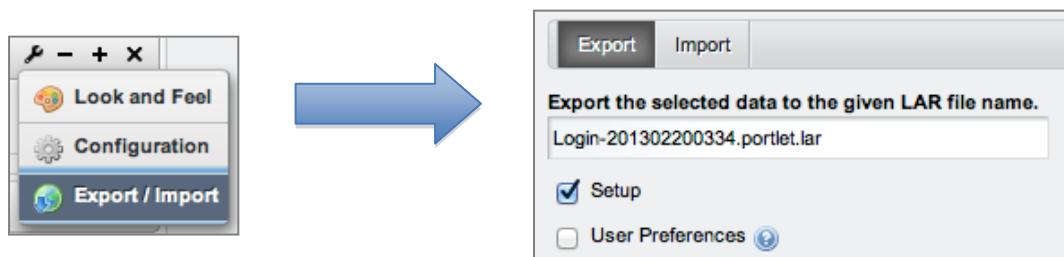
Code Changes @ <http://code.google.com/p/lr-book/source/detail?r=51>

10.5 Portlet Data Handlers – Export / Import

In Section [10.2.2 Setting preferences the Liferay Way](#), the moment we insert the tags for Configuration Action class in ‘**liferay-portlet.xml**’ a new link starts appearing in the options pull-down menu of the Portlet. Have you ever wondered what this option is meant for? Let’s set the stage with our own Signin Portlet lying in our welcome page and give some theoretical background to this scintillating feature of Liferay. No doubt these kind of features have made Liferay the world’s most loved and admired portal framework. The developers love it and users admire it.

10.5.1 Export / Import on Signin Portlet

Login to the portal (*if you’re not logged in already*) and go to the Export / Import option of the Signin Portlet. Liferay takes you to another page as shown here.



Just with the option “Setup” checked, click “Export” to export the settings of this Portlet in the form of a LAR file. LAR originally stands for “Liferay Archive”, a file that is very similar in nature to a JAR (*Java Archive*) or a WAR (*Web Archive*) file. Let’s inflate this file and see its contents. I used the following command in the prompt to do the inflation of this LAR that is nothing but a compressed format of some files.

```
jar xvf Login-201302200400.portlet.lar
```

The command ran successfully producing the following output.

```
inflated: groups/10179/portlets/58/preferences/layout/0/10611/portlet-preferences.xml
inflated: groups/10179/portlets/58/10611/portlet.xml
inflated: groups/10179/links.xml
inflated: groups/10179/comments.xml
inflated: groups/10179/expando-tables.xml
inflated: groups/10179/locks.xml
inflated: groups/10179/ratings.xml
inflated: manifest.xml
```

You see a list of XML files. Let’s open these files and check what they contain. But before we do that, let’s know what these numbers “**10179**”, “**58**” and “**10611**” mean. Let me tell you instead of keeping it suspense as we have many more things to cover in this section. “**10179**” refers to the groupId. Whatever the value you’re getting while doing this task, you can verify it with “group_” table in the database. “**58**” refers to the portletId of the Portlet for which we have exported the settings. “**10611**” refers to

the “plid” (*page layout id*) of the page where the original Portlet was lying. You can confirm this by checking with the “layout” table and its “plid” column. Now let’s go in the details of every file that was originally a part of the exported LAR file. (*For your convenience, I’ve even checked in the whole “groups” folder to our SVN repository and the contents can be browsed by hitting [this link](#).*).

For the time being let’s ignore we are directly inside “**10179**” folder. We have to focus on the “**portlet.xml**” that contains the complete packaging information about the Portlet. It has the reference to “**portlet-preferences.xml**” that has the details about the Portlet settings. In essence, a LAR file is nothing but a completely packaged version of a Portlet with its data, settings, comments, ratings, links and locks. Liferay has this kind of packaging mechanism in three different levels – a site level, a page level and a Portlet level. What we are doing at the moment is packaging at the Portlet level. The packaging details of other two levels are not in the scope of this book and hence not covered.

10.5.2 Export / Import on Library Portlet

We have got a fair understanding of the export feature. What about Import? You’re going to try this out in our own Library Portlet. I am dedicating this sub-section for you to complete this task without getting into any other details of Liferay’s style of packaging the information / data. There are two parts to this exercise. For the Library Portlet staying in the “My Library” page, set proper values for both “max-books-limit” and “libraryName”. The first one is set through “Preferences” option and the other one is set by going to “configuration → Setup”. Once both these values are properly set, go to “Export / Import” and Export the “Setup” and save the resultant LAR file in your preferred location.

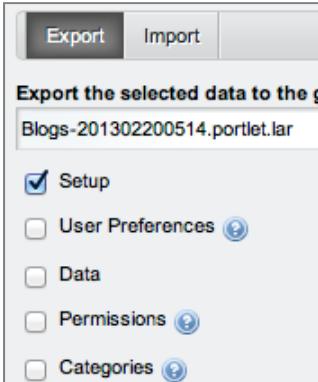
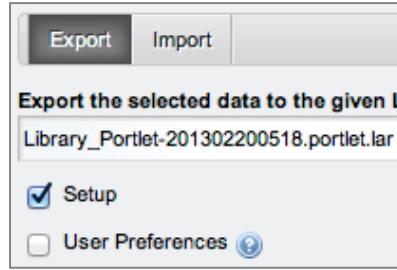
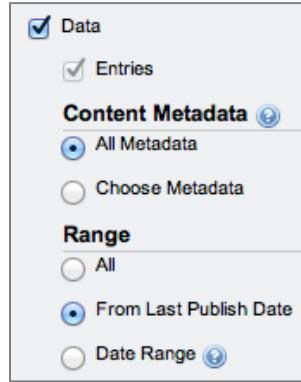
Coming to the second part, create another page by name “Library 2” (*if it is not there already*) and drag & drop the Library Portlet. Once the Portlet is available in the page, click the “Export / Import” option and import the LAR file originally exported from the other Portlet. After successful import, check the values for “max-books-limit” and “libraryName” for this instance of Library Portlet. They’ll be exactly the same like what is there for the original Portlet. If you have done this exercise, then I am sure you have got a fair idea about the whole concept. Now coming to the most important questions, what is the significance of this packaging mechanism?

The LAR files are extremely powerful for backing up the Portlet data or it’s settings. They are also used to share the settings and data between various environments. Say for example, the developer has created some sample data that he wants to send to his colleague in a different geographical location. Instead of sending the whole database , it will suffice if he just sends the LAR file and the person at the other end does an import and reproduces the exact settings and data of the original guy.

10.5.3 Packaging Portlet Data

I don’t know if you have tried Export / Import with some of the Portlets that come bundled with Liferay. Examples of these protlets are Blogs, Message Boards and

Document Library. When you do an Export and then Import from these portlets, not only the settings but also the data gets properly transferred between the source and target. The “Export / Import” page itself looks differently for these portlets with more options the significant one being the option to choose “Data” in the form of a checkbox. The following table shows the “Export / Import” pages of both Blogs Portlet and our Library Portlet.

Blog Portlet	Library Portlet	Checking “Data”
		

The Blog portlet’s Export / Import page has got three additional options when compared to that of Library Portlet. They are “**data**”, “**permissions**” and “**Categories**”. When you check “**data**”, more options are shown as you see above. How to achieve exactly the same thing for our Library Portlet, so that the Portlet can be packaged with all its data, the primary entities and its dependencies? This is going to be the discussion for rest of this section. Let’s start! In the process we are going to get introduced to Portlet Data Handlers. We’ll follow four steps to reach our goal.

Step 1: Let’s begin by creating a new class “**LibraryPortletDataHandler**” under “**com.library.lar**” package, extending [BasePortletDataHandler](#). Once the class is created with empty body, make its appropriate entry in “**liferay-portlet.xml**”. The original contents of the new class will look like below.

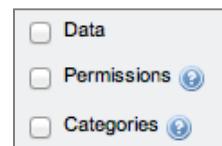
```
package com.library.lar;
import com.liferay.portal.kernel.lar.BasePortletDataHandler;

public class LibraryPortletDataHandler
    extends BasePortletDataHandler {
```

Insert this entry in “**liferay-portlet.xml**” between “`<friendly-url-routes>`” and “`<control-panel-entry-category>`” entries.

```
<portlet-data-handler-class>
    com.library.lar.LibraryPortletDataHandler
</portlet-data-handler-class>
```

The moment you save the changes and check the Portlet’s “Export / Import” page, the additional three options appear. You can hover on the icon with a question mark to know more about these options. Check “**data**” and more options underneath will be shown.



Step 2: With the basic model in place, we are now going to write the implementation for exporting the actual data – all the books that are originally part of this Library Portlet instance and their dependent data to any level. We'll start with overriding three methods from the base class and declaring one static variable in the same class.

```
public boolean isAlwaysExportable() {
    return true;
}

public boolean isPublishToLiveByDefault() {
    return true;
}

public PortletDataHandlerControl[] getExportControls() {
    PortletDataHandlerBoolean _books =
        new PortletDataHandlerBoolean(
            _NAMESPACE, "books", true, true);
    return new PortletDataHandlerControl[] { _books };
}

private static final String _NAMESPACE = "lms";
```

The above code will require the following additional imports.

```
import com.liferay.portal.kernel.lar.PortletDataHandlerBoolean;
import com.liferay.portal.kernel.lar.PortletDataHandlerControl;
```

Next, we'll override the most important method that does the actual exporting job.

```
protected String doExportData(PortletDataContext portletDataContext,
    String portletId, PortletPreferences portletPreferences)
    throws Exception {

    long companyId = portletDataContext.getCompanyId();
    long groupId = portletDataContext.getScopeGroupId();
    List<LMSBook> books =
        LMSBookLocalServiceUtil.getLibraryBooks(companyId, groupId);

    if (Validator.isNull(books) || books.isEmpty())
        return StringPool.BLANK;

    Document document = SAXReaderUtil.createDocument();
    Element rootElement = document.addElement("library-data");
    rootElement.addAttribute("group-id", Long.toString(groupId));
    for (LMSBook lmsBook : books) {
        exportBook(portletDataContext, rootElement, lmsBook);
    }

    return document.formattedString();
}
```

The above method calls another private method “`exportBook`” which appends a book node to the XML “`document`”. Let's write that function too.

```
private void exportBook(PortletDataContext portletDataContext,
    Element rootElement, LMSBook lmsBook) {
```

```

// each book data is represented by a XML
StringBuilder path = new StringBuilder();
path.append(portletDataContext.getPortletPath(_NAMESPACE));
path.append("/books/");
path.append(lmsBook.getBookId());
path.append(".xml");

Element element = rootElement.addElement("book");
try {
    portletDataContext.addClassModel(
        element, path.toString(), lmsBook, _NAMESPACE);
} catch (PortalException e) {
    e.printStackTrace();
} catch (SystemException e) {
    e.printStackTrace();
}
}

```

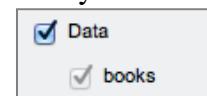
Make all additional imports that are required by these two new methods.

```

import java.util.List;
import javax.portlet.PortletPreferences;
import com.liferay.portal.kernel.exception.PortalException;
import com.liferay.portal.kernel.exception.SystemException;
import com.liferay.portal.kernel.lar.PortletDataContext;
import com.liferay.portal.kernel.util.StringPool;
import com.liferay.portal.kernel.util.Validator;
import com.liferay.portal.kernel.xml.Document;
import com.liferay.portal.kernel.xml.Element;
import com.liferay.portal.kernel.xml.SAXReaderUtil;
import com.slayer.model.LMSBook;
import com.slayer.service.LMSBookLocalServiceUtil;

```

Step 3: This is the verification step for whatever we have done so far. We have successfully written the export functionality. Go back to our Library portlet's "Export / Import" and click "Export". Just observe that the Data option is checked by default. Also the option underneath "books" is checked and disabled by default. This is the result of "getExportControls" method we have written in our PortletDataHandler class.



Your exported LAR file will be inside the "downloads" folder. Run the "jar xvf" command once again to explode the contents of this LAR file and check what's inside. For me, it gave the following output (*I've not captured the inflation of other xml files – links, comments, expand-tables, locks, ratings and manifest*)

```

inflated: groups/10179/portlets/lms/books/76.xml
inflated: groups/10179/portlets/lms/books/77.xml
inflated: groups/10179/portlets/lms/books/78.xml
inflated: groups/10179/portlets/library_WAR_libraryportlet/10809/portlet-data.xml
inflated: groups/10179/portlets/library_WAR_libraryportlet/-/portlet-preferences.xml
inflated: groups/10179/portlets/library_WAR_libraryportlet/10809/portlet.xml

```

Open each of the XML files – 76.xml, 77.xml, etc. Each file has data about one book. The file "portlet-data.xml" contains the aggregation of the data in these files. The entries in this file look something like this in my case.

```

<library-data group-id="10179">
    <book path="/groups/10179/portlets/lms/books/76.xml" />
    <book path="/groups/10179/portlets/lms/books/77.xml" />
    <book path="/groups/10179/portlets/lms/books/78.xml" />
</library-data>

```

Great! You have happily done the export part. What about importing this data in the other Portlet that is sitting on the other page without any books? Let's see this next.

Step 4: The importing feature requires one more method to be overridden in our PortletDataHandler class. Let's do that and see it in action.

```

protected PortletPreferences doImportData(
    PortletDataContext portletDataContext, String portletId,
    PortletPreferences portletPreferences, String data)
throws Exception {

    Document document = SAXReaderUtil.read(data);

    if (Validator.isNull(document)) return null;

    Element rootElement = document.getRootElement();

    for (Element bookElement : rootElement.elements("book")) {
        String path = bookElement.attributeValue("path");

        if (!portletDataContext.isPathNotProcessed(path))
            continue;

        LMSBook lmsBook = (LMSBook)
            portletDataContext.getZipEntryAsObject(path);
        importBook(portletDataContext, bookElement, lmsBook);
    }
    return null;
}

```

Even at the time of import, the user can set many options. These options can be programmatically controlled in our PortletDataHandler class. Like how we overridden the “getExportControls” method we have to override “getImportControls” to control the display of these options that appear during import. At the bare minimum, let's override this method and call “getExportControls” as we would like to have the same controls as that of Export.



```

public PortletDataHandlerControl[] getImportControls() {
    return getExportControls();
}

```

The “doImportData” above method calls another private method “importBook” that will have the actual implementation for importing the book. That means take each XML and insert a new record to the database with a different “companyId” and “groupId” combination based on environment where the physical import happens.

```

private void importBook(PortletDataContext portletDataContext,
                      Element bookElement, LMSBook lmsBook) {

    // Getting the serviceContext object
    ServiceContext serviceContext =
        portletDataContext.createServiceContext(
            bookElement, lmsBook, _NAMESPACE);

    String bookTitle = lmsBook.getBookTitle();
    String author = lmsBook.getAuthor();
    LMSBook importedBook =
        LMSBookLocalServiceUtil.insertBook(
            bookTitle, author, serviceContext);
    try {
        portletDataContext.importClassedModel(
            lmsBook, importedBook, _NAMESPACE);
    } catch (PortalException e) {
        e.printStackTrace();
    } catch (SystemException e) {
        e.printStackTrace();
    }
}

```

Now, try importing the LAR into an instance of Library Portlet that is sitting inside another site, say for example Humanities or LifeSciences. The data should properly get imported. Congratulations! You have completed learning one of the finest features of Liferay. Before you move on, just have a look at the various “*PortletDataHandler” classes that are out there inside the Portal source by pressing **Ctrl+Shift+R** in Eclipse.

Code Changes @ <http://code.google.com/p/lr-book/source/detail?r=53>

10.6 Inter-Portlet Communication

By entering this section, we are shifting to the other logical part of this chapter. Hope, you remember, in the beginning of the chapter I mentioned that this chapter would be divided into two logical parts. The first part covered the various configuration and externalization strategies in Liferay. This part, the second one, will bring to us another great feature of Portlet 2.0, Inter-Portlet Communication. This concept was originally part of the Portlet 2.0 specification and Liferay has built some extentions on top of it. Before we get into the details, let's see what it means.

Portlets have partners with whom they can communicate, talk, and exchange messages either directly through the environment where they are living (browser) or through the one who is controlling them from behind (portal server). A generic name given for their peaceful co-existence inside a browser or a portlet container is called Inter-Portlet Communication (IPC). Unofficially, this kind of communication is also called as Portlet-Portlet Communication (PPC).

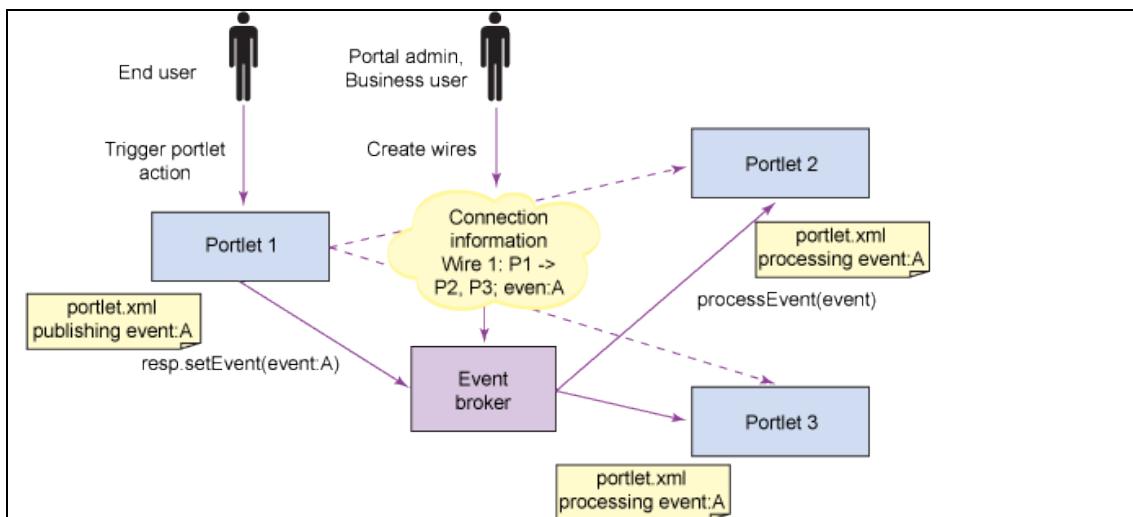
The JSR-286 or Portlet 2.0 specification defines two different means of coordination / communication between the Portlet that want to have partnership – 1) Events and 2) Public Render Parameters. On top of these two ways, we can achieve IPC through other non-standard ways. We are going to see all of them in this section and the next

section. We'll dedicate this section for standard methods of IPC and the next section for the other methods; couple of them is Liferay specific extensions. Before we get into nitty-gritties of actual implementation let's see one major benefit of IPC. It helps in increasing user experience by decoupling a bigger application into smaller parts (portlets).

In any case of communication between two different objects or entities, there is a source and a destination, trigger and a target, a publisher and a subscriber or a producer and a consumer. Finally it's all one and the same. The Portlet 2.0 specifications calls the parties involved in a communication as a ***publisher*** and a ***listener***. A publisher is the one who triggers or fires an even and the listener listens to that event and acts accordingly based on the message. Again, a trigger could be a simple ***message*** or on instruction or it can have a ***payload*** attached. The payload between two communicating Portlet is nothing but the ***data*** interchanged between them. This kind of communication is mostly asynchronous. This means once the publisher triggers an event, it does not know what are the consequences of that event.

10.6.1 Server-Side Eventing

Portlet 2.0 specification allows portlets to send and receive events. As mentioned before, the model is a loosely coupled model where the portal application acts as broker (*mediator*) between the different portlets and distributes the events. The following diagram depicts how the coordination works.

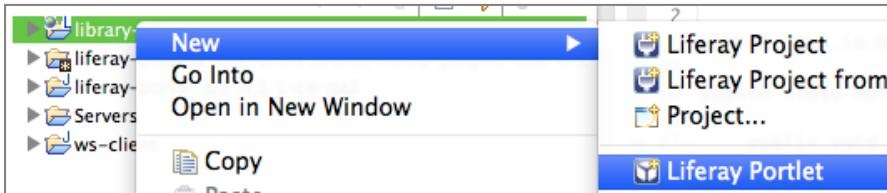


Portlet 1 defines that it can publish some event A in its “**portlet.xml**”. Portlets 2 and 3 define that they are able to receive event A in their deployment descriptors. The portal administrator has now put these three portlets on a page and created connections between the sending portlet Portlet 1 and the receiving portlets Portlet 2 and Portlet 3. Now the user interacts with Portlet 1 and triggers an action on Portlet 1. As a result of this action, the portlet issues event A. After the portlet has finished its action processing the event broker component, which is typically part of a portal runtime, it looks for targets of this event. Thus, the event broker calls the `processEvent` life cycle method introduced with version 2.0 on Portlet 2 and 3 with event A as payload. After

the action and event processing have finished, the rendering lifecycle starts and the portlets can create new markup based on the state updates that occurred in the action and event phases. Sounds very technical? Don't worry let's see this in action.

In the exercise we are going to do now, let's keep our Library Portlet as a *listener*. This demands us to create another new Portlet that will act as *publisher* to fire event. We'll cover the whole process of this server-side eventing mechanism in four steps.

Step 1: Create a triggering Portlet inside the same plugin project (library-portlet) by selecting the appropriate option from Eclipse as shown in the picture.



In the resulting dialog give the settings as below and click "Finish".

Portlet class	TriggerPortlet
Java package	com.library
Superclass	com.liferay.util.bridges.mvc.MVCPortlet

Add this Portlet to the left hand side in "My Library" page and set it's borders. Now you should see the triggering Portlet appearing nicely on the page. (*Just to reiterate the new Portlet is on the left-hand side and our original Library Portlet is on the right-hand side. This orientation will make our IPC understanding much better. You do something on one side and there is some impact on the other side of the page.*) Rename the Portlet title as "Quick Add". What's next?

Step 2: Make changes to "**trigger/view.jsp**" of this new Portlet, so that it presents a quick form to add a book into our Library. Replace the existing contents with,

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

<portlet:actionURL var="quickAddURL" name="quickAdd" />

<aui:form action="<%=" quickAddURL.toString() %>">
    <aui:input name="bookTitle" />
    <aui:input name="author" />
    <aui:button type="submit" />
</aui:form>
```

Optionally you can add validations to the above form. Swiftly add an entry for "**book-title**" in "**Language.properties**", so that the label appears properly and also helps in making it multilingual. Don't forget to make the following "**<resource-bundle>**" entry in "**portlet.xml**" for the new Portlet.

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

Try hitting “Save” button and it’ll throw “`java.lang.NoSuchMethodException`” exception on the server console. Let’s fix this by adding a new `processAction` method “” in the new Portlet class “**TriggerPortlet.java**”.

```
public void quickAdd(ActionRequest request,
    ActionResponse response) throws IOException, PortletException {

    // Retrieve parameters
    String bookTitle = ParamUtil.getString(request, "bookTitle");
    String author = ParamUtil.getString(request, "author");

    // Prepare the payload
    LMSBook lmsBook = new LMSBookImpl();
    lmsBook.setBookTitle(bookTitle);
    lmsBook.setAuthor(author);

    // Fire the event attaching the payload
    QName qName = new QName("http://liferay.com", "lmsBook", "x");
    response.setEvent(qName, lmsBook);
}
```

You will make necessary imports and the import for `QName` is `javax.xml.namespace.QName`. The last step in the above code “`response.setEvent(qName, lmsBook);`” will actually register / fire the event which is already registered with the portal server. But unfortunately we have neither registered an event publisher nor an event listener with our portal server through “**portlet.xml**” deployment descriptor. We are going to do that in our next step.

Step 3: This step involves registering the events with the portal server and configuring the portlets that are associated with those events, identifying who is the publisher of an event and who are all going to listen to an event. Open “**portlet.xml**” and let’s register an event first with the below code. The first is event’s qualified name and the second parameter denotes the type of its payload. The block has to be inserted just above the closing “`</portlet-app>`” tag.

```
<event-definition xmlns:x="http://liferay.com">
    <qname>x:lmsBook</qname>
    <value-type>com.slayer.model.LMSBook</value-type>
</event-definition>
```

The next logical step is to identify in the same file who is going to publish this event and who are going to listen to this event. Inside “trigger” Portlet block, you insert this declarative just before the closing “`</portlet>`” tag.

```
<supported-publishing-event xmlns:x="http://liferay.com">
    <qname>x:lmsBook</qname>
</supported-publishing-event>
```

Similarly make our Library Portlet as the listener by inserting the below declarative in the appropriate place. Note this is a “`processing-event`” unlike the earlier one, meaning this guy is going to listen for any event with the `qName` “`x:lmsBook`”.

```
<supported-processing-event xmlns:x="http://liferay.com">
    <qname>x:lmsBook</qname>
```

```
</supported-processing-event>
```

Step 4: The only thing left now is writing a handler in our Library Portlet (listener) to catch the event and process it appropriately, restoring the original object (payload) from the event. For this we'll define either a new method "quickAdd" in our "**LibraryPortlet.java**" and annotate it as OR override "processEvent" method and write our code there. Both do the same thing and they are shown in two successive blocks of code.

```
public void processEvent(EventRequest request,
                        EventResponse response)
    throws PortletException, IOException {

    Event event = request.getEvent();
    if (event.getName().equalsIgnoreCase("lmsBook")) {
        //process the event
    }
}
```

```
@ProcessEvent(qname = "{http://liferay.com}lmsBook")
public void quickAdd(EventRequest request, EventResponse response)
    throws PortletException, IOException {
    // process the event
}
```

Let's take the second approach, as it is much cleaner and comprehensible. Insert the below code as the method body.

```
Event event = request.getEvent();

LMSBook lmsBook = (LMSBook) event.getValue();

ServiceContext serviceContext = null;
try {
    serviceContext = ServiceContextFactory.getInstance(request);
} catch (PortalException e) {
    e.printStackTrace();
} catch (SystemException e) {
    e.printStackTrace();
}

LMSBookLocalServiceUtil.insertBook(
    lmsBook.getBookTitle(),
    lmsBook.getAuthor(),
    serviceContext);
```

See in the above code, how we have obtained an instance of "serviceContext". Save the changes and check the Portlet now in action. You quickly add a book and it will get saved by invoking an annotated "processEvent" method of our "LibraryPortlet.java". The one who is mediating the whole communication between these portlets is the portal server (Portlet container). We are going to a quick task before we move on to the next topic in this series. This is a task you have to do by yourself.

Do it Yourself: Disable “add-process-action-success-action” for trigger Portlet and show the success page immediately after “Save” is clicked on this Portlet. This task will help you to see the IPC in action more obviously.

10.6.2 Public Render Parameters (PRP)

This is the second standard way of achieving IPC, but less powerful than the previous one but a lightweight alternative for server-side eventing. I know you must have completed the task of the previous sub-section by setting a renderParameter to the portletResponse object.

```
response.setRenderParameter("jspPage", LibraryConstants.PAGE_SUCCESS);
```

First let's ask the question, what is a “render” parameter? Just like Servlets a Portlet does not maintain state within the Portlet object. Many requests, from many different users, may use an instance of a portlet at the same time. Render parameters are used to maintain the state of a Portlet that is otherwise stateless. A “public render” parameter is one that is common to many portlets and hence called as public. By default all render parameters are private in nature. Public render parameters allow JSR 286 portlets to share navigational state information. They are especially useful for coordinating the multiple navigation or viewer portlets that display different information items that are all related to the same parameter name.

The portal stores all portlet render parameters, including public render parameters, as an encoded part of the current portal URL. In the steps that are about to follow let's see how to configure a PRP and how to access them in various portlets directly from the portal server. Let's think of a good use-case to illustrate this. Let's define a parameter called “**DAILY_GREETING**”. Whenever any member comes to the Library he'll be greeted with this message. The Librarian can keep changing this message everyday morning through the interface provided to him. For convenience sake, we'll use the same “Quick Add” Portlet for setting this message.

Step 1: First and foremost we have to register a public render parameter. Since this concept is a part of the original JSR-286 specification, we'll declare a PRP inside “**portlet.xml**”. Insert this code immediately after the event-definition.

```
<public-render-parameter>
  <identifier>DAILY_GREETING</identifier>
  <qname xmlns:x="http://liferay.com">x:DAILY_GREETING</qname>
</public-render-parameter>
```

Once this is done, in the same file, we have to identify the portlets that are aware of this PRP with the help of “**<supported-public-render-parameter>**”. For both our portlets make the following entry before the ending “**</portlet>**” tag.

```
<supported-public-render-parameter>
  DAILY_GREETING
</supported-public-render-parameter>
```

That's it. You're done with the configuration part. Next comes the usage part.

Step 2: Create a new form in the “view.jsp” of trigger Portlet and when submitted make it to invoke “`setGreeting`” action method in it’s Portlet class. Let’s do this.

```
<hr/>
<portlet:actionURL var="setGreetingURL" name="setGreeting" />

<aui:form action="<% setGreetingURL.toString() %>">
    <aui:input name="dailyGreeting" />
    <aui:button type="submit" />
</aui:form>
```

Add an entry for “daily-greeting” in “Language.properties” and define this new method in “TriggerPortlet.java” class.

```
public void setGreeting(ActionRequest actionRequest,
                        ActionResponse actionResponse)
                        throws IOException, PortletException {

    String dailyGreeting =
        ParamUtil.getString(actionRequest, "dailyGreeting");
    actionResponse.setRenderParameter(
        "DAILY_GREETING", dailyGreeting);
}
```

Yes, the PRP has been successfully set. Now, how to read it from our actual Library Portlet where this message has to be flashed when the member first logs in? A PRP is retrieved like any other parameter either inside the Portlet class or inside the JSP of the portlets that are made aware of this PRP.

Step 3: Open “view.jsp” of library Portlet and insert the below line for daily greeting to appear gracefully for the visiting member.

```
<h1>
    <u>Greeting for Today</u>:
    <%= ParamUtil.getString(renderRequest, "DAILY_GREETING", "Hi") %>
</h1>
```

Save the changes and check the interaction that is happening between the two Portlets now. The main purpose of PRP is to pass parameters from one Portlet to the other Portlet, mostly in the same page. The Portlet container as before moderates this parameter passing but the limitation is only String literals can be set as parameters (payloads) and not objects as in the case of Eventing. The other limitation is the sharing scope always applies to all parameters of all portlets on a page; you can currently not control sharing of public render parameters below the page level. But Liferay has a solution for this problem.

Click configuration option for one of these portlets and you will find a “Communication” tab newly appearing there. Through this we can ignore a PRP in the current Portlet.

The screenshot shows the 'Communication' tab of a portlet settings dialog. A note says: 'Set up the communication among the portlets that use public render parameters. For each of the public parameters possible to ignore the values coming from other portlets or to read the value from another parameter.' Below is a table:

Shared Parameter	Ignore	Read Value from Parameter
DAILY_GREETING	<input type="checkbox"/>	DAILY_GREETING

Congratulations! You have successfully tried out an example of Public Render Parameters, setting them in one Portlet and retrieving them in another portlet.

Code Changes @ <http://code.google.com/p/lr-book/source/detail?r=54>

10.7 Non-Standard ways of IPC

In this section, let's see some more ways of accomplishing Inter-Portlet communication. Few of them are Liferay specific. We'll begin this section with Portlet Session sharing, and then move on to see client-side eventing to finally conclude with Portlet URL invocation. Will try to keep this section short .

10.7.1 Portlet Session Sharing

We have heard about HttpSession, an object that maintains the state of a user within the context of an application. What is a [PortletSession](#)? Conceptually it is exactly the same as HttpSession but its scope is limited to only a Portlet. The PortletSession interface provides a way to identify a user across more than one request and to store transient information about that user. A PortletSession is created per user client per portlet application. A portlet can bind an object attribute into a PortletSession by name. The portlet session is based on the HttpSession. Therefore all HttpSession listeners do apply to the portlet session and attributes set in the portlet session are visible in the HttpSession and vice versa.

The PortletSession interface defines two scopes for storing objects:

- APPLICATION_SCOPE – All objects stored in the session using this scope must be available to all the portlets, servlets and JSPs that belongs to the same portlet application and that handles a request identified as being a part of the same session.
- PORTLET_SCOPE – Objects stored in the session using this scope must be available to the portlet during requests for the same portlet window that the objects where stored from.

Inside any JSP of a Portlet, the “portletSession” is available as an implicit object and is injected by the tag “`<portlet:defineObjects/>`”. Inside the Portlet class the object is retrieved from the portletRequest object. The “request” object could be of one of the four types – actionRequest, renderRequest, eventRequest or resourceRequest.

```
PortletSession portletSession = request.getPortletSession();
```

Once this object is available, you can bind objects (*attributes*) to it. By default it is set with “PORTLET_SCOPE”. Let’s play around with couple of scenarios of how to set this within the Portlet so that the value is available for the entire duration of the current user session.

Scenario 1: Open “**view.jsp**” of our Library Portlet and append this code purely purpose and we’ll remove post our testing. Once the value is set in the portletSession, you navigate to any page and come back; the value will be still there. This is not the case with public render parameters. We have used the default PORTLET_SCOPE.

```
<% portletSession.setAttribute("SAY_HELLO", "Good Morning"); %>
<h2>Message from Portlet Session:
<%= portletSession.getAttribute("SAY_HELLO") %></h2>
```

Scenario 2: Just append this one line in the “**view.jsp**” of trigger Portlet. It will print null, as what the other guy has set is not available in this Portlet. Inorder for this value to be available in the trigger Portlet, let’s go back to the “**view.jsp**” of Library Portlet and set this attribute with a higher scope.

```
<% portletSession.setAttribute("SAY_HELLO", "Good Morning",
    PortletSession.APPLICATION_SCOPE); %>
```

Try retrieving it in the “**view.jsp**” of trigger Portlet and the value should get displayed now as what has been set by the other guy.

```
<h2><%= portletSession.getAttribute("SAY_HELLO",
    PortletSession.APPLICATION_SCOPE) %></h2>
```

Do it Yourself: We have used public render parameter to set the global “daily-greeting” message. Unfortunately it is not the right use-case to use PRP. Change the code so that the greeting-message is set in the Portlet session with APPLICATION_SCOPE. You will have to make modifications to “setGreeting” method of “TriggerPortlet.java” and display the message in our Library Portlet’s “**view.jsp**”. This is a classical example of achieving IPC by making use of Portlet Session sharing.

10.7.2 Sharing session variable across WARs

The Portlet 2.0 IPC mechanisms (public render parameters, Events) are restricted to ACTION-to-VIEW. What I mean by ACTION-to-VIEW is on “action phase” of one portlet information is shared and made available to specific/all portlet’s “view phase”. There are use cases where we need an IPC mechanism to share information between portlets of different WARs in VIEW-to-VIEW. It means one portlet will share information from its VIEW phase and will be available to other Portlet in VIEW phase. By default Each WAR has its own session and will not be shared with other WARs. Liferay provides a mechanism by which Portlets can share session attributes across WARs.

How to leverage this feature of Liferay? Portlets that would like to share session attributes with portlets sitting in other WARs need to add following entry in “**liferay-portlet.xml**”. By default the value for this declarative is `true`.

```
<private-session-attributes>false</private-session-attributes>
```

The next step is to use Namespace prefix to Set/Get shared session attributes. By default “`LIFERAY_SHARED_`” prefix is used for sharing session attribute to other WARs. It can be customized through the “**session.shared.attributes**” entry in “**portal.properties**”.

```
portletSession.setAttribute("LIFERAY_SHARED_MYOBJECT",  
    value, PortletSession.APPLICATION_SCOPE);
```

Other portlet(s) in different WAR can access like,

```
portletSession.getAttribute("LIFERAY_SHARED_MYOBJECT",  
    PortletSession.APPLICATION_SCOPE);
```

10.7.3 Obtaining a handle to HttpSession

There will be situations where you want to set some value inside the HttpSession. For example, a Portlet has to set something in the HttpSession that can be retrieved by a theme and displayed to the user. We cannot use PortletSession in such scenarios, as the theme is not aware of PortletSession. In such situations, we should make use of the following code to get a handle to the HttpSession inside the Portlet class and bind any object to that handle.

```
HttpServletRequest httpServletRequest =  
    PortalUtil.getHttpServletRequest(actionRequest);  
HttpSession httpSession = httpServletRequest.getSession();
```

10.7.4 Client-side Eventing

This non-standard mechanism of Inter-Portlet communication works purely based on manipulating the browser’s DOM object with the help of JavaScript. Liferay provides two JavaScript functions to achieve this kind of IPC – `Liferay.fire()` and `Liferay.on()`. Let’s get our hands wet with a quick and cute example. Append the below code in the “**view.jsp**” files of trigger and Library Portlets respectively.

```
<hr/>  
<aui:form name="frm1">  
    <aui:input name="sayHello" />  
    <aui:button value="Poke"  
        onClick="javascript:sayHello();"/>  
</aui:form>  
  
<script>  
    function sayHello() {  
        var frm = document.<portlet:namespace/>frm1;  
        var data = frm.<portlet:namespace/>sayHello.value;
```

```

        var payload = {helloTo : data};
        Liferay.fire("sayHelloEvent", payload);
    }
</script>

```

In the above code, we are firing the event with a unique name for the event and it's payload that is nothing but the JSON representation of data. On the other side, we'll listen to this event and catch the payload with the following code.

```

<aui:script>
    AUI().ready(function(A){
        Liferay.on("sayHelloEvent", function(payload){
            alert('I am in the other Portlet:' + payload.helloTo);
        });
    });
</aui:script>

```

Using this mechanism of IPC, you can create portals that are extremely responsive and provides a very rich user experience. The next topic I would like to present in a new section, as there are many things to be covered there and also keeping its significance in achieving inter-portlet communication.

10.8 Portlet URL Invocation

Using this approach, we can dynamically create and invoke any kind of URL on another portlet. Liferay provides two ways of creating a Portlet URL – one is using a special kind of tag and the other uses a special API that Liferay provides. Let's see both the options here with a real example. Let's assume that we have to provide some quick links on the left-hand side. When these links are clicked an appropriate page opens in the other Portlet lying on the other side of the page.

10.8.1 Using the Tag

Open “**view.jsp**” of trigger Portlet and append below code. Save the file and check the Portlet. You should see a link that when clicked will open the “**update.jsp**” on the other Portlet.

```

<liferay-portlet:renderURL plid="<% plid %>" var="addBookURL"
    portletName="library_WAR_libraryportlet">
    <portlet:param name="jspPage"
        value="<% LibraryConstants.PAGE_UPDATE %>" />
</liferay-portlet:renderURL>
<hr/><aui:a href="<% addBookURL.toString() %>" label="Add Book" />

```

For the new tag to work, include the corresponding taglib URI in “**init.jsp**”.

