



F r o m T e c h n o l o g i e s t o S o l u t i o n s

Liferay Portal 5.2 Systems Development

Build Java-based custom intranet systems on top of Liferay portal



Jonas X. Yuan

PACKT
PUBLISHING

Liferay Portal 5.2 Systems Development

Build Java-based custom intranet systems on top of Liferay portal

Jonas X. Yuan



BIRMINGHAM - MUMBAI



This material is copyright and is licensed for the sole use by Matt Bedsaul on 27th May 2009
1 Microsoft Way, , Redmond, , 98052

Liferay Portal 5.2 Systems Development

Copyright © 2009 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, Packt Publishing, nor its dealers or distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: May 2009

Production Reference: 1190509

Published by Packt Publishing Ltd.
32 Lincoln Road
Olton
Birmingham, B27 6PA, UK.

ISBN 978-1-847194-70-1

www.packtpub.com

Cover Image by Leo Cornall (leocornall@hotmail.com)



This material is copyright and is licensed for the sole use by Matt Bedsaul on 27th May 2009
1 Microsoft Way, , Redmond, , 98052

Credits

Author

Jonas X. Yuan

Project Team Leader

Abhijeet Deobhakta

Reviewers

Christianto Sahat
Steve Rogers

Editorial Team Leader

Gagandeep Singh

Acquisition Editor

Sarah Cullington

Project Coordinator

Lata Basantani

Development Editor

Dilip Venkatesh

Indexer

Hemangini Bari

Technical Editors

Aanchal Kumar
John Antony

Proofreader

Camille Guy

Copy Editors

Sneha Kulkarni
Sumathi Sridhar

Production Coordinator

Dolly Dasilva

Cover Work

Dolly Dasilva

About the author

Dr. Jonas X. Yuan is a Senior Technical Analyst at CIGNEX. He holds a Ph. D. in Computer Science from University of Zurich specializing in Integrity Control in Federated Database Systems. He earned his M.S. and B.S. degrees from China, where he conducted research on expert systems for predicting landslides. Jonas is experienced in Systems Development Life Cycle (SDLC). Previously, he worked as a Project Manager and a Technical Architect in Web GIS (Geographic Information System). He has deep, hands-on skills in J2EE technologies. Most importantly, he had developed a BPEL (Business Process Execution Language) Engine called BPELPower from scratch in NASA data center. He has a strong experience on content management and publishing such as Media/Games/Publishing. He is also an expert in Liferay portal, Alfresco Content Management Systems (CMS), OpenX Ad Serving, Intalio | BPM, Pentaho Business Intelligence, LDAP, and SSO.

He has also authored the book: *Liferay Portal Enterprise Intranets*; ISBN 978-1-84719-272-1.



This material is copyright and is licensed for the sole use by Matt Bedsaul on 27th May 2009
1 Microsoft Way, , Redmond, , 98052

Acknowledgement

I would like to thank the team members at Liferay, especially, Raymond Auge, Brian Chan, Bryan Cheung, Jorge Ferrer, Michael Young, Jerry Niu, Ed Shin, Craig Kaneko, Brian Kim, Bruno Farache, Thiago Moreira, Amos Fong, Scott Lee and David Truong for providing the valuable information and all the support.

My special thanks to all my team members at CIGNEX for making this book a reality. I would like to thank Paul Anthony, Munwar Shariff, and Rajesh Devidasani for their encouragement and great support. Our sales and pre-sales team Amit Babaria, Harish Ramachandran, helped me understand what the customers are looking for. Our consulting team Robert Chen, Venkata Challagulla, Harshad Bakshi, and Zankar Shah presented me the various flavors of Liferay implementations with real-life examples. I am thankful to them.

I sincerely thank and appreciate Sarah Cullington and Dilip Venkatesh, Senior Acquisition Editor and Development Editor, respectively, at Packt Publishing for criticizing and fixing my writing style. Thanks to Lata Basantani, Aanchal Kumar, John Antony, and the entire team at Packt Publishing. It is really joyful to work with them.

Last but not the least, I would like to thank my parents and my wife, Linda, for their love, understanding, and encouragement. My special thanks to my wonderful and understanding kid, Joshua.

About the reviewer

Christianto Sahat was born and raised in Jakarta, Indonesia. He decided to find a job abroad and see the world because "the trees in the village don't teach me anything anymore". He graduated from the local university in Electronics Engineering with digital design skill, and then switched to software development, especially, Java technology and Liferay portal. He has been working on many projects in insurance, banking and public sector projects for many years, and now works as a freelance portal developer specializing in Liferay portal development. He enjoys all kinds of water and sea sports such as from wind surfing, diving, and underwater hockey. Currently he lives in Singapore.

I would like to thank S. Resmiana Limpong, my mother, who struggled so hard to raise me as a single parent, even though it was a very tough period for her. Without her I won't be here, exploring and learning about Java and Liferay technologies and seeing the world.