```

<%@ taglib uri="http://liferay.com/tld/portlet"
    prefix="liferay-portlet" %>

```

10.8.2 Using the Util class

In this step, we'll see how to dynamically create a link using the utility class provided by Liferay. This code can be even used inside the Portlet class to forward the control to a different page after an action is performed. This time we'll introduce another link that will take the user to the list page of our Library Portlet. Paste this in “**view.jsp**”.

```
<%
    PortletURL listPageURL = PortletURLFactoryUtil.create(
        request, "library_WAR_libraryportlet",
        plid, PortletRequest.RENDER_PHASE);
    listPageURL.setParameter("jspPage",
        LibraryConstants.PAGE_LIST);
%>
<br/><aui:a href="<%=" listPageURL.toString() %>" label="Show Books" />
```

Make the necessary imports in “**init.jsp**”, so that they are commonly available.

```
<%@page import="javax.portlet.PortletRequest" %>
<%@page import="com.liferay.portlet.PortletURLFactoryUtil" %>
```

10.8.3 Namespace problem during ActionURL call

In the previous two cases you have to take care of one thing if you're forming an actionURL to the other portlet. Suppose you're submitting a form in Portlet A, that directly goes and invokes an action in Portlet B. In Portlet A (trigger), you have written the code like the below in the “**view.jsp**” of trigger Portlet.

```
<liferay-portlet:actionURL var="quickAddNewURL"
    name="<%=" LibraryConstants.ACTION_UPDATE_BOOK %>"
    portletName="library_WAR_libraryportlet" />

<aui:form action="<%=" quickAddNewURL %>">
    <aui:input name="bookTitle" />
    <aui:input name="author" />
    <aui:button type="submit" />
</aui:form>
```

The code works perfectly fine and data gets properly inserted in the underlying table. But unfortunately the values for bookTitle and author fields are coming empty. How to fix this? This is a classical problem that will be faced normally during this kind of IPC invocation. In such scenarios, we need to set the “`useNamespace`” of “`<aui:form>`” to false. Check the submission after making the change to the form and confirm now the data is going properly to the database.

One more thing you have to notice here. Whenever we use “`var`” to create a Portlet URL either using Portlet 2.0 or “`liferay-portlet`” tag, we need not have to use the “`toString()`” function on it, as the variable that is created is already of String format.

10.8.4 Calling Portlet In Another Page

Using the above two methods, you can even invoke a Portlet that is sitting in another page. All we need is the “plid” of the other page. As long as we know the name or friendly URL of the other page where the target is lying, we can get it’s plid and supply it for dynamically forming the Portlet URL. In one the applications we have developed at mPower Global, we have used the following methods written in **LMSUtil** to achieve this purpose. You may feel free to use this code in your applications as well.

```
public static long getPlidByName(long groupId, Locale locale,
        String name, boolean privateLayout) {
    long plid = 0l;
    List<Layout> layouts = null;
    try {
        layouts = LayoutLocalServiceUtil.getLayouts(
            groupId, privateLayout, "portlet");
    } catch (SystemException e) {
        e.printStackTrace();
    }

    if (Validator.isNotNull(layouts)) {
        for (Layout layout: layouts) {
            if (layout.getName(locale).equalsIgnoreCase(name)) {
                plid = layout.getPlid();
                break;
            }
        }
    }
    return plid;
}
```

```
public static long getPlidByFriendlyURL(long groupId,
        String friendlyURL, boolean privateLayout) {
    long plid = 0l;
    try {
        Layout layout =
            LayoutLocalServiceUtil.getFriendlyURLLayout(
                groupId, privateLayout, friendlyURL);
        if (Validator.isNotNull(layout)) {
            plid = layout.getPlid();
        }
    } catch (PortalException e) {
        e.printStackTrace();
    } catch (SystemException e) {
        e.printStackTrace();
    }
    return plid;
}
```

The invocation from the JSP will look like below. The “targetPlid” can now be used in the formation of the target portlet URL.

```
long targetPlid = LMSUtil.getPlidByName(
    themeDisplay.getScopeGroupId(), themeDisplay.getLocale(),
    "<target page name>", false);
```

I strongly recommend you should try this yourself and see the power of Liferay with your own eyes. When I tried this myself, I was very excited ☺.

10.8.5 Dynamic URL Formation thru JavaScript

Liferay lets you build PortletURLs using JavaScript on the fly in the browser itself. This will drastically improve the client side performance especially if a long list of rows has to be displayed and there are links in each row item. Liferay provide the power to simply integrate your pure javascript files (.js) with PortletURLs, without over passing it as parameter for your javascript constructor. This is how the basic URL formation JavaScript will look like.

```
<aui:script>
    AUI().ready('liferay-portlet-url', function(A){
        var portletURL = new Liferay.PortletURL();
        portletURL.setParameter("key1", "value");
        portletURL.setPortletId(50);
        alert( "that is the url: " + portletURL.toString());
    });
</aui:script>
```

You can literally play with the JavaScript Liferay.PortletURL very nicely if you know the various methods it provides.

```
setCopyCurrentRenderParameters:
function(copyCurrentRenderParameters);
setDoAsUserId: function(doAsUserId);
setEncrypt: function(encrypt);
setEscapeXML: function(escapeXML);
setLifecycle: function(lifecycle);
setName: function(name);
setParameter: function(key, value);
setPlid: function(plid);
setPortletConfiguration: function(portletConfiguration);
setPortletId: function(portletId);
setPortletMode: function(portletMode);
setResourceId: function(resourceId);
setSecure: function(secure);
setWindowState: function(windowState);
toString: function();
```

There are also some convenient methods inside this JavaScript package that helps you in creating the four different types of Portlet URL's.

```
// = new Liferay.PortletURL('ACTION_PHASE');
var actionURL = Liferay.PortletURL.createActionURL();

// = new Liferay.PortletURL('RENDER_PHASE');
var renderURL = Liferay.PortletURL.createRenderURL();

// = new Liferay.PortletURL('RESOURCE_PHASE');
var resourceURL = Liferay.PortletURL.createResourceURL();

var permissionURL = Liferay.PortletURL.createPermissionURL(
portletResource, modelResource, modelResourceDesc, resourcePrimKey);
```

10.9 Customizing Portlet Based on Query String

Though this topic is slightly different from the context of this chapter, I feel this still has relevance and is worth covering here. In one of our recent projects we had a very unique requirement. There is a registration page for a normal user and for a company. The registration form is almost the same but in the case of a company registration there will be an additional block for entering the address details. Instead of creating two different portlets and placing them in two different pages, one for normal user and one for company, we have effectively made use of the feature of reading from the URL query string. But unfortunately inside the Portlet JSP or Portlet class there is no direct way of getting the values from the query string. Then how we got it working? See the solution below.

The URL for the company registration page will look something like, [http://\[host\]/web/guest/register?show-address=true](http://[host]/web/guest/register?show-address=true). In the case of a normal user, he has to click on a link / button that will have the value false for showAddress parameter in the URL's query string. Based on this value, we had to show the address block or otherwise. In the “**view.jsp**” of the registration Portlet all we have to do is to get the handle for the original Servlet Request and retrieve the parameter from there.

```
<%
    boolean showAddressBlock = GetterUtil.getBoolean(
        PortalUtil.getOriginalServletRequest(request)
            .getParameter("show-address"), false);
%>
<c:if test="<% showAddressBlock %>">
    <h2>The address block is shown ....</h2>
</c:if>
```

If the same thing has to be done in your Portlet class, the code will be,

```
HttpServletRequest httpRequest =
    PortalUtil.getHttpServletRequest(portletRequest);
HttpServletRequest originalRequest =
    PortalUtil.getOriginalServletRequest(httpRequest);
```

See how cool is this? Liferay is rocking and you too... mastering it.

10.9.1 Another Classical Usecase

Let me present you another case of making use of this great feature by getting the original servlet request object with the help of PortalUtil utility class. Have you seen the invitation Portlet of Liferay? If not, you can check now under the “Community” category. I’ll explain you what you need to do with this Portlet and how to enable it for our portal’s referral program where users get credits for referring their friends.

Community
Bookmarks
Directory
Invitation
My Sites
Page Comments
Page Flags
Page Ratings

Let’s divide the whole use case into three parts and attack the whole functionality.

Part 1: Develop a hook to customize the invitation Portlet or develop a new Portlet mimicking the same functionality. User “A” (*a registered user of the portal*) will send invitation to his friends and relatives. The invitation mail will contain a link with the inviterId as part of the query string for the registration URL. Inside the code, you will dynamically form the registration page URL like below and append the inviter userId along with it before firing the email.

```
PortalUtil.getCreateAccountURL(request, themeDisplay);
```

Part 2: In this part, we need to customize the “**create_account.jsp**” page of the portal with the help of a JSP hook to inject a hidden variable in the form based on the inviterId parameter in the query string. You may put this as “Expando” token that can be directly retrieved in the service layer API that creates a user account. The referrer Id has to be stored in a new custom attribute for the “user_” table.

Part 3: Develop a Portlet that will show the list of people who joined the portal through a given user. When he logs in the portal, he should see the people who joined by virtue of getting his invitation and the bonus points he earned. This is something related to what is called as “Social Equity” that we’ll see how Liferay helps in achieving this from various angles.

Wish you all the best for carrying out this fabulous task. I am looking out for people who can work with me on developing these components. If you feel you are one of them, shoot me an email to ahmed@mpowerglobal.com with subject “WWU” (*Want to Work With you*).

Summary

Hope you enjoyed this great chapter where we discussed in length about the various configuration options that are available with Liferay. The various topics covered are:

- Static Portlet Preferences
- Dynamic Portlet Preferences
- Reading values from properties files
- Continuation of firing emails from portlets
- Portlet Data handlers with the option to Export and Import

One thing we did not cover explicitly is the externalization of labels and message of a Portlet through language resource bundle files, as this has been covered in the previous chapter in Section [9.5 Portlet Internationalization \(I18N\)](#).

The second logical part of the chapter explained in length about the various mechanisms of achieving Inter Portlet Communication between the portlets. This included Server-Side Eventing, Public Render Parameters, Client-Side Eventing, Portlet Session Sharing and various ways of Portlet URL invocation. One topic I did not cover is the AJAX based invocation of a Portlet URL ,as I am keeping it for one of our later chapters. This we will see when we discuss about resource requests and forming resource URLs from the Portlet and invoking them.

Finally we concluded the chapter by covering the topic of reading the values directly from the query string. This is again a way of configuring the Portlet and passing the parameter to it. I also gave you a very comprehensive exercise and told if you're able do to this successfully, then we can work together in some real projects.

In Chapter 11 where we are going go see in detail the various sub-frameworks of Liferay.

11. Liferay Frameworks – Part 1

The chapter covers

- Security and Permissions
 - Portlet Permissions
 - More Security Layers and Custom Utility
 - Model Permissions
 - Liferay Asset Framework
 - Attaching Tags and Categories
 - Publishing Assets through Asset Publisher
 - Asset Publisher for Books, Final Touches
-

This chapter and the next are solely dedicated for knowing, understanding and implementing many sub-frameworks of Liferay. Liferay itself is a comprehensive portal framework. Under the hood it has many sub-frameworks each of them can be considered as a well established, highly isolated framework in itself. We will get to know some of them. Infact these frameworks make Liferay the most distinguished and robust portal platform that was ever made, even surpassing the capabilities of many matured commercial portal servers out there in the market.

This chapter will predominantly cover three important frameworks, starting with the Security and Permission, then moving on to Asset framework and concluding the chapter with File Storage framework. These frameworks provide set of API's and configurations that make the Developers life easy and helps in enabling some core functionalities of the portal without much efforts. Let's get started.

11.1 Security and Permissions

Everything in this world is protected in some way or the other, right from our homes, offices, the banks we use and the schools we go. Without a proper security in place things get exposed for potential attacks and theft. The same rule applies for portals as well. If we don't put proper security systems in place, there are very high chances of the portal resources / assets getting stolen or the portal itself getting compromised by attackers and professional hackers. Hence, as developers, it is extremely important to know our enemies and understand the threats while developing portal and Portlet applications. A slight negligence can lead to major sabotages and catastrophes.

The JSR-286 (Portlet 2.0) specification defines a simple security scheme using portlet roles and their mapping to portal roles. On top of this basic security infrastructure, Liferay provides a fine-grained and extremely powerful permissions system that we can use to implement access security in our custom portlets. While security is general concept permission refers to the access control over various portal and Portlet resources.

11.1.1 Overview of Portlet 2.0 Security

The JSR specification defines the means to specify the roles that will be used by each portlet within its definition in “**portlet.xml**”. The default roles are *administrator*, *guest*, *user* and *power-user*. You can open this file for our Library Portlet and check for yourself. The entries are like the ones below. For these four roles defined in here in “**portlet.xml**” there is a set of corresponding entries in “**liferay-portlet.xml**” file.

Entries in “portlet.xml” (For every portlet)
<pre><security-role-ref> <role-name>administrator</role-name> </security-role-ref> <security-role-ref> <role-name>guest</role-name> </security-role-ref> <security-role-ref> <role-name>power-user</role-name> </security-role-ref> <security-role-ref> <role-name>user</role-name> </security-role-ref></pre>

```
<security-role-ref>
    <role-name>administrator</role-name>
</security-role-ref>
<security-role-ref>
    <role-name>guest</role-name>
</security-role-ref>
<security-role-ref>
    <role-name>power-user</role-name>
</security-role-ref>
<security-role-ref>
    <role-name>user</role-name>
</security-role-ref>
```

These roles need to be mapped to specific roles within the portal. The reason for this mapping is to provide a means for the deployer of a portlet to resolve conflicts between roles that have the same name but are from different portlets. In order to do the mapping, it is necessary to use portal-specific configuration file (*deployment descriptor*).

Entries in “liferay-portlet.xml” (Common for all portlets)
<pre><role-mapper> <role-name>administrator</role-name> <role-link>Administrator</role-link> </role-mapper> <role-mapper></pre>

```
<role-mapper>
    <role-name>administrator</role-name>
    <role-link>Administrator</role-link>
</role-mapper>
<role-mapper>
```

```
<role-name>guest</role-name>
  <role-link>Guest</role-link>
</role-mapper>
<role-mapper>
  <role-name>power-user</role-name>
    <role-link>Power User</role-link>
</role-mapper>
<role-mapper>
  <role-name>user</role-name>
    <role-link>User</role-link>
</role-mapper>
```

This means that if a portlet definition references the role “power-user”, that Portlet will be mapped to the Liferay role in its database called “Power User”. In your Portlet code, you can then use methods as defined in Portlet specification, specifically for checking the permission.

```
portletRequest.isUserInRole("guest");
portletRequest.getRemoteUser();
portletRequest.getUserPrincipal();
```

In reality we will hardly use Portlet 2.0 permissions system in our custom portlets as they are too primitive in nature and cannot fulfill our complex security requirements. Instead Liferay uses its own permission system directly to achieve more fine-grained security for our Portlet resources.

11.1.2 Overview of Liferay's Permission System

Liferay's permission system is a very flexible mechanism to define the actions a given user can perform within the general context of the portal or within a given portlet and its data. Portal and portlet developers break down the operations that can be performed within the portlet, and/or on the data they create, into distinct "actions". The act of granting the ability to perform a given action to an individual user, role, user group, etc. is of course the act of granting that permission. In this way, portal/portlet developers are not tied to named roles, user group, or any other authorization model in particular; all they need to care about is clearly defining the different types of operations required to suite the business logic of their applications.

Once the actions have been identified and configured, the system administrator is free to grant permissions to perform those actions to users, groups, communities, etc, either directly or through roles, using the portal's administration tools or specifically within portlet's configurations UIs. The purpose of this section is to explain how a portlet developer can use this service during development to give the portal administrator the same level of control over permissions that he has over the portlets that come bundled with Liferay. Before we get into the implementation details we should familiarize ourselves with few terminologies around this subject. They are:

- **Resource** – A generic term for any object represented in the portal. Examples of resources include portlets (e.g. Message Boards, Calendar, etc.), Java classes (e.g. Message Board Topics, Calendar Events, Library Books, etc.), and files (e.g. documents, images, etc.)

-
- **Permission** – An action on a resource. For example, the view action with respect to viewing the calendar portlet is defined as permission in Liferay.
 - **Action** – Refers to an operation / function that can be performed on a given Portlet, eg. Could be “Add a Book”, “Delete a Book”, “Delete many books”, etc. on our Library Portlet.

Permission in Liferay can be achieved at six levels:

1. **Server Level** – Only Omni administrators can do certain operations / functions.
2. **Portal Level** – These actions can be performed only by people who manage a portal instance within the given installation of Liferay
3. **Page Level** – We can set permission on pages, so that only people who've got the required permissions can access them.
4. **Portlet Level** – There are some generic permission that can be set for every Portlet that is deployed into the portal server. An image is shown below to illustrate the common permissions on a Portlet.
5. **Action Level** – We can define permission for every action that can happen within a given Portlet. We have seen an example of this just now.
6. **Data Level** – This kind of permissions are called as model permissions, which is usually set on every individual record or an asset that the Portlet owns.

Role	Add to Page	Configuration	View	Permissions
Guest	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
Owner	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
Portal Content Reviewer	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Power User	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
User	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Site Content Reviewer	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Site Member	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>

Save

11.1.3 Liferay's Permissioning Algorithm

There are six permission algorithms that Liferay has used over the years for checking permissions. Liferay 5 introduced algorithm 5 that was based on RBAC (*an acronym for Role Based Access Control*) system. Liferay 6 used algorithm 6 that was an optimized version of algorithm 5 that improved performance by using a reduced set of database tables and storing the permissioning information as bits. It's important to note that once a permission algorithm is configured and resources are created, the algorithm cannot be changed; otherwise the existing permissions will be lost.

For all new deployments it is strongly recommended you use algorithm 6. For deployments that are using other algorithms it's recommended you migrate to algorithm 6 using the migration tools provided in the Control Panel (*the option will appear on only if you have started the server with a permission algorithm less than*

-
- 6). The entry “`permissions.user.check.algorithm=6`” in “**portal.properties**” controls the permission algorithm being used by the portal server.

Before we get onto the actual implementation, I strongly recommend that you go through this [Liferay’s Wiki](#) page that explains the inner details of how the permissions system works in Liferay, what are the entities and their relationships. You should also note that the diagram shown in this article was created for Liferay version 4.4 and is only valid for algorithms 1 to 4. Several things have changed since then so it should not be taken as an accurate source of information. Anyway it’s still very useful to get a general understanding of the permission system. We’ll look into some of these tables when we implement permissions for our Library Portlet.

11.2 Portlet Permissions

In the previous section we have seen the various levels where permissions can be enforced – *server, portal, page, Portlet, actions* and *data* (models). In this section, we’ll see in greater detail about Portlet permissions that include the common permissions available for all Portlets and the ones that can be injected specifically on a Portlet based on its actions / functions. First, let’s get ourselves familiarized with what is available by default. Initially you may think why we should do all these dirty tasks. But very soon you will realize the significance of them.

11.2.1 Understanding the backend tables

Without disturbing our current “library-portlet”, create a new Liferay project from Eclipse and call it as “quick-portlet”. Deploy this Portlet into the server with the help of “Add and Remove” option. The moment the Portlet gets deployed some tables in the database get impacted. What are they?

Impact 1: You will see a new record in the “**portlet**” table that has columns – *id_, companyId, portletId, roles, active_*.

<code>id_</code>	<code>companyId</code>	<code>portletId</code>	<code>roles</code>	<code>active_</code>
20001	10153	quick_WAR_quickportlet		1

Impact 2: Four records get inserted into “**resourceaction**” table with columns – *resourceActionId, name, actionId and bitwiseValue*.

<code>resourceActionId</code>	<code>name</code>	<code>actionId</code>	<code>bitwiseValue</code>
1602	quick_WAR_quickportlet	ADD_TO_PAGE	2
1603	quick_WAR_quickportlet	CONFIGURATION	4
1604	quick_WAR_quickportlet	PERMISSIONS	8
1601	quick_WAR_quickportlet	VIEW	1

Impact 3: Four records make entry into “**resourcepermission**” table.

resourcePermissionId	companyId	name	scope	primKey	roleId	ownerId	actionIds
3301	10153	quick_WAR_quickportlet	1	10153	10160	0	2
3302	10153	quick_WAR_quickportlet	1	10153	10161	0	2
3303	10153	quick_WAR_quickportlet	1	10153	10163	0	2
3304	10153	quick_WAR_quickportlet	1	10153	10164	0	2

The “roleId” column in the above table is a Foreign Key to the “role_” table shown below. The four roleIds map to the four roles we mentioned in the “portlet.xml” and mapped them in the “liferay-portlet.xml” with actual portal roles.

roleId	companyId	classNameId	classPK	name
10160	10153	10004	10160	Administrator
10161	10153	10004	10161	Guest
10162	10153	10004	10162	Owner
10163	10153	10004	10163	Power User

We have seen three major impacts happening to the database soon after the Portlet gets deployed to the server. Now let's see the impacts after the portlet is placed on a portal page. Login as admin and Drag & drop the new Portlet in the welcome page and let's quickly examine the additional impacts that have happened to the database and come back to perform some basic tasks.

Impact 1: First and foremost the “typeSettings” column in the “layout” table has got updated with the id of the new Portlet we added to the page, with the information, which column of the current layout this Portlet is sitting, either “30” column or “70” column.

Impact 2: Three additional records got inserted into the “resourcepermission” table.

scope	primKey	roleId	ownerId	actionIds
4	10611_LAYOUT_quick1_WAR_quick1portlet	10162	0	15
4	10611_LAYOUT_quick1_WAR_quick1portlet	10169	0	1
4	10611_LAYOUT_quick1_WAR_quick1portlet	10161	0	1

Note that the actionId, roleId and primKey columns of this table now and compare them with the other four rows that got inserted earlier.

11.2.2 BitWise permissioning unleashed

Coming back to the Portlet that is lying in the browser, let's go to its configuration dialog. You should see the following four permissions in this very basic MVCPortlet. This is in the form of a two-dimensional matrix. X-axis has the various actions and Y-axis displays various portal roles.

View	Add to Page	Configuration	Permissions
------	-------------	---------------	-------------

Task 1: Uncheck the “View” for “Guest” role and save the settings. Go to the database and you will see that for one record in the “resourcepermission” table the value of “actionIds” column has become from “1” to “0”. This has happened for the

roleId corresponding to “Guest”. This means the guest can literally do anything on this Portlet. Logout from the portal and see how the Portlet is appearing now for the “Guest” user. It should flash the following message.

You do not have the roles required to access this portlet.

The role that corresponds to the “value” is “Owner” is 15. If you quickly go back to the configuration dialog and check that row, you will see all the boxes checked for that role. In binary logic the Boolean value “1 1 1 1” is nothing but decimal value “15”.

Owner	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
-------	-------------------------------------	-------------------------------------	-------------------------------------	-------------------------------------

You can do the conversion yourself and check through [this link](#). I am sure, now you got a fair idea about how this whole permission algorithm works in Liferay. For this time being if you understand this much it is more than enough. You will not believe the permissions systems of Liferay are one of the most complex pieces in the whole stake. With its complexity comes its versatility. Now go back to the “**resourceaction**” and check the “bitwiseValue” column. This column denotes the position of an action in that two-dimentional matrix. In the UI, the ordering may be different.

Permissions	Configuration	Add to Page	View	Decimal
0	0	0	1	1
0	0	1	0	2
0	1	0	0	4
1	0	0	0	8

Task 2: Enable to view permission to “Guest” and ensure that the Portlet is available for the guest. If you don’t do this then we have to login everytime to check the new features we are going to build in the rest of this book. Now go to our Library Portlet and see the “Permissions” tab under the portlet’s “configuration” option. There are seven Portlet actions by default unlike the four in the previous case.

Preferences	action.CONFIG	View	Add to Page	Access in Control Panel	Configuration	Permissions
-------------	---------------	------	-------------	-------------------------	---------------	-------------

The extra options are coming because we have enabled extra features of this Portlet like; edit mode, config mode and access in Control Panel. Correspondingly there are seven records in the “**resourceaction**” table.

resourceActionId	name	actionId	bitwiseValue
1001	library_WAR_libraryportlet	VIEW	1
1002	library_WAR_libraryportlet	ADD_TO_PAGE	2
1003	library_WAR_libraryportlet	CONFIGURATION	4
1004	library_WAR_libraryportlet	PERMISSIONS	8
1101	library_WAR_libraryportlet	ACCESS_IN_CONTROL_PANEL	16
1301	library_WAR_libraryportlet	PREFERENCES	32
1302	library_WAR_libraryportlet	CONFIG	64

11.2.3 Adding New Portlet Permissions

With the basic understanding, now we are going to get into our actual implementation. What we need to do if we have to give the same level of flexibility for all the custom actions that we are going to have in our Library Management System. Say for e.g. only users with the role “Librarian” should be able to add the book. In future, the portal administration can extend the ability to add a book even by the “Asst. Librarian” without touching the code. He should be able to do this on the fly just through the UI. How we are going to inject this kind of flexibility and agility into our Portlet? That’s the topic for the rest of this section before we move on to Model permissions.

Adding permissions to any Portlet consists of four main steps, also known as DRAC.

1. Define all resources and their permissions, usually done in an XML file.
2. Register all the resources defined in step 1 into the permissions system. This is known as *adding resources*.
3. Associate the necessary permissions with resources.
4. Check permission before returning resources.

Let’s apply these four steps in our own Library Portlet and enable it to provide that level of flexibility.

Step 1: Create a new file “**default.xml**” under “**src/resource-actions**” (*this folder has to be created first*).

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE resource-action-mapping PUBLIC
  "-//Liferay//DTD Resource Action Mapping 6.1.0//EN"
  "http://www.liferay.com/dtd/liferay-resource-action-mapping_6_1_0.dtd">
<resource-action-mapping>
  <portlet-resource>
    <portlet-name>library</portlet-name>
    <permissions>
      <supports>
        <action-key>ACCESS_IN_CONTROL_PANEL</action-key>
        <action-key>ADD_TO_PAGE</action-key>
        <action-key>CONFIGURATION</action-key>
        <action-key>VIEW</action-key>
        <action-key>ADD_ENTRY</action-key>
      </supports>
      <site-member-defaults>
        <action-key>VIEW</action-key>
      </site-member-defaults>
      <guest-defaults>
        <action-key>VIEW</action-key>
      </guest-defaults>
      <guest-unsupported>
        <action-key>ACCESS_IN_CONTROL_PANEL</action-key>
        <action-key>CONFIGURATION</action-key>
      </guest-unsupported>
    </permissions>
  </portlet-resource>
</resource-action-mapping>
```

```
</resource-action-mapping>
```

In this file, we are defining a new “`<portlet-resource>`” with the id of the portlet in “`<portlet-name>`” tag and all its set of actions in “`<supports>`” tag including the default ones and the ones that we want to add specifically for this portlet. The “`<action-key>`” element defines the name of the action. The name will be translated as specified in “**Language.properties**”. For example, if the action key is “VIEW”, then the key in this will be “action.VIEW”. In our case, we have added “ADD_ENTRY” as the new custom action on this Portlet. Apart from “`<supports>`” we can include the following six declaratives at the time of defining our new resource.

1. The “`<community-defaults>`” element specifies the actions that community members are permitted to perform by default. This element is deprecated, so use the “`<site-member-defaults>`” element instead.
2. The “`<guest-defaults>`” element specifies the actions that guest users are permitted to perform by default.
3. The “`<guest-unsupported>`” element specifies the actions that guests are never permitted to perform. This disables the ability to assign permissions for these actions. Only define actions here if you wish to prevent anyone from granting permissions to perform these actions.
4. The “`<layout-manager>`” element specifies the actions that layout managers are permitted to perform. If omitted, then layout-managers are granted permissions on all supported actions. If included, then layout managers can only perform actions specified in this element.
5. The “`<owner-defaults>`” element specifies the actions that the creator of the resource is permitted to perform. If omitted, then owners are granted permissions on all supported actions.
6. The “`<site-member-defaults>`” element specifies the actions that site members are permitted to perform by default.

Step 2: The next logical step is the “Register”. We need to tell the portal server through some way to register this information into the permissions system. In the case of Portlet permissions, making an entry in “**portlet.properties**” file and re-deploying the Portlet, so that the “**default.xml**” file is read and the contents are registered with the permissions system, usually does it. Open “**portlet.properties**” that we have created in Section [10.3 Reading from “properties” files](#) and insert this entry.

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

After saving and the deployment happened properly, check the “Permissions” tab under the portlet’s configuration option. You will see the new action appearing there. This means it has got properly registered into the Liferay’s permissioning system. I strongly urge you to look into the back-end tables and know the impact that has been made there by this “Register” step.



Step 3: The third step is to associate OR dis-associate a particular role from performing this action. The owner of that resource usually does this from the

administration UI. So it's no big deal and we have already done that. Let's go to the permissions tab under the Portlet's configuration option and can perform this step.

Step 4: The last and final step is to check permissions before returning the resource or providing access to the resource. “Add Entry” is permission on the Portlet resource. So we are going to surround the “[Add new Book »](#)” link before displaying to a user. This will prevent anyone to enter a book into our Library Management System without authorisation. We are going to put this protection at three levels. The first level of security we will see here and I'll cover the rest in the next section.

Level 1 Security: In the first level of security we are going to protect the link or the button that people usually use to add a book into the Library. For this we are going to make use of one of the below guys,

- The implicit object “permissionChecker” available inside the JSP page by virtue of using the tag “`<liferay-theme:defineObjects/>`”. To know the methods of this object you should go through the apidocs of the corresponding interface [PermissionChecker](#).
- A utility class provided by Liferay for this purpose, [PortletPermissionUtil](#).

Let's go with the second one as it is much more refined to use than directly using “permissionChecker” object. Open “**init.jsp**” of Library Portlet and append the following code within a set of scriptlet tags. We are keeping this variable in “**init.jsp**” so that it is available for all other JSP's of the plugin project.

```
boolean canAddBook = PortletPermissionUtil.contains(  
    permissionChecker, layout,  
    portletDisplay.getRootPortletId(), ActionKeys.ADD_ENTRY);
```

In parallel, open “**view.jsp**” of this Portlet and surround the “[Add new Book »](#)” with the condition.

```
<c:if test="<%= canAddBook %>">  
    <a href="<%= updateBookURL %>">  
        <liferay-ui:message key="add-new-book" /> &raquo;  
    </a>  
</c:if>
```

Save the changes and check the Portlet now, by logging out of the portal. As a guest you should no longer see this link. But when you login again, the link is available. But if you go to the Permissions tab of this Portlet and specifically check the option to allow users to “Add Entry”, then this link will appear even for the guest user. You can confirm this as well. But after testing, un-check this option once again as we have still not completed this topic.

Role	Preferences	action.CONFIG	View	Add to Page	Access in Control Panel	Configuration	Permissions	Add Entry
Guest	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>

The class [ActionKeys](#) contains all the standard action names. This does not mean that you cannot define your own action. You can very much do that. When you define a new custom action name, you just have to make an entry in the “**Language.properties**” file with the word “action” prefixed to the new name similar

to the one below. This will ensure the proper display of this action in the configuration dialog and other places.

```
action.ADD_ENTRY=Add Entry
```

11.3 More Security Layers and Custom Utility

This section is just a continuation of the previous section, as I wanted few topics to get logically separated from the previous ones. This section will cover how to implement more levels of security and also how to implement a custom permission checker utility class for every custom Portlet that we develop.

11.3.1 Level-2 Security for Protecting Resources

Now, we have made sure that the “Guest” does not have permission to add a book. But what if he knows the URL and directly hit the URL from the browser? He can comfortably go to the book add form directly and add the book. This is a major security loophole in our application. We need to prevent this as well. The best place to incorporate the 2nd level of security is in our Portlet class. Before the “**update.jsp**” page being served, we need to check if the user has the permission to access this page. Only then give him the access, else redirect him to an error page with a proper display message. How does this sound? Can we do this now?

We need to either override the “*doview*” or “*render*” in our “**LibraryPortlet.java**”. I think we already have the “*render*” overridden. Let’s insert a new line there “*checkBeforeServe(request);*” and implement the actual private method that will take care to check the permission before serving “**update.jsp**”.

```
private void checkBeforeServe(RenderRequest request)
    throws PortletException {

    String jspPage = ParamUtil.getString(request, "jspPage");

    if (jspPage.equalsIgnoreCase(LibraryConstants.PAGE_UPDATE)) {
        ThemeDisplay themeDisplay = (ThemeDisplay)
            request.getAttribute(WebKeys.THEME_DISPLAY);

        StringBuilder portletName = new StringBuilder()
            .append(getPortletName())
            .append("_WAR_")
            .append(getPortletName())
            .append("portlet");

        try {
            PortletPermissionUtil.check(
                themeDisplay.getPermissionChecker(),
                portletName.toString(),
                ActionKeys.ADD_ENTRY);
        } catch (PortalException | SystemException e) {
            throw new PortletException(e.getMessage());
        }
    }
}
```

```
}
```

Now you try hitting the URL directly and the portlet should show the error. But the guest can view the books list page without any issues.

```
http://localhost:8080/web/guest/my-
library?p_p_id=library_WAR_libraryportlet&p_p_lifecycle=0&p_p_state=n
ormal&p_p_mode=view&p_p_col_id=column-
2&p_p_col_count=1&_library_WAR_libraryportlet_jspPage=%2Fhtml%2Flibra
ry%2Fupdate.jsp
```

11.3.2 Level-3 Security Protection for Local / Remote API's

Do you remember we have earlier exposed the “`insertBook`” API as a remote service? This guarantees that anyone can just invoke this method through a web service or a JSON service and add a book into our Library. We need to protect this as well, so that no one ever can access this API remotely without having proper credentials.

In “`<Entity>ServiceImpl`” a method is already available for getting an instance of `permissionChecker` object – “`getPermissionChecker()`”. So, before performing a database operation or calling a localService API, we can check the permission something very similar to how we did in the previous case.

In “`<Entity>LocalServiceImpl`”, unfortunately this method is not available and we have to rely on the “`serviceContext`” object that is available and which has the following permission related methods that can be invoked before serving a resource or performing an operation. It has these following methods to check the permissions.

```
serviceContext.getGroupPermissions();
serviceContext.getGuestOrUserId();
serviceContext.getGuestPermissions();
serviceContext.isAddGroupPermissions();
serviceContext.isAddGuestPermissions();
```

But most of these methods are applicable only for model permissions that we are going to see next. Liferay provides mitigation for getting the “`permissionChecker`” object inside this file.

```
PermissionChecker permissionChecker =
    PermissionThreadLocal .getPermissionChecker();
```

11.3.3 Writing a Custom Permission Class

Don't you think it will be a good idea to have a custom wrapper utility class that will help us to perform the security checks inside the JSP or JAVA code? This is exactly how Liferay has defined custom permission util classes for most of the portlets that come bundled with it. Let's create a new class “**LibraryPermissionUtil.java**” inside

the package “com.library.security” and create few easy to use API’s there as shown here, including the Portlet name declared as a class level field.

```
static final String portletId = "library_WAR_libraryportlet";

public static boolean hasPermissionToAddBook(
    PermissionChecker permissionChecker) {

    boolean hasPermission = false;

    try {
        hasPermission = PortletPermissionUtil.contains(
            permissionChecker,
            portletId, ActionKeys.ADD_ENTRY);
    } catch (PortalException e) {
        e.printStackTrace();
    } catch (SystemException e) {
        e.printStackTrace();
    }

    return hasPermission;
}

public static boolean hasPermissionToAddBook(
    PortletRequest portletRequest) {

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

    return hasPermissionToAddBook(
        themeDisplay.getPermissionChecker());
}
```

Make the necessary imports.

```
import javax.portlet.PortletRequest;
import com.liferay.portal.kernel.exception.PortalException;
import com.liferay.portal.kernel.exception.SystemException;
import com.liferay.portal.kernel.util.WebKeys;
import com.liferay.portal.security.permission.ActionKeys;
import com.liferay.portal.security.permission.PermissionChecker;
import com.liferay.portal.service.permission.PortletPermissionUtil;
import com.liferay.portal.theme.ThemeDisplay;
```

Once methods like these are in place, you can invoke them directly either from the JSP or Portlet class without the need for defining anything in the “**init.jsp**”. These custom API’s will also give more readability to the code that we are writing. In the “**view.jsp**” you may surround the Add Book link with just one call like,

```
LibraryPermissionUtil.hasPermissionToAddBook(permissionChecker)
```

I am sure; you have understood the Portlet permissions in its fullest form with all these examples. I know it is the trickiest topics in the whole of Liferay. If you were still not comfortably, I’d prefer you go over these sections again before moving on with the model permissions that we are going to cover in detail in the upcoming section.

11.4 Model Permissions

If we have to talk about the good things about Liferay, the discussion is never going to get over that soon as the list is really long. Out of the items in that long list, we are covering a very small subset in this book. One feature in this subset is Model Permissions. By building this feature inside the portal itself Liferay has achieved what only few major databases have implemented. This feature is all about setting the permissions at every record level of any given table.

Once a book is created in our Library, we can make the application is such a way that, the administrator can set permission on that one individual book. The permissions could be who can view that particular book, who can update its information, who can borrow that book and who can remove that book from our Library. Sounds interesting, isn't it? Let's see this in full-blown action.

11.4.1 Defining and Registering Model Permissions

Similar to Portlet permissions, model permissions also have the same four steps of Define, Register, Associate and Check (DRAC). The "define" happens in the same XML file where we defined the Portlet permissions. Open those file and insert this block after “`<portlet-resource>`” block. The only difference is that, this block has a “`<model-name>`” and the reference to the Portlet from where this model is being accessed with the help of a “`<portlet-ref>`” tag.

```
<model-resource>
    <model-name>com.slayer.model.LMSBook</model-name>
    <portlet-ref>
        <portlet-name>library</portlet-name>
    </portlet-ref>
    <permissions>
        <supports>
            <action-key>ADD_ENTRY</action-key>
            <action-key>DELETE</action-key>
            <action-key>UPDATE</action-key>
            <action-key>PERMISSIONS</action-key>
            <action-key>VIEW</action-key>
        </supports>
        <site-member-defaults>
            <action-key>VIEW</action-key>
        </site-member-defaults>
        <guest-defaults>
            <action-key>VIEW</action-key>
        </guest-defaults>
        <guest-unsupported>
            <action-key>ADD_ENTRY</action-key>
            <action-key>PERMISSIONS</action-key>
            <action-key>DELETE</action-key>
        </guest-unsupported>
    </permissions>
</model-resource>
```

The next step is to register every book as a resource when it is getting added to our database. Only when it is designated as a “resource” the permissions are application on that resource. For doing this, just insert this block of code in “insertBook” of “**LMSBookLocalServiceImpl**” immediately after the book is inserted into the database. The object “`resourceLocalService`” is available in this class by default through Spring’s DI (*Dependency Injection*).

```
try {
    resourceLocalService.addModelResources(lmsBook, serviceContext);
} catch (PortalException e) {
    e.printStackTrace();
} catch (SystemException e) {
    e.printStackTrace();
}
```

There are other methods also available but here we are using the most appropriate one. Save the changes and add a book into our Library. Let’s see what’s happening in the back-end. To model, a book is added to the Library apart from an entry into the “lms_lmsBook” table, you can see the following impacts happening there.

Impact 1: Five records get inserted into “**resourceaction**” table.

resourceActionId	name	actionId	bitwiseValue
1905	com.slayer.model.LMSBook	VIEW	1
1901	com.slayer.model.LMSBook	ADD_ENTRY	2
1902	com.slayer.model.LMSBook	DELETE	4
1903	com.slayer.model.LMSBook	UPDATE	8
1904	com.slayer.model.LMSBook	PERMISSIONS	16

Impact 2: One record found a way into the “**resourcepermission**” table.

resourcePermissionId	companyId	name	scope	primKey	roleId	ownerId	actionIds
3401	10153	com.slayer.model.LMSBook	4	105	10162	10195	31

The binary value of actionIds 31 is “1 1 1 1 1”, this means all the five operations of VIEW, ADD_ENTRY, DELETE, UPDATE and PERMISSIONS can be performed by a user having the roleId of “10162” (Guest). Having got a fair idea of defining and registering the model permissions, let’s move on to see how to enable the same in the UI so that the administrator can do this on the fly.

11.4.2 Setting Permissions While Adding Book

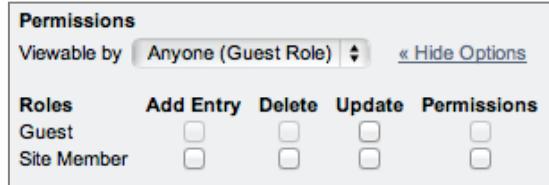
In this sub-section we’ll see how to dynamically set the model permissions on one instance of book at the time of adding the book to the Library. Liferay provides a convenient tag for this purpose. Let’s use that tag in our “**update.jsp**”.

Step 1: Insert the below code inside the form, just before the “Save” button.

```
<aui:field-wrapper label="Permissions">
    <liferay-ui:input-permissions
        modelName="<% LMSBook.class.getName() %>" />
```

```
</aui:field-wrapper>
```

The “`modelName`” attribute refers to model / entity for which the control is being used. Save and check the form now, you should see the “Permissions” option part of the add book form. There is an option to show more options and less option. There is a drop-down to select the role.



Step 2: Now the next step is to save this information at the time of persisting the book into our tables. For this you need not have to literally do anything. The moment you have this control in the form, the “`serviceContext`” object inside the action method has all the selections set. You can actually test this by putting some SOP statements inside the “`updateBook`” method of our Portlet class that will print the values of “`getGroupPermissions`” and “`getGuestPermissions`”. Based on the items that you select in the form at the time of adding the book, additional records get added to the back-end tables that set the permissions on this model. This is how the data got stored for one book that I added, with respect to three different roles.

resourcePermissionId	companyId	name	scope	primaryKey	roleId	ownerId	actionIds
3708	10153	com.slayer.model.LMSBook	4	111	10161	0	9
3706	10153	com.slayer.model.LMSBook	4	111	10162	10195	31
3707	10153	com.slayer.model.LMSBook	4	111	10169	0	30

11.4.3 UI to Control Permissions of Every Book in List

In the concluded sub-section, we’ll set the permissions at the times of adding a book to the Library. If the administrator wants to control the permissions after it gets added, then what’s the solution? This sub-section has the answer. Let’s directly get on to doing some changes to our “`actions.jsp`” in order to achieve this. Open this file and make these changes.

Change 1: Include the following taglib URL so that we can use Liferay security related tags in our JSP.

```
<%@ taglib uri="http://liferay.com/tld/security"
           prefix="liferay-security" %>
```

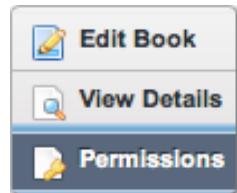
Change 2: Create an URL by using one of the tags of the taglib we just included.

```
<liferay-security:permissionsURL
    modelResource="<%=_LMSBook.class.getName()%>"
    modelResourceDescription="<%=_book.getBookTitle()%>"
    resourcePrimKey="<%=_Long.toString(book.getBookId())%>"
    var="permissionsURL" />
```

Change 3: Create a new item inside “`<liferay-ui:icon-menu>`” for the new link.

```
<liferay-ui:icon image="permissions" url="<%=_permissionsURL%>" />
```

Save the changes and check the list of books now to confirm that the new URL is appearing in the actions menu and click that link takes you to a page where you can set the permissions for that particular book entry. After saving your setting in this page, you can come back to the list view by clicking the “Back” link.



11.4.4 Showing Only those Books with VIEW Permission

For any member who is coming to the library we have to show only the books that have the VIEW permission enabled. This is a best exercise for you to test and understand what we have learnt so far. While fetching the books to get displayed using the “getLibraryBooks” method of the DTO class, we have to pull only those books that can be “viewed”. Let’s modify the original method as below to apply this filter.

```
public List<LMSBook> getLibraryBooks(long companyId, long groupId)
    throws SystemException {
    List<LMSBook> books =
        lmsBookPersistence.findByCompanyId_GroupId(
            companyId, groupId);

    List<LMSBook> result = null;

    if (Validator.isNotNull(books) && !books.isEmpty()) {

        result = new ArrayList<LMSBook>();
        String className = LMSBook.class.getName();
        PermissionChecker permissionChecker =
            PermissionThreadLocal.getPermissionChecker();

        for (LMSBook book : books) {
            if (permissionChecker.hasPermission(groupId,
                className, book.getBookId(), ActionKeys.VIEW)) {
                result.add(book);
            }
        }
    }
    return result;
}
```

Test the new API by removing the view permission on certain books by guests and check the book list as a non-logged in user.

11.4.5 Permissions filtering and “filterFindBy” Methods

Are you too excited about writing the above code and testing it positive? If you hate writing too much code, there is a blessing that Service Builder is about to offer. After you define the model permissions in the “**default.xml**”, the Service Layer automatically detects it and generates “**filterFindBy**” methods for all the finder tags that you have defined in the “service.xml”.

Re-run the Service Builder and the new methods are ready for you. This is called as permissions filtering. But note that these are generated only under the following conditions and never expect them to be readily available. The conditions are:

1. Entity has a simple primitive primary key, “**bookId**” in our case.
2. Entity has permission check registered in “**resource-actions/default.xml**”.
3. Entity has both “**userId**” and “**groupId**” fields.
4. Entity has a finder tag with “**groupId**” as one of the finder columns.

The improved “`getLibraryBooks`” method will look like below with permission filtering in place. This feature of the Service Builder has literally saved you from performing all sorts of gymnastics.

```
public List<LMSBook> getLibraryBooks(long companyId, long groupId)
    throws SystemException {
    return lmsBookPersistence
        .filterFindByCompanyId_GroupId(companyId, groupId);
}
```

In olden days developers loved to write lots of code but eventually ending up making everything buggy (*I used to be one amongst them a decade back*). The modern age developers are smart and they love writing less code. They are very much aware that more code means more bugs and more headaches.

11.4.6 Hiding an Action Based on Permission

It is equally important to show or hide some actions in the list of actions appearing in the list based on the permissions of the user currently logged in this is exactly similar to the Portlet permissions action. But here in this case, the actions are being performed on every single record (*entity*). It is exactly the same like how we protected the “Add new Book” with a condition. Say, for e.g. we want to give the ability to update the book information only to certain user having one particular role. I am quickly going to show you what change have to be done in our “**actions.jsp**” to selectively show the “Edit Book” option.

Step 1: Add this new method in “**LibraryPermissionUtil**”.

```
public static boolean canEditThisBook(long groupId,
    PermissionChecker permissionChecker, long bookId) {

    return permissionChecker.hasPermission(groupId,
        LMSBook.class.getName(), bookId, ActionKeys.UPDATE);
}
```

Step 2: Surround the “Edit Book” link with this code replacing the original link.

```
<c:if test="<% LibraryPermissionUtil.canEditThisBook(
    themeDisplay.getScopeGroupId(),
    permissionChecker, book.getBookId()) %>">
    <liferay-ui:icon image="edit" message="Edit Book"
        url="<% editBookURL.toString() %>" />
</c:if>
```

To test this, remove the “Update” permission on certain roles for some books and the link should not appear for those books when users with that role logs in and checks the list of books in our Library. Congratulations! You have successfully mastered the subject of model permissions. Now it is time for you to play around with some of the default Liferay portlets (*Blogs*) and check how model permissions are being used.

Code Changes @ <http://code.google.com/p/lr-book/source/detail?r=61>

11.4.7 Removing a Resource During Delete

Friend, we have inserted a resource at the time of adding a book into our Library. Don’t you think we have to remove the same resource while deleting a book from our Library? If we don’t do this, then the “**resourcepermission**” table will be filled with orphaned records without a reference to the parent, hence putting them under the dangling reference category a key violation of referential integrity.

Hence, whenever you write the code to delete a book, don’t forget to write a line to remove the “resource” as well, so that the corresponding record in the underlying resource tables get deleted too. This is how your code will look like in our existing “LibraryPortlet.java” Portlet class where we have defined the “deleteBook” method.

```
// delete the resource
LMSBook lmsBook = null;
try {
    lmsBook = LMSBookLocalServiceUtil.fetchLMSBook(bookId);
} catch (SystemException e) {
    e.printStackTrace();
}
try {
    ResourceLocalServiceUtil.deleteResource(
        lmsBook, ResourceConstants.SCOPe_INDIVIDUAL);
} catch (PortalException e) {
    e.printStackTrace();
} catch (SystemException e) {
    e.printStackTrace();
}
```

Now delete a book from the Library that you recently created. This should also delete the corresponding records from the “**resourcepermission**” table.

11.4.8 Exploring Permissions from Admin UI

Whatever we learnt so far cannot be perfected unless and until we spend some quality time playing around with the permissions system through the administration interface.

- 1) Login as portal administrator; go to **Control Panel → Portal → Roles**. Find out what are the types of roles and list down how many roles are there in each type.
- 2) Create a new Regular Role called “Librarian” and “Define Permissions” on that role. Once you click “Define Permissions” you will get this page to configure permissions on that Role.

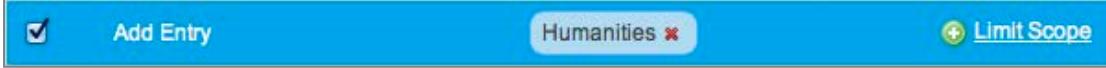
3) From the Drop-down list “Add Permissions” select “Library Portlet”. Out of six headings in the list as shown, our Portlet will appear under *Site Applications*.

- | | |
|---|---|
| <ul style="list-style-type: none"> • Portal • Site Content • Site Applications | <ul style="list-style-type: none"> • Control Panel: Personal • Control Panel: Site • Control Panel: Portal |
|---|---|

4) After selecting “Library Portlet”, you will see its list of actions. For every action listed out there, the default scope is “Portal”. Scope is another dimension for permission that gives the context in which a user can perform an action by virtue of having a particular role.

<input type="checkbox"/>	Action	Scope	
<input type="checkbox"/>	Access in Control Panel	Portal	Limit Scope
<input type="checkbox"/>	Add Entry	Portal	Limit Scope
<input type="checkbox"/>	Add to Page	Portal	Limit Scope
<input type="checkbox"/>	Configuration	Portal	Limit Scope
<input type="checkbox"/>	Permissions	Portal	Limit Scope
<input type="checkbox"/>	Preferences	Portal	Limit Scope
<input type="checkbox"/>	View	Portal	Limit Scope
<input type="checkbox"/>	action.CONFIG	Portal	Limit Scope

5) When you click on the “Limit Scope” link you will get this dialog with all existing sites listed out. Currently we have five sites. For any user having a Librarian role, we can restrict to perform an operation only within certain sites and not in the portal level. The next picture illustrates this concept. There is even a search box to find a site if there are many sites currently hosted in our Liferay installation.



Trust me, if you have to master the roles and permissions system of Liferay the BEST method is nothing but “**Trial and Error**”. This is the most complicated, yet sophisticated framework of Liferay. If you have to master it there are no other shortcuts than experimenting yourself with various scenarios. Once you start practicing it, you will start realizing the incredible power and flexibility it brings to the table.

Code Changes @ <http://code.google.com/p/lr-book/source/detail?r=63>

11.5 Liferay Asset Framework

Welcome to another world filled with all sophisticated features of Liferay. One of them is Asset Framework, a solid platform to bucket all the valuable resources of Liferay, so that they can leverage so many common things. Before we actually get in the subject let's find out the literal meaning of “asset”. The disctionary.com defines “asset” as 1) a useful and desirable thing or quality 2) a single item of ownership having exchange value. In the software parlance and especially Liferay the term asset is used as a generic way to refer to any type of content regardless of whether it's purely text, an external file, a URL, an image, a record in our online book library, etc. From now on, whenever the word asset is used, think of it as a generic way to refer to documents, blog entries, bookmarks, wiki pages, etc. Even a “user” is an asset.

11.5.1 Library Book as an Asset

Liferay's asset framework provides a set of functionalities that are common to several different content types. It was initially created to be able to add tags to blog entries, wiki pages, web content, etc without having to reimplement this same functionality over and over. Since then, it has grown to add more functionalities and it has been made possible to use the framework for custom applications even if they are implemented within a plugin and that's what we are going to do in the rest of this section. We'll register a book in our online Library as an asset and leverage all the benefits of the overarching asset framework.

Here are the main functionalities that you will be able to reuse by virtue of this great framework:

1. Associate tags to custom content types (*new tags will be created automatically when the author assigns them to the content*).
2. Associate categories to custom content types (*authors will only be allowed to select from predefined categories within several predefined vocabularies*)
3. Manage tags from the control panel (*including merging tags*)
4. Manage categories from the control panel (*including creating complex hierachies*).

-
5. Keep track of the number of visualizations of an asset.
 6. Publish your content using the Asset Publisher portlet. Asset Publisher is able to publish dynamic lists of assets or manually selected lists of assets.
 7. It is also able to show a summary view of an asset and offer a link to the full view if desired. Because of this it will save your time since for many use cases it will make it unnecessary to develop custom portlets for your custom content types.
 8. At the time of publishing assets you also have the options to allow users to perform various social networking features.

If these functionalities seem useful for and entity defined in your custom portlet, then you might be wondering, what do I've to do to benefit from them? The subsequent sub-sections will answer this question. We'll see all these eight points in great detail as we move forward.

11.5.2 Resources Verses Assets

During the course of previous sections, we have seen about a “resource”. The primary objective of a resource is to leverage the benefits of Liferay’s permissions system. Where as, an asset encompasses lot more things. But the registration process is one and the same. The registration of an “asset” is nothing different than the registration of a “resource”. Whenever a custom content is created you need to let the asset framework know about it. This is nothing but the combination of Definition and Registration process in DRAC. This process is quite simple. You just need to invoke a method of the asset framework. When invoking this method you will also let the framework know about the tags and/or categories of the content that was just created / authored. Let’s apply the process for our Library book involving two simple steps.

Step 1: Optionally inject “assetEntryLocalService” object through Spring into our “<Entity>*ServiceImpl” files by inserting the following tag at the end of entity definition in “service.xml” and re-run the service builder.

```
<reference entity="AssetEntry"
           package-path="com.liferay.portlet.asset" />
```

If you don’t want to an injection like this you may still use the corresponding utility class, “AssetEntryLocalServiceUtil” inside any of our DTO classes and invoke the methods of interest.

Step 2: At the end of “insertBook” in “LMSBookLocalServiceImpl” insert this below block of code to register the newly created book as an asset.

```
try {
    assetEntryLocalService.updateEntry(
        serviceContext.getUserId(), // userId
        serviceContext.getScopeGroupId(), // groupId
        lmsBook.getClass().getName(), // className
        lmsBook.getPrimaryKey(), // classPK
        serviceContext.getAssetCategoryIds(), // categoryIds
        serviceContext.getAssetTagNames()); // tag Names
} catch (PortalException e) {
```

```

        e.printStackTrace();
    } catch (SystemException e) {
        e.printStackTrace();
    }
}

```

There is couple of more variants of this “updateEntry” method. The first is too simple and the other is too elaborate. You can use any of these API’s of your choice and based on the need in the current application.

```

public AssetEntry updateEntry(
    String className, long classPK, Date publishDate,
    Date expirationDate, boolean visible);

```

The other variant of this method accepts too many parameters.

```

public AssetEntry updateEntry(
    long userId, long groupId, String className, long classPK,
    String classUuid, long classTypeId, long[] categoryIds,
    String[] tagNames, boolean visible, Date startDate, Date endDate,
    Date publishDate, Date expirationDate, String mimeType,
    String title, String description, String summary, String url,
    String layoutUuid, int height, int width, Integer priority,
    boolean sync);

```

Save the changes and add a book to our Library. Once this is done, let’s see the impacts to the database tables and especially the asset related tables. Out of the list of following [ten tables](#) related with “asset” framework, there is one entry that shows up quite prominently inside “assetentry” table (*shown with an asterisk*).

assetCategory	assetCategoryProperty
assetentries_assetcategories	assetentries_assettags
assetentry*	assetlink
assettag	assettagproperty
assettagstats	assetvocabulary

You can confirm that for the newly added book, there is an equivalent entry into the “assetentry” table as shown below.

entryId	groupId	companyId	userId	classNameId	classPK	visible	viewCount
20601	10179	10153	10195	10901	112	1	0

11.5.3 Summary of Parameters passed to updateEntry

Here is a quick summary of the most important parameters of “updateEntry” methods:

- **userId** – is the identifier of the user who created the content
- **groupId** – identifies the scope in which the content has been created. If your content does not support scopes, you can just pass 0 as the value.
- **className** – identifies the type of asset. By convention we recommend that it is the name of the Java class that represents your content type, but you can actually use any String you want as long as you’re sure that it is unique.

-
- **classPK** – identifies the specific content being created among any other of the same type. It is usually the primary key of the table where the custom content is stored.
 - **classUuid** – this parameter can optionally be used to specify a secondary identifier that is guaranteed to be unique universally. Having this type of identifier is especially useful if your contents will be exported and imported across separate portals.
 - **assetCategoryIds** and **assetTagNames** – they both represent the categories and tags that have been selected by the author of the content. The asset framework will store them for you.
 - **visible** – specifies whether this content should be shown at all by Asset Publisher. By default the value is true.
 - **title**, **description** and **summary** – they are descriptive fields that will be used by the Asset Publisher when displaying entries of your content type.
 - **publishDate** and **expirationDate** – these parameters can be optionally specified to let Asset Publisher know that it should not show the content before a given publication date or after a given expiration date.

All other fields are optional and might not make sense in all cases; hence you can just ignore them.

11.5.4 Proper Deletion of an Asset

Exactly how we did for a “resource”, when one of your custom content is deleted you should also let the Asset Framework know to clean up the information stored and also to make sure that the Asset Publisher doesn't show any information for a content that has been deleted. The signature of method to perform this is:

```
public void deleteEntry(String className, long classPK)
    throws PortalException, SystemException;
```

Inside our Portlet class or the DTO class the code for deleting the asset entry immediately after deleting the actual asset (*book*) will look like:

```
try {
    AssetEntryLocalServiceUtil.deleteEntry(
        LMSBook.class.getName(), lmsBook.getPrimaryKey());
} catch (PortalException e) {
    e.printStackTrace();
} catch (SystemException e) {
    e.printStackTrace();
}
```

11.6 Attaching Tags and Categories

In the previous section we have registered and designated a book in our Library as a valid asset. In this section we'll see how to really make use of its characteristic of being an asset in the asset framework. The first and foremost thing to leverage out of the asset framework is assigning tags and categories. This is going to be of great help

while listing the asset through “Asset Publisher” Portlet or searching the assets through “Faceted Search”. The next couple of sub-sections are going to explain the concept and the associated impacts happening to the database tables related to assets.

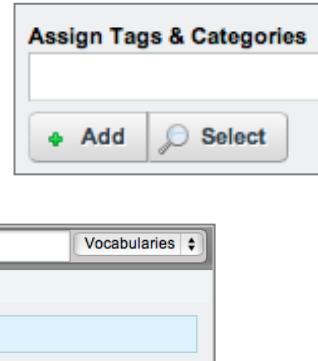
11.6.1 Changes to the User Interface

In order to attach tags and categories to our asset, we have to include the appropriate control to the book updation form. This is exactly similar to how we use control to set permissions on a resource. Open “**update.jsp**” and insert this block of code to make the new control appear in the form.

```
<aui:field-wrapper label="Assign Tags & Categories">
    <liferay-ui:asset-tags-selector
        className="<%=" LMSBook.class.getName() %>" 
        classPK="<%=" lmsBook.getPrimaryKey() %>"/>

    <liferay-ui:asset-categories-selector
        className="<%=" LMSBook.class.getName() %>" 
        classPK="<%=" lmsBook.getPrimaryKey() %>"/>
</aui:field-wrapper>
```

After saving, this control appears in our add book form. You can keep adding new tags on the fly or “Select” from the existing repository of tags. The categories will not appear by default. You have to go to “Categories” option in the Control Panel and add them to appear in our add book form.



If you’re using AlloyUI tags, the code will get further reduced. As the UI rendered by AUI is not that intuitive, let’s use the above code it self.

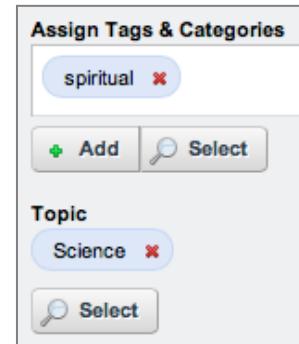
```
<aui:input name="tags" type="assetTags" />
<aui:input name="categories" type="assetCategories" />
```

11.6.2 How the information is stored?

Let’s add a new book to our Library with one tag and one category selected and see what’s happening at the backend. As before Liferay portal server, using the two methods of “`serviceContext`” object properly sets the values. You don’t have to literally worry about anything.

- `setAssetCategoryIds(assetCategoryIds);`
- `setAssetTagNames(assetTagNames);`

Soon after the addition go to database to check. Definitely there is one record insertion to “`assetentry`” table.



Impact 1: One record has been inserted into “**assetcategory**” table. This was actually happened when the new category was added through the control panel.

parentCategoryId	leftCategoryId	rightCategoryId	name	title
0	2	3	Science	<?xml version...

Impact 2: One record has got into “**assettag**” table.

userId	userName	createDate	modifiedDate	name	assetCount
10195	Test Test	2013-02-25...	2013-02-25...	spiritual	1

Impact 3: There is one record each in “**assetentries_assetcategories**” and “**assetentries_assettags**” tables that give the association with the actual “**assetentry**”.

entryId	categoryId
20801	20704

entryId	tagId
20801	20802

11.6.3 Showing Summary in Book Details Page

The next task is to make the list of tags and categories associated with a Library book in its detail page. This information will help our online library member to quickly decide whether to borrow this book or not. Open your “**detail.jsp**” and insert the below code in its proper location in order to make the summary appear when a book is viewed by the member. The code is very similar to the previous one used in the form, except the fact that it’ll be “**summary**” instead of “**selector**”.

```
<aui:field-wrapper label="Tags & Categories">
    <liferay-ui:asset-tags-summary
        className="<% LMSBook.class.getName() %>"
        classPK="<% lmsBook.getPrimaryKey() %>" />
    <liferay-ui:asset-categories-summary
        className="<% LMSBook.class.getName() %>"
        classPK="<% lmsBook.getPrimaryKey() %>" />
</aui:field-wrapper>
```

In both tags you can also use an optional parameter called “**portletURL**”. When specifying this parameter each of the tags will be a link built with the provided URL and adding a “tag” parameter or a “categoryId” parameter. This is very useful in order to provide support for tags navigation and categories navigation within your custom portlet. But you will need to take care of implementing this functionality yourself in the code of the portlet by reading the values of those two parameters and using the AssetEntryService to query the database for entries based on the specified tag or category. For your information the categories and tags of Liferay are also called as **Taxonomies** and **Folksonomies**, respectively.

Code Changes @ <http://code.google.com/p/lr-book/source/detail?r=64>

11.7 Publishing Assets (*Books*) through Asset Publisher

The other greatest benefit if designating our custom contents as assets is the ability to publish them through the Asset Publisher Portlet. It has been designed to display multiple assets. It has quite a few configuration options that we'll cover in this chapter. By default, abstracts (previews) of recently published assets are displayed by the Asset Publisher portlet and links to their full views are provided. You can configure the Asset Publisher portlet to display a table of assets, a list of asset titles, or the full content of assets.

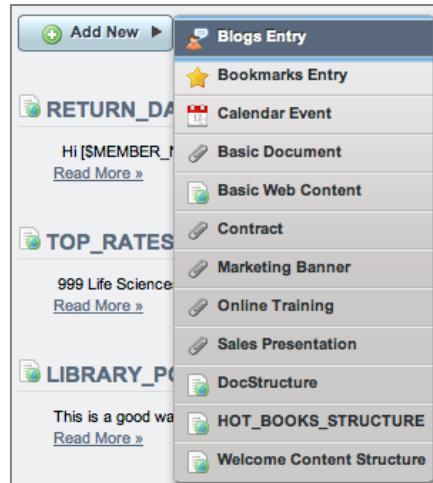
11.7.1 Configuring Asset Publisher Portlet

You can configure the Asset Publisher to display only certain kinds of assets and you choose how many items to display in a list. The Asset Publisher portlet is very useful for displaying chosen types of content, for displaying recent content, and for allowing users to browse content by tags and categories. The Asset Publisher is designed to integrate with the Tags Navigation and Categories Navigation portlets to allow this kind of filtering on the assets. The picture shows the various configuration options and settings available for displaying assets through “Asset Publisher” Portlet. Overall it is an excellent tool to display our contents in whatever way we want.



You can add an asset directly from the Asset Publisher Portlet. The various other options are:

- Support for several type of list displays
- Dynamic selection based on tags
- Manual selection of assets
- Support for grouping the assets
- Flexible selection of metadata information
- Quick creation and edition of assets
- Support for rating of assets
- Support for commenting the assets
- Configurable display options
- Pagination of listed assets



11.7.2 Asset Publisher Display – Basic Infrastructure

This sub section and the next bring us to the actual implementation for displaying our book through the Asset Publisher Portlet. We'll also try to give the option to add a book directly from the Asset Publisher Portlet. In order to be able to display your assets the Asset Publisher needs to know how to access some metadata of them and also needs you to provide templates for the different type of views that it can display (*abstract and full view*). You can provide all this information to the Asset Publisher through a pair of classes that implement the [AssetRenderer](#) interface and the [AssetRendererFactory](#) interface respectively. Let's see some explanation about these two important interfaces that helps in publishing custom assets.

AssetRenderer

This is the class that provides metadata information about one specific asset and is also able to check for permissions to edit or view it for the current user. It is also responsible for rendering the asset for the different templates (abstract, and full content), by forwarding to a specific JSP. It is also recommended that instead of implementing the interface directly, you extend from the [BaseAssetRenderer](#) class, that provides with nice defaults and more robustness for methods that could be added to the interface in the future.

AssetRendererFactory

This is the class that knows how to retrieve specific assets from the persistent storage from the “**classPK**” (*that is usually the primary key, but can be anything you have chosen when invoking the updateAsset method used to add or update the asset*). This class should be able to grab the asset from a “**groupId**” (*that identifies an scope of data*) and “**urlTitle**” (*which is a title that can be used in friendly URLs to refer uniquele to the asset within a given scope*).

Finally, it can also provide a URL that the Asset Publisher can use when a user wants to add a new asset of your custom type. This URL should point to your own portlet.

There are other less important methods, but you can avoid implementing them by extending from [BaseAssetRendererFactory](#). Extending from this class, instead of implementing the interface directly will also make your code more robust if there are changes in the interface in future versions of Liferay, since the base implementation will provide custom implementations for them.

11.7.3 Actual Implementation of Showing the Books

Having seen so much of theory, we are finally getting into the implementation part of making our Library books appear inside an instance of Asset Publisher Portlet. This involves four steps. Let's follow these steps religiously.

Step 1: Create the pages for showing the abstract view of a book and a full view of the book. Create two JSP files under “/html/library/asset”, “**abstract.jsp**” and “**full_content.jsp**” with just one line for including the “**init.jsp**”. Optionally you can put some title to differentiate between these two pages while they are viewed.

```
<%@include file="/html/library/init.jsp"%>  
  
<h1>This is abstract view of a book</h1>
```

Step 2: Write a new class “**LMSBookAssetRenderer**” making it to extend “**BaseAssetRenderer**” under the new package “**com.library.asset**”. Once the class is in place, let's make the following changes.

a) Define a local private variable and a parameterized constructor.

```
private LMSBook _lmsBook;  
  
public LMSBookAssetRenderer(LMSBook lmsBook) {  
    _lmsBook = lmsBook;  
}
```

b) Override the following getter methods of the base class.

```
public long getClassPK() {  
    return _lmsBook.getPrimaryKey();  
}  
  
public long getGroupId() {  
    return _lmsBook.getGroupId();  
}  
  
public String getSummary(Locale locale) {  
    return _lmsBook.getBookTitle();  
}  
  
public String getTitle(Locale locale) {  
    return _lmsBook.getBookTitle();  
}  
  
public long getUserId() {  
    return _lmsBook.getUserId();  
}
```

```

}

public String getUuid() {
    return _lmsBook.getUuid();
}

public String getUserName() {

    String userName = StringPool.BLANK;
    try {
        User user = UserLocalServiceUtil.fetchUser(getUserId());
        userName = user.getFullName();
    } catch (SystemException e) {
        e.printStackTrace();
    }
    return userName;
}

```

- c) Override and implement the “render” method that renders that actual page. Optionally you can define the new page paths in “**LibraryConstants.java**”.

```

public String render(RenderRequest request,
    RenderResponse response, String template)
    throws Exception {

    request.setAttribute("ASSET_ENTRY", _lmsBook);
    String page = "/html/library/asset/abstract.jsp";

    if (template.equals(TEMPLATE_FULL_CONTENT)) {
        page = "/html/library/asset/full_content.jsp";
    }

    return page;
}

```

Step 3: Let’s write a new class “**LMSBookAssetRendererFactory**” extending “**BaseAssetRendererFactory**” under the same package overriding these three methods to begin with.

```

public AssetRenderer getAssetRenderer(long bookId, int type)
    throws PortalException, SystemException {

    LMSBook lmsBook = LMSBookLocalServiceUtil.fetchLMSBook(bookId);
    return new LMSBookAssetRenderer(lmsBook);
}

public String getClassName() {
    return LMSBook.class.getName();
}

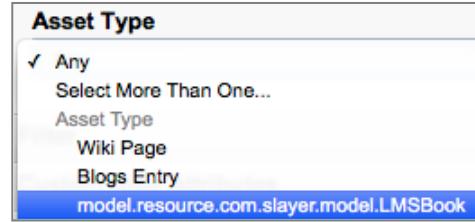
public String getType() {
    return "book";
}

```

Step 4: Coming to the most important step, we need to link our Library Portlet with the new AssetRendererFactory class that we just created. This linking happens in “**liferay-portlet.xml**” as usual. Open this file and insert this entry in the appropriate place, so that there is no conflict with its corresponding DTD.

```
<asset-renderer-factory>
    com.library.asset.LMSBookAssetRendererFactory
</asset-renderer-factory>
```

- a)** Save your changes and check. When you go to the configuration settings of an Asset Publisher Portlet, you should see a new option to select the Library books and display them within the Asset Publisher Portlet. But the only issue is it is showing a long class name. Let's fix it now.



- b)** Put the below entry in our Portlet's "**Language_en.properties**" file and check the configuration now. A proper label should appear. If I put this entry in "**Language.properties**", the label is not showing up.

```
model.resource.com.slayer.model.LMSBook=Library Book
```

Code Changes @ <http://code.google.com/p/lr-book/source/detail?r=65>

11.8 Asset Publisher for Books – Finishing Touches

In this section, we'll give couple of more finishing touches to the way we have made our Library books getting filtered through the Asset Publisher Portlet. First and foremost you can apply all kinds of configuration options. Check all the options under the "Display Settings" section and see the behavior of the full view of the book. The options to print and comments are not showing up even after checking these options. We'll find out what could have caused these issues. In the interim, we'll complete the other stuff and make the functionality as complete as possible.

<input checked="" type="checkbox"/> Enable Print
<input checked="" type="checkbox"/> Enable Flags
<input checked="" type="checkbox"/> Enable Permissions
<input checked="" type="checkbox"/> Enable Related Assets
<input checked="" type="checkbox"/> Enable Ratings
<input checked="" type="checkbox"/> Enable Comments
<input checked="" type="checkbox"/> Enable Comment Ratings
<input checked="" type="checkbox"/> Enable Social Bookmarks

11.8.1 Improving abstract view and full view pages

Let's go back to the place where we started and put some real contents inside the abstract and full_content views. Open these two JSP files and insert this one line scriplet. You can always do all other further beautifications as you wish.

```
<%>
    LMSBook lmsBook = (LMSBook)request.getAttribute("ASSET_ENTRY");
%>
```

Do you rememeber? This object we have set in the overridden "render" method of "**LMSBookAssetRenderer**" class.

11.8.2 Making book “addable” from Asset Publisher

If you observe closely, the “Add New” option is not coming when we select the asset as “Library Book”. How to make them appear, so that we can add the book directly from here than going over to the Library Portlet? It’s quite simple. All you have to do is to introduce a “getURLAdd” in our “**LMSBookAssetRendererFactory**”. That’s it!

```
public PortletURL getURLAdd(
    LiferayPortletRequest liferayPortletRequest,
    LiferayPortletResponse liferayPortletResponse)
throws PortalException, SystemException {

    HttpServletRequest request =
        liferayPortletRequest.getHttpServletRequest();

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

    if (!LibraryPermissionUtil.hasPermissionToAddBook(
        themeDisplay.getPermissionChecker())) {
        return null;
    }

    PortletURL portletURL = PortletURLFactoryUtil.create(
        request, "library_WAR_libraryportlet",
        getControlPanelPlid(themeDisplay),
        PortletRequest.RENDER_PHASE);
    portletURL.setParameter("jspPage",
        LibraryConstants.PAGE_UPDATE);

    return portletURL;
}
```

Once you save the changes and check the Asset Publisher now. Click on this link and the popup opens.



11.8.3 Enabling the “Print” option

One thing which is still left is the ability to see the “Print” link in the full content view of the book. To enable this, you have to add the following three methods to our “**LMSBookAssetRenderer**” class (*ensure the print option is checked in the Portlet configuration*).

```
public boolean isConvertible() {
    return true;
}

public boolean isLocalizable() {
    return true;
}

public boolean isPrintable() {
    return true;
```

```
}
```

11.8.4 Links to comments

Inspite of checking the option to “Enable Comment” we are not seeing that still. What we need to do to really enable this option? Just override another method inside our “**LMSBookAssetRenderer**” class.

```
public String getDiscussionPath() {  
    return "edit_entry_discussion";  
}
```

Now, you should see the feature to add comments on this Library book.

No comments yet. Be the first.  [Subscribe to Comments](#)

Code Changes @ <http://code.google.com/p/lr-book/source/detail?r=66>

11.8.5 New Friendly URL through Asset Publisher

One last thing before we conclude. If observe the URL in the browser location bar while viewing a book in the full_content view. It automatically appears in the FriendlyURL format. This is another great advantage of using Asset Publisher for publishing the various assets of the Liferay portal and making it uniquely traceable.

```
/web/guest/what-we-do/-/asset_publisher/eresq1heRGhP/book/id/23201
```

The value in red is the particular instance of the Asset Publisher Portlet. Based on this value the portal server properly redirect the request to the page where the instance is place and shows the asset in its full view mode.

Congratulations! You have completed learning and experimenting one of the strong features of Liferay. Sure you are enjoying going through these sections that are covered in detail; the Asset Framework and Asset Publisher Portlet. There are lot more things you can achieve on these premises. Some of them we'll see in the later chapters. The next chapter is going to cover lot more powerful frameworks of Liferay, starting with the storage framework.

Summary

In this chapter we have touched many important sub-frameworks of Liferay. We started with Liferay's security and Permissions framework. This is an extremely powerful framework that provides security at Portal level, porlet level and model levels. We have also seen how to define custom utility classes for checking the permissions before performing an operation, exactly the same way that Liferay has implemented this in various places of the portal source code.

In the next part of this chapter, we have given complete attention towards understanding and implementing Liferay's Asset Framework, another extremely powerful framework of Liferay that helps in leveraging many useful features for all assets defined in the system.

12. Liferay Frameworks – Part 2

This chapter covers

- Liferay Storage Framework
 - MVCPortlet and Large Applications
 - Showing Book Cover in the List Page
 - File Processing and Liferay
 - Server Side Validation
 - Custom Fields Framework
 - Enabling Custom Fields for Library Portlet
-

This chapter will cover another set of extremely powerful and exciting mini frameworks within Liferay. We'll start the chapter with Liferay's storage framework using which you can store anything to the back-end, index them, search them and retrieve them. We'll see how to store and retrieve images and files. Our discussion will move forward to see how the Liferay MVC framework can cater to the needs of a large application. Then we'll see how to perform server side validations and showing messages to the user using SessionMessages and SessionErrors API's.

The chapter will end with a complete overview of custom fields and how to enable custom fields for our Library Portlet. We'll see all the concepts behind expando tables and how to programmatically manipulate them.

12.1 Liferay Storage Framework

The File Storage Framework allows storing files using the back-end of Liferay's Document Library System. By using this framework you won't have to worry about clustering or backups since they'll be already taken care of by the Document Library itself. The wiki and the message boards of Liferay to store attached files in pages and posts respectively use this framework, for example. The Document Library of Liferay has the ability to process both files and images. We'll cover both these options in this section. While discussing these topics we'll also cover couple of other interesting topics. I am going to keep that as a suspense now and tell you later.

12.1.1 Uploading Cover Page of the Book

Let's start this section with Image processing capabilities of Liferay. This is a part of Liferay's storage framework. Let's tie this subject with a core requirement of our Library Management System. After a book has been added, we'd like to upload the cover page of the book. This will help the library members to get a good overview about the book they are going to borrow just by looking at the cover page. The exercise consists of four steps.

Step 1: The first thing we are going to do is to upload all the book cover images to the "image" table of Liferay instead of creating a new table. This table has seven columns and a column to store the actual image that's being uploaded to the portal.