I'd like to thank the Liferay team as well for creating a very good and free portal software, indirectly giving support to reduce digital divide between the first and third world countries, and giving a chance to local software developers to make a new business from this software, creating many jobs. Now I know how to work for a much better purpose than just to earn money. Special thanks to Raymond Auge and Jorge Ferrer who still manage to find time to answer questions on Liferay's forum. You inspire me a lot guys.

Table of Contents

Preface	1
Chapter 1: Introducing Liferay Portal Architecture and Framework	9
What's Liferay portal?	9
Liferay portal	10
Liferay CMS and WCM	11
Liferay collaboration and social networking software	12
Why Liferay portal?	13
A rich, friendly, intuitive, and collaborative end user experience	13
A single point of access to all information	14
High adaptability to the demands of fast-changing market	15
Highest values	15
Architecture and framework	16
Service oriented architecture: SOA	16
Enterprise service bus: ESB	17
Portal development strategies	18
Extension environment	18
Plugins SDK environment	19
Development strategies	20
Summary	22
Chapter 2: Working with JSR-286 Portlets	23
Experiencing Liferay portal and portlets	24
What is a portal?	24
What is a portlet?	25
What is a portlet container?	26
Why JSR-286 portlets?	27
Using JSR-286 portlets	27
Understanding portlet life cycle	27
Utilizing portlet modes	29

Table of Contents

Employing window states	31
What's the difference between a portlet and a servlet?	33
Use cookies, document head section elements, and HTTP headers	34
Employing portlet configuration, context, request, response, and preferences	35
Using portlet configuration	37
Employing portlet context	38
Using portlet request	38
Employing portlet response	39
Working with portlet preference	39
Extending JSR-286 portlets	40
Using portlet filters	40
Using portlet-managed modes	42
Utilizing container runtime options	42
Serving resources	43
Using Resource URL	43
Using caching levels of resources	44
Serving the AJAX data directly from the portlet	45
Utilizing other features	45
Using JAVA 5 features	45
Employing caching features	46
Sharing runtime ID	47
Using taglib	47
Coordinating portlets	48
Sharing data via the session	50
Using PortletSession	50
Using PortletContext	51
Using page parameters	51
Using portlet events	52
Sending events	52
Receiving events	53
Employing public render parameters	54
Summary	55
Chapter 3: ServiceBuilder and Development Environments	57
Setting up Ext	58
Required tools	58
JDK	58
Ant	59
Databases	59
MySQL	59
Application servers	60
Tomcat	61

Table of Contents

IDE	62
Eclipse IDE	63
Workspace	63
Subclipse	64
Tomcat plugins	65
Portal source code	67
Building Ext	67
Getting portal source code	68
Source structures and Ant targets	69
Updating Tomcat to support Ext development	70
Customizing properties	71
Building via Ant	73
Navigating Ext structures	73
Deploying Ext	74
Configuring database	74
Using Ant deploy	75
View portal structures in Tomcat	76
Fast-deploy in Ext	77
Using ServiceBuilder in Ext	78
Viewing portlet development structures	79
Building services	80
Create service XML	81
Build services	82
What's happening?	83
Navigating portlet specification	84
Setting up Plugins SDK	86
Building Plugins SDK project	87
Deploying plugins	88
Fast development of plugins with Tomcat	89
Using development environments efficiently	90
How does Ext work?	91
When do we use Ext?	91
Summary	92
Chapter 4: Experiencing Struts Portlets	93
Developing a JSP portlet	94
Defining the JSP portlet	94
Changing the title and category	97
Using JSP portlet efficiently	98
Fast deploy	98
Employing JSP portlet	99

Table of Contents

Constructing a basic Struts portlet	100
Defining a Struts portlet	101
Specifying the page flow and page layout	103
Creating JSP pages	105
Changing the title and category	107
Building an advanced Struts portlet	108
Adding an action	108
Creating an action	109
Defining the action	110
Adding a form in JSP page	111
Creating success and error pages	111
Interacting with the database	112
Creating a database structure	113
Creating methods to add and retrieve records	115
Updating existing files	116
Retrieving records from the database	118
Redirecting	118
Updating the action	119
Updating action paths	120
Updating existing JSP files	120
Adding more actions	121
Creating methods to edit and delete records	122
Updating the action	123
Creating actions menu JSP file	123
Updating existing JSP files	124
Setting up permissions	126
Setting up permissions in the backend	128
Setting up permissions in frontend	130
Deploying	132
Using Struts efficiently	133
Why use Struts?	134
Why use tiles?	134
When do we use Struts?	134
Chapter 5: Managing Pages	135
Extending Communities portlet	136
Building Ext Communities portlet	137
Constructing the portlet	137
Setting up actions	140
Setting up page flow and page layout	141
Preparing JSP files	143
Setting up the Ext Communities portlet in the backend	144
Creating database structure	145
Creating methods to update, delete, and retrieve	147
Updating the action classes	148