Field	Type	Null	Key
imageId	bigint(20)	NO	PRI
modifiedDate	datetime	YES	
text_	longtext	YES	
type_	varchar(75)	YES	
height	int(11)	YES	
width	int(11)	YES	
size_	int(11)	YES	MUL

To benefit from this existing infrastructure, we are going to make use of either "**ImageLocalServiceUtil**" or injecting "**imageLocalService**" object into our DTO class. Let's go with the second approach and this requires a new "**reference**" tag to be defined in "**service.xml**" and the service layer re-generated.

```
<reference package-path="com.liferay.portal" entity="Image" />
```

Step 2: Let's get our back-end ready first. Let create a new method in "**LMSBookLocalServiceImpl**" to obtain the image through the "**serviceContext**" object and inserting it to the database by invoking the method on "**imageLocalService**". Give a generic name for this new method.

```
public void attachFiles(long bookId, ServiceContext serviceContext) {
    // uploading the image
    File coverImage = (File)
        serviceContext.getAttribute("COVER_IMAGE");
```

```

    if (Validator.isNotNull(coverImage)) {
        try {
            imageLocalService.updateImage(
                bookId, coverImage);
        } catch (PortalException e) {
            e.printStackTrace();
        } catch (SystemException e) {
            e.printStackTrace();
        }
    }
}

```

Rerun the Service Builder as we have added a new method. Our back-end is ready. It's time to work on the front-end.

Step 3: Let's begin making changes to our front-end. We are not going to disturb our original add book form “**update.jsp**”. There are reasons for it. The first reason is we did not want to clutter that form with too much of information. It already has many fields. The second reason will come obvious very soon.

a) Create a new file “**upload.jsp**” with a simple form.

```

<%@include file="/html/library/init.jsp"%>
<%
    String bookId = ParamUtil.getString(request, "bookId");
    String redirectURL = ParamUtil.getString(request, "redirectURL");
%>
<portlet:actionURL var="uploadFilesURL" name="UploadFiles" />

<aui:form action=<%= uploadFilesURL %>
    enctype="multipart/form-data">
    <aui:input type="hidden" name="bookId" value=<%= bookId %> />
    <aui:input type="hidden" name="redirectURL"
               value=<%= redirectURL %> />
    <aui:input name="coverImage" type="file" />
    <aui:button type="submit" />
</aui:form>

```

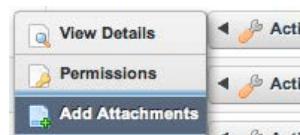
Note that there are two hidden variables “bookId” and “redirectURL”. The first one is required to store the actual image and the second is for redirecting the request back to the list page after the image gets uploaded to the system.

b) Open “**actions.jsp**” and include a new link to show up under the “Actions” icon menu.

```

<liferay-ui:icon url=<%= attachFilesURL %>
    image="add_article" message="add-attachments" />

```



The “attachFilesURL” will be declared like this,

```

<portlet:renderURL var="attachFilesURL">
    <portlet:param name="jspPage"
                  value="<%= LibraryConstants.PAGE_UPLOAD %>" />
    <portlet:param name="redirectURL"
                  value="<%= themeDisplay.getURLCurrent() %>" />
    <portlet:param name="bookId"
                  value="<%= Long.toString(book.getBookId()) %>" />

```

```
</portlet:renderURL>
```

Save changes and confirm everything is working fine. But we have still not written the code in our Portlet class to process the uploaded image and set it in our “serviceContext” object so that the DTO API can retrieve the image from there and insert into the database.

What should be the value for “image” attribute of “liferay-ui:icon” tag?

It is nothing but the name of an icon inside your current themes “/images/common” folder. The icon file should be present inside this folder for properly displaying the icon in the icon-menu.

Step 4: We are going to write the gluing code in our “**LibraryPortlet.java**” that’ll get the image uploaded from the form and pass it to the back-end API by putting that file into the “serviceContext” object.

```
public void UploadFiles(ActionRequest actionRequest,
                        ActionResponse actionResponse)
                        throws IOException, PortletException {

    UploadPortletRequest uploadRequest =
        PortalUtil.getUploadPortletRequest(actionRequest);

    File coverImage = uploadRequest.getFile("coverImage");

    if (coverImage.getTotalSpace() > 0) {
        long bookId =
            ParamUtil.getLong(uploadRequest, "bookId");
        try {
            ServiceContext serviceContext =
                ServiceContextFactory
                    .getInstance(actionRequest);

            serviceContext
                .setAttribute("COVER_IMAGE", coverImage);
            LMSBookLocalServiceUtil
                .attachFiles(bookId, serviceContext);
        } catch (PortalException e) {
            e.printStackTrace();
        } catch (SystemException e) {
            e.printStackTrace();
        }
    }

    // redirecting to original list page
    actionResponse.sendRedirect(
        ParamUtil.getString(uploadRequest, "redirectURL"));
}
```

Now, for any book that is already in the library, click “Add Attachments” to upload the cover image. After you upload, you will be taken back to the original list page. But any idea what has happened behind your computer screen? Let’s examine what this guy has done.

```
LMSBookLocalServiceUtil.attachFiles(bookId, serviceContext);
```

12.1.2 Image upload – Backend Impacts

After the successful upload of the cover page image, the following two changes have happened in the backend – one to our actual “image” table and the other to our document library repository on the server’s file system. Let’s see both of them here.

Impact 1: One record got inserted into the “image” table with the `imageId` as the “`bookId`” that we used at the time of invoking “`imageLocalService.updateImage`” inside our DTO class.

imageId	modifiedDate	text_	type_	height	width	size_
107	2013-02-26...		jpg	480	640	59177

This record contains all the attributes of the image you uploaded. But did u notice the “`text_`” field that is supposed the image data is empty? Surprisingly where did the image data go? Don’t panic. It is safely sitting in our file system. Let’s see where exactly in the file system. This is the next impact.

Impact 2: The actual file is sitting inside the folder where allt he document library files are stored by default that is, “`{LIFERAY_HOME}/data/document_library`”.

-0
---0
----107.jpg

In the upcoming sub-section we’ll see more details about the document library and it’s storage mechanism using Jackrabbit.

12.1.3 Jackrabbit and Document Library

The document library module of Liferay by default uses an apache library called “**jackrabbit**” to store the uploaded files in the file system rather than storing in the database. There are many advantages of storing the files in the filesystem; the most important of them is the speed with which they can be retrieved. [Apache Jackrabbit](#) content repository is a fully conforming implementation of the Content Repository for Java Technology API. A content repository is a hierarchical content store with support for structured and unstructured content, full text search, versioning, transactions, observation, and more. In Liferay the jackrabbit settings are controlled by the file, “`{LIFERAY_HOME}/data/jackrabbit/repository.xml`”.

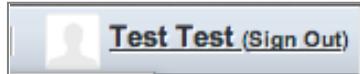
Apart from the settings in this file, Liferay has numerous settings in “**portal.properties**” that control the document library data storage behavior. The primary storage is decided by “`d1.store.impl`” entry in this file. The default value is, “`com.liferay.portlet.documentlibrary.store.FileSystemStore`”. Apart from this, we can enable other stores as well. They are:

```
com.liferay.portlet.documentlibrary.store.AdvancedFileSystemStore
com.liferay.portlet.documentlibrary.store.CMISStore
com.liferay.portlet.documentlibrary.store.DBStore
com.liferay.portlet.documentlibrary.store.JCRStore
com.liferay.portlet.documentlibrary.store.S3Store
```

All other document library related settings start with the prefix “`dl.`”. You can go through these options to know details of how you can control / override them through “`portal-ext.properties`”.

Do it Yourself:

Login to portal. Click on the link at the top right corner.
Upload your image and check what's happening.



12.2 MVCPortlet and Large Applications

Friend, have you ever wondered why I've used an action name “`UploadFiles`” that starts with a capital letter? Don't you think this is something against the general [Java coding standards](#)? I've also told in the beginning of this section that we'll write this as a new action method instead of making our original “`updateBook`” to handle the new upload request.

12.2.1 The two main reasons

First and foremost, the first reason being, whenever we use the form attribute, `enctype="multipart/form-data"`, we have to retrieve the form elements from “`uploadRequest`” as they are no longer available in the original “`actionRequest`”. You do the conversion with the help of this statement,

```
UploadPortletRequest uploadRequest =  
    PortalUtil.getUploadPortletRequest(actionRequest);
```

Coming to the second most important reason that compelled me to keep this as a separate action method in our Portlet class, I do not encourage people to write code against java and [Liferay coding standards](#). Having said this, now we are coming to the most important aspect that made me to break my own principles.

Just go back and check the number of lines in our “`LibraryPortlet.java`”. We have already exceeded 400 lines of code. This is a violation of the rule to keep the number of lines in a java class within an optimal limit. If our Portlet application is going to be really big there will be more number of action methods and it's going to drastically increase the size of this one single file. Ultimately the maintainability is going to get compromised. More over if each developer in the team is working on one action; we'll get into integration problems as everyone is working on the same file and updating it.

If you have already worked on other MVC frameworks like Struts and JSF, you will definitely hate this monolithic approach of keeping everything at one single place. In struts, you would have written every action as a separate action class. In JSF you would have written them as Controller classes. Then why not in Liferay MVC framework? Even LiferayMVC allows you to break down the actions into multiple class files. All you have to do is to following some simple naming conventions and the portal server takes care of the rest. There are only four requirements to do this.

-
1. The new action class should implement the [ActionCommand](#) interface that has only one method defined, “processCommand”.
 2. The actual class name should end with the suffix “**ActionCommand**”, e.g., **UploadFilesActionCommand.java**.
 3. We need to define a new “`<init-param>`” in our “**portlet.xml**” with name as “action.package.prefix” and value the actual java package where all the action classes are residing.
 4. All our action classes should reside inside the package we have defined above.

12.2.2 Moving our code to a new File

This sub-section is dedicated for actually implementing an `ActionCommand` class and ensuring that everything is working fine as usual. Let’s get started. I am going with the assumption that you have already inserted this block in your “**portlet.xml**” satisfying one of the three key requirements.

```
<init-param>
    <name>action.package.prefix</name>
    <value>com.library.action</value>
</init-param>
```

Step 1: Create a new class under package “**com.library.action**” with the name “**UploadFilesActionCommand**” implementing “`ActionCommand`” interface. The empty class will look like below.

```
package com.library.action;

import javax.portlet.PortletException;
import javax.portlet.PortletRequest;
import javax.portlet.PortletResponse;
import com.liferay.util.bridges.mvc.ActionCommand;

public class UploadFilesActionCommand implements ActionCommand {

    public boolean processCommand(PortletRequest portletRequest,
        PortletResponse portletResponse)
        throws PortletException {
        return true;
    }
}
```

The method “`processCommand`” returns either `true` or `false` based on the successful execution of the action being performed.

Step 2: Remove the method “`UploadFiles`” we wrote inside our “**LibraryPortlet**” earlier and move the body of that method (*plain insertion*) inside this new method, “`processCommand`”. No other changes are required in our JSP files.

```
ActionRequest actionRequest = (ActionRequest) portletRequest;
ActionResponse actionResponse = (ActionResponse) portletResponse;
```

```

UploadPortletRequest uploadRequest =
    PortalUtil.getUploadPortletRequest(actionRequest);

File coverImage = uploadRequest.getFile("coverImage");

if (coverImage.getTotalSpace() > 0) {
    long bookId = ParamUtil.getLong(uploadRequest, "bookId");
    try {
        ServiceContext serviceContext =
            ServiceContextFactory
                .getInstance(actionRequest);

        serviceContext.setAttribute("COVER_IMAGE", coverImage);
        LMSBookLocalServiceUtil
            .attachFiles(bookId, serviceContext);
    } catch (PortalException e) {
        e.printStackTrace();
    } catch (SystemException e) {
        e.printStackTrace();
    }
}

// redirecting to original list page
try {
    actionResponse.sendRedirect(
        ParamUtil.getString(uploadRequest, "redirectURL"));
} catch (IOException e) {
    e.printStackTrace();
}

```

Test the upload process now and confirm that everything is working fine. You should see the new record getting inserted in our “**image**” table. If you still don’t believe put some SOP statements in the above method to reconfirm and see for yourself.

But unfortunately this approach has broken our earlier functionality. If we go with individual action classes, then the action methods in our Portlet class are not getting executed. So, this is a design decision that you need to take at the beginning of the project itself. For the time being, I am going to revert back the changes we just made, so that everything works as before. But if you want to move all your methods into individual classes, you can do that as well. It’s up to you to decide what route you want to take.

Code Changes @ <http://code.google.com/p/lr-book/source/detail?r=67>

12.3 Showing Book Cover in the List Page

Our Assistant Librarian has uploaded the cover page images for all the books in our Library. But he also feels it’ll be nice if the cover image is shown in the list page where the member comes and looks out for a book.

12.3.1 Direct fetching of an Image

Using a servlet we can directly pull an image sitting in our backend. Let's include a new column in our “list.jsp” to see this in action and then discuss the details.

```
<%
    long bookId = book.getBookId();
    StringBuilder imageURL = new StringBuilder()
        .append(themeDisplay.getPathImage())
        .append("/book/?img_id=")
        .append(bookId)
        .append("&t=")
        .append(WebServerServletTokenUtil.getToken(bookId));
%>
<liferay-ui:search-container-column-text name="Cover Page">
    <img src=<%= imageURL.toString() %>
        height="125px" width="100px" />
</liferay-ui:search-container-column-text>
```

Check the list now. It should properly display the cover image of every book. For books without an image, it'll display an icon with question mark (or a X mark), indicating the image is not available. If we analyse the generated image URL it is of the following format.

```
/image/book/?img_id=104&t=1361863335548
```

This URL actually invokes a Servlet that is part of our Liferay server and returns the value of the image. If you check the “web.xml” inside PORTAL_SRC, you will find this entry.

```
<servlet-mapping>
    <servlet-name>Web Server Servlet</servlet-name>
    <url-pattern>/image/*</url-pattern>
</servlet-mapping>
```

“Web Server Servlet” resolves to an actual servlet, [WebServerServlet](#). You can go through the code for this servlet and understand what it does. The next question is why we have used “[WebServerServletTokenUtil](#)” to generate a unique token? This helps in getting the image from the portal server cache when the second time the same URL is invoked. The cache maintains the original image id and its unique token.

12.3.2 Serving the Image through serveResource

There are few issues in the previous approach to retrieve the image.

1. In the page that shows a list of books, displaying the actual image is going to adversely affect the page load time. Usually in lists, we show only the “thumbnail” view and in the details page, we should show the actual image.
2. While showing the image, we have hardcoding the height and image of the image using those attributes of the HTML “IMG” tag. But this is not right, as the image

appearance will be skewed. The height and width of the image has to be proportional with the image's original dimensions that are captured in the “image” table.

3. If the cover page is not available for a book, then we have to show a default image instead of showing the image not available icon.

In this sub-section, we are going to mitigate the first one out of the above two problems by introducing a “serveResource” method in our Portlet class that will do all the required massaging of the image. This exercise requires two steps.

Step 1: Let's override the “serveResource” in our “**LibraryPortlet.java**” Portlet class and make it like the one below.

```
public void serveResource(ResourceRequest resourceRequest,
    ResourceResponse resourceResponse) throws IOException,
    PortletException {

    String cmd =
        ParamUtil.getString(resourceRequest, Constants.CMD);

    if (cmd.equalsIgnoreCase("serveImage")) {
        long imageId =
            ParamUtil.getLong(resourceRequest, "imageId");

        Image image = null;
        try {
            image = ImageLocalServiceUtil.fetchImage(imageId);
        } catch (SystemException e) {
            e.printStackTrace();
        }

        if (Validator.isNotNull(image)) {
            byte[] bytes = image.getTextObj();

            HttpServletResponse response = PortalUtil
                .getHttpServletResponse(resourceResponse);

            response.setContentType(image.getType());
            ServletResponseUtil.write(response, bytes);
        }
    }
}
```

Step 2: Replace the code we have written earlier through a StringBuilder to form the image URL with this code.

```
ResourceURL imageURL = renderResponse.createResourceURL();
imageURL.setParameter(Constants.CMD, "serveImage");
imageURL.setParameter("imageId", String.valueOf(bookId));
imageURL.setParameter("t",
    WebServerServletTokenUtil.getToken(bookId));
```

Save the changes and everything should work as before, except that now our Portlet with the help of “serveResource” method is serving the image. We have not done any massaging on the original image. It is displayed “as-is” basis. You can right click on one of the images and save it to your drive.

12.3.3 Scaling the image as a Thumbnail

Just check the photo you uploaded through “My Account”. How it is appearing on your window’s top right corner? We are going to do the same for the cover page image for our books. Definitely not as small as the one you’re seeing. To achieve this, we are going to modify our “serveResource” method to scale the original image as a thumbnail and then serve the image to the front-end. We are going to scale the image to 20% of its original size. Instead of writing the code for scaling in the Portlet class, let’s define a new general-purpose static method “getScaledImage” in our utility class “**LMSUtil.java**”. The first parameter is the actual image object and the second is the scaling factor, which is nothing but a percentage value.

```
public static byte[] getScaledImage(Image image, int percent) {  
  
    byte[] bytes = null;  
    ImageBag imageBag = null;  
    try {  
        imageBag = ImageToolUtil.read(image.getTextObj());  
    } catch (IOException e) {  
        e.printStackTrace();  
    }  
  
    if (Validator.isNull(imageBag)) return null;  
  
    RenderedImage renderedImage = imageBag.getRenderedImage();  
  
    long height = Math.round(image.getHeight() * percent/100);  
    long width = Math.round(image.getWidth() * percent/100);  
  
    renderedImage = ImageToolUtil.scale( renderedImage,  
                                         (int) height, (int) width);  
    try {  
        bytes = ImageToolUtil.getBytes(renderedImage,  
                                         image.getType());  
    } catch (IOException e) {  
        e.printStackTrace();  
    }  
  
    return bytes;  
}
```

Make the necessary imports in this class required by the new method.

```
import java.awt.image.RenderedImage;  
import java.io.IOException;  
import com.liferay.portal.kernel.image.ImageBag;  
import com.liferay.portal.kernel.image.ImageToolUtil;  
import com.liferay.portal.kernel.util.Validator;  
import com.liferay.portal.model.Image;
```

Go back to our “**LibraryPortlet.java**” and insert this line just before we write the response object using [ServletResponseUtil](#).

```
bytes = LMSUtil.getScaledImage(image, 20);
```

Check the new image now and save it. Its size should be 20% less than the original image. See how powerful these Liferay features are? Imagine how much effort will be required if you have to do all these things yourself in your application. Liferay has infact saved you from doing all these low level things. In many of the default Liferay portlets this image scaling is used. One good example is the shopping Portlet.

Code Changes @ <http://code.google.com/p/lr-book/source/detail?r=69>

Do it Yourself: In “**list.jsp**” we no longer require the attributes “`height="125px"` `width="100px"`” for image tag as this makes the scaled image appear in a blurred fashion. Remove these attributes and check how the cover images are appearing now.

12.4 File Processing and Liferay

This section will introduce you to another great feature of the Document Library API. Using this API, you can directly store and retrieve files in the file system unlike the last one where we stored the metadata about the images in the database and the actual image in the file system. We’ll see these capabilities with some nice use cases related with your Library Portlet. Imagine if we want to improvise our “Add Attachments” with the ability to attach the sample chapters of the library book, so that any member can read the sample chapter before blocking a book for borrowing.

12.4.1 Simple Storage to DLStore

This process of storing the file consists of five steps. Let’s implement those here.

Step 1: Update our “**service.xml**” to have a new field for our “**LMSBook**” entity and manually alter the underlying table to have this column. Re-run Service Builder.

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

```
ALTER TABLE lms_lmsbook ADD COLUMN sampleChapter VARCHAR(75) NULL  
AFTER groupId;
```

Step 2: Open “**upload.jsp**” and add a new field in the form to upload this file. Make a corresponding entry in “**Language.properties**” for the new label, “`sampleChapter=Sample Chapter`”.

```
<aui:input name="sampleChapter" type="file" />
```

Step 3: Make changes to “**UploadFiles**” method in our “**LibraryPortlet.java**” to get this file from the “**uploadRequest**” and set it in the “**serviceContext**” object, so that we can process it properly in the backend API. I’ve made some changes to the original code, in order to make it cleaner.

```
ServiceContext serviceContext = null;  
try {
```

```

        serviceContext =
            ServiceContextFactory.getInstance(actionRequest);
    } catch (PortalException e) {
        e.printStackTrace();
    } catch (SystemException e) {
        e.printStackTrace();
    }

    File coverImage = uploadRequest.getFile("coverImage");

    if (coverImage.getTotalSpace() > 0) {
        serviceContext.setAttribute("COVER_IMAGE", coverImage);
    }

    File sampleChapterFile = uploadRequest.getFile("sampleChapter");
    if (sampleChapterFile.getTotalSpace() > 0) {
        serviceContext.setAttribute(
            "SAMPLE CHAPTER", sampleChapterFile);
        String fileName = uploadRequest.getFileName("sampleChapter");
        serviceContext.setAttribute("FILE_NAME", fileName);
    }

    long bookId = ParamUtil.getLong(uploadRequest, "bookId");
    LMSBookLocalServiceUtil.attachFiles(bookId, serviceContext);

```

Step 4: We'll start with creating a static method in "**LMSUtil.java**" that helps in creating a folder in our file system if it is not there already.

```

public static void createFolder(String folderName, long companyId) {
    long repositoryId = CompanyConstants.SYSTEM;
    boolean folderExists = false;

    try {
        folderExists = DLStoreUtil.hasDirectory(
            companyId, repositoryId, folderName);
    } catch (PortalException e) {
        e.printStackTrace();
    } catch (SystemException e) {
        e.printStackTrace();
    }
    if (!folderExists) {
        try {
            DLStoreUtil.addDirectory(
                companyId, repositoryId, folderName);
        } catch (PortalException e) {
            e.printStackTrace();
        } catch (SystemException e) {
            e.printStackTrace();
        }
    }
}

```

Step 5: Let's now put our attention towards the actual API, "`attachFiles`" in our DTO class (**LMSBookLocalServiceImpl**) and make necessary changes to store the file to the file system using "`DLFileStore`". Update the existing method with these additional lines of code.

```

File sampleChapter = (File)
    serviceContext.getAttribute("SAMPLE CHAPTER");

```

```

if (Validator.isNotNull(sampleChapter)) {
    LMSBook lmsBook = null;
    try {
        lmsBook = fetchLMSBook(bookId);
    } catch (SystemException e) {
        e.printStackTrace();
    }

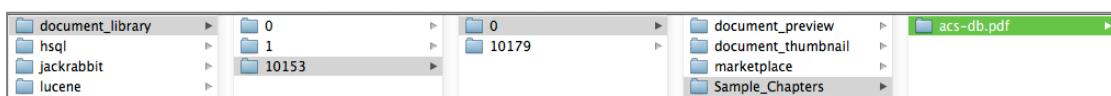
    if (Validator.isNotNull(lmsBook)) {
        // 1. update the original object
        String fileName = (String)
            serviceContext.getAttribute("FILE_NAME");
        lmsBook.setSampleChapter(fileName);
        try {
            updateLMSBook(lmsBook);
        } catch (SystemException e) {
            e.printStackTrace();
        }

        // 2. create a folder to store this file
        long companyId = serviceContext.getCompanyId();
        String folderName = "Sample_Chapters";
        LMSUtil.createFolder(folderName, companyId);

        // 3. Saving the file now
        String filePath =
            folderName + StringPool.SLASH + fileName;
        try {
            DLStoreUtil.addFile(companyId,
                CompanyConstants.SYSTEM, filePath, sampleChapter);
        } catch (PortalException e) {
            e.printStackTrace();
        } catch (SystemException e) {
            e.printStackTrace();
        }
    }
}

```

Go back to your Portlet's "Add Attachments" and upload the sample chapter for one of the books in Library. Let's see what is happening in the backend. I did the same and the book has got successfully uploaded to the file system. The following is the screenshot of the directory tree in my Macbook.



You can confirm with your machine. In the whole process nothing has happened to the database and no records got inserted to any of the tables, except the "lmsBook" record has got updated with the name of the sample chapter we just uploaded.

uuid_	companyId	userId	userName	groupId	sampleChapter
da4423fe-d2...	10153	10157		10179	acs-db.pdf

Congratulations! You have successfully saved the file to the DLFileStore. All these magic has been single handedly performed by one class [DLStoreUtil](#). This class is the

window to access the underlying file system repository. You can check all other methods of this powerful class. The next thing is to retrieve the file and make it available to our library members.

Code Changes @ <http://code.google.com/p/lr-book/source/detail?r=70>

12.4.2 Retrieving a file from DLFileStore

The file has been successfully saved to the file system repository. The next task is to show a link to the members of our Library to view the sample chapter for any book they'd like to borrow. Let's complete the UI changes first.

Step 1: Open “list.jsp” and introduce a new column that will have a link to open the sample chapter of a book. The column will be empty for books without a sample chapter.

```
<%
    ResourceURL chapterURL = renderResponse.createResourceURL();
    chapterURL.setParameter(Constants.CMD, "serveFile");
    chapterURL.setParameter("bookId", String.valueOf(bookId));
    chapterURL.setParameter("fileName", book.getSampleChapter());
    chapterURL.setParameter("t",
        WebServerServletTokenUtil.getToken(bookId));
%>
<liferay-ui:search-container-column-text property="sampleChapter"
    name="Sample Chapter" href="<% chapterURL.toString() %>" />
```

After saving and checking the changes, the link will appear with the name of the file. When you click the link, nothing will happen, as we have still not written the required logic in our “serveResource” method of the Portlet class.

Step 2: Open “LibraryPortlet.java” and insert this code in the existing method “serveResource”.

```
if (cmd.equalsIgnoreCase("serveFile")) {
    String fileName =
        ParamUtil.getString(resourceRequest, "fileName");

    long companyId = PortalUtil.getCompanyId(resourceRequest);
    long repositoryId = CompanyConstants.SYSTEM;
    String filePath =
        "Sample_Chapters" + StringPool.SLASH + fileName;

    File file = null;
    try {
        file = DLStoreUtil.getFile(
            companyId, repositoryId, filePath);
    } catch (PortalException e) {
        e.printStackTrace();
    } catch (SystemException e) {
        e.printStackTrace();
    }

    if (Validator.isNotNull(file)) {
```

```

        byte[] bytes = FileUtil.getBytes(file);
        String contentType =
            new MimetypesFileTypeMap().getContentType(file);

        // preparing the response
        HttpServletResponse response =
            PortalUtil.getHttpServletResponse(resourceResponse);
        response.setContentType(contentType);
        response.setHeader("Content-Disposition",
            "attachment; filename= " + fileName);
        ServletResponseUtil.write(response, bytes);
    }
}

```

Great! When you click the link now, it will download the actual chapter sample into your machines “**downloads**” folder. Congratulations! You have successfully retrieved a file stored in our file system again using our great friend, “**DLStoreUtil**”.

Code Changes @ <http://code.google.com/p/lr-book/source/detail?r=71>

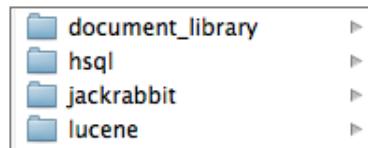
12.4.3 Data Backup Strategy on a Real Server

In this sub-section, I’d like to cover something very important related to taking backups of the portal data on a regular basis, usually done daily. I normally cover this topic in my course on Liferay Server Administration. But I wanted to give a quick heads-up on this topic as it is worth covering it here and making you aware of these details. In a real production server where Liferay is deployed and running, we need to take back up for two important chunks of data. In the event of any server crash, we can always recover the data from the backup that is made on another remote server. We use [Amazon S3](#) (Simple Storage Service) to backup the data from our production servers, either using a utility called [S3Sync](#) or another service called, [Tarsnap](#).

Coming to the main subject, what needs to be backed up? Remember the backup process is automated and executed by a CRON job running on the server. No human being will be sitting and doing the job on a daily basis.

1) The first thing that needs to be backed up is the database dump. We normally take the dump of the database. In case of MySQL the command is “mysqldump”. This command give a “*.sql” file as the output. Then this file is zipped to “*.sql.gz”. The file name is generally the name of the database suffixed with the date on which the backup is taken, eg. “**lportal_20130227.sql**”.

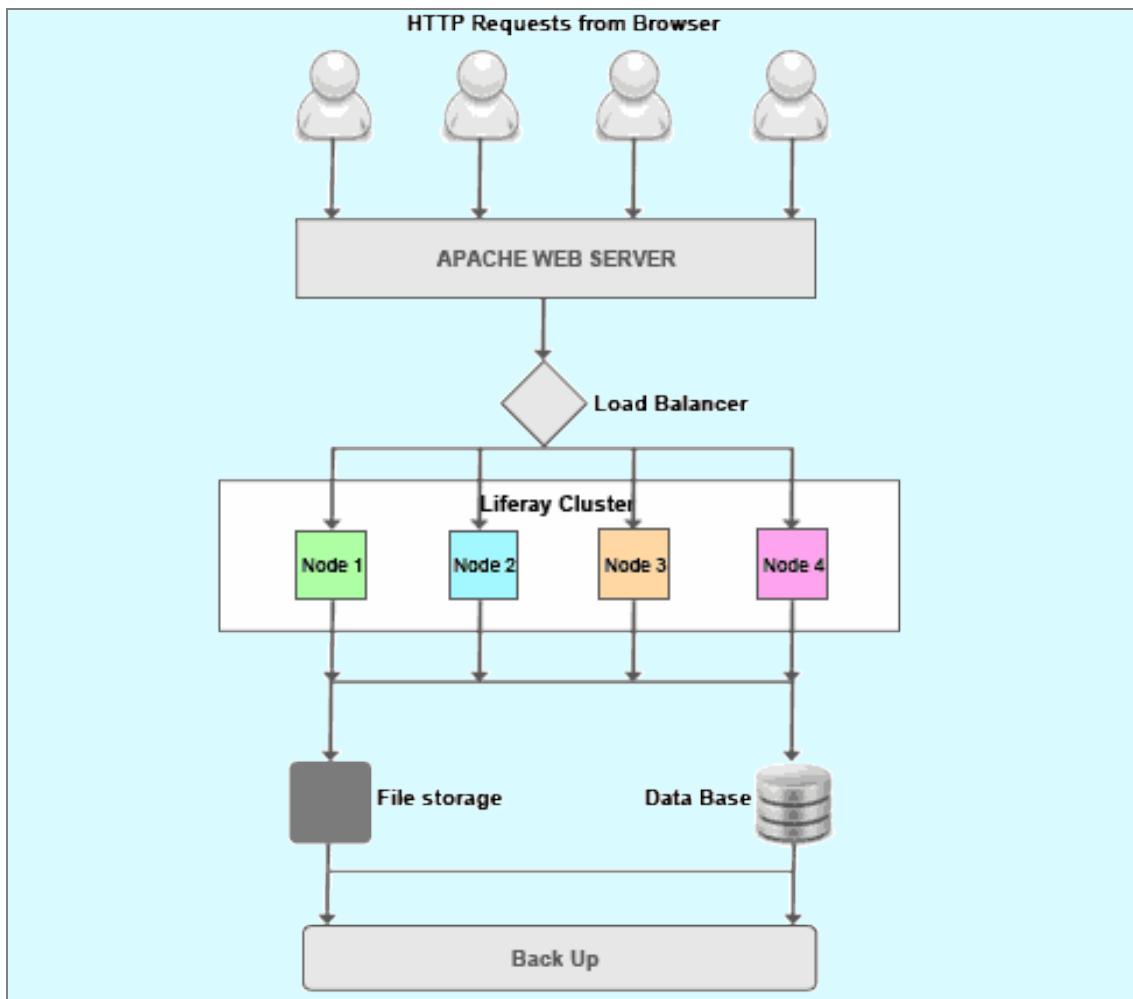
2) The next important thing that needs to be backed up is the “**data**” folder inside {LIFERAY_HOME}. This folder contains four sub folders. Since we don’t use Hipersonic SQL any more, delete folder “**hsq1**”.



The “lucene” folder maintains the indexes. We’ll cover this in detail when we discuss about “Searching and Indexing”. The “jackrabbit” folder contains one configuration file Jackrabbit. So, out of these four folders, we need to backup the folder “document_library”. The following command packages the whole folder as one tar zip file. For unzipping you have to use “**tar xvzf document_library.tgz**”.

```
tar cvzf document_library.tgz document_library
```

In a clustered environment, the “**data**” folder will reside on a SAN (Storage Area Network), so that all nodes can access and store files in the same location. The database will be common for all nodes. We have to do few settings when we host Liferay on a clustered environment for the sake of load balancing and disaster recovery. The details are beyond the scope of this book. If you’re really curious to know the details, you may get in touch with me. I’m giving a rough schema of a clustered Liferay deployment on a real production environment.



12.5 Server Side Validation

This section is going to introduce you to the server-side validation and how we can return some messages to the user. First let's use some browser side validation in our “**upload.jsp**” to limit the files that can be uploaded for each of the fields.

12.5.1 Client Side validation to check file Extension

Update the first “file” field in “**upload.jsp**” to have an additional validation rule to check the extension of the file that is getting uploaded.

```
<aui:input name="coverImage" type="file">
    <aui:validator name="acceptFiles">
        'jpg, png'
    </aui:validator>
</aui:input>
```



The “`acceptFiles`” validator is for use with input type="file". It only allows file uploads with given extensions. Specify multiple values either comma delimited 'jpg, png', whitespace delimited 'jpg png', or pipe 'jpg|png' delimited. Do not include the period before the extension. When I tried to upload a file “data.csv” it showed the error message – *Please enter a value with a valid extension (jpg, png)*. There is detailed description of client side validation in [Section 4.5.3 Form Validation](#).

With client side validation in this case, we have accomplished only half of what we actually wanted to achieve. On top of file extension validation, if we want to do file size validation there is no provision in the browser itself, as we'll not come to know the file size unless and until it is uploaded and available on the server side. This brings us to a situation where server-side validation is inevitable.

12.5.2 Server-Side Validation

Let's implement the required changes to our “**LibraryPortlet.java**” and other files to do this server-side validation and show the same upload form in case of any exceptions with display of proper error message to the user.

Step 1: We'll define the max size limit of a book's cover image to be uploaded in “**portlet.properties**”. Let's keep a value of 20 KB (20480 bytes).

```
book.cover.image.max.size=20480
```

As a good practice you should define a new variable in “**LibraryConstants.java**” for this key and refer the value in other files. This has been emphasized again and again the whole of this book.

```
static final String PROP_BOOK_COVER_IMAGE_MAX_SIZE =
    "book.cover.image.max.size";
```

Step 2: In “**upload.jsp**”, insert this line to show the error message.

```
<liferay-ui:error key="image-size-exceeded"
    message="error-image-size-exceeded" />
```

The value for the “`message`” attribute could be a plain text or an entry in “**Language.properties**” file, so that the translation is automatically taken care of. Let’s make a new entry in “**Language.properties**” file.

```
error-image-size-exceeded=The size of the image has exceeded limit.
```

Instead of “`message`” we can also have an “`exception`” attribute. If this exception is set for the “`SessionErrors`” the message of that exception will be displayed to the end user who is uploading the file.

Step 3: Replace the original line of code in the method “`UploadFiles`” of our Portlet class “**LibraryPortlet.java**” with the block of code given next to it.

```
serviceContext.setAttribute("COVER_IMAGE", coverImage);

long fileSize = FileUtils.getBytes(coverImage).length;

long sizeLimit = GetterUtil.getLong(
    PortletProps.get(
        LibraryConstants.PROP_BOOK_COVER_IMAGE_MAX_SIZE));

if (fileSize <= sizeLimit) {
    serviceContext.setAttribute("COVER_IMAGE", coverImage);
} else {
    SessionErrors.add(actionRequest, "image-size-exceeded");
    actionResponse.setRenderParameter("jspPage",
        LibraryConstants.PAGE_UPLOAD);
}
```

One final thing is left before you could test the changes. Surround the last redirect statement with the condition “`SessionErrors.isEmpty(actionRequest)`”.

Save changes and check the upload functionality now. Upload an image less than 20KB to test the happy path and an image more than 20KB size to test negative case. Like how we have used [SessionErrors](#) API to display the error messages, we can use [SessionMessages](#) to transmit meaningful success messages to the frontend. Instead of binding (*adding*) a message to we can directly bind an exception. You can check all the methods available with these API’s. In the JSP a success message has to be displayed with another variant of “`liferay-ui:message`”,

```
<liferay-ui:success key="" message="" />
```

Code Changes @ <http://code.google.com/p/lr-book/source/detail?r=72>

12.5.3 Various types of “aui:input” fields

As we are discussing about the UI form and validations, I thought it would be good to mention few things about the AUI tags that we use in constructing those HTML

forms in a nice and elegant manner. In the majority of the forms we have developed so far, we have used “**aui:input**” tag. I am not sure whether you’re aware of the various types of this tag. Based on the type the rendering is different. Of course the default type is “**text**” which renders a simple text field.

```
<aui:input type="image" />           <aui:input type="radio" />
<aui:input type="assetCategories" />    <aui:input type="timeZone" />
<aui:input type="checkbox" />          <aui:input type="textArea" />
<aui:input type="assetTags" />          <aui:input type="file" />
```

Now, I am going to show you an example of how you can construct an input field that accepts a date, for e.g. the “dateOfReturn” for a book that is being borrowed. For the purpose of testing, we’ll put the new code in our “**upload.jsp**” itself.

```
<%
    Calendar dateOfReturn = CalendarFactoryUtil.getCalendar();
    dateOfReturn.setTime(lmsBook.getCreateDate());
    LMSBook lmsBook = LMSBookLocalServiceUtil
        .fetchLMSBook(Long.valueOf(bookId)); %>
<aui:input
    name="dateOfReturn" model="<%=" LMSBook.class %>"
    bean="<%=" lmsBook %>" value="<%=" dateOfReturn %>" />
```

Make the necessary imports.

```
<%@page import="java.util.Calendar"%>
<%@page import="com.liferay.portal.kernel.util.CalendarFactoryUtil"%>
```

To make the tag display just a date, you have to give it model hints. This is how you tell Liferay how to display this field. You do so using a file that was generated when Service Builder created Java classes to manipulate your data. Open the file “**portlet-model-hints.xml**” under “**src/META-INF**” folder. Scroll down in the file until you find the field for “dateOfReturn”, and replace it with the following code. Since it is not there in our “**LMSBook**” entity, you can add this new entry there as well.

```
<field name="dateOfReturn" type="Date">
    <hint name="year-range-delta">10</hint>
    <hint name="year-range-future">true</hint>
    <hint name="show-time">false</hint>
</field>
```

Save the changes and check the form now. The date picker field should nicely appear as shown.



There is one Wiki Article [<http://bit.ly/16wgvqO>] that talks about Model Hints. If you’re really curious to know more about this topic, you can make use of this article.

12.6 Custom Fields Framework

From our discussion on Liferay's Storage Framework, we are entering into another exciting framework that helps the users of our Portlets to dynamically created fields to the existing entities and do all operations on those newly created fields as if they are the original fields of the entity. All this can be done on the fly without even writing one single line of code. This section and the next section will tear this subject apart and make you know all the nuts and bolts of this fabulous framework. Let me start giving you some background to get you familiarized.

12.6.1 Control Panel and Custom Fields

Let's start our discussion with something that is already there Liferay. Login a Portal administrator, go to the Control Panel and click "Custom Fields" under "Portal" section. You will get to a page where you will see a default list of entities. You can define custom fields on these portal level entities. In the screenshot below I've shown only few of them as sample.

Resource	Custom Fields	
Blogs Entry		Edit
Bookmarks Entry		Edit
Bookmarks Folder		Edit
Calendar Event		Edit
Documents and Media Document		Edit

Now click "Edit" against the "User" entity and create couple of custom fields on it, say, "**Education**" and "**Occupation**". While creating a new field, you specify a key and a type. The custom field **key** is used to access the field programmatically through the "`liferay-ui:custom-attribute`" tag. On every custom field you can set permissions after it is being defined. You can edit and delete a custom field.

The screenshot shows the "User" entity in the Liferay Control Panel. A "Add Custom Field" button is visible. Two custom fields are listed: "Education" (Key: Education) and "Occupation" (Key: Occupation). Both fields are of type "Text" and are set to "False" for both "Hidden" and "Searchable". The "Edit" button is highlighted for the "Education" field. To the right of each field are "Actions" buttons for "Permissions" and "Delete".

The type of a custom field could be either "**preset**" or "**primitive**" and one of below.

Preset Types	Primitive Types
Selection of Integer Values	True/False – Boolean and Date
Selection of Decimal Values	Decimal Number (32 & 64-bits)
Selection of Text Values	Group of Decimal Numbers (32 & 64-bits)

Text Box	Integer (16,32 & 64-bits)
Text Box - Indexed	Group of Integers (16,32 & 64-bit)
Text Field - Secret	Decimal Number of Integer (64-bit)
Text Field - Indexed	Group of Decimal Numbers or Integers (64-bit)
	Text and Group of Text Values

This concept of creating custom fields dynamically in Liferay is called as “Expando”. It is a “generic” service that allows you to dynamically define a collection of data. The data can be:

- Typed (boolean, Date, double, int, long, short, String, and arrays of all those)
- Associated with a specific entity (e.g. “com.liferay.portal.model.User”)
- Arranged into any number of “columns”
- Available to plugins and other WARS
- Accessed from Velocity templates
- Accessed via Liferay’s JSON API through AJAX

In the next section we’ll see how the backend tables get impacted as soon as these new custom fields are defined on the given portal entities. These tables can also be programmatically manipulated to create custom fields and set values in those fields directly from our hooks, themes and portlets.

12.6.2 Impact to the backend tables

With two new custom fields defined on the “User” entity, let’s see what effects have taken place at the backend database level. Liferay has defined four tables where the custom fields and their values are stored and retrieved. What are those four tables?

ExpandoTable: This table stores the logical representation of the new table. The “classNameId” maps to the “classname_” table. In my case “10005” refers to the “User” entity.

tableId	companyId	classNameId	name
21724	10153	10005	CUSTOM_FIELDS
10411	10153	10005	MP

ExpandoColumn: This table defines the columns of the logical table with a column name, type, defaultData and additional settings.

columnId	companyId	tableId	name	type_	defaultData	typeSettings
21725	10153	21724	Education	15		index-type=1
21727	10153	21724	Occupation	15		index-type=1

We have already seen the various types of a custom field. The class [ExpandoColumnConstants](#) has all the types and their labels defined as constants. The “defaultData” can be set by going to the “Edit” option of the field. The “typeSettings” columns contain the “newline-separated” values of some additional settings (attributes) of a custom field. These settings are listed below and can be set through “Edit” option on that field. You observe that the custom fields that we created for the “User” entity are in this table.

Hidden	The field's value is never shown in any user interface besides this one. This allows the field to be used for some more obscure and advanced purposes such as acting as a placeholder for custom permissions.
Visible with Update	A user with update permission can view this field. This setting overrides the value of hidden in this case.
Searchability	The value of the field will be indexed when the entity (such as User) is modified. Only <code>java.lang.String</code> fields can be made searchable. Note that when a field is newly made searchable, the indexes must be updated before the data is available to search. Searchability can be either on the field name or its key.
Secret	Typing will be hidden on the screen. Use this for passwords and other secret information.
Height	This will set the height of the input field in pixels.
Width	This will set the width of the input field in pixels.

An example of a fully loaded value for the “typeSettings column” will look like what you see here. They are key value pairs separated by “\n” (*new line*) character. Liferay uses this approach of smartly storing the data in many other tables.

```
height=0
index-type=2
hidden=0
width=30
secret=1
visible-with-update-permission=1
```

ExpandoRow: Every record in this table is a logical representation of a row (record) in the logical table. The table has fields – rowId, companyId, tableId and classPK.

rowId_	companyId	tableId	classPK
10805	10153	10411	10195

ExpandoValue: Every record of this table contains the value for an actual combination of a logical table, a logical row and a logical column.

valueId	companyId	tableId	columnId	rowId_	classNameId	classPK	data_
10806	10153	10411	10412	10805	10005	10195	H/93NNpOjGD...

12.6.2 Activities for You

Having got a strong understanding of custom fields and Expando, I am giving you two activities for you to do yourself. Of course, they are quite simple and I am sure it's going to be a cakewalk for you. In the process of doing the second activity you will get to install a new Official plugin from Liferay, “Web Form”. This plugin allows you to create dynamic forms on the fly and present those forms to the portal users.

Activity one: Login to the portal and go to “My Account” page. Click “Custom Fields” from the “Miscellaneous” section on the right side to see a form as shown. Enter some values and save the form. Find out how the values are now stored in the “**ExpandoRow**” and “**ExpandoValue**” tables.

Three	Two	One
Comments Custom Fields (Modified) <input type="button" value="Save"/> <input type="button" value="Cancel"/>	Custom Fields <hr/> Education M.S (Software Systems) Occupation Liferay Learner	Miscellaneous Announcements Display Settings Comments Custom Fields

Activity two: Coming to our second activity, go to “Marketplace” from the Control Panel and install “**Web Form Portlet**”. You will find it under the “Productivity” category in the Liferay Marketplace Store. Click on it and you will get the details of it. From this page, click “Free” to fill a form and purchase it. Once it is available in your “**Purchased**” list of plugins, you can download it and then install it. After successful install, create a new page “Contact Us” and drag+drop this Portlet on the page.



Create a simple “Contact Us” form. Submit the form with sample data and check how the data is getting saved in the four “Expando” tables.

12.7 Enabling Custom Fields for Library Portlet

We have seen so many ways of configuring and customizing our Library Portlet in Chapter 10 [Configuration & Communication](#). Don’t you think it will be even nicer if we provide the Librarians (*our actual users*) to define custom fields on the book they add to the Library? Say, for example, currently we don’t have a field called “ISBN” for a book. But one of the librarians wants this field to be there in the form. Yes, customers are always very demanding. They seldom understand the pain of developers. What to do? We have to honor them as they pay our bills. We want to build our Portlet in such a way that whatever field is required in the future; the users themselves should be able to add it dynamically through the administrative interface, like how we were able to do for some of the existing portal level entities. Let’s get started.

12.7.1 Defining the custom fields

There are two things to be done in this step of defining the custom fields for our custom entity.

Step 1: Create a new class that will help in displaying the “**LMSBook**” in the list of entities under Control Panel, that accept custom fields. Call this class as “**LMSBookCustomFieldsDisplay**” under package “**com.library.custom**” extending **BaseCustomAttributesDisplay** [<http://bit.ly/YvmFQs>] and overriding just one method of the base class.

```

package com.library.custom;

import
com.liferay.portlet.expando.model.BaseCustomAttributesDisplay;
import com.slayer.model.LMSBook;

public class LMSBookCustomFieldsDisplay
    extends BaseCustomAttributesDisplay {
    public String getClassName() {
        return LMSBook.class.getName();
    }
}

```

Step 2: Make an entry in “**liferay-portlet.xml**” pointing to the class we just created. This entry should come after the ending “`</asset-renderer-factory>`”.

```

<custom-attributes-display>
    com.library.custom.LMSBookCustomFieldsDisplay
</custom-attributes-display>

```

Now check the **Control Panel → Portal Section → Custom Fields**. It should have a new entry for our “LMSBook”.



Click “Edit” to create couple of new custom fields for this entity. Also set the “View” permission for “User” on these fields.

isbn					« Back
Role	Delete	Permissions	Update	View	
Guest	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
Librarian	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
Owner	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	
Portal Content Reviewer	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
Power User	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
User	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	

Check the database tables. You should find new records inserted.

ExpandoTable

tableId	companyId	classNameId	name
19501	10153	10901	CUSTOM_FIELDS

ExpandoColumn

columnId	companyId	tableId	name	type_	defaultData	typeSettings
21801	10153	19501	ISBN	15		index-type=1

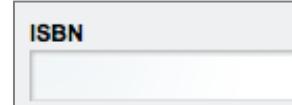
12.7.2 Code Level changes

We are done with the configuration changes required for defining the new custom fields. Now it is time to enable it in our code, so that they are available in the UI form and also get saved to the database when a book is added. We'll perform the following two steps.

Step 1: Insert the below block to our “update.jsp” in the form, in order to make the new custom field appear at the time of adding a book to the library.

```
<liferay-ui:custom-attributes-available
    className="<% LMSBook.class.getName() %>">
    <liferay-ui:custom-attribute-list
        classPK="<% lmsBook.getBookId() %>"
        className="<% LMSBook.class.getName() %>"
        editable="<% true %>" label="<% true %>" />
</liferay-ui:custom-attributes-available>
```

There is one more tag “`liferay-ui:custom-attribute`” that helps in only one field of the list of fields. In this tag, we also have to pass the “**key**” of the custom field. The usage of above tag automatically injects the custom fields into the form. For the field to appear properly “`editable`” and “`label`” attributes have to be set to “`true`”.



For each field in this list, a form element is rendered whose format is like,

```
<div class="aui-field-wrapper-content" id="aui_3_4_0_1_1854">
    <label class="aui-field-label"> ISBN </label>
    <input type="hidden" value="ISBN"
        name="_library_WAR_libraryportlet_ExpandoAttributeName--ISBN--">
    <input id="pqai_ISBN" class="lfr-input-text" type="text" value=""
        style="" name="_library_WAR_libraryportlet_ExpandoAttribute--ISBN--">
</div>
```

“`ExpandoAttributeName--`” and “`ExpandoAttribute--`” fields have got great significance in Liferay. They are called as Expando tokens. Liferay automatically sets these values into the “`serviceContext`” object with the need for us to do anything extra in our Portlet class.

Step 2: With the values properly set in “`serviceContext`” all you have to do now is to update “`insertBook`” in our DTO class, “**LMSBookLocalServiceImpl**” with this one single line. That's it ☺.

```
lmsBook.setExpandoBridgeAttributes(serviceContext);
```

After this change, try adding a new book into our Library. The book should get added and in parallel you should see new records getting inserted into “**ExpandoRow**” and “**ExpandoValue**” tables. Let's do one quick test to confirm everything is fine from the UI perspective. Try editing the book you just added. In the “Edit Book” the original value for the custom field should remain intact.

Congratulations! You have done a splendid job. Keep up the good work in your real life application too. Wait, just one thing I forgot to mention. Let's catch up before

jumping to the next chapter. Whenever inside your code, whether JSP, Portlet class or DTO classes, as long as you have the handle to an actual entity, you can retrieve any custom field with the help of a statement like,

```
lmsBook.getExpandoBridge().getAttribute(attributeName);
```

This guy returns an object that implements “Serializable” interface. You just have to cast it to whatever type you want.

Code Changes @ <http://code.google.com/p/lr-book/source/detail?r=73>

Summary

This chapter covered the second part of the most important sub-frameworks of Liferay. We started with a general discussion on Storage Framework. We implemented how to store and image and how to retrieve an image. In the middle, we have seen another great feature of Liferay that lets us to write every action as a separate class instead of keeping everything inside one Portlet class. The next section covered showing the book cover as a thumbnail image. Converting an actual image as a thumbnail is a framework in itself and Liferay has that support inbuilt.

Then we moved on to discuss the file processing mechanisms in Liferay using the DLStoreUtil API. We continued our discussion to see the server side validations are done. In the process we have seen many other related topics. Finally we spent a good amount of time to understand and implement the custom field’s framework using which we can dynamically create custom fields for our custom entities.

13. Liferay Collaboration Frameworks

This Chapter Covers...

- Liferay Subscription Framework
 - Liferay and AJAX using AlloyUI
 - Commenting and Rating Framework
 - Social Activity Framework
 - Social Equity Framework
 - Business Process (BPM) Framework
-

Collaboration is the key to success. Collaboration is working with each other to do a task. It is a recursive process where two or more people (users) or organisations work together to realise shared goals. A portal platform like Liferay provides various ways and tools to help the portal users to interact and collaborate in order to make the portal a lively and happening place. More user collaboration means more traffic to the portal and more popularity.

In this chapter, we'll see the various ways in which the collaboration can happen between the portal users, either directly or indirectly. Collaboration can be either one way or two ways. The words Collaboration and Social are used almost synonymously. Wikipedia gives this definition – “*The term social refers to a characteristic of living organisms as applied to populations of humans and other animals. It always refers to the interaction of organisms with other organisms and to their collective co-existence, irrespective of whether they are aware of it or not, and irrespective of whether the interaction is voluntary or involuntary*”.

The following table shows the various *out-of-the-box* Portlets of Liferay that help in achieving collaboration between the portal users.

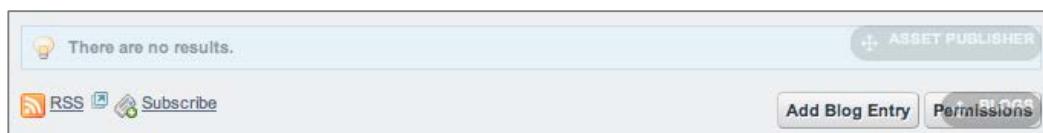
Collaboration	Community	Social	Content
Blogs	Bookmarks	Activities	Asset Publisher
Blogs Aggregator	Directory	Group Statistics	Polls and Display
Calendar	Invitation	Walls	Knowledge Base
Message Boards	Page Comments	User Statistics	Related Assets
Recent Bloggers	Page flags	Requests	Content Search
Wiki	Page Ratings	Where is user?	Workflow

13.1 Liferay Subscription Framework

Let's start this chapter with one of the many ways collaboration can be triggered. Subscription helps in enabling collaboration between the portal and its users. Using this framework the users can subscribe to any Portlet or a feature or an asset in the portal. By virtue of subscription we can enable notification being sent from the portal whenever there are changes happening in the objects for which users have subscribed. Say for example, one online library member has subscribed for a book asset. He'll be notified whenever any changes happen on that book asset. The first sub-section explains the basic concept and the next one shows how to implement subscription in our own Library Portlet.

13.1.1 Knowing the basics

Let's spend some time understanding the basics of subscription mechanism. Using this feature user can subscribe to or un-subscribe from any addressable entity of the portal or our custom portlets. Login to the portal as administrator and create a new page called "Blogs" and add the "Blogs" Portlet onto this page. Initially there are no blogs shown.



Click "Subscribe" link and let's see what's happening at the backend. A new record gets inserted into the "**subscription**" table with the following values. The "**classNameId**" refers to the "**BlogsEntry**" entity and "**classPK**" refers to the groupId on which this Portlet is currently placed.

subscriptionId	companyId	userId	userName	createDate	modifiedDate	classNameId	classPK	frequency
23502	10153	10195	Test Test	2013-02-28...	2013-02-28...	10007	10179	instant

As a next step, create a new sample blog and publish it. Once the blog appears on the page, click its title to view the blog details. At the bottom of the page, you'll see a link to subscribe to the comment. Click on this link and see what is happening to the "subscription" table. You should see another record here. Check "classPK" value.

No comments yet. Be the first. [Subscribe to Comments](#)

13.1.2 Enabling Subscription for Library Portlet

We're going to do exactly the same thing for our Library Portlet. We'll make the following two ways our members can subscribe to our Library.

- 1) *Library Subscription* – in this case, any new book added to the library, all the members who have subscribed have to be notified announcing the availability of the new book so that they can come and borrow it from our Library.

2) Book Subscription – If a member wants a book, but currently another member has borrowed that book. So, he can subscribe for this book expressing his interest to borrow it next. As soon as the other member returns the book, this member has to be notified that this book is available now, so that he can go to the library and borrow it.

With this requirement in mind, we'll go back to our discussion room to brainstorm how to implement this feature. One of the developers in the IT team has already read this book and had an idea of what Liferay offers to achieve this kind of functionality. He proposed the solution and everyone unanimously agreed. Let's see the actual solution now. Unfortunately, Liferay has not provided a ready-made taglib to show the “subscribe” or “unsubscribe” link. This gives us the room to create our own taglib for this purpose in the next section.

Library Subscription

Step 1: Append the following code to our portlet's “view.jsp”.

```
<% boolean isSubscribed =
    SubscriptionLocalServiceUtil.isSubscribed(
        company.getCompanyId(), user.getUserId(),
        LMSBook.class.getName(), scopeGroupId);
String currentURL = themeDisplay.getURLCurrent();
%>
<c:choose>
    <c:when test="<% isSubscribed %>">
        <portlet:actionURL var="unsubscribeURL" name="subscribe">
            <portlet:param name="<% Constants.CMD %>" value="<% Constants.UNSUBSCRIBE %>" />
            <portlet:param name="redirectURL" value="<% currentURL %>" />
        </portlet:actionURL>

        <liferay-ui:icon image="unsubscribe" label="<% true %>" url="<% unsubscribeURL %>" />
    </c:when>
    <c:otherwise>
        <portlet:actionURL var="subscribeURL" name="subscribe">
            <portlet:param name="<% Constants.CMD %>" value="<% Constants.SUBSCRIBE %>" />
            <portlet:param name="redirectURL" value="<% currentURL %>" />
        </portlet:actionURL>

        <liferay-ui:icon image="subscribe" label="<% true %>" url="<% subscribeURL %>" />
    </c:otherwise>
</c:choose>
```

This whole block you'll embed within the condition to check if the user is currently signed in, “`<c:if test="<% themeDisplay.isSignedIn() %>">...</c:if>`”.

Step 2: Write a new processAction method “subscribe” in our Portlet class “**LibraryPortlet.java**” and making it responsible for do the actual job.

```
public void subscribe(ActionRequest actionRequest,
                      ActionResponse actionResponse)
```

```

        throws IOException, PortletException {

    String cmd = ParamUtil.getString(actionRequest, Constants.CMD);
    ThemeDisplay themeDisplay = (ThemeDisplay)
        actionRequest.getAttribute(WebKeys.THEME_DISPLAY);

    long userId = themeDisplay.getUserId();
    long groupId = themeDisplay.getScopeGroupId();
    String className = LMSBook.class.getName();
    long classPK = groupId;
    String frequency = SubscriptionConstants.FREQUENCY_INSTANT;

    try {
        if (cmd.equalsIgnoreCase(Constants.SUBSCRIBE))
            SubscriptionLocalServiceUtil.addSubscription(
                userId, groupId, className, classPK, frequency);
        else
            SubscriptionLocalServiceUtil.deleteSubscription(
                userId, className, classPK);
    } catch (PortalException e) {
        e.printStackTrace();
    } catch (SystemException e) {
        e.printStackTrace();
    }
    actionResponse.sendRedirect(
        ParamUtil.getString(actionRequest, "redirectURL"));
}

```

Book Subscription

In the case of subscribing for a particular book, you'll append exactly the same code in "**detail.jsp**" and to make it work you need to do the following two changes.

Change 1: In the JSP code of book details page, pass the additional parameter "**bookId**" in both *subscribe* and *unsubscribe* action URLs.

```
<portlet:param name="bookId" value="<% lmsBook.getBookId() %>" />
```

Change 2: In our Portlet class, "**LibraryPortlet.java**" slightly modify one line of the code to handle both Library Subscription and Book Subscription.

```
long classPK = ParamUtil.getLong(actionRequest, "bookId", groupId);
```

Code Changes @ <http://code.google.com/p/lr-book/source/detail?r=74>

Do it Yourself: There are two activities you're going to do by yourself.

Activity 1: Create a method in "**LMSUtil.java**" that will do the actual job of updating the "**subscription**" table. This will help us to invoke this method from anywhere. Make changes to "**LibraryPortlet.java**" accordingly. The method signature is,

```
public static void applySubscription(
    PortletRequest portletRequest, String cmd);
```

Activity 2: Create a JSP fragment, “**subscription.jspf**” under “**html/library/social**” and cut+paste the contents that we appended to “**view.jsp**” in step 1 into this file. In lieu of the previous code, insert an include statement in “**view.jsp**”. The above two changes should leave the portlet’s functionality undisturbed. This activity will put the base for converting the whole stuff as a simple taglib.

```
<%@include file="/html/library/social/subscription.jspf" %>
```

Code Changes @ <http://code.google.com/p/lr-book/source/detail?r=75>

13.2 Liferay and AJAX using AlloyUI

One of the key requirements of a highly collaborative portal is AJAX-ifying the links. There are many benefits of this. In the previous example, just to change the state of user subscription, we’re making an action request, and updating the web screen all over again. The time factor is compromised followed by the space factor. Just for one update, we’re loading the whole HTML all over again served from the server. Imagine one million people doing the same thing on your portal and how much of data transfer happens between your server and those one million browsers.

Remember, these data transfers are not free. The hosting companies that host your Liferay applications charge them. So, while architecting your application, you need to take care of even these subtle non-functional requirements. Saving every byte of data transfer will be a huge cost saving on your hosting front, every month. This is one of the reasons; companies are now coming up with RIA’s and SPA’s. It is not just the user experience, but definitely it is the primary objective of AJAX-ifying your requests that updates only the portion of the screen and not the entire web page. By the way, if you’re new to AJAX, I feel you should spend some time going through this w3schools.com tutorial and get yourself comfortable.

13.2.1 Getting the ground ready for AJAX

From the browser’s JavaScript, whenever you want to hit our Portlet through an AJAX call, you should construct a “**resourceRequest**” and set its Window State as “**EXCLUSIVE**”. This is a special windowState that Liferay provides on top of what has been provided by Portlet 2.0 specification. As the name suggests an URL whose Window State is “**EXCLUSIVE**”, then the URL directly hits the Portlet and NO other Portlets in the same page (*as the original portlet*) get impacted. We’ll take four steps to complete this exercise.

Step 1: Convert the action we had earlier in our Portlet class into `serveResource` method. Insert the following block of code in the existing “`serveResource`” method of our Portlet class.

```
if (cmd.equalsIgnoreCase(Constants.SUBSCRIBE)
    || cmd.equalsIgnoreCase(Constants.UNSUBSCRIBE)) {
    LMSUtil.applySubscription(resourceRequest, cmd);
```

```

// setting the response in JSON format
JSONObject jsonObj = JSONFactoryUtil.createJSONObject();
jsonObj.put("subscribed",
            cmd.equalsIgnoreCase(Constants.SUBSCRIBE));
HttpServletResponse response = PortalUtil
        .getHttpServletResponse(resourceResponse);
PrintWriter pw = response.getWriter();
pw.write(jsonObj.toString());
pw.close();
}

```

Note we're calling the “`applySubscription`” method of “**LMSUtil**” which you did as an activity in the previous section. We've created a “`jsonObj`” and set some values into it before flushing it back into the “`response`”, so that the guy who is making the AJAX call to our Portlet can pick up the same object. The approach is very similar to using “`ServletResponseUtil`” that we adopted in our other examples of serving cover images and sample chapters. If we've not used “`PrintWriter`” then the code will look like below. Also in the above code we've not explicitly set the `contentType` of the response, as JSON is nothing but a form of text. Optionally you can set it as well. Liferay has all the standard Mime types defined in [ContentTypes.java](#).

```
ServletResponseUtil.write(response, jsonObj.toString().getBytes());
```

Step 2: Let's go back to our “**subscription.jspf**” which you created as part of an activity in the last section. The new block will replace the original block embedded within, “`<c:if test="<%=> themeDisplay.isSignedIn() %>" ...</c:if>`”. Let's put the basic skeleton in place.

```

<%@page import="javax.portlet.ResourceURL" %>
<%
    boolean isSubscribed =
        SubscriptionLocalServiceUtil.isSubscribed(
            company.getCompanyId(), user.getUserId(),
            LMSBook.class.getName(), scopeGroupId);

    String label = isSubscribed?
        Constants.UNSUBSCRIBE : Constants.SUBSCRIBE;
%>

<aui:a href="javascript:void();" label="<%=> label %>"
       onClick="javascript:toggleState(this);" />

<aui:script>
    AUI().ready('liferay-portlet-url', 'aui-io-request',
        function(A){
            alert('I am inside the function');
        }
    );
</aui:script>

```

We've enabled two Liferay JavaScript modules, “`liferay-portlet-url`” and “`'aui-io-request'`. Check the Portlet now, it should alert a message “**I am inside the function**” on loading of the page. Now replace the alert message with this code:

```
var portletId = '<%= portletDisplay.getRootPortletId() %>';
// form the URL
```

```

var subscriptionURL =
    Liferay.PortletURL.createResourceURL();
subscriptionURL
    .setPortletId(portletId);
subscriptionURL
    .setParameter('<%= Constants.CMD %>', '<%= label %>');
}

toggleState = function(anchor) {
    alert('I am royally clicked');
}

```

Save the changes and check now. Click on the “subscription” link and you’ll get the alert for “I am royally clicked”. We’re almost close to our solution. Next replace this alert message with this block of code that makes the AJAX call.

```

// make the AJAX call
A.io.request(subscriptionURL.toString(), {
    method: 'GET',
    dataType: 'json',
    on: {
        success: function() {
            var result = this.get("responseData").subscribed;
            alert(result);
        }
    }
});

```

Thanks for being very patient. This is how you’ve to work with JavaScript in your real world. JavaScript is very difficult to debug. Hence, it is highly recommended to go step-by-step. Whatever you’ve done so far, make sure that it is working fine. Please understand the code that we’ve written. We’ve formed a resourceURL using Liferay JavaScript API and then invoking the URL through AJAX. On “success” of that invocation, we’re processing the JSON response that is coming back from our Portlet class. Everything should work fine, except that everytime you’ve to refresh the page to make a new request as the “subscribe” / “unsubscribe” link and the underlying resourceURL are not toggling as a result of the AJAX call. We need put some more code inside the “success” block to change these attributes of the “anchor” link dynamically. Let’s do this in our next step.

Step 3: Replace the “alert” inside the “success” block with the actual logic to dynamically change the link and the label.

```

var linkText = (result === true)? 'UnSubscribe' : 'Subscribe';
A.one('#'+anchor.id).html(linkText);
subscriptionURL.setParameter('<%= Constants.CMD %>',
    linkText);

```

Now check our subscription is working perfectly fine. The link should get toggled without any problems. One final thing we’re going to do to improve the usability. Usually during AJAX calls, a loading mask is placed to indicate, something is happening behind the screens. This gives a nice feeling for the end-user and gives him the satisfaction that his actions on the browser (*clicks and other events*) are making some serious back-end calls.

Step 4: Let's introduce the loading mask to indicate some processing is in progress before the link changes itself. For this, we've to load the “aui-loading-mask” Liferay JavaScript module. Append the new module to the existing link of modules. Don't forgot to put the comma at the end of this line after adding the new module.

```
AUI().ready('liferay-portlet-url', 'aui-io-request', 'aui-loading-mask',
```

Before and after the block of code you introduced in Step 3, insert these blocks.

```
var layer = A.one('#portlet_'+portletId);
if (typeof(layer.loadingmask) == 'undefined') {
    layer.plug(A.LoadingMask, { background: '#000' });
}
layer.loadingmask.toggle();
```

```
setTimeout(function(){layer.loadingmask.toggle()};500);
```

Now you can enjoy with the overlay that comes up when you click the subscription link. Don't you think it's cool? To reinforce your learnig so far, I recommend you should read this [Liferay Wiki](#) on Liferay and AJAX written by Liferay's UI guru, [Nate Cavanaugh](#).

13.2.2 AJAX Loading Library Welcome Message

I strongly felt, this is a very good context to quickly cover another important and interesting tweek to load Web Content through AJAX. Hope you remember in Section [9.6.1 Embedding Web Content in a Portlet](#), we embedded a journal article (Web Content) inside our Portlet'sing a “liferay-ui” tag. The Web Content is getting properly loaded, but there is one issue. Though the issue is not quite serious but there is a good scope for improvising the way the article gets loaded. Currently the loading happens in the server-side and the contents are then served to the browser. Don't you think it will be nicer if the content is loaded through AJAX asynchronously and simulataneously when the Portlet gets loaded. This way there is going to be a huge performance gain. Let's do this change in two steps in our “view.jsp”.

Step 1: Open “view.jsp”. Replace the below “liferay-ui” tag with a “div” tag given immediately after it.

```
<liferay-ui:journal-article articleId="LIBRARY_WELCOME_MESSAGE"
    groupId="<% themeDisplay.getScopeGroupId() %>" />
```

```
<div id="LibraryWelcomeMessage"></div>
```

Step 2: The next step is to dynamically fetch the contents of the web content, “LIBRARY_WELCOME_MESSAGE” and load it, completely through AJAX without the other parts of the Portlet knowing about it. For the “” tag, add an attribute to load the Liferay JavaScript module required for making AJAX calls, “use='aui-io-request'”. Inside same function (*block*) insert the following code to do the dynamic content loading.

```
// loading the welcome article
```

```

var url = '/c/journal/view_article_content?' +
    'groupId=<%= themeDisplay.getScopeGroupId() %>' +
    '&articleId=LIBRARY_WELCOME_MESSAGE';
A.io.request(url, {
    method: 'GET',
    on: {
        success: function() {
            var data = this.responseText;
            A.one('#LibraryWelcomeMessage').html(data);
        }
    }
});

```

The URL above is nothing by a normal URL that returns any web content with an ID. You can actually hit this URL in your browser and verify the actual article is getting display on the browser screen. The only requirement is that the article should have view permission for Guest role.

```
http://localhost:8080/c/journal/view_article_content?groupId=10179&articleId=LIBRARY_WELCOME_MESSAGE
```

Code Changes @ <http://code.google.com/p/lr-book/source/detail?r=78>

13.2.3 Liferay JavaScript Loading Sequence

Before we move on to our next topic, let's quickly skim through the sequence in which the various JavaScript events are triggered during a typical portal page load inside a browser. This information will certainly help you to decide where exactly to write your custom JavaScript code.

1) This callback is executed as soon as the HTML in the page has finished loading (*minus any portlets loaded via AJAX*).

```
AUI().ready(function(A) {
    // custom code here...
});
```

2) This event gets executed after each Portlet on the page has loaded. The callback receives two parameters: `portletId` and `node`. `portletId` is the id of the Portlet that was just loaded. `node` is the Alloy Node object of the same portlet.

```
Liferay.Portlet.ready(function(portletId, node) {
    // custom code here...
});
```

3) This is an automatic event that gets triggered after everything including AJAX portlets has finished loading.

```
Liferay.on('allPortletsReady', function() {
    // custom code here...
});
```

13.3 Commenting and Rating Framework

Another great way of triggering collaboration in the portal is to enable users to write comment on the portal assets and also rate them. These actions are very much part of social activities and in the later sections we'll see how to measure these social activities and give rewards to the users helping them to earn their social equity in the portal. Users social equity will keep going up based on their participation and contribution. When users participate and contribute the asset's popularity will proportionately go up. In this section, we'll see how to enable commenting feature in our Library Portlet, so that our library members can start writing comments after they read a book. These comments will serve as good review generated by the users themselves, as they are the best people to criticize on the books in our Library. This is how the portal users automatically generate the portal contents.

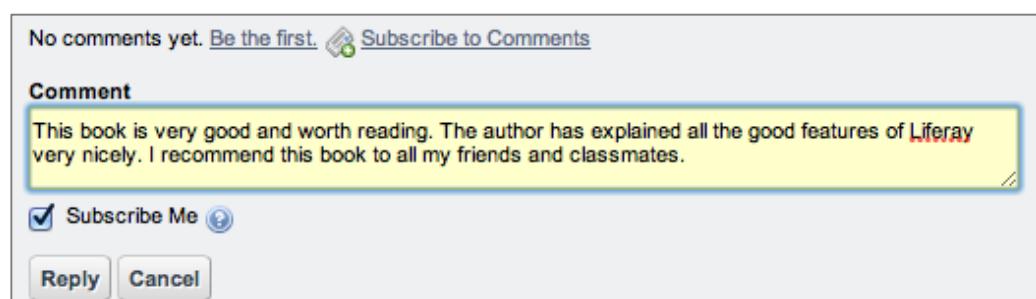
13.3.1 Enabling Comments on a Library Book

In this sub-section, we'll see how to enable comments for our library books apart from what we can do through the Asset Publisher Portlet. The exercise consists of just three steps.

Step 1: Append the below code in “**detail.jsp**”, so that whoever views a book can write some comments on the book.

```
<portlet:actionURL var="discussionURL" name="discussOnThisBook" />
<liferay-ui:discussion
    classPK="<% lmsBook.getPrimaryKey() %>"
    userId="<% themeDisplay.getUserId() %>"
    className="<% LMSBook.class.getName() %>"
    subject="<% lmsBook.getBookTitle() %>"
    formAction="<% discussionURL %>"
    ratingsEnabled="<% true %>"
    redirect="<% themeDisplay.getURLCurrent() %>" />
```

Check the detail page now. Let's write some good comments on this book.



There is a checkbox “Subscribe Me”. If the user checks this box then he/she is automatically subscribed to the comments written on this book. Now click “Reply” and you'll encounter an error. This is because, we've still not written an action method “**discussOnThisBook**” in our Portlet class. This is what we're going to do in the next step. Be prepared, this is a very special action method as it is going to do something very differently.

Step 2: Open our Portlet class “**LibraryPortlet.java**” and insert this new method.

```
public void discussOnThisBook(ActionRequest actionRequest,
    ActionResponse actionResponse)
        throws IOException, PortletException {

    String className =
        "com.liferay.portlet.messageboards.action." +
        "EditDiscussionAction";
    String methodName = "processAction";

    String[] parameterTypeNames = {
        "org.apache.struts.action.ActionMapping",
        "org.apache.struts.action.ActionForm",
        PortletConfig.class.getName(),
        ActionRequest.class.getName(),
        ActionResponse.class.getName()
    };

    Object[] arguments = {
        null,
        null,
        getPortletConfig(),
        actionRequest,
        actionResponse
    };

    try {
        PortalClassInvoker.invoke(true, className,
            methodName, parameterTypeNames, arguments);
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

The only required import is:

```
import com.liferay.portal.kernel.util.PortalClassInvoker;
```

Are the contents of this method looking like Greek and Latin for you? Don’t panic. It’s quite simple. I’ll explain. All we’re doing here is invoking an Action class of the portal to save the comments the user just entered. We could’ve written the whole logic in saving the comments to the database. But there are so many other things Liferay takes care of before saving the comments. If we write our own stuff, then chances are that, we’ll get deprived of these extra goodies. Hence, we’re delegating the task to one of the portal classes that is exclusively meant for this purpose. From our Portlet action we’re invoking a Portal action using [Java Reflection API](#). `PortalClassInvoker` class hides the details of this invocation and makes it very easy for us. The “`invoke`” method of this class takes five arguments.

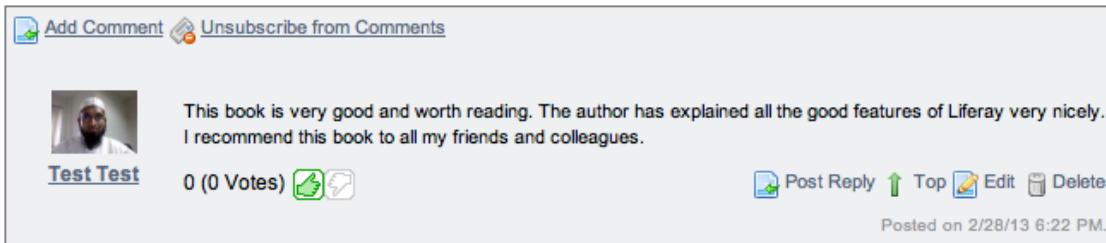
1. Boolean to denote whether a new instance of this class has to be created or not
2. The class name of the portal action class that has the method to be invoked
3. The actual method name of this class that has to be invoked
4. The type names of the parameters of this method, in the form of String array

5. The actual objects that need to be passed for these arguments in the form of an array of java objects

Alternatively, parameters 2,3 and 4 can be packaged in the form of a [MethodKey](#) object and a different “*invoke*” method can be called whose signature is,

```
PortalClassInvoker.invoke(true, methodKey, arguments);
```

Save the changes and try writing the first comment on a book. It should properly work and the comment will get displayed as below.



Look at this image. The owner can Edit or Delete this comment. Others can “Post Reply” on this comment and also give either thumbs-up or thumbs-down rating. The thumbnail of the person who posted the comment is also appearing. This triggers a whole lot of discussion and collaboration between the members of our Library. In the backdrop we’ve leveraged the existing infrastructure of Liferay’s discussion forum (*Message Boards*) to capture these discussion threads. Let’s analyse what has happened in the backend in our next step.

Step 3: A closer look at the database tables revealed the following information. When people start writing comments on a book, a new discussion thread starts with messages being shared on that particular thread. Both the parent (**mbthread**) and child (**mbmessage**) tables are captured and shown here for your convenience.

threadId	rootMessageId	rootMessageUserId	messageCount	viewCount	lastPostByUserId
21002	21001	10195	3	0	10195

The record in “mbthread” contains the messageId of the first message in this thread (*rootMessageId*), which user has started the discussion (*rootMessageUserId*), who is the last person to post on this thread (*lastPostByUserId*), total number of messages in this thread (*messageCount*) and how many times this thread has been viewed (*viewCount*). Now coming to the children of this record, it has the following records. Each record has a reference to the discussion thread (*threadId*), the *classNameId* (LMSBook), *classPK* (instance of the book), *subject*, *body* and *format* of the message.

messageId	classNameId	classPK	threadId	rootMessageId	parentMessageId	subject	body	format
21001	10901	114	21002	21001	0	114	114	bbcode
24705	10901	114	21002	21001	21001	This is a good...	This is a good...	bbcode
24708	10901	114	21002	21001	21001	this is the firs...	this is the firs...	bbcode

Apart from these columns these tables have other audit fields like *categoryId* (-1), *groupId*, *companyId*, *userId*, *userName*, *createDate* and *modifiedDate*. Here, I’d also like to quickly mention about the message format. As per the “**portal.properties**” all

the messages in the message board are in “**bbcode**” format. The other format could be “**html**”. You can get the details of BBCode from <http://www.bbcode.org>.

```
message.boards.message.formats=bbcode,html  
message.boards.message.formats.default=bbcode
```

Code Changes @ <http://code.google.com/p/lr-book/source/detail?r=76>

13.3.2 Liferay's Rating System

I am not sure if you've observed the small little icons below every comment written on a particular book. Let me explain it.

0 (0 Votes)  

This *cutie* appears as we've mentioned “**ratingsEnabled**” as an attribute for showing the comments link using “**liferay-ui:discussion**”. You press either thumbs-up or thumbs-down icons and let's see what happens to the backend. The request gets submitted as an AJAX request and in the back-end two tables are getting affected. The first table is “**ratingssentry**” and the second one is “**ratingssstats**” which maintains the statistics of the rating on each individual asset in the Liferay portal. The snapshot of these entries are shown here one below the other.

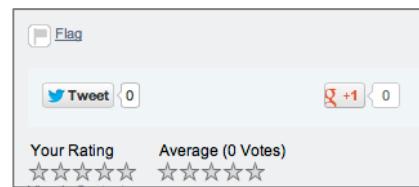
entryId	classNameId	classPK	score
24802	10117	24708	1

statsId	classNameId	classPK	totalEntries	totalScore	averageScore
24803	10117	24708	1	1	1

The above rating system is appearing just because you've enabled it in the tag used to display the discussions on a book. If you want to the rating system to be integrated as part of your application in a stand-alone fashion, what we're going to do? This is the subject of our next topic. Let's move one.

13.3.3 Stand-alone Rating System

You must have already seen what you're seeing here in the full_content view of the book asset published through Asset Publisher. The controls with stars appear after checking the “Enable Ratings” option in the Portlet's configuration page.



Imagine for some reason we've not attached our Library book with the Asset Framework and we've to have this rating feature in the book details view itself. How to achieve this? Liferay provide a very convenient taglibs that help you in placing these rating controls wherever you want. All you need to supply are the classNameId, classPK and the type of rating system that you want to display, whether five star system or thumbs-up/thumbs-down system. Let's see this in action. Open our “**detail.jsp**” and append this tag with minimal attributes.

When you check the details of a book now, you'll see the rating control appearing there cutely. Given your ratings.



Now check our two tables “**ratingsentry**” and “**ratingsstats**”. You’ll see new records appearing there for this instance of library book.

You can control the visual appearance and settings of this rating control with the help of other parameters. They are:

<code>numberOfStars</code>	The default value is five. You can either show more or less stars based on this value.
<code>ratingsEntry</code>	This is an optional field that you can supply, nothing but the entry of the ratingsEntry in the underlying table.
<code>type</code>	There are only two types – “stars” and “thumbs”. The default is “stars”.
<code>ratingsStats</code>	The value is optional and this is nothing but the reference to the record in “ ratingsstats ” table
<code>url</code>	Usually this is not passed.

Code Changes @ <http://code.google.com/p/lr-book/source/detail?r=77>

13.3.4 Inserting Social Bookmarks

This interesting taglib is Liferay's own implementation of other social bookmarking tools available over the Internet like [SocialTwist](#) and [AddThis](#). Inserting this code anywhere inside our JSP's will display the icons to bookmark the current link with twitter and facebook as shown in the picture below. The code to get this box is below,



```
<liferay-ui:social-bookmarks
    url="<%=" themeDisplay.getURLCurrent() %>" 
    title="<%=" lmsBook.getBookTitle() %>"/>
```

The other attributes of this tag apart from “url” and “title” are:

<code>displayStyle</code>	This tells the orientation of this box, either “horizontal” or “vertical”. The default is “horizontal”.
<code>target</code>	This is the normal HTML target that can take one of the values from – “_blank”, “_parent”, “_self” and “_top”.
<code>types</code>	By default it takes the types specified in “ portal.properties ”. For every type you should have a corresponding JSP defined. See the details for the entries in “ portal.properties ”.

Entries in “**portal.properties**” file:

```
social.bookmark.types=twitter,facebook,plusone
social.bookmark.jsp[facebook]=\
    /html/taglib/ui/social_bookmark/facebook.jsp
social.bookmark.jsp[plusone]=\
    /html/taglib/ui/social_bookmark/plusone.jsp
social.bookmark.jsp[twitter]=\
    /html/taglib/ui/social_bookmark/twitter.jsp
```

Code Changes @ <http://code.google.com/p/lr-book/source/detail?r=82>

13.4 Social Activity Framework

Let's take the case of a user Activity of writing a comment on a Library Book as seen in the previous section. With some massaging we'll make this normal activity as a social activity in the portal. Any social activity has a direct relationship with the ways uses collaborate in the portal. A social activity will potentially trigger a collaboration life cycle and it'll also earn the user some points in the portal called as "Social Equity". This concept of Social Equity will be explained in great length in the upcoming section. In this section we're going to see two things:

1. How to transform a "normal activity" to a "social actity"?
2. How to track all the social activities of users via "Activities" Portlet?

13.4.1 Transformation to a Social Activity

Whenever a logged in user "writes a comment" on a book, we're going to designate this activity as a "Social Activity" of the portal or within his current community or group. Define a new method in our Portlet class "**LibraryPortlet.java**".

```
private void logSocialActivity(PortletRequest portletRequest) {  
    ThemeDisplay themeDisplay = (ThemeDisplay)  
        portletRequest.getAttribute(WebKeys.THEME_DISPLAY);  
    long userId = themeDisplay.getUserId();  
    long groupId = themeDisplay.getScopeGroupId();  
    long classPK = ParamUtil.getLong(portletRequest, "classPK");  
  
    try {  
        SocialActivityLocalServiceUtil.addActivity(  
            userId, groupId, LMSBook.class.getName(),  
            classPK, 1,  
            "User commented on book: " + classPK, classPK);  
    } catch (PortalException e) {  
        e.printStackTrace();  
    } catch (SystemException e) {  
        e.printStackTrace();  
    }  
}
```

Append the following one line of code to the "discussOnThisBook" in the same Portlet class to invoke this new method after a comment is being made.

```
logSocialActivity(actionRequest);
```

Save the changes and submit a comment on one of your favorite books in the Library and check the back-end tables to see the impacts. After a book comment, two records got inserted into the "**socialactivity**" table.

activityId	mirrorActivityId	classNameId	classPK	type_	receiverUserId
301	0	10901	114	1	114
302	301	10901	114	1	10195

Some of the parameters that are passed at the time of creating these entries are,

-
- The **type** identifies the type of an activity. You're free to define this however you want. See [WikiActivityKeys](#) for an example. If you want you can also create a new file with name, “**LibraryActivityKeys.java**”.
 - The **extraData** parameter is a string that can contain any additional info you want.
 - The **receiverUserId** is the user who the activity is done to.

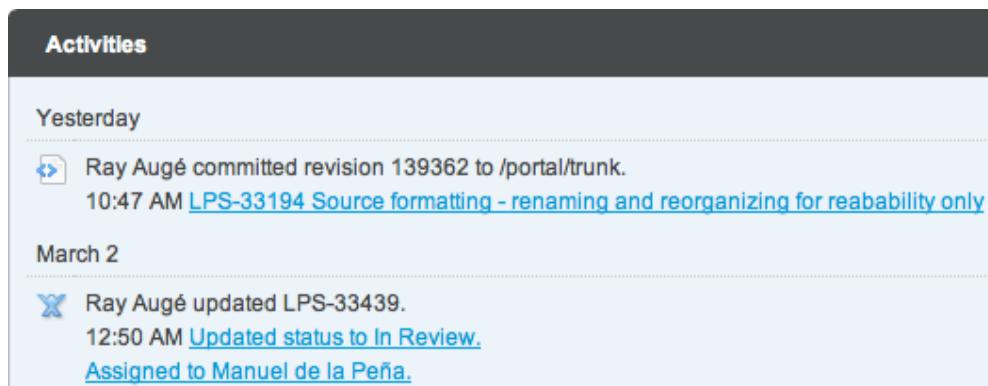
There is one more API of “`SocialActivityLocalServiceUtil`” for removing all social activities of an asset whenever the asset itself is getting deleted from the portal.

```
SocialActivityLocalServiceUtil.deleteActivities(
    className, classPK);
```

In the next section we're going to see how to make our Portlet interpret these social activities and publish them as items in the “Activities” Portlet of Liferay.

13.4.2 Social Activity Intrepreter and Tracking

If you're a regular visiter of Liferay.com website, then you must have probably noticed the “Activities” Portlet appearing in the profile pages of all users out there. Let me show you from the profile page of one of my favorite guys at Liferay, [Ray Auge](#). Though there are many entries there, I am just showing couple of them for your perusal. Most of his activities are related to code check in as he is part of Liferay's core engineering team.



The screenshot shows the Liferay Activities portlet. It has a dark header bar with the word "Activities". Below it, a list of activities is displayed. The first entry is for "Yesterday": "Ray Augé committed revision 139362 to /portal/trunk." followed by a link "10:47 AM [LPS-33194 Source formatting - renaming and reorganizing for readability only](#)". The second entry is for "March 2": "Ray Augé updated LPS-33439." followed by a link "12:50 AM [Updated status to In Review.](#)" and another link "[Assigned to Manuel de la Peña.](#)".

Assume that we've this similar feature in our own online Library Management Portal, where one use will be able to see the portal activities of the other user and vice-versa. Though this Portlet is usually added to a user's profile page, in our case, we'll drag and drop this Portlet in our Library's welcome page itself. Initially you'll not find any entries there. The next thing is to interpret every entry that gets into the “socialactivity” and make it appear there so that the Library users come to know who is doing what in the Library. The process consists of four steps.

Step 1: Create a new class “**LibraryActivityInterpreter.java**” under the package “`com.library.social`”, extending [BaseSocialActivityInterpreter](#) and override the “`doInterpret`” method. This method of this class returns an object of type [SocialActivityFeedEntry](#), with parameters **link**, **title** and **body**.

```
package com.library.social;
```

```

import com.liferay.portal.kernel.util.StringPool;
import com.liferay.portal.theme.ThemeDisplay;
import
com.liferay.portlet.social.model.BaseSocialActivityInterpreter;
import com.liferay.portlet.social.model.SocialActivity;
import com.liferay.portlet.social.model.SocialActivityFeedEntry;
import com.slayer.model.LMSBook;
import com.slayer.service.LMSBookLocalServiceUtil;

public class LibraryActivityInterpreter
        extends BaseSocialActivityInterpreter {

    public String[] getClassNames() {
        return new String[]{LMSBook.class.getName()};
    }

    protected SocialActivityFeedEntry doInterpret(
        SocialActivity activity, ThemeDisplay themeDisplay)
        throws Exception {

        long bookId = activity.getClassPK();
        LMSBook lmsBook =
            LMSBookLocalServiceUtil.fetchLMSBook(bookId);

        String link = getLink(themeDisplay, bookId);
        String title = getTitle(activity,
            lmsBook.getBookTitle(), link, themeDisplay);
        String body = StringPool.BLANK;

        return new SocialActivityFeedEntry(link, title, body);
    }
}

```

Step 2: Create a private method “getLink”.

```

private String getLink(ThemeDisplay themeDisplay, long bookId) {
    StringBuilder sb = new StringBuilder()
        .append(themeDisplay.getPathFriendlyURLPublic())
        .append("/guest/my-library/-/library/detail/")
        .append(bookId);

    return sb.toString();
}

```

Step 3: Create a private method “getTitle”.

```

private String getTitle(SocialActivity activity,
        String content, String link, ThemeDisplay themeDisplay) {
    String userName = getUserName(
        activity.getUserId(), themeDisplay);

    String text = wrapLink(link, content);

    String groupName = getGroupName(
        activity.getGroupId(), themeDisplay);

    String pattern =
        "{0} has commented on the book \"{1}\" in the group, {2}";

```

```
        return themeDisplay.translate(
            pattern, new Object[] {userName, text, groupName});
    }
```

Step 4: One final thing to be done before you see everything working in a perfect harmony. This step is to associate our Portlet and the new social activity interpreter class. This entry should go inside our “**liferay-portlet.xml**”. This entry should come before “**control-panel-entry**” entries in this file.

```
<social-activity-interpreter-class>
    com.library.social.LibraryActivityInterpreter
</social-activity-interpreter-class>
```

Save all the changes and you could see whenever user write comments on the Library books, they start appearing in our “Activities” Portlet. Congratulations! You’ve integrated the commenting feature on a Library book with the Social Activities Portlet. Our next item is how to attach this social activity with the Liferay’s social rewarding system called Social Equity.

Code Changes @ <http://code.google.com/p/lr-book/source/detail?r=79>

13.5 Social Equity Framework

One of the key requirements of a good collaboration portal is the ability to track the various activities of its users and to appropriately reward them with points in order to keep them more engaged and glued to the portal. Every portal technology has it’s own way of naming this feature. Some call it as “**Loyalty Program**”, some call it as “**Miles**”, etc. Liferay calls it with an entirely new nomenclature, “**Social Equity**” which is a direct result of a “**Social Activity**” performed by a user in the portal. This is a brand new feature in Liferay 6 onwards. This system of awarding users for their activities itself has been made as a complete framework in Liferay. Let’s spend some time to understand the basic concepts before moving on to integrate this awesome feature with our own Library Portlet. We’ll start our discussion with understanding the evolution of “Community Equity” Model.

13.5.1 Understanding “Community Equity” Model

The objective of this system is to build a dynamic social capital system by measuring the contribution and participation of a user and the information value of an asset on which he acts. A person can gain such equity through certain activities performed in various Liferay sites (*communities*). The activities that award equities include,

- Adding contributions (wikis, blogs, custom assets)
- Rating, Commenting, Viewing and Voting on assets
- Viewing Contents, Bookmarking and Sharing
- Searching and Tagging (*yet to be implemented*)
- Inviting new members to join (*yet to be implemented*)

The following are the four main values used to describe engagement in communities:

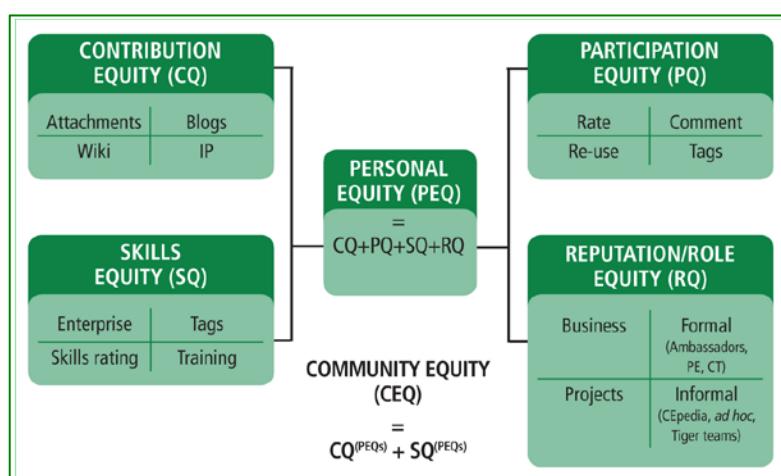
Information Equity (IQ): This equity shows the importance and quality (measured by popularity) of the information contained in an asset. It is calculated through social activities related to the information.

Contribution Equity (CQ): The contribution equity measures the contribution of a person to the community. This value is calculated from the information equity of the contributed assets.

Participation Equity (PQ): This equity measures the active participation of a person. It is calculated by measuring the feedback a person has provided to other community contributions (assets). Viewing a contribution can be translated as feedback as well.

Personal Equity (PEQ): The ultimate result that describes a person's achievements and participation in the community. This value is the sum of a person's contribution and participation equities.

The Liferay’s model of Social Equity is based on the white paper and calculations of Peter Reiser from SUN Microsystems. The following is the “Community Equity” model adapted from Sun Microsystems’s original model. Off course, we’re not going to get into the details of the complete model. The idea is to just give you and overview of how Liferay has styled its Social Equity calculations based on this innovative model.



13.5.2 Measuring Social Activity

When there is a lot of user activities and interactions in a portal, it will be very helpful if we've tools to separate them out into various buckets and measure the quality of the user contributions. Liferay contains a lot of applications that end users can use to communicate with each other and provide information. Some of this information is good and helpful and some of it can be rather useless. Using Liferay's Social Activity feature will help show which users are making real, valuable contributions and reward them accordingly.

To activate Social Activity, you'll first need to determine which collaboration applications you want to use Social Activity. There are currently three types of content you can use with Social Activity - Blogs Entries, Message Board Messages, and Wiki Pages. Social Activity tracks three metrics from within each of these applications two are for the user - **Participation** and **Contribution** - and the other, **Popularity**, is for the asset involved. Now, we'll see how to do this from the Control Panel. Login to the portal as a "Site Admin" and go to the Control Panel. Under the Web Site section, click on the link "Social Activity" to get the settings form as shown.

Enable Social Activity for:

Blogs Entry When a User:
 Message Boards Mess...
 Wiki Page

The user gets 5 participation point(s) and 0 contribution point(s).
The asset gets 0 popularity point(s).

Let's activate Social Activity for Blogs Entries. Check the box next to Blog Entry. You now have options to set point values and limits on several different actions for blogs. You'll notice each item on the list has dropdowns you can use to set the number of participation and contribution points; popularity points are tied directly to contribution points. In addition to that, you can expand the box by clicking Limits in the top right of each list item. You can use this to set a limit on how many times a user can perform this activity with a specific asset and receive the requisite points. For some activities, you can set limits on both participation and contribution points but on new content creation you can only set limits on participation points.

After setting your preferences, click "Save". Let's now look into the changes that have happened in the back-end. There are five tables, related with social activities and social equity. One of them, "socialactivity", we've already seen. The other four tables where the settings, limits and counters get stored are:

socialactivityachievement	socialactivitycounter
socialactivitylimit	socialactivitysetting

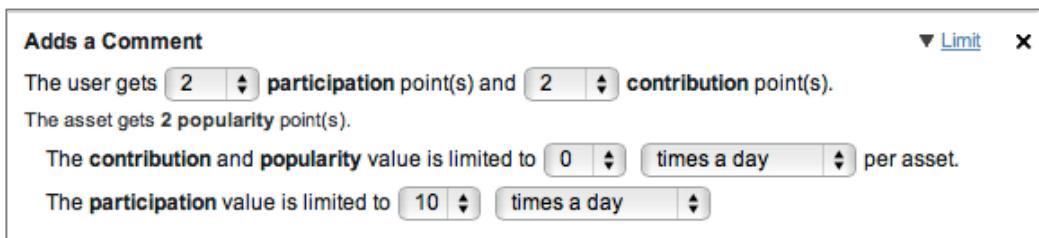
The settings we just saved are entered into the "**socialactivitysetting**" table.

activitySettingId	classNameId	activityType	name	value
25806	10007	10001	contribution	{"enabled":true,"li...
25808	10007	10001	popularity	{"enabled":true,"li...
25813	10007	10003	participation	{"enabled":true,"li...
25901	10901	10005	participation	{"enabled":true,"li...

The entries of the “value” column looks like below. These intial values come from the social activity configuration file, “**liferay-social.xml**” for that Portlet.

value
{"enabled":true,"limitValue":0,"ownerType":3,"value":0,"limitEnabled":true,"limitPeriod":1}
{"enabled":true,"limitValue":0,"ownerType":2,"value":0,"limitEnabled":true,"limitPeriod":1}
{"enabled":true,"limitValue":0,"ownerType":1,"value":0,"limitEnabled":true,"limitPeriod":1}
{"enabled":true,"limitValue":1,"ownerType":1,"value":5,"limitEnabled":true,"limitPeriod":1}

Except for all actions that do not involve the creation of a new asset, all of the contribution points always go to the original asset creator and all popularity points go to the original asset. That means if Votes on a Blog is set to have 1 Participation point and 5 Contribution points (and therefore 5 Popularity points), the user who votes on the asset will receive 1 participation point, the user who created the asset will receive 5 contribution points, and the asset will receive 5 popularity points. The screenshot below shows how to set these limits on a given action.



Having got a general idea of how the configuration for the social activity and social equity works, let's now move our focus to our Library Portlet. We'd like to hook the commenting on library books to the social equity system and give reward points for the users who write comments on the books.

13.5.3 Enabling Social Activity for Library Portlet

Step 1: Create a new file “**liferay-social.xml**” under “**docroot/WEB-INF**” with the following contents. For a detailed description of the entries in this file, please refer to its corresponding DTD “**liferay-social_6_1_0.dtd**”.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE liferay-social PUBLIC
        "-//Liferay//DTD Social 6.1.0//EN"
        "http://www.liferay.com/dtd/liferay-social_6_1_0.dtd">

<liferay-social>
    <activity>
        <model-name>com.slayer.model.LMSBook</model-name>
        <activity-type>
${com.liferay.portlet.social.model.SocialActivityConstants.TYPE_ADD_COMMENT}
            </activity-type>
            <language-key>ADD_COMMENT</language-key>
            <log-activity>true</log-activity>
            <contribution-limit enabled="false" />
            <participation-value>5</participation-value>
            <participation-limit period="day">
                10</participation-limit>
```

```

<counter>
    <name>asset.comments</name>
    <owner-type>asset</owner-type>
</counter>
<counter>
    <name>creator.comments</name>
    <owner-type>creator</owner-type>
</counter>
<counter>
    <name>user.comments</name>
    <owner-type>actor</owner-type>
</counter>
</activity>
</liferay-social>

```

Once this file is ready, you should manually deploy the Portlet using the “**ant deploy**” target. You can do this by right clicking on the project from Eclipse and choosing Liferay → SDK → deploy. Once this Portlet is deployed successfully, you should get these log messages on the server console, confirming the linking between our Portlet and the social system has happened properly.

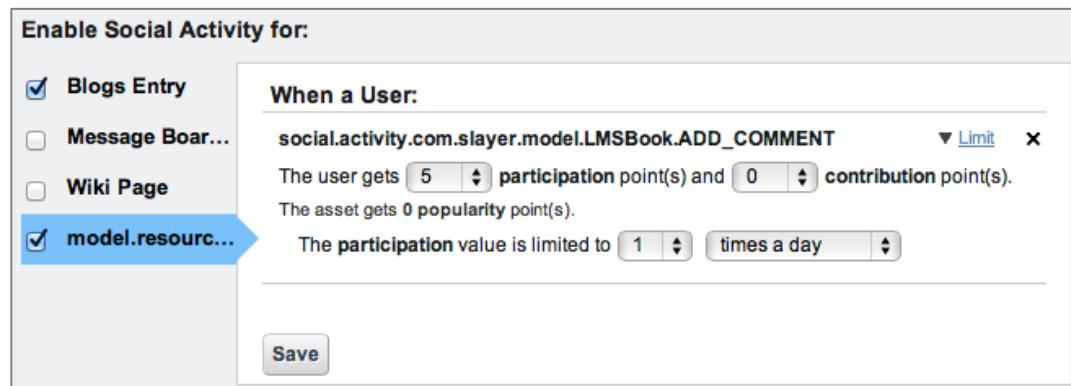
```

12:46:19,364 INFO [pool-2-thread-4][SocialHotDeployListener:86]
Registering social for library-portlet
12:46:19,366 INFO [pool-2-thread-4][SocialHotDeployListener:95]
Social for library-portlet is available for use

```

Step 2: With the social system activated and by virtue of setting “`<log-activity>true</log-activity>`” in our “**liferay-social.xml**”, we don’t have to programmatically create a social activity whenever a user comments on a book. Hence, remove the line from the “`discussOnThisBook`” of our Portlet class “**LibraryPortlet.java**” – `logSocialActivity(actionRequest);`

Step 3: Login to the control panel and check the “Social Activity” link. You should see a new item there with the initial values as configured in “**liferay-social.xml**”. Check the box against our newly added Social Activity item and save the changes.



Save the changes and write some comments on one of the books in our Library and see the changes that are happening to the five tables related with Liferay’s social system. The popularity of the Book asset goes up proportionately based on the user’s contribution points. It’s easy to assign points. You can arbitrarily assign points for just about anything. The challenge is making the points significant in some way. As mentioned before, the primary purpose of social activity tracking is to make sure that

users who regularly contribute to the portal and participate in discussions are recognized as such. So the central piece of the social equity display is the User Statistics Portlet. This most important central piece of the Social Equity system we're going to see next.

Code Changes @ <http://code.google.com/p/lr-book/source/detail?r=80>

13.5.4 User Statistics and Top Users Portlets

Liferay does not end the story here. It also provides some cute powerful portlets that display the statistics of the user's participation, contribution and assets popularity. The User Statistics portlet displays a list of users ranked by an amalgamation of their participation and contribution scores. By clicking on the Configuration icon for the portlet, you can change some of the specifics of the rankings. There are four check boxes that you can enable or disable:



Rank by Contribution: If this is checked, a user's contribution score will be used as a factor in calculating their rank.

Rank by Participation: If this is checked, a user's participation score will be used as a factor in calculating their rank.

Show Header Text: Determines whether the title shows or only the rankings.

Show Totals: Toggles the display of the users activity score next to their name.

A screenshot of the Liferay User Statistics portlet displaying the top users. The portlet header states: "Top users out of 2. Ranking is based on participation and contribution." It lists two users:

- Test Test**: Rank: 1, Contribution Score: 0 (Total: 0), Participation Score: 5 (Total: 5)
- aa aa aa**: Rank: 2, Contribution Score: 0 (Total: 0), Participation Score: 5 (Total: 5)

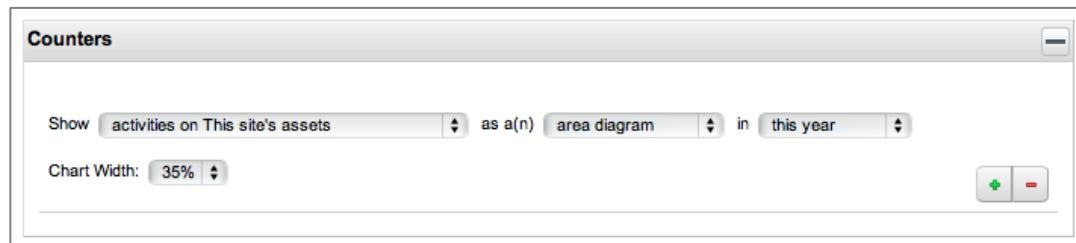
The user names are displayed with their profile pictures.

Display Additional Activity Counters: You can toggle the display of any number of other pieces of information next to the users name in the statistics, ranging from the number of comments on assets a user has created to the number of wiki articles that the user has created. If you want to display multiple data points, you can click the plus button to add one and the minus button to remove one. You can have as many data

points displayed as you want, but displaying too many might make the User Statistics portlet appear very cluttered.

13.5.5 Group Statistics Portlet

Apart from displaying user statistics, there is another interesting Portlet that Liferay provides – Group Statistics. This Portlet displays the participation, contribution and popularity of the assets across the entire group. The following is the configuration settings of the Group Statistics Portlet.



This Portlet can show multiple things in various charts for either this year or the last 12 months. As with the case of User Statistics, we can configure multiple items to be displayed. The items that can be shown through this Portlet are:

“On / To / Of” Site’s Assets	“Of / By” Site’s Users
Activities	Achievements & Activities
Attachments added	Attachments Added
Cancelled Subscriptions	Blog entries and updates on them
Votes and Comments	Votes, Comments and Subscriptions
Subscriptions	Message Board Posts
Popularity	Wiki Pages and updates on them

There are a wide-ranging number of actions that you can provide social credit for. Users can receive credit for everything from subscribing to a blog to writing wiki articles. You can easily tweak the numbers in the control panel if it becomes clear that certain activities are weighted too high or too low. Social Activity can be an invaluable tool for portals that are heavily driven by community-created content. It allows you to easily recognize users who are major contributors and it indicates to new users whose advice will be most trustworthy. Social Activity is easy to set up and can be configured differently for each site, increasing the flexibility of your portal.

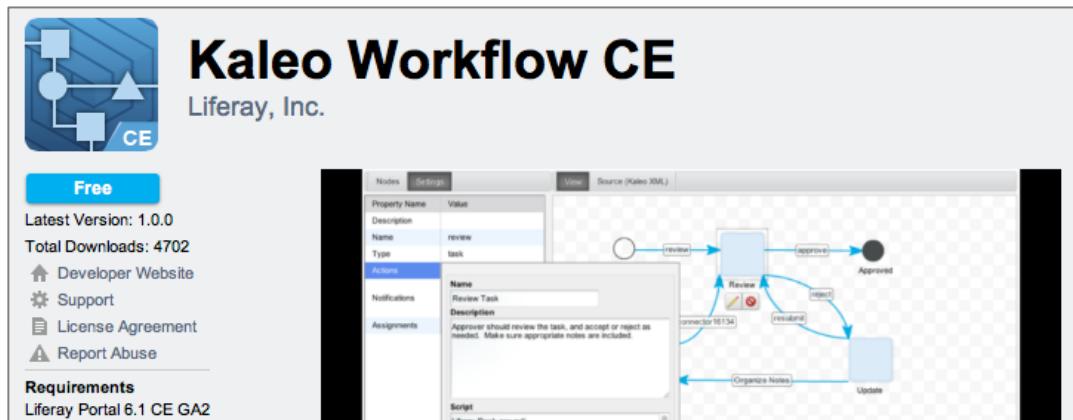
13.6 Business Process (BPM) Framework

Any matured business will have a well-defined process. The art and science of managing the processes of a business is called as Business Process Management (BPM). While developing applications for a business we've to take this important aspect into consideration. The simplest use-case of a business process is the Leave Application process. The employee logs in to the company's HR system and applies for leave for specific number of days. The leave application then goes to his immediate manager who can either approve the leave or decline the request or forward the application to his boss.

Usually we begin with simulating the whole process in a sheet of paper. But, thanks to the advancement in the field of computing, there are tools and standards available to design the whole workflow process. These are called as BPM Modelling tools. The latest standard for BPM can be found at <http://www.bpmn.org>. In the above example, “Leave Application” is the entity or model that is taken through a workflow process.

13.6.1 Liferay and Workflow

Liferay has an inbuilt workflow engine called as Kaleo. It is a very lightweight yet powerful workflow engine that comes as a separate plugin and available in the Liferay marketplace. You've to first get this plugin download from the marketplace and installed to your local Liferay server.



Once the plugin is successfully installed, you'll see some obvious changes inside your Control Panel.

Portal Section	Website Section	User Section
Workflow	Workflow Configuration	My Workflow Tasks My Submissions

If you go inside the “Workflow” link of Portal Section, you'll find an option to configure the default workflow for the whole portal. You can assign a workflow process for any of those given entities and see the effects.

13.6.2 Our Library and Workflow

Having seen the basic workflow features of Liferay, we're now going to implement workflow process for our Library book as well. It all starts with a request for a book from the member of a Library. Then it goes through a whole lot of process involving the finance team, procurement team and the librarian before the book finally gets commissioned into the library and made available for all the members of the Library. Let's see one portion of this whole process in action as we move on. It comprises of four steps before we could inject workflow capabilities for our Library Portlet.

Step 1: Create a new Workflow handler class, “LibraryWorkflowHandler” that extends [BaseWorkflowHandler](#). Put this new class under “com.library.workflow” package. Make the initial contents of this file to look like,

```
package com.library.workflow;

public class LibraryWorkflowHandler extends BaseWorkflowHandler {
    static final String CLASS_NAME = LMSBook.class.getName();

    public String getClassName() {
        return CLASS_NAME;
    }

    public String getType(Locale locale) {
        return null;
    }

    public Object updateStatus(int status, Map<String,
        Serializable> workflowContext)
        throws PortalException, SystemException {
        return null;
    }
}
```

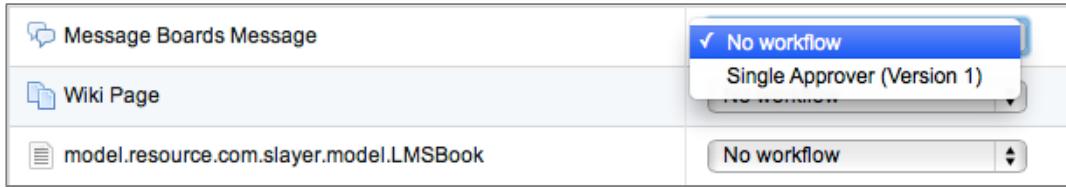
Make the necessary imports,

```
import java.io.Serializable;
import java.util.Locale;
import java.util.Map;
import com.liferay.portal.kernel.exception.PortalException;
import com.liferay.portal.kernel.exception.SystemException;
import com.liferay.portal.kernel.workflow.BaseWorkflowHandler;
import com.slayer.model.LMSBook;
```

Step 2: Insert an entry in “**liferay-portlet.xml**” pointing to this new handler class. This is how we’re registering our Portlet with the workflow handler. Insert this entry between “**custom-attributes-display**” and “**action-url-redirect**” tags.

```
<workflow-handler>
    com.library.workflow.LibraryWorkflowHandler
</workflow-handler>
```

Save the changes and do a manual deployment of our Library Portlet, if required. Login as Portal Admin and go to the Control Panel. Under the “Portal” section, click the “**Workflow**” and then click “**Default Configuration**”. Our new entity will appear in the list along with the existing entities that can get pushed into a workflow process.



Step 3: Next step is to make the entity workflow aware. Inorder to do this, add the following columns to our LMSBook entity in “**service.xml**” and re-generate the service layer. These fields are required to keep track of the workflow status during every transition. Optionally you can inject a reference of [WorkflowInstanceLink](#) into our DTO class, “**LMSBookLocalServiceImpl.java**”.

```
<!-- Workflow fields -->
<column name="status" type="int" />
<column name="statusByUserId" type="long" />
<column name="statusByUserName" type="String" />
<column name="statusDate" type="Date" />

<reference package-path="com.liferay.portal"
           entity="WorkflowInstanceLink" />
```

Mere running of the service builder will not automatically create these new columns in the underlying database. We've to manually do it by running this script.

```
ALTER TABLE lms_lmsbook
  ADD COLUMN status INT NULL AFTER sampleChapter,
  ADD COLUMN statusByUserId BIGINT(20) NULL AFTER status,
  ADD COLUMN statusByUserName VARCHAR(75) NULL AFTER statusByUserId,
  ADD COLUMN statusDate DATETIME NULL AFTER statusByUserName;
```

Step 4: At the time of a new book getting added to our Library we've to initiate the workflow process, so that the book can go through the required workflow process before getting officially commissioned into our Library. Let's trigger the new workflow instance at the middle of our “*insertBook*” method in our DTO class, “**LMSBookLocalServiceImpl.java**”, just before persisting the new book.

```
// workflow status
lmsBook.setStatus(WorkflowConstants.STATUS_PENDING);
lmsBook.setStatusByUserId(serviceContext.getUserId());
lmsBook.setStatusDate(new java.util.Date());
```

Then immediately after the object is persisted, initiate the workflow process by invoking the below code. Insert this code at the end of the method. This method “*startWorkflowInstance*” takes seven parameters.

```
try {
    WorkflowHandlerRegistryUtil.startWorkflowInstance(
        lmsBook.getCompanyId(),          // companyId
        lmsBook.getGroupId(),           // groupId
        lmsBook.getUserId(),            // userId
        LMSBook.class.getName(),       // className
        lmsBook.getPrimaryKey(),        // classPK
        lmsBook,                      // model
```

```

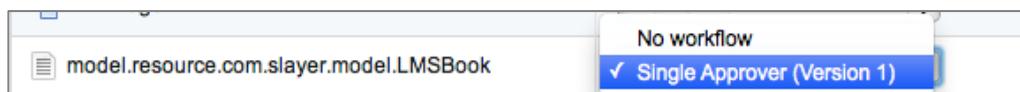
        serviceContext);
    } catch (PortalException e1) {
        e1.printStackTrace();
    } catch (SystemException e1) {
        e1.printStackTrace();
    }
}
// serviceContext

```

Save all the changes and let's move onto the next section to see all the changes that are happening both at the database level and at the UI level.

13.6.3 Database and UI changes

With all the above changes to our Portlet, check the database tables after adding a new book to our Library. Except the new record getting inserted into the “lms_lmsBook” table there are no other effects to other tables in the database. Now let activate the workflow for our custom entity “**LMSBook**” from the UI and check which table is getting impacted. In the workflow “Default Configuration” tab, change from “**No Workflow**” to “**Single Approver (Version 1)**” against our custom entity and save the settings.



This new setting will get saved in the table “**workflowdefinitionlink**” back in our database. The groupId is “0” because the setting is at the portal level.

workflowDefinitionLinkId	groupId	classNameId	typePK	workflowDefinitionName	workflowDefinitionVersion
28601	0	10901	0	Single Approver	1

After attaching our book entity with the workflow, now create a new book and check the additional impacts happening to the database.

Table Name	Records	Table Name	Records
Workflowinstancelink	One	Kaleolog	Four
Kaleoinstance	One	Kaleotaskassignmentinstance	Five
Kaleoinstancetoken	One	Kaleotaskinstancetoken	One

Now let's examine the columns of these tables and get a closer look at these kaleo workflow tables, starting with the “**workflowinstancelink**” table. I leave it to you to check the relationship between these tables and the other kaleo tables (*there are total 16 of them*).

workflowInstancelinkId	groupId	classNameId	classPK	workflowInstanceId
28637	10179	10901	141	28634

Coming to the User interface, you'll observe the following changes in the Control Panel. You can see them one after the other.

- Under the “Website” section, you'll see the link “Workflow Configuration”. You can use this place to override any settings that are made at the portal level. Workflow

can be customized for every site of the portal.



2) Under the “User” section, you’ll find two links. These links will start appearing as soon as you deploy Kaleo workflow Portlet. But now, you’ll see new entries appearing under both these headings. The first one shows all the workflow tasks that are pending for the currently logged in user to take an action. The second one lists all the tasks that are submitted by the currently logged in user. These items have to be worked upon by some other users of the portal and they’ll see these items under their “My Workflow Tasks” section. Both these headings are appearing by virtue of our custom WorkflowHandler class.

Click “My Workflow Tasks” to see the list of recently added books that are pending for approval. They’ll not be visible to the library members unless and until they get reviewed and approved by the Library Manager. This demands you to slightly change the filter logic to display only the books whose status is “**approved**”.

Assigned to My Roles					
Task	Asset Title	Asset Type	Last Activity Date	Due Date	
Review	kaleo	null	3/5/13 12:00 PM	Assign to Me	Actions
Review	ccc	null	3/4/13 11:59 AM	Update Due Date	Actions
Review	111	null	3/4/13 11:38 PM	Never	Actions

Click on one of these items and you’ll be taken to the “assets” abstract view as shown in the next page. This happens because we’ve already hooked our entity with asset framework. Unfortunately the Asset Type column is showing as “null”. We need to show a proper value. Go back to our “**LibraryWorkflowHandler.java**” class and modify the “`getType`” method as below and check the list now.

```
public String getType(Locale locale) {
    return ResourceActionsUtil.getModelResource(
        locale, CLASS_NAME);
}
```

13.6.4 Custom Handling of workflow events

In this final sub-section of this topic, we'll see how to set the workflow status of any given entity; in our case it is the Library Book. Open the main workflow handler class for our Portlet, “**LibraryWorkflowHandler.java**” and define this method “`updateStatus`” that takes three parameters.

```
public Object updateStatus(int status, Map<String,
    Serializable> workflowContext)
    throws PortalException, SystemException {

    if (status > WorkflowConstants.STATUS_APPROVED) return null;

    long statusByUserId = GetterUtil.getLong(
        (String)workflowContext.get(
            WorkflowConstants.CONTEXT_USER_ID));

    long resourcePrimKey = GetterUtil.getLong(
        (String)workflowContext.get(
            WorkflowConstants.CONTEXT_ENTRY_CLASS_PK));

    LMSBook lmsBook =
        LMSBookLocalServiceUtil.fetchLMSBook(resourcePrimKey);

    lmsBook.setStatus(WorkflowConstants.STATUS_APPROVED);
    lmsBook.setStatusByUserId(statusByUserId);
    lmsBook.setStatusDate(new java.util.Date());

    return LMSBookLocalServiceUtil.updateLMSBook(lmsBook);
}
```

The following is the list of attributes that can be retrieved from “workflowContext” object, like how it is done in the above code.

Attribute Name	Constant defined in “WorkflowConstants.java”
serviceContext	<i>CONTEXT_SERVICE_CONTEXT</i>
entryClassName	<i>CONTEXT_ENTRY_CLASS_NAME</i>
groupId	<i>CONTEXT_GROUP_ID</i>
entryType	<i>CONTEXT_ENTRY_TYPE</i>
userId	<i>CONTEXT_USER_ID</i>
taskComments	<i>CONTEXT_TASK_COMMENTS</i>
companyId	<i>CONTEXT_COMPANY_ID</i>
entryClassPK	<i>CONTEXT_ENTRY_CLASS_PK</i>
transitionName	<i>CONTEXT_TRANSITION_NAME</i>

Code Changes @ <http://code.google.com/p/lr-book/source/detail?r=81>

Summary

This chapter covered Liferay’s collaboration framework.

14. A Tour of Advanced API's

This chapter covers

- Portlet Messaging – MessageBus
 - Indexing and Search
 - Advanced Search Features
 - Device Detection API
 - Creating a Custom Taglib
 - More on Portlet + Web Content Integration
 - Form Navigator Tag
-

This last chapter of this book will take you through a tour of some advanced API's. The first section will be on MessageBus – a powerful mechanism to pass messages between the various components of the portal both synchronously and asynchronously. Then we'll move on to see Liferay's indexing and search capabilities. Under the hood Liferay uses Lucene for relentlessly doing this job. Once the indexing is done in the way we can the data can be searched in multiple ways. There is one section dedicated to cover all the advanced search features of Liferay with the help of underlying Lucene search engine.

We'll quickly catch up with the Device Detection API's. Using these API's we can make the programs intelligent enough to understand the devices from where they are being accessed. The programs can behave differently for different devices. The Portlet contents shown in a browser may not be same inside an android phone. These API's helps in building applications that can work on any device making them ubiquitous.

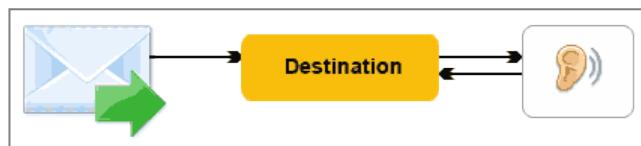
After the discussion on Device Detection API, we'll do a complete and hands-on implementation of a custom taglib within a Portlet plugin. We'll take the example of the subscription link for Library and a book and see how to make it as a taglib so that the code can be reused.

The last two sections may not be considered as API's, but they are more of a best practice. The first one will discuss how to externalize the content and design of a book details page in the form of a Web Content and the next one will discuss about an UI pattern called as Form Navigator.

14.1 Portlet Messaging – MessageBus

In Section [13.1.2 Enabling Subscription for Library Portlet](#), we have seen how to make use of Liferay's Subscription Framework to enable library members to subscribe to our Library, so that any major updates that are happening in the Library, they get notified. One example could be addition of new book in to the Library. All members who have subscribed (*opted-in*) for email notifications should be notified about this event and get traction for the new book amongst the members. Imagine there are one thousand users who subscribed for the Library.

So, the requirement is to fire one thousand emails immediately after the new book has been reviewed and approved for availability to the members. Do you think sending one thousand emails synchronously is a good idea? After pressing the “Approve” button the Librarian has to literally wait for almost 15-20 minutes till the backend process completes and he could see the next screen. This is not at all a good thing that you'd like to see if you're the librarian. Then what else could be the solution? The answer is to use a well-established middleware messaging system based on [JMS](#). Liferay has an inbuilt messaging system that is lightweight implementation of JMS. The upcoming sub-sections will present the various aspects of this powerful Messaging API. The following image is a super simplification of this model.



14.1.1 Integrating Subscriptions to MessageBus

Contrary to convention, in this first sub-section, I am going to explain you how we are going to achieve the above use case of sending emails to all members who have subscribed for notifications from our Library. In the subsequent sub-sections we'll see various other uses of MessageBus and the ways of configuring it for different scenarios and situations. First things First. Let's implement this and keep moving. The implementation comprises of four steps.

Step 1: The control center for the MessageBus to work in perfect harmony is the “**messaging-spring.xml**”. Create this file under “src/META-INF” of our plugin with the following contents. Soon I am going to explain each section of this file.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans default-destroy-method="destroy"
       default-init-method="afterPropertiesSet"
       xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

    <!-- Listener(s) -->
    <bean id="listener.library"
          class="com.library.listener.NewBookListener" />

    <!-- Destination(s) -->

```

```

<bean id="library.destination"
      class="com.liferay.portal.kernel.messaging.ParallelDestination">
    <property name="name" value="destination/library" />
  </bean>

<!-- Configurator -->
<bean id="messagingConfigurator"
      class="com.liferay.portal.kernel.messaging.config.PluginMessagingConfigurator">

  <property name="messageListeners">
    <map key-type="java.lang.String"
         value-type="java.util.List">
      <entry key="destination/library">
        <list
          value-type="com.liferay.portal.kernel.messaging.MessageListener">
          <ref bean="listener.library" />
        </list>
      </entry>
    </map>
  </property>
  <property name="destinations">
    <list>
      <ref bean="library.destination" />
    </list>
  </property>
</bean>
</beans>

```

A MessageBus System configuration consists of four important components:

1. **Message Bus** – Manages transfer of messages from message senders to message listeners
2. **Destinations** – Are addresses or endpoints to which listeners register to receive messages
3. **Listeners** – Consume messages received at destinations. They receive all messages sent to their registered destinations.
4. **Senders** – Invoke the Message Bus to send messages to destinations

In the above file, we have mentioned our Message Listener, a Destination for the Message and the Configurator that does the routing of messages between the Senders, Listeners and Destinations.

Step 2: In this step we'll identify the place and context we are going to send messages to our MessageBus. This is nothing but registering a message with the Destination from where the Listener will pick up the message. The “sender” will be the place where a book is finally getting commissioned into our Library. I think the ideal place to write this code is our WorkflowHandler class, “**LibraryWorkflowHandler.java**” where the final status of the newly added book is changing from PENDING to APPROVED. Let's open this file and insert the below code at the end of “updateStatus” method, just before the return statement.

```

// Sending New Book Message to the MessageBus
Message message = new Message();
message.setPayload(lmsBook);
String destinationName = "destination/library";
MessageBusUtil.sendMessage(destinationName, message);

```

Make the necessary imports in this file.

```
import com.liferay.portal.kernel.messaging.Message;
import com.liferay.portal.kernel.messaging.MessageBusUtil;
```

Apart from the “`setPayload`” method, you can use the following methods to set values / objects into a “`message`” that is the super encapsulating payload for our `MessageBus`.

```
message.setDestinationName(destinationName);
message.setValues(values); // values is a Map<String, Object> type
message.put(key, value); // value is of type "Object"
```

Similarly the “`MessageBusUtil`” utility class also has got many methods to send the message across the `MessageBus`. We are done with polling the destination with a message. The next step is to create the listener that is listening to the changes happening in the destination. In the listener we’ll write the code of what to be done after listening to a message sent by the Sender.

Step 3: This step is to write the Listener who’ll listen to any message coming via the destination as configured in “`messaging-spring.xml`” in [step 1](#). Create a new class “`NewBookListener.java`” extending [`BaseMessageListener`](#) under the package “`com.library.listener`” and overriding the method “`doReceive`”. Insert this code in the method body. We are giving the “`groupId`” as “`classPK`”. The reason being these subscriptions is for the entire library and not on one particular library book.

```
System.out.println("inside doReceive method....");

LMSBook lmsBook = (LMSBook) message.getPayload();

long companyId = lmsBook.getCompanyId();
long classPK = lmsBook.getGroupId();

List<Subscription> librarySubscriptions =
    SubscriptionLocalServiceUtil.getSubscriptions(
        companyId, LMSBook.class.getName(), classPK);

for (Subscription subscription: librarySubscriptions) {
    User user = UserLocalServiceUtil.fetchUser(
        subscription.getUserId());

    // send notification to the user.
}
```

Step 4: This is the last step but a very important step. Without this step the other steps are of no use. You have to register your new “`messaging-spring.xml`” file into the portal context, to make the portal aware of the new message bus. Open your plugin’s “`web.xml`” file and insert this block at the end, just before the closing “`</web-app>`”.

```
<context-param>
    <param-name>portalContextConfigLocation</param-name>
    <param-value>
        /WEB-INF/classes/META-INF/messaging-spring.xml
    </param-value>
</context-param>
```

You're now going to test this new feature. You may have to manually deploy the plugin for the changes to take effect fully. Add a new book into our Library and take it through workflow. When you finally "Approve" the book, you should see the message "inside doReceive method...." appearing on the server console. This confirms that the MessageBus is in action now and the listener is listening to the destination.

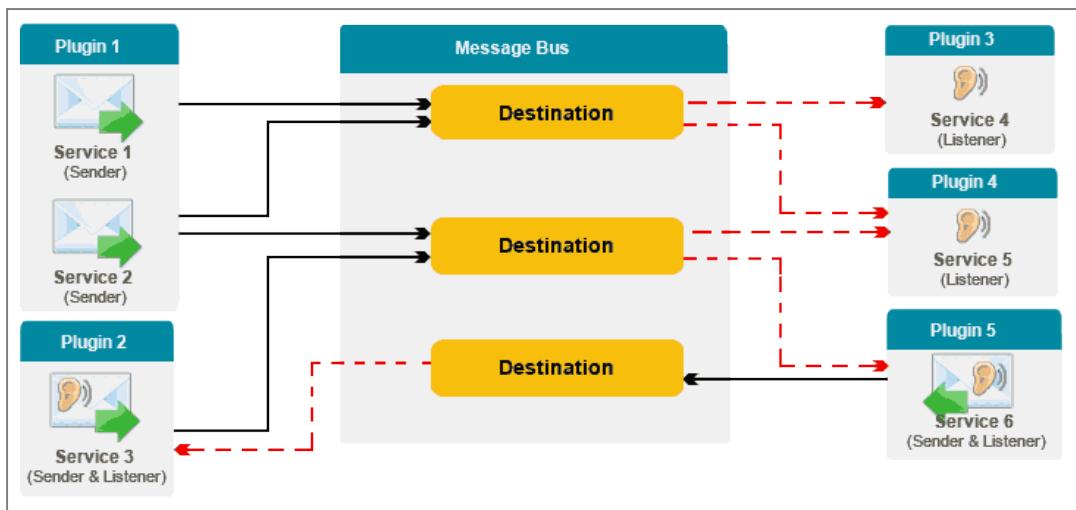
14.1.2 MessageBus Usage Situations

As we have already seen in the introduction of this section, the Message Bus is a service level API used to exchange messages within Liferay. The Message Bus exchanges messages containing Strings and Objects, providing loose coupling between message producers and consumers. For this reason, you won't have class loading issues. The Message Bus is located in portal-kernel, in the global class loader, making it accessible to every deployed webapp. Common uses for Message Bus include:

- Sending search index write events
- Sending subscription emails
- Handling messages at scheduler endpoints
- Running asynchronous processes

Your services can send messages to one or more destinations. And your services can listen to one or more destinations. An individual service can be both a message sender and a message listener. Messages can be shared within the two services of the same plugging, or between the services of two different plugins or between a plugin and the portal (ROOT) context. From a plugin you can send message to a destination that is already defined in "**messaging-spring-core.xml**". The same rule applies to listening to a destination already defined there.

In essence, the possibilities are endless. You can send messages between any two entities that are currently part of the portal server. The figure below depicts services sending messages to one or more destinations and services listening to one or more destinations.



Messaging to the MessageBus can be of two types. We need to choose one of them based on our requirement. In the first case, we may require the response from the Message Bus instantly the second is a “**send-and-forget**” category. Both of them are explained below.

1) Synchronous messaging – After sending a message, the sender blocks waiting for a response from a recipient to move forward. This kind of messaging is very similar to our RPC based web services. The only difference is the messaging is brokered / orchestrated by the Message Bus infrastructure.

2) Asynchronous messaging – After sending a message, the sender is free to continue processing. The sender can be configured to receive a callback or can simply “send and forget.” We’ll cover both synchronous and asynchronous messaging implementation details in the next sub-section.

- **Callback** – The sender can include a callback destination key as the response destination for the message. The recipient (listener) can then send a response message back to the sender via this response destination.
- **Send-and-Forget** – The sender includes no callback information in the message sent and simply continues with processing. The one we have implemented for our Library subscription belongs to this category.

Message Types include using either Message or JSONObject classes. Within Liferay core services, we typically serialize and deserialize in JSON. Messages can be delivered either **serially** or in **Parallel** destination. In the example we just did, we have used [ParallelDestination](#).

Code Changes @ <http://code.google.com/p/lr-book/source/detail?r=83>

Congratulations, you have learnt one of the powerful API’s of Liferay that helps in seamlessly sending and receiving messages between components. This is a lightweight equivalent of a full-blown JMS system like [ActiveMQ](#). In lieu of Liferay’s MessageBus, if you want to use some other Messaging System, you can do that without any limitations and nothing is going to stop you. But Liferay MessageBus is really very powerful.

14.2 Indexing and Search

We are stepping into another very interesting and exciting topic – Indexing and Search. So far, we have heard about indexing at the database level. A database index is a data structure that improves the speed of data retrieval operations on a database table at the cost of slower writes and less storage space. Indices can be created using one or more columns of a database table, providing the basis for both rapid random lookups and efficient access of ordered records. But what about the indexing at the application level? Lucene is the answer.

Liferay has used Apache Lucene to do the indexing and searching of data at the application level. Apache Lucene is a high-performance, full-featured text search engine library written entirely in Java. It is a technology suitable for nearly any application that requires full-text search, especially cross-platform. Apache Lucene is an open source project freely downloadable. But Lucene is not tightly coupled with Liferay. We can replace Lucene with any other component that does indexing and searching, for example Solr. For more details on Lucene library, you can visit <http://lucene.apache.org/core/>.

14.2.1 Enabling the Indexer Class

In this sub-section, we'll see how to associate an Indexer class for our Portlet that will help in indexing and searching of the Portlet's data in an efficient manner. The process consists of just two steps to begin with. The later sub-section of this section will go into various other details of indexing and searching.

Step 1: Open “**liferay-portlet.xml**” and put this entry for a new indexer class. This entry should just go before the “`<scheduler-entry>`” entry.

```
<indexer-class>com.library.search.LibraryIndexer</indexer-class>
```

Step 2: The Portlet will complain during its deployment till we write an actual indexer class under the specified package and over-ride some of the basic methods of the base indexer class. Create a new class “**LibraryIndexer.java**” inside package “`com.library.search`”. The new class should extend `BaseIndexer` and override some basic methods as shown below. There are total of nine methods – two of them public and seven protected.

```
package com.library.search;

import java.util.Locale;
import javax.portlet.PortletURL;
import com.liferay.portal.kernel.search.BaseIndexer;
import com.liferay.portal.kernel.search.Document;
import com.liferay.portal.kernel.search.SearchContext;
import com.liferay.portal.kernel.search.Summary;

public class LibraryIndexer extends BaseIndexer {

    public String[] getClassNames() {
        return null;
    }
}
```

```

    }

    public String getPortletId() {
        return null;
    }

    protected void doDelete(Object arg0) throws Exception {
        // empty
    }

    protected Document doGetDocument(Object obj)
        throws Exception {
        return null;
    }

    protected Summary doGetSummary(Document document,
        Locale locale, String snippet, PortletURL portletURL)
        throws Exception {
        return null;
    }

    protected void doReindex(Object obj) throws Exception {
        // empty
    }

    protected void doReindex(String[] args) throws Exception {
        // empty
    }

    protected void doReindex(String className, long classPK)
        throws Exception {
        //empty
    }

    protected String getPortletId(SearchContext searchContext) {
        return null;
    }
}

```

Let's go one method at a time starting with "getclassNames". Modify it as below.

```

public String[] getClassNames() {
    String[] CLASS_NAMES = {LMSBook.class.getName()};
    return CLASS_NAMES;
}

```

In the "getPortletId" method return the actual portletId for our Library Portlet.

```
return "library_WAR_libraryportlet";
```

Put some meaningful SOP statements inside all other methods. After doing these changes, deploy the Portlet manually and let's verify the Portlet is loading the new indexer class. Login to the portal as Omni Administrator and go to Control Panel → Server Section → Plugins Installation. You will see the list of plugins that are currently installed. Navigate to the page that contains our Library Portlet. It should have a new button "**ReIndex**". Try clicking this button and find out which method of our newly written Indexer class is being invoked.

Language Package: Unknown	Yes	
Library Portlet Package: Unknown	Yes	Reindex
License Manager Package: Unknown	Yes	

14.2.2 Indexing during Book insertion

Our next step is to do proper indexing as and when a book gets added to our Library, so that when members search, the books can be searched directly from the index than querying the database. This kind of searching from the index is going to drastically improve the performance. Where to invoke this code? Ideally it should be inside the method of our DTO class where we insert the book into the database through the persistence layer. Immediately after the book is persisted you can invoke the call to index the book into the file system. You can also opt to write this code inside your WorkflowHandler class “after” the book is officially reviewed and approved for inclusion in our Library. Let’s go with the first approach just to explain the point.

Step 1: Open your DTO class “**LMSBookLocalServiceImpl.java**” and insert the below code at the end of “`insertBook`” just before the “`return`” statement.

```
// indexer
Indexer indexer =
    IndexerRegistryUtil.getIndexer(LMSBook.class);
try {
    indexer.reindex(lmsBook);
} catch (SearchException e) {
    e.printStackTrace();
}
```

Make the additional imports that are required.

```
import com.liferay.portal.kernel.search.Indexer;
import com.liferay.portal.kernel.search.IndexerRegistryUtil;
```

Save the changes and try adding a book into our Library through the Portlet. You will observe from the server console that the “`doReindex`” method of our indexer class is invoked twice.

```
inside doReindex(Object obj)
inside doReindex(Object obj)
```

Why this is called twice? Let’s fix this issue before we proceed with putting some real logic inside this method to do the indexing of the book data. A closer analysis will reveal that the service layer will automatically do the indexing as soon as an Indexer is attached with a Portlet. The call is invoked twice because of these statements in the same method “`insertBook`”.

```
1 lmsBook = LMSBookLocalServiceUtil.addLMSBook(lmsBook);
```

```
2 | indexer.reindex(lmsBook);
```

Now you have two options. It is upto you to take up one of them.

Option 1: Remove the block that contains the second statement. This option does not required to make use of [IndexerRegistryUtil](#) inside our DTO class.

Option 2: Replace the first statement with a direct call to the persistence layer. From our DTO class it is always preferred to directly invoke the persistenceLayer.

```
lmsBook = lmsBookPersistence.update(lmsBook, false);
```

Step 2: Now we'll write the logic inside our “doReindex” method.

```
protected void doReindex(Object obj) throws Exception {
    LMSBook lmsBook = (LMSBook) obj;
    Document document = getDocument(lmsBook);
    SearchEngineUtil.updateDocument(getSearchEngineId(),
        lmsBook.getCompanyId(), document);
}
```

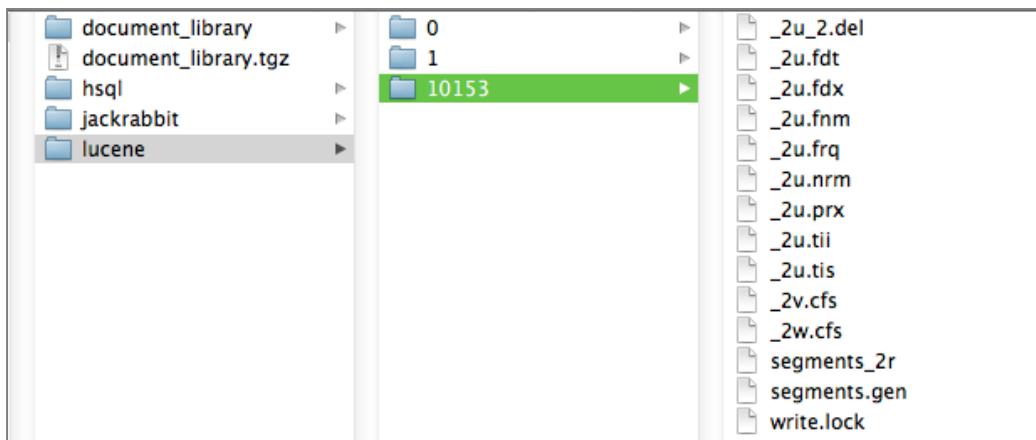
The “getDocument” will invoke the “doGetDocument” of our Indexer class. Let's write the logic for this method as well.

```
protected Document doGetDocument(Object obj) throws Exception {
    LMSBook lmsBook = (LMSBook) obj;
    Document document =
        getBaseModelDocument(getPortletId(), lmsBook);
    return document;
}
```

Save all changes and try to insert book into our Library. Our Indexer class should now start indexing the book information properly. But how do you confirm this fact? All the indexes go into your server's file system by default unless configured otherwise. The following two entries in “**portal.properties**” control this aspect.

```
lucene.store.type=file
lucene.dir=${liferay.home}/data/lucene/
```

Let's go and check this folder under our {LIFERAY_HOME}. You will find the indexes there. Every time you add a book, the number of files inside this folder will go up by one. The folder “**10153**” is the companyId. It may be different in your case.



Before moving to the next topic, let's examine the contents of these index files. Unfortunately, if you open any of these in your favorite text editor, you will not understand anything. The contents will be very cryptic. As going into the contents of these files is not any use for a human being like you and me, let's see programmatically what a "document" contains. Insert a SOP statement inside our "doGetDocument" method and then try adding a book.

```
System.out.println("document ==> " + document);
```

In the server console, you will see the following information displayed in one line. I've broken that up for the sake of readability and understanding.

Field of the "document" object	Constant in "Field.java"
entryClassName=[com.slayer.model.LMSBook]	ENTRY_CLASS_NAME
uid=[library_WAR_libraryportlet_PORTLET_160]	UID
scopeGroupId=[10179]	SCOPE_GROUP_ID
groupId=[10179]	GROUP_ID
assetCategoryIds=[]	ASSET_CATEGORY_IDS
status=[1]	STATUS
userId=[10195]	USER_ID
userName=[test test]	USER_NAME
companyId=[10153]	COMPANY_ID
createDate=[20130306084735]	CREATE_DATE
portletId=[library_WAR_libraryportlet]	PORTLET_ID
entryClassPK=[160]	CLASS_PK
assetTagNames=[]	ASSET_TAG_NAMES

This conveys that the index comprises of "documents". A document is a representation of an entity, e.g. "LMSBook" in the file system in a serialized format. A document consists of fields. A "field" could be a "keyword" or a "text". The documents in the index are "searchable" with the help of these fields, especially "keywords". In the next sub-section we are going to write the new search functionality with the help of the new keywords that we want to introduce. Before we move on let's quickly read out the summary of what we have seen so far.

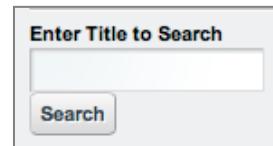
- An **index** contains a sequence of **documents**

- A **document** is a sequence of **fields**
- A **field** is a named sequence of **expressions**
- An **expression** is a **string** (string)

14.2.3 Writing new Search Functionality

Till the last sub-section we have seen just the indexing part, only 50% of our real goal. The real goal is to search the entities based on its indexes. Without putting this into place, the real objective of indexing will not be met. First and foremost we have to decide on what fields the indexed entities would get searched. This is the most important thing before we write the search functionality. Based on these fields, we have to alter the way the original indexing of the entity is done.

Let's go back to our simple search for Library books that is appearing in the default view of our Portlet. This search is invoking our original database search. We'll change this to invoke our new search that searches the index.



Step 1: We'd like the new search to search the books either by “bookTitle” or “author” fields. But unfortunately, when the books got indexed and stored as documents in the file system, these fields were not captured and the indexing done on them. And obviously for this use case, we don't want to search the books based on the default set of fields on which the book data got indexed. How to add these fields (keywords) during the time of indexing a book? Go back to our “doGetDocument” of the Indexer class and insert these lines just before returning the “document”.

```
document.addKeyword(Field.TITLE, lmsBook.getBookTitle());
document.addKeyword("author", lmsBook.getAuthor());
document.addText(Field.TITLE, lmsBook.getBookTitle());
```

Step 2: Yes, try adding new books into our Library. They'll get indexed with these additional fields. But what about the old books we have already added? We have to go to the Control Panel and click the “ReIndex” button. When you click this button, it is just going to invoke the method “doReindex(String[] args)” of Indexer class, but unfortunately nothing is going to happen as we have not written any logic to do the re-indexing of the whole set of books in the Library. Let's write logic to re-index all the books in the Library. The first element of the String array “args” contains the current companyId. Re-write this method as below and then click “ReIndex” as admin.

```
protected void doReindex(String[] args) throws Exception {
    long companyId = Long.valueOf(args[0]);
    DynamicQuery dynamicQuery =
        DynamicQueryFactoryUtil.forClass(LMSBook.class);
    dynamicQuery.add(
        RestrictionsFactoryUtil.eq("companyId", companyId));

    List<LMSBook> books =
        LMSBookLocalServiceUtil.dynamicQuery(dynamicQuery);

    for (LMSBook book: books) {
```

```

        if (book.getGroupId() > 0) reindex(book);
    }
}

```

Now in the filesystem, you should see all the books properly indexed with the new fields. Hurray!! You just have one more step to do a real search on these indexes and fetch the relevant results.

Step 3: This step is to make us work on our application to get everything hooked properly. In this step we'll introduce two new API's in our DTO class, “**LMSBookLocalServiceImpl.java**” that will search the lucene index and fetch the results. Let's write this new method in this class and re-run the Service Builder for the changes to percolate to the entire service layer.

a) The first API will search indexes and return “Hits” and will have three steps.

1. Preparing a Search Context
2. Preparing a Query to search
3. Firing the query to get hits

```

public Hits getHits(String keyword, long companyId, long groupId) {
    // 1. Preparing a Search Context
    SearchContext searchContext = new SearchContext();
    searchContext.setCompanyId(companyId);

    String[] CLASS_NAMES = { LMSBook.class.getName() };
    searchContext.setEntryClassNames(CLASS_NAMES);

    long[] groupIds = {groupId};
    searchContext.setGroupIds(groupIds);

    // 2. Preparing a Query to search
    BooleanQuery searchQuery =
        BooleanQueryFactoryUtil.create(searchContext);
    String[] terms = {Field.TITLE, "author"};

    try {
        searchQuery.addTerms(terms, keyword);
    } catch (ParseException e) {
        e.printStackTrace();
    }

    // 3. Firing the query to get hits
    Hits hits = null;
    try {
        hits = SearchEngineUtil.search(
            searchContext, searchQuery);
    } catch (SearchException e) {
        e.printStackTrace();
    }
    return hits;
}

```

b) The second API will make a call to the first API and wrap the results as LMSBook objects and return the results. It has just two steps as shown below.

```

public List<LMSBook> searchIndex()

```

```

        String keyword, long companyId, long groupId)
        throws SystemException {

    Hits hits = getHits(keyword, companyId, groupId);

    // 1. return null if no results
    if (Validator.isNull(hits) || hits.getLength() == 0)
        return null;

    // 2. Convert results into a List of LMSBook objects
    List<LMSBook> books = new ArrayList<LMSBook>();
    for (Document document : hits.getDocs()) {
        long bookId = GetterUtil.getLong(
            document.get(Field.ENTRY_CLASS_PK));
        LMSBook book = fetchLMSBook(bookId);
        books.add(book);
    }

    return books;
}

```

Step 4: One “searchIndex” API is ready. All we have to do now is to invoke this new API from our “searchBooks” method of the Portlet class. Open the “**LibraryPortlet.java**” and change the call to the search API to this new call.

```

List<LMSBook> lmsBooks = LMSBookLocalServiceUtil
    .searchIndex(searchTerm,
        themeDisplay.getCompanyId(), themeDisplay.getScopeGroupId());

```

You’re finally done. Go back to the Portlet in the browser and search the books, either by bookTitle or by author name. You should get the proper results filtered by these two fields. Congratulations! You have successfully implemented the first level of indexing the data and then searching the data inorder to fetch the actual results.

Just one small thing, I’d like to mention here. In step 2 of the “searchIndex” method we have written in the DTO class, we are fetching the actual “**ImsBook**” object from the database / cache. Instead of doing this, you can instantiate a new object of this type and populate its field with the values that are there in the “document” object and put into the list. This will further improve the performance. But the only caveat is this minified “**ImsBook**” object may not contain all the data you may want to show in the book details page. But it should be definitely sufficient to display the results in the results page.

```

LMSBook book = new LMSBookImpl();
book.setBookId(bookId);
book.setBookTitle(document.get(Field.TITLE));
book.setUserId(Long.valueOf(document.get(Field.USER_ID)));

```

Code Changes @ <http://code.google.com/p/lr-book/source/detail?r=84>

14.3 Advanced Search Features

Lucene provides very rich and diverse features around searching the indexed contents. Once we properly indexed the data, it is upto the application programmer to decide how to search the index. Complex search queries can be formed and run against the indexes in order to get the search results called as “Hits”. The following are the various search scenarios that are possible with Lucene Search.

- Ranked searching – best results returned first
- Many powerful query types: phrase queries, wildcard queries, proximity queries, range queries and more
- Fielded searching (e.g. title, author, contents)
- Sorting of search results by any field
- Multiple-index searching with merged results
- Allows simultaneous update and searching
- Flexible facetting, highlighting, joins and result grouping
- Fast, memory-efficient and typo-tolerant suggesters
- Pluggable ranking models, including the Vector Space Model and Okapi BM25
- Configurable storage engine (codecs)

In the next sub-section we are going to see another Fielded search, this time searching the books by both “bookTitle” and “author”.

14.3.1 Search with multiple fields

Imagine there is a form to search the book on multiple fields. In the previous section, we have seen how to search both on the fields “**bookTitle**” and “**author**”. In this example we are going to see how to separately search on these fields. Open our Portlet’s “**view.jsp**” and insert the following form in its appropriate place. After inserting this code, you will get the form as shown on the right.



```
<portlet:actionURL var="searchAdvancedURL"
    name="searchAdvanced" />

<aui:form action="<% = searchAdvancedURL %>">
    <aui:select name="searchType"
        label="Search Type" inlineLabel="true">
        <aui:option value="<% = false %>" label="Any"/>
        <aui:option value="<% = true %>" label="All"/>
    </aui:select>
    <aui:input name="bookTitle" />
    <aui:input name="author" />
    <aui:button type="submit" value="Search" />
</aui:form>
```

The next step is to write three methods in our DTO class “**LMSBookLocalServiceImpl**” that will search the Lucene index based on the search criteria selected by the user. The first method will set a criterion in the `searchQuery`.

```

private void appendSearchTerm(String field, String value,
    boolean isAndSearch, BooleanQuery searchQuery) {

    if (Validator.isNotNull(value)) {
        if (isAndSearch) {
            searchQuery.addRequiredTerm(field, value, true);
        } else {
            try {
                searchQuery.addTerm(field, value, true);
            } catch (ParseException e) {
                e.printStackTrace();
            }
        }
    }
}

```

The second method returns the Hits. The third method will wrap these hits in the form of LMSBook objects and return to the Portlet class.

```

protected Hits getHits(
    long companyId, long groupId, String bookTitle,
    String author, boolean andSearch)
    throws SystemException {

    // 1. Preparing the Search Context
    SearchContext searchContext = new SearchContext();
    searchContext.setAndSearch(andSearch);
    searchContext.setCompanyId(companyId);
    long[] groupIds = {groupId};
    searchContext.setGroupIds(groupIds);

    BooleanQuery searchQuery =
        BooleanQueryFactoryUtil.create(searchContext);
    appendSearchTerm(Field.TITLE, bookTitle,
        searchContext.isAndSearch(), searchQuery);
    appendSearchTerm("author", author,
        searchContext.isAndSearch(), searchQuery);

    // 2. Firing the query and getting the hits
    Hits hits = null;
    try {
        hits = SearchEngineUtil.search(
            searchContext, searchQuery);
    } catch (SearchException e) {
        e.printStackTrace();
    }

    return hits;
}

```

The third method that returns the results will be like this. After introducing the third method rerun the Service Builder.

```

public List<LMSBook> advancedSearch(long companyId, long groupId,
    String bookTitle, String author, boolean andSearch) {

    Hits hits = null;
    try {

```

```

        hits = getHits(
            companyId, groupId, bookTitle, author, andSearch);
    } catch (SystemException e) {
        e.printStackTrace();
    }
    if (hits == null || hits.getLength() == 0) return null;

    List<LMSBook> books = new ArrayList<LMSBook>();
    for (int i=0; i<hits.getLength(); i++) {
        Document doc = hits.doc(i);
        long bookId = GetterUtil.getLong(
            doc.get(Field.ENTRY_CLASS_PK));
        try {
            LMSBook book = fetchLMSBook(bookId);
            books.add(book);
        } catch (SystemException e) {
            e.printStackTrace();
        }
    }
    return books;
}

```

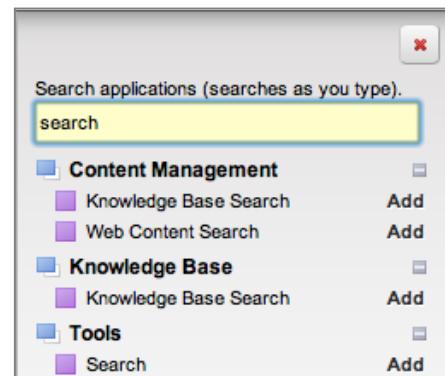
Now, I leave things to you to write the appropriate method in our Portlet class “**LibraryPortlet.java**” to call the method of our DTO class and show the results accordingly. You can extend the same example to search with more fields. Whatever search can be performed on a database, can be done on an index as well. In the next sub-section we are going to see the Faceted Search and how Liferay supports the concept of faceted search.

Code Changes @ <http://code.google.com/p/lr-book/source/detail?r=89>

14.3.2 Faceted Search

Searching is an important aspect of any portal. Users should able to search and locate the relevant contents they want without any hassles. Liferay provides three important tools to search contents and the assets of the portal. They are:

1. **Knowledge Base Search** – searches the knowledge base articles.
2. **Web Contents Search** – searches the web contents
3. **Faceted Search** – searches all portal assets



The objective of this sub-section is to quickly show you how to register our custom asset (Library Book) into the faceted system of Liferay, so that anyone who comes and makes a general search, our Library books that satisfy the search criteria should also appear as part of the search results. Create a new page by name “Search” and drag & drop the Faceted Search Portlet under “Tools” category. Once the Portlet is in the page, click on its configuration to make some changes.

In the “Basic” Display Settings, you see four check boxes. They are primarily to control the display of the results out of a Faceted Search. In the Other Settings section, you have four options. Two of them are meant for testing the search behavior and forth, Display Open Search Results is to display the results in Open Search format. This we’ll see in detail in the next sub-section.

Now coming to the actual stuff, click on the “Advanced” radio button where you can configure a list of assets that get searched when the user invokes a faceted search. The only prerequisite is that the asset in question should have an indexer properly defined and configured. When you click this option, you see the full configuration in the JSON format. The entities / assets that get searched are listed too.

Display Settings

Basic
 Advanced

Display Asset Type Facet
 Display Asset Tags Facet
 Display Asset Categories Facet
 Display Modified Range Facet

Other Settings

Display Results in Document Form

View in Context

Display Main Query

Display Open Search Results

```
"data": {
    "values": [
        "com.liferay.portlet.bookmarks.model.BookmarksEntry",
        "com.liferay.portlet.blogs.model.BlogsEntry",
        "com.liferay.portlet.calendar.model.CalEvent",
        "com.liferay.portlet.messageboards.model.MBMessage",
        "com.liferay.portlet.wiki.model.WikiPage",
        "com.liferay.portal.model.User",
        "com.slayer.model.LMSBook"
    ],
    "frequencyThreshold": 1
},
```

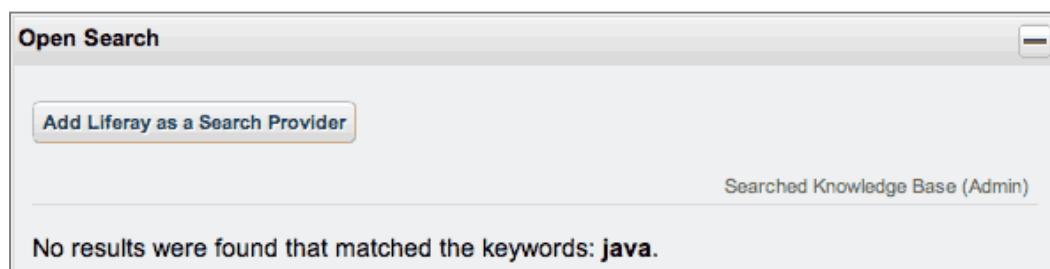
To the existing list of entities, add our custom entity / asset as well, as highlighted in a different color above. Save the changes and trigger a search by giving the book name or author name of some of the books in our Library. The only other requirement for the books to appear in the faceted search results apart from properly getting indexed is to have an entry in the “assetentry” table and having an “asset” status. Otherwise the results are not getting properly displayed. The following is the result of a search that I’ve performed for a “Java” book in our Library. You will notice the keywords that are used to search the assets are highlighted in yellow, helping the end-user to quickly identify the relevant contents what he/she is searching.



It is worth spending some time to know a little about faceted search with the help of this Wiki Article, http://en.wikipedia.org/wiki/Faceted_search. The blog written by Ray Auge is also a great source of information [<http://bit.ly/WrW1eH>].

14.3.3 Open Search Integration

One of the options that you will see in the configuration of the Faceted Search Portlet is “Display Open Search Results”. What does it mean? From the context of the faceted Portlet it means that this Portlet will show results from third party Open Search plugins. Let’s check this option and save the configuration. Go back to the faceted search Portlet and search for your favorite book. The results will show up in the normal search results. But unfortunately the books will not show up in the new “Open Search” Portlet that appears below the Faceted Search results Portlet as shown here.



Now our first task is to make our book appear as part of the Open Search results. Post this we’ll see some theoretical background about Open Search and how Liferay can serve as an Open Search Provider. The first exercise comprises of three steps. The only prerequisite before doing this exercise is to have an Indexer class properly configured for our Portlet.

Step 1: Create a new class for open search integration, “**LibraryOpenSearchImpl**” inside the package “**com.library.search**”. This new open search class should extend **HitsOpenSearchImpl**. Let’s begin with overriding three basic methods of the base class by putting some contents inside the bodies of those methods.

```
package com.library.search;

import com.liferay.portal.kernel.search.HitsOpenSearchImpl;

public class LibraryOpenSearchImpl extends HitsOpenSearchImpl {

    public String getPortletId() {
        return "library_WAR_libraryportlet";
    }

    public String getSearchPath() {
        return "/c/library/open_search";
    }

    public String getTitle(String keywords) {
        return "Library Open Search for " + keywords;
    }
}
```

Step 2: Make an entry for this new class in “**liferay-portlet.xml**”. This entry should come immediate after the “`<indexer-class>`” entry.

```
<open-search-class>
    com.library.search.LibraryOpenSearchImpl
</open-search-class>
```

After these changes are deployed, you will see the new title, “Library Portlet” along with the text “Searched Knowledge Base (Admin)” in the Open Search Portlet. This confirms that the Open Search class has been deployed properly and got hooked with our Portlet. But you will see some error messages in the server console. We need to do couple of things in our portlet’s indexer class to get rid of these exceptions and allow the open search Portlet to show the proper results.

```
inside search method....
11:28:27,115 WARN [http-bio-8080-exec-11][HitsOpenSearchImpl:47]
class com.library.search.LibraryOpenSearchImpl does not implement
getIndexer()
inside doGetSummary
11:28:27,143 ERROR [http-bio-8080-exec-11][search_jsp:1873] Error
displaying content of type com.library.search.LibraryOpenSearchImpl
com.liferay.portal.kernel.search.SearchException:
java.lang.NullPointerException
atcom.liferay.portal.kernel.search.HitsOpenSearchImpl.search(HitsOpen
SearchImpl.java:212)
```

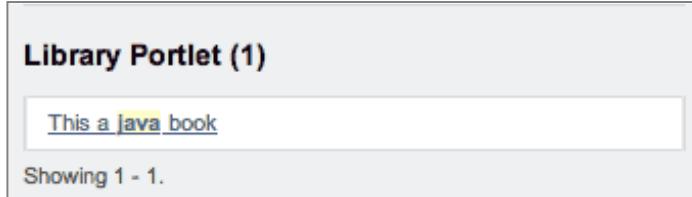
Step 3: Open our Indexer class, “**LibraryIndexer.java**” and override the “`doGetSummary`” method as shown below. This is required for the Open Search to show the results properly in its standard format.

```
protected Summary doGetSummary(Document document, Locale local,
    String snippet, PortletURL portletURL) throws Exception {
    return new Summary(
        document.get(Field.TITLE), snippet, portletURL);
}
```

The other requirement is to insert the following three lines inside the “`doGetDocument`” of our Indexer class, just before the return statement. This is to avoid a parsing exception happening in the “`search`” method of `HitsOpenSearchImpl`.

```
Date modifiedDate = lmsBook.getModifiedDate();
if (modifiedDate == null) modifiedDate = new Date();
document.addDate(Field.MODIFIED_DATE, modifiedDate);
```

Save all your changes and make a search. The Open Search Portlet should now show the results in its format as shown here. Congratulations! This is one of the fantastic jobs you have done so far. Now it is time to know little bit about OpenSearch and How Liferay can be an OpenSearch provider. Visit opensearch.org to know about OpenSearch results format.



Code Changes @ <http://code.google.com/p/lr-book/source/detail?r=86>

14.3.4 Fixing the URL Issue

After doing the search, you get the search results both the main search results and under the Open Search section. Now click on the links appearing on the bookTitle. The first one takes you to an URL, <http://localhost:8080/web/guest/null> and the second one takes you to the default Portlet view of our Library Portlet instead of showing the details view of the Portlet. In this sub-section, we are going to quickly fix these two issues, so that when an user clicks on these links, it should take him to the right place.

First Link: Open your “**LMSBookAssetRenderer.java**” and override the public “`getURLViewInContext`” method as shown here.

```
public String getURLViewInContext(
    LiferayPortletRequest liferayPortletRequest,
    LiferayPortletResponse liferayPortletResponse,
    String noSuchEntryRedirect) throws Exception {
    return noSuchEntryRedirect;
}
```

Second Link: Open your “**LibraryOpenSearchImpl.java**” and override the protected “`getURL`” method as shown here.

```
protected String getURL(ThemeDisplay themeDisplay, long groupId,
    Document result, PortletURL portletURL) throws Exception {
    portletURL.setParameter("jspPage",
        LibraryConstants.PAGE_DETAILS);
    long bookId = GetterUtil.getLong(
        result.get(Field.ENTRY_CLASS_PK));
    portletURL.setParameter("bookId", String.valueOf(bookId));

    return super.getURL(
        themeDisplay, groupId, result, portletURL);
}
```

Save the changes and confirm that the links are working fine now.

Code Changes @ <http://code.google.com/p/lr-book/source/detail?r=92>

14.4 Device Detection API

During the last decade once a web application is developed we used to rigoursly test the application in all major web broswers starting from Internet Explorer 6 upto the latest version of Firefox and Safari, primarily to ensure that the application that we developed works well in all browsers. If, in case, some feature is not working on a particular browser, then we write some different code to make it work on that browser using some alternate code. Inside the java or JSP code the type of brower used by the client is usually obtained with this code, where “request” is an object of HttpServletRequest.

```
String userAgent = request.getHeader("user-agent");
```

But things have really changed in the last few years. With the advent of latest web and JavaScript frameworks and standard tag libraries the need for browser compatibility tests are gradually diminishing. These frameworks and libraries are developed keeping these browser compatibility issues in mind relieving the developers of this dirty job. With this good news on one side, there is a bad news as well. With the exponential increase in the Internet usage and advent of many new devices apart from the conventional PC based web browsers, there is an extremely high demand for testing our applications against all these devices. How to make your portal content and portlets appear inside the small screens of these hand-held devices and smart phone, exactly the same way they get displayed on web browsers? The answer to this million dollars question is Liferay’s Device Detection API.

14.4.1 Insalling the Device Detection plugin

The Device Detection API is used for detecting the capabilities of a device that is making a request to your portal. In addition, the Device Detection API allows Liferay to detect which mobile device or operating system is being used for any given request and alters the rendering of pages based on the detected device. To install this feature, you will need to install the Device Recognition Provider app from Liferay Marketplace. Based on your Liferay edition, you can select the appropriate link for more info and download information. For Community Edition of Liferay the link to download this plugin is given below. There is another version for Enterprise Edition.
<http://www.liferay.com/marketplace/-/mp/application/15193341>.

The Device Recognition plugin, which is bundled inside the Device Recognition Provider app, uses a device database called WURFL to determine the capabilities of your device. You can visit their site for more information at <http://wurfl.sourceforge.net/>. You could create your own plugin to use your own device's database. Let's go through some simple ways to use the Device Detection API and its capabilities.

14.4.2 Using the Device API

We'll go over a couple of code snippets that will help you get started. The object Device can be obtained from the themeDisplay object like this:

```
Device device = themeDisplay.getDevice();
```

For reference, you can view the API in the Device Javadocs. Using some of the methods from the javadocs, here is an example that obtains the dimensions of a particular device:

```
Dimensions dimensions = device.getScreenSize();
float height = dimensions.getHeight();
float width = dimensions.getWidth();
```

The required imports are:

```
import com.liferay.portal.kernel.mobile.device.Device;
import com.liferay.portal.kernel.mobile.device.Dimensions;
```

Now, your device can obtain the Device object and can obtain the dimensions of a device. Of course, you can acquire many other values that take care of those trivial problems that arise when sending content to different devices. Simply refer to the previously mentioned Device javadocs for assistance. Let's go through some device capabilities in the next sub-section.

14.4.3 Device Capabilities

Most of the capabilities of a device can be detected, but this depends on the device detection implementation you're using. For the Device Recognition plugin, you can

view its device database's (WURFL) list of capabilities here [<http://bit.ly/Zepdon>]. For an example, you can obtain the capability of a brand name by using this code:

```
//String brandName = device.getBrand();  
String brandName = device.getCapability("brand_name");
```

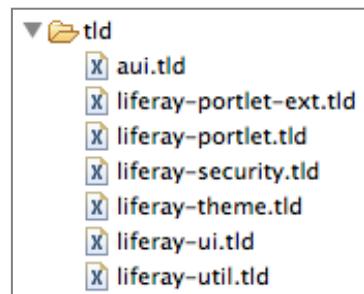
Furthermore, you can grab many other values such as “`model_name`”, “`marketing_name`”, “`release_date`”, etc. Also, there are Boolean values that can be acquired that include: `is_wireless_device`, `is_tablet`, etc. Keeping the capabilities list in mind when configuring your device is very helpful. You're able to detect the capabilities of a device making a request to your portal by using the Device Detection API. Overall, this is an extremely powerful API that helps you to write your applications in such a way that they behave properly in all the devices.

14.5 Creating a Custom Taglib

Custom tags are user-defined JSP language elements that encapsulate recurring tasks. Custom tags are distributed in a tag library, which defines a set of related custom tags and contains the objects that implement the tags. In the whole of this book, we have seen many instances, where we have used many tags that come by default in Liferay. The following are some of the tag libraries used by the portal. Apart from this list, there are also a bunch of tag libraries provided by Struts framework as majority of the Portlet that come bundled with Liferay are based on Struts MVC framework. Actually you will find these entries inside the “`init.jsp`” under “`{PORTAL_SRC}/portal-web/html/common`”.

No.	Taglib URI	Prefix	Type
1	<code>http://java.sun.com/jsp/jstl/core</code>	<code>c</code>	JSTL
2	<code>http://java.sun.com/jsp/jstl/fmt</code>	<code>fmt</code>	JSTL
3	<code>http://java.sun.com/jsp/jstl/functions</code>	<code>fn</code>	JSTL
4	<code>http://java.sun.com/jsp/jstl/sql</code>	<code>sql</code>	JSTL
5	<code>http://java.sun.com/jsp/jstl/xml</code>	<code>x</code>	JSTL
6	<code>http://java.sun.com/portlet_2_0</code>	<code>portlet</code>	JSR-286
7	<code>http://liferay.com/tld/aui</code>	<code>aui</code>	Liferay
8	<code>http://liferay.com/tld/portlet</code>	<code>liferay-portlet</code>	Liferay
9	<code>http://liferay.com/tld/security</code>	<code>liferay-security</code>	Liferay
10	<code>http://liferay.com/tld/theme</code>	<code>liferay-theme</code>	Liferay
11	<code>http://liferay.com/tld/ui</code>	<code>liferay-ui</code>	Liferay
12	<code>http://liferay.com/tld/util</code>	<code>liferay-util</code>	Liferay

Any new plugin project that we create out of Liferay's Plugins SDK / Eclipse IDE has the support for the tag libraries that are shown on your right. But we have to inject the taglib that we want to use through our “`init.jsp`” as we have done in the case of our Library Portlet. In the deployed app's “`web.xml`” file, you will find the entries for all the taglibs that are used by our Portlet. The below entries are from “`init.jsp`”.



```

<%@taglib uri="http://java.sun.com/portlet_2_0" prefix="portlet" %>
<%@taglib uri="http://liferay.com/tld/aui" prefix="aui" %>
<%@taglib uri="http://liferay.com/tld/ui" prefix="liferay-ui" %>
<%@taglib uri="http://java.sun.com/jsp/jstl/fmt" prefix="fmt" %>
<%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@taglib uri="http://liferay.com/tld/theme" prefix="liferay-theme" %>
<%@taglib uri="http://liferay.com/tld/portlet"
prefix="liferay-portlet" %>
<%@taglib uri="http://liferay.com/tld/library" prefix="library-ui" %>

```

Ok, have you ever wondered where the code required for these tags are lying inside the PORTAL_SRC? Let me give a quick snapshot, so that you can have a look inside the PORTAL_SRC to confirm this fact yourself.

JSP Files	TLD Files	JAVA Files
 taglib <ul style="list-style-type: none">  aui  portlet  theme  ui <ul style="list-style-type: none">  init-ext.jsp  init.jsp  taglib-init-ext.jspf  taglib-init.jsp 	 META-INF <ul style="list-style-type: none">  aui.tld  faces-config.xml  liferay-faces.taglib.xml  liferay-faces.tld  liferay-portlet-ext.tld  liferay-portlet.tld  liferay-security.tld  liferay-theme.tld  liferay-ui.tld  liferay-util.tld 	 taglib <ul style="list-style-type: none">  aui  core  faces  portlet  portletext  security  theme  ui  util <ul style="list-style-type: none">  aui.xml  liferay-aui.xml

A “tag” is a combination of three things – a Tag Java file, a TLD file and a set of JSP files (optional). In the above table, you can see all three three elements of a tag. The fourth important thing is to associate the taglib that is containing the tag via “**web.xml**” at the time of deployment.

14.5.1 Creating a custom tag for Library Portlet

In this sub-section, we’ll see how to create a tag lib for something that are we are going to re-use again in our Library Portlet. We have seen the link to “subscribe” and “unsubscribe”. We have already used the same code in two places. Imagine this feature will be required in many other places of the same plugin; it is not a good idea to copy paste the entire code all over again. It will be nice if we encapsulate the code in the form of a taglib and wherever this link is required, we just have to make use of the tag library tag. This demands us to create a new taglib and a tag within it to provide this link. Let’s plunge into action.

Remember, we made an entry in our Portlet’s “**view.jsp**” to show this subscription link. In the whole of this exercise, our task is to convert this into a cute taglib.

```
<%@include file="/html/library/social/subscription.jspf" %>
```

The process comprises of six steps. They are:

1. Creating a TLD (Tag Library Definition) file and putting an entry there.

-
2. Creating the TagLib java class
 3. Define the new taglib in “**web.xml**”
 4. Creating a corresponding JSP for the new tag (*optional*).
 5. Including the new taglib in “**init.jsp**”
 6. Invoking our taglib in any of our JSP files

Now, let's get in the details of each of these steps.

Step 1: Create “**library.tld**” under “**docroot/WEB-INF/tld**” with the below contents.

```
<?xml version="1.0"?>

<taglib
    version="2.0"
    xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
    http://java.sun.com/xml/ns/j2ee/web-jsptaglibrary_2_0.xsd">

    <tlib-version>1.0</tlib-version>
    <short-name>library-ui</short-name>
    <uri>http://liferay.com/tld/library</uri>
    <tag>
        <name>subscribe</name>
        <tag-class>com.library.taglib.SubscribeTag</tag-class>
        <body-content>JSP</body-content>
        <attribute>
            <name>className</name>
            <required>true</required>
            <rteprvalue>true</rteprvalue>
        </attribute>
        <attribute>
            <name>classPK</name>
            <required>true</required>
            <rteprvalue>true</rteprvalue>
        </attribute>
    </tag>
</taglib>
```

Here, we are defining the tag to take two attributes – “**className**” and “**classPK**” and both are mandatory attributes. The next step is to create a Java class for this.

Step 2: Create a new Java class “**SubscribeTag.java**” extending [IncludeTag](#), under the package “**com.library.taglib**” with the following basic structure.

```
package com.library.taglib;

import com.liferay.taglib.util.IncludeTag;

public class SubscribeTag extends IncludeTag {
    final String _PAGE = "/html/library/taglib/subscription.jsp";
    public void setClassName(String className) {
        this.className = className;
    }

    public void setClassPK(long classPK) {
        this.classPK = classPK;
```

```

    }
    private String className;
    private long classPK;
}

```

We have defined two variables each corresponds to the attribute of the tag. These attributes have setter methods defined. We have also defined a page that maps the JSP page that will contain the UI for the tag. Our new class does not end here. We need to define three more methods for the taglib to function properly.

- a) Override the “`setAttributes`” method and setting the two attributes to the `HttpServletRequest` object, “`request`”.

```

protected void setAttributes(HttpServletRequest request) {
    request.setAttribute("library:className", className);
    request.setAttribute("library:classPK", classPK);
}

```

- b) Override the “`doEndTag`” method that makes a call to “`include`” method.

```

public int doEndTag() throws JspException {
    try {
        include(_PAGE);
    } catch (Exception e) {
        e.printStackTrace();
    }
    return super.doEndTag();
}

```

- c) Override the “`include`” method with the following code.

```

protected void include(String page) throws Exception {
    ServletContext servletContext =
        pageContext.getServletContext();
    RequestDispatcher requestDispatcher
        = servletContext.getRequestDispatcher(page);
    requestDispatcher.include(getServletRequest(),
        new PipingServletResponse(
            pageContext, isTrimNewLines()));
}

```

Step 3: As per our initial plan, the third step is to define our new taglib in “**web.xml**”. Open this file for our Library Portlet and insert the following entry in its appropriate location.

```

<taglib>
    <taglib-uri>http://liferay.com/tld/library</taglib-uri>
    <taglib-location>/WEB-INF/tld/library.tld</taglib-location>
</taglib>

```

Step 4: The fourth step is to write JSP for this taglib. Since, we already have the implementation of providing a subscription link in the “**subscription.jspf**” under “**html/library/social**”. We’ll just copy the contents from the file and paste into the new file “**subscription.jsp**” under “**/html/library/taglib**”. The best thing is to just copy the file itself into the new location and change its file extention to “**.jsp**”. After

you change the file extention, don't forget to include the “**init.jsp**” and make some necessary imports to make the new file work.

```
<%@include file="/html/library/init.jsp" %>
<%@page import="com.liferay.portal.kernel.util.Constants" %>
<%@page
import="com.liferay.portal.service.SubscriptionLocalServiceUtil" %>
```

Step 5: This step is to include the new taglib URI definition in our “**init.jsp**” so that our new tag can be used in all other JSP’s of our Portlet. Open the “**init.jsp**” and insert this line in it’s appropriate position.

```
<%@taglib uri="http://liferay.com/tld/library" prefix="library-ui" %>
```

Step 6: The last and final step is the actual invocation of the new tag. Let’s first try this in our “**view.jsp**” helping our members to subscribe for the Library itself. Comment out the original line that included the “**subscription.jspf**” and append the following code for invoking the new tag in lieu of the original code.

```
<%-- <%@include file="/html/library/social/subscription.jspf" %> --%>
<library-ui:subscribe classPK="<%=\ themeDisplay.getScopeGroupId() %>" 
className="<%=\ LMSBook.class.getName() %>" />
```

Now, save all your changes and view our Portlet. The “**subscription**” link should appear as before with all its glamour and exuberance. Remember, the link will appear only if the user is logged in. So, don’t get surprised if you don’t see this link when you’re not logged in to the portal. Try to do the same thing for our “**detail.jsp**” and make the subscription link appear there as well. Here, you will have to pass the actual Id of the book for the “**classPK**” attribute of the tag.

Congratulations! You have created a new taglib with one tag and successfully used in in our JSP files. See how simple and powerful it is. I am sure you have never imagined this kind of simplicity that Liferay brings to our tables.

Code Changes @ <http://code.google.com/p/lr-book/source/detail?r=87>

14.6 More on Portlet + Web Content Integration

We have seen Asset Framework in Section [11.7.3 Actual Implementation of Showing the Books](#). We have seen the marriage between our Portlet and web contents in Section [9.6.1 Embedding Web Content in a Portlet](#). This section is a combination of these two things. I am going to show you one powerful way of externalizing the content from the data. The externalization of the content happens in the form of Web Content and the data is fetched from the database and the merging is done to produce the final output. From the Asset Publisher Portlet, lets see the full_content view of a book.

Liferay book

[« Back](#)



This is full view of a book



0



0



0

Average (0 Votes)



[View in Context »](#)

No comments yet. [Be the first.](#)

For the purpose of testing in the assets “**full_content.jsp**” we have just put one line “This is full view of a book”. But ideally the contents should should the complete details of an actual book in the library with book title, details of the author, who has currently and all other relevant information. One option is to write the whole HTML in the “**full_content.jsp**”. But this is not a scalable model, as this does not allow constant updates to the content style and formatting. It also has a developer dependency. The Librarian wants the flexibility to update the look and feel of book details when viewed through Asset Publisher Portlet.

Let’s begin completing the code for “**full_content.jsp**” under “**html/library/asset**”. Just replace the existing code with this one.

```
<%@page import="com.liferay.portal.kernel.util.StringUtil"%>
<%@page import="com.liferay.portlet.journal.NoSuchArticleException"%>
<%@page import="com.liferay.portlet.journalcontent.util.JournalContent"%>
<%@page import="com.liferay.portlet.journal.model.JournalArticle"%>
<%@page import="com.liferay.portlet.journal.service.JournalArticleLocalServiceUtil"%>

<%@include file="/html/library/init.jsp"%>
<%
    LMSBook lmsBook = (LMSBook)request.getAttribute("ASSET_ENTRY");
    String articleId = "LIBRARY_BOOK_FULL_CONTENT_VIEW";

    JournalArticle journalArticle = null;
    String[] tokens = {
        "[{BOOK_TITLE}]", "[{AUTHOR}]", "[{CREATE_DATE}]"
    };
    try {
        journalArticle =
            JournalArticleLocalServiceUtil.getArticle(
                themeDisplay.getScopeGroupId(), articleId);
    } catch (NoSuchArticleException nsae) {
        StringBuilder sb = new StringBuilder()
            .append("<b>No Web Content found with name:</b>")
            .append(articleId)
            .append("<br/>Create this article with tags")
            .append("<ul>");

        for (int i=0; i<tokens.length; i++) {
```

```

        sb.append(" <li>").append(tokens[i]).append(" </li> ");
    }
    sb.append(" </ul> ");
    out.write(sb.toString());
}

if (Validator.isNotNull(journalArticle)) {
    String content = JournalArticleLocalServiceUtil
        .getArticleContent(journalArticle, null, null,
            themeDisplay.getLanguageId(), themeDisplay);

    String[] values = {
        lmsBook.getBookTitle(), lmsBook.getAuthor(),
        lmsBook.getCreateDate().toString()};
    content = StringUtil.replace(content, tokens, values);
    out.write(content);
}
%>

```

Check the Portlet now in the browser and the output will be like the one below, as we have still not created a web content with the article Id as the one specified.

Liferay book

No Web Content found with name:LIBRARY_BOOK_FULL_CONTENT_VIEW

Create this article with tags

- {{BOOK_TITLE}}
- {{AUTHOR}}
- {{CREATE_DATE}}

[Print](#)

In the last step, let's create the actual article by logging in the portal and going to the "Web Content" section in the Control Panel. Let the article be created with this initial content later on you can do any amount of styling and decoration as you wish.

This is the detailed view of the book {{BOOK_TITLE}}, written by {{AUTHOR}}. The book has been added to our Library on {{CREATE_DATE}}.

Thank you for viewing this book.

That's all. Now go back and check the full content view of the book from Asset Publisher. You will see the book details appearing nicely.

Liferay book

This is the detailed view of the book Liferay book, written by veena. The book has been added to our Library on Fri Mar 08 14:42:41 GMT 2013.

Thank you for viewing this book.

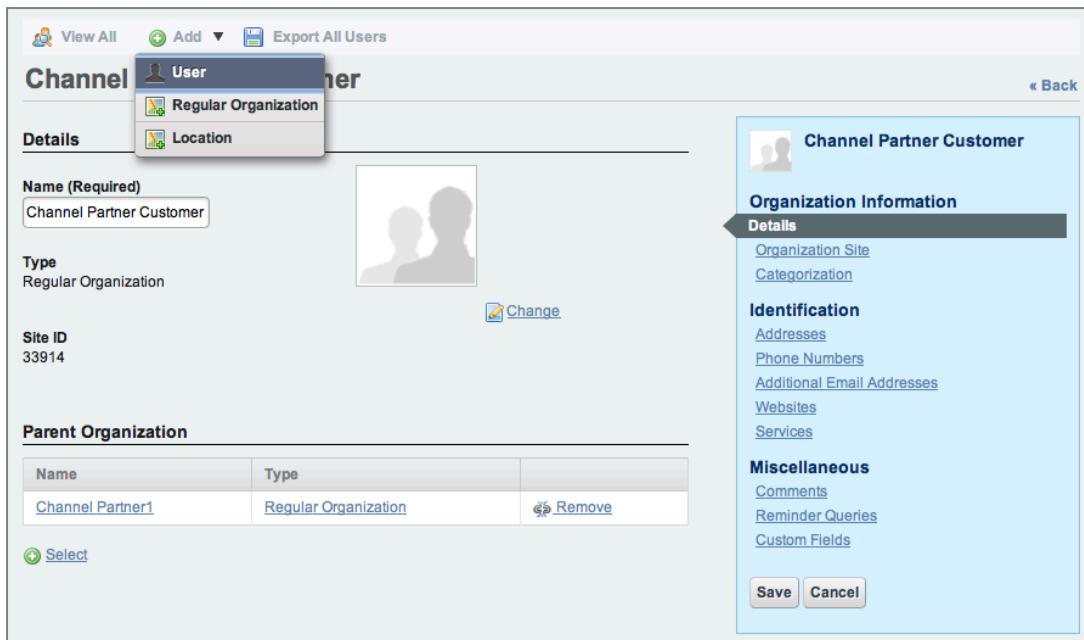
[Print](#)

Congratulations! You have learnt how to use the **JournalArticleLocalServiceUtil** API to dynamically present the content and change it on the fly to show the book details. The librarian can do this himself without depending on the developers.

Code Changes @ <http://code.google.com/p/lr-book/source/detail?r=88>

14.7 Form Navigator Tag

This is going to be the last section of this book. In this section, we'll discuss how to replicate a feature of Liferay that you can see in many places inside the Control Panel. Once such example is the place to manage the Users and Organizations. Go there and you can see a large form broken down into multiple smaller forms and categorized into various sections. Whenever you modify something on one form, it displays the message as "modified" helping the user to know which section has been modified.

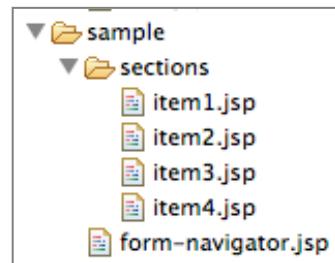


The objective of this section is to know how to implement this feature using one of the powerful tags provided by Liferay. Infact it is quite simple and you need not have to write hundreds of lines of code to accomplish this feature.

Step 1: Create a new file “form-navigator.jsp” inside “html/sample”.

```
<%@include file="/html/library/init.jsp" %>
<%
    String[] categoryNames = {"Category1", "Category2"};
    String[] Category1 = {"item1", "item2"};
    String[] Category2 = {"item3", "item4"};
    String[][] categorySections = {Category1, Category2};
%>
<aui:form>
    <aui:field-wrapper label="NaviForm Sample">
        <liferay-ui:form-navigator
            categoryNames="<%="categoryNames %>"
            categorySections="<%="categorySections %>"
            showButtons="<%="true %>"
            jspPath="/html/sample/sections/" />
    </aui:field-wrapper>
</aui:form>
```

Step 2: Create JSP files for every section under the folder “**html/sample/sections**” with the exact names. Put some unique text or form elements in each of these JSP files. In the next step we’ll create a sample link from the “**view.jsp**” to come to this form navigator page.



Step 3: Open our “**view.jsp**” and append these lines.

```
<portlet:renderURL var="formNavigatorURL">
    <portlet:param name="jspPage"
        value="/html/sample/form-navigator.jsp" />
</portlet:renderURL>
<aui:a href="<%=" formNavigatorURL %>" label="Form Navigator" />
```

Save the changes and click on the link in the Portlet. You will see the form navigator in action. Congratulations! You can use this feature to breakdown a large form into multiple smaller forms separated logically into categories and headings.

Code Changes @ <http://code.google.com/p/lr-book/source/detail?r=91>

Summary

This last and final chapter of this book covered some very important API's and frameworks whatever we were not able to cover in the previous chapters for various reasons. The most important of them is the Indexing and Searching capabilities of Liferay with the help of Apache Lucene Search Engine. We have seen the advanced search mechanisms and faceted search. We have also covered the concepts of Open Search and steps to implement Open Search.

We have covered Message Bus and implemented a good use case for understanding the concept of messaging between two components. The message bus can handle both synchronous and asynchronous messages.

We concluded this chapter with two topics that are equally important. In the first one we discussed about integrating Web Content and Asset Publisher. In the last section we have seen the Form Navigator UI component of Liferay.

Appendix

A – Liferay Portlet XML Entries

We have seen the usage and details of many of the tags in “**liferay-portlet.xml**” through this book. But unfortunately we didn’t have the chance to look into some other entries that are also equally important. I am presenting a quick look at all those entries, so that you may use them in your real world portlets as needed. The remaining un-used entries are given in the alphabetical order and can serve as a good reference.

A.1 “action-timeout”

The default value of this entry is “0”. If set to a value greater than 0, and “**monitoring-spring.xml**” is enabled via the property “**spring.configs**” in “**portal.properties**”, and the property “**monitoring.portlet.action.request**” is set to true, then the portlet’s action phase processing will be timed. If the execution time is longer than action-timeout, it will be recorded as a timeout request processing. The time unit is millisecond.

A.2 “add-default-resource”

If this value is set to false and the portlet does not belong to the page but has been dynamically added, then the user will not have permissions to view the portlet. If the value is set to true, the default portlet resources and permissions are added to the page, and the user can then view the portlet. This is useful (*and necessary*) for portlets that need to be dynamically added to a page. However, to prevent security loopholes, the default value is false. The following properties in “**portal.properties**” allow security checks to be configured around this behavior.

- portlet.add.default.resource.check.enabled
- portlet.add.default.resource.check.whitelist

A.3 “ajaxable”

The default value for this entry is true. If false, then this portlet can never be displayed via Ajax.

A.4 “atom-collection-adapter”

The value for this entry must be a class that implements AtomCollectionAdapter [<http://bit.ly/ZFVkNP>] or extends BaseAtomCollectionAdapter [<http://bit.ly/ZiP8eE>].

A.5 “*autopropagated-parameters*”

Set this value to a comma-delimited list of parameter names that will be automatically propagated through the portlet.

A.6 “*facebook-integration*”

Set this value to either “**fbml**” or “**iframe**”. The default value is “**iframe**” because IFrame integration will work without requiring any changes to your code. See the Message Boards portlet for minor changes that were made to make it FBML compliant. Note that the Liferay tag libraries already output FBML automatically if Facebook makes a request.

A.7 “*include*”

Set this value to true to if the portlet is available to the portal. If set to false, the portlet is not available to the portal. The default value is true. Portlets those are not included as parts of the portal are never available to the user to be made active or inactive. As far the user knows, the portlets do not even exist in the system. This allows the Liferay developers to bundle a lot of portlets in one core package, and yet allow custom deployments to turn on or off individual portlets or sets of portlets. This follows the Siebel and Microsoft model of bundling everything in one core package, but using XML configuration or registry settings to turn on and off features or sets of features.

Liferay does not recommend that custom deployers modify the core source by removing specific portlets because this prevents an easy upgrade process in the future. The best way to turn on and off portlets is to set the include element. The advantage of this way of doing things is that it becomes very easy to deploy Liferay. All features are available in one package. The disadvantage is that by not utilizing all of the portlets, you’re wasting disk space and may even take a small but static memory footprint. However, we feel that the extra disk space and memory usage is a cheap price to pay in order to provide an easy installation and upgrade path.

A.8 “*layout-cacheable*”

Set this flag to true if the data contained in this portlet can/will never change unless the layout or Journal portlet entry is changed.

A.9 “*maximize-edit*”

Set this value to true if the portlet goes into the maximized state when the user goes into the edit mode. This only affects the default portal icons and not what the portlet developer may programmatically set. The default value is false. Same explanation holds good for “*maximize-help*” as well.

A.10 “parent-struts-path”

This value must be the struts-path of another portlet in the same web application. The current portlet will be able to use all the struts mappings of the parent without duplicating them in “[struts-config.xml](#)”.

A.11 “permission-propagator”

This value must be a class that implements the interface PermissionPropagator [[http://bit.ly/Y8BLiV](#)] or extends the class BasePermissionPropagator [[http://bit.ly/WIVMC3](#)] and is called to propagate permissions.

A.12 “poller-processor-class”

This value must be a class that implements PollerProcessor [[http://bit.ly/ZkNlpB](#)] or extends the class BasePollerProcessor [[http://bit.ly/14Uykfr](#)] and is triggered by the JavaScript class Liferay.Poller. It allows a portlet to use polling to be notified of data changes. See Liferay’s Chat portlet for a real world implementation of this feature.

A.13 “pop-message-listener-class”

This value must be a class that implements the interface MessageListener [[http://bit.ly/ZHawtZ](#)] and is called when processing emails.

A.14 “pop-up-print”

Set this value to true if the portlet goes into the pop up state when the user goes into the print mode. This only affects the default portal icons and not what the portlet developer may programmatically set. The default value is true.

A.15 “portlet-layout-listener-class”

This value must be a class that implements the interface PortletLayoutListener [[http://bit.ly/15LvnQH](#)] and is called when a portlet is added, moved, or removed from a layout. See JournalContentPortletLayoutListener [[http://bit.ly/14UzV5d](#)] in your portal source for an example of this type of class.

A.16 “portlet-url-class”

This value must be a class that extends PortletURLImplWrapper [[http://bit.ly/YqOadZ](#)]. Set this class to override the default portlet URL implementation. See StrutsActionPortletURL [[http://bit.ly/10KDP3P](#)] in your portal source as an example.

A.16 “*private-request-attributes*”

Set this value to true if the portlet does not share request attributes with the portal or any other portlet. The default value is true. The property “**request.shared.attributes**” in “**portal.properties**” specifies which request attributes are shared even when the private-request-attributes value is true.

A.17 “*private-session-attributes*”

Set this value to true if the portlet does not share session attributes with the portal. The default value is true. The property “**session.shared.attributes**” in “**portal.properties**” specifies which session attributes are shared even when the private-session-attributes value is true.

A.18 “*remoteable*”

Set this value to true if the portlet can be used remotely like in WSRP. If set to false, the portlet will not be available remotely. The default value is false.

A.19 “*render-timeout*”

Same as “action-timeout” but this property is for render requests.

A.20 “*render-weight*”

The default value is 1. If set to a value less than “1” the portlet is rendered in parallel. If set to a value of “1” or greater, then the portlet is rendered serially. Portlets with a greater render weight have greater priority and will be rendered before portlets with a lower render weight. If the ajaxable value is set to false, then render-weight is always set to 1 if it is set to a value less than 1. This means ajaxable can override render-weight if ajaxable is set to false.

A.21 “*restore-current-view*”

Set this value to true if the portlet restores to the current view when toggling between maximized and normal states. If set to “false”, the portlet will reset the current view when toggling between maximized and normal states. The default value is true.

A.22 “*show-portlet-access-denied*”

Set this value to true if users are shown the portlet with an access denied message if they do not have access to the portlet. If the value is set to “false” then users are never shown the portlet if they do not have access to the portlet. The default value is set in “**portal.properties**”.

A.23 “*show-portlet-inactive*”

Set this value to true if users are shown the portlet with an inactive message if the portlet is inactive. If set to false, users are never shown the portlet if the portlet is inactive. The default value is set in “**portal.properties**”.

A.24 “*social-request-interpreter-class*”

This value must be a class that implements SocialRequestInterpreter [<http://bit.ly/YkVj2x>] or extends the class BaseSocialRequestInterpreter [<http://bit.ly/13ScAor>] and is called to interpret requests into friendly messages that are easily understandable by a human being.

A.25 “*struts-path*”

Assume that the value is “mail”. This tells the portal that all requests with the path mail/* are considered part of this portlet’s scope. Users who request paths that match mail/* will only be granted access if they also have access to this portlet. This is true for both portlet requests and regular servlet requests.

A.26 “*system*”

Set this value to true if the portlet is a system portlet that a user cannot manually add to their page. The default value is false.

A.27 “*url-encoder-class*”

This value must be a class that implements URLEncoder. Use this to set a custom URLEncoder [<http://bit.ly/12NNjwi>] that is used by the RenderResponse class to implement the encodeURL method. This is useful if you need to add custom logic to rewrite URLs.

A.28 “*use-default-template*”

Set this value to true if the portlet uses the default template to decorate and wrap content. Setting this to false allows the developer to own and maintain the portlet’s entire outputted content. The default value is true. The most common use of this is if you want the portlet to look different from the other portlets or if you want the portlet to not have borders around the outputted content.

A.29 “user-principal-strategy”

Set this value to either "userId" or "screenName". Calling request.getRemoteUser() will normally return the user id. However, some portlets may need the user principal returned to be screen name instead.

A.30 “virtual-path”

This value sets the virtual path used to override the default servlet context path. For example, suppose your portlet is deployed to the servlet path "/test-portlet". By default, the portal will return "/test-portlet" for the servlet context path. You can override it by setting virtual-path to "/virtual" and have the portal return "/virtual" for the servlet context path. The default value is "" which means this is not used.

A.31 “webdav-storage-class”

This value must be a class that implements WebDAVStorage [<http://bit.ly/X553vn>] or extends the class BaseWebDAVStorage [<http://bit.ly/ZI7rfh>] and allows data to be exposed via the WebDAV protocol. See DLWebDAVStorageImpl in your portal source for an example.

A.32 “webdav-storage-token”

This value is the WebDAV directory name for data managed by this portlet.

A.33 “xml-rpc-method-class”

This value must be a class that implements Method [<http://bit.ly/16qAJCn>] and allows data to be exposed via the XML-RPC protocol. See in our portal source, PingbackMethodImpl [<http://bit.ly/XHiLIG>] for details.

M P Ahmed Hasan

Author