Table of Contents

Setting up the Ext Communities portlet in the frontend	149
Updating and deleting Community customized columns	149
Retrieving community-customized columns	150
Customizing the Manage Pages portlet	150
Building a standalone layout management portlet	152
Constructing the portlet	152
Setting up the action	153
Setting up page flow and page layout	155
Preparing JSP files	155
Setting up the Ext Layout Management portlet in the backend	156
Creating a database structure	156
Creating methods to update, delete, and retrieve	158
Updating the action class	160
Setting up the layout management portlet in the frontend	160
Customizing page management with more features	162
Adding localized feature	162
Extending model for locale	163
Customizing language properties	164
Displaying multiple languages	166
Employing tabs	169
Applying layout templates dynamically	170
Setting up pages, layout templates, and portlets mappings	171
Adding layout templates	171
Displaying layout templates by sections	173
Tracking pages	175
Using communities and layout page efficiently	176
Employing group, community, and permissions	176
Using communities, layout pages, comments, and ratings	177
Extending the community and layout pages	178
Summary	178
Chapter 6: Customizing the WYSIWYG Editor	179
Configuring the WYSIWYG editor	180
Extending the Ant target for fast deployment	180
Upgrading the WYSIWYG editor: FCKeditor	181
Setting up the FCKeditor	182
Adding customized icons	182
Employing default configuration	183
Adding templates and styles in FCKeditor	184
Constructing styles and formats	186
Preparing CSS styles in themes	186
Employing customized CSS styles from themes	187
Customizing styles	188
Building templates	189

[v]

Table of Contents

Inserting images and links from different services	190
Configuring a File Browser Connector with Liferay portal services	191
Configuring the services for images, documents, and pages	191
Browsing images and links	192
Preparing Liferay portal services	193
Customizing the File Browser Connector with RESTful services	195
Adding advanced search view features	195
Adding advanced search functions to links and images	198
Preparing RESTful services	205
Inserting content-rich flashes into Web Content	207
Querying flashes	208
Adding single flash SWF, videos, and slideshows to journal articles	210
Adding advanced search views	211
Adding advanced search functions	211
Adding flash objects	213
Adding video queue and video list as part of journal articles	215
Putting a video list into journal articles	216
Setting up video queue in journal articles	217
Adding games and playlists as part of journal articles	218
Playing games beside text message	220
Employing playlist as visualization of text information	220
Preparing RESTful services	221
Using the WYSIWYG editor FCKeditor efficiently	222
Extending the file browser connector	222
Employing the WYSIWYG editor in portlets	223
Employing the WYSIWYG editor in the Web Content portlet	223
Using Liferay display tag	224
Adding the WYSIWYG editor in a custom portlet	224
When do we use the WYSIWYG editor?	225
Summary	225
Chapter 7: Customizing CMS and WCM	227
Managing Terms of Use dynamically	228
Customizing static Terms of Use	228
Building dynamic Terms of Use	229
Constructing featured content	230
Customizing the Web Content Display portlet	231
Creating the Ext Web Content Display portlet	232
Building a view action	234
Setting up structure and template	235
Building a structure	236
Preparing the icon images	236
Building a template	237
Building featured content articles	238
Preparing images	238
Building an article with images and text	239

Customizing the Web Content List portlet	239
Constructing the Ext Web Content List portlet	240
Building a view action	242
Setting up the view page	243
Adding custom article types	243
Consuming custom article types	243
Customizing the Asset Publisher portlet	246
Adding a large-size image and a medium-size image in Web Content	246
Building the Ext Asset Publisher portlet	249
Extending view with tags	252
Configuring tags	252
Setting up default tags	252
Updating views	253
Building dynamic articles with recently added content and related content	255
Displaying journal articles through asset ID	257
Showing touts with article ID	258
Adding Velocity services	258
Building touts structure and template	260
Building article touts	262
Listing recently added content	262
Exhibiting related content	264
Building dynamic articles with polls	266
Adding template node poll	267
Updating the Web Content portlet with template node poll	269
Associating journal articles with polls	270
Extending CMS and WCM	271
Employing articles, structures, and templates	271
Using journal template—Velocity templates	272
Enjoying the Web Content search portlet	273
Tagging content	273
Extending Image Gallery and Document Library	275
Adding Velocity templates in Asset Publisher	275
Summary	276
Chapter 8: Building a Personalized Community	277
Sharing content with friends	278
Building print preview as article template	279
Sharing applications on any web site by widget	280
Sending emails with sharing links to friends	282
Building the Share portlet	282
Setting up view action and email	282
Setting up the view page with jQuery	283

Table of Contents

Preparing jQuery service	285
Building the article template	286
Setting up the most popular journal articles	286
Adding a view counter in the Web Content Display portlet	287
Setting up VM service	289
Building article template for the most popular journal articles	290
Setting up the default article type	290
Setting up the article template	291
Putting all article templates together	291
Having a handle on view counter for assets	292
Using journal article tokens	292
Get view count on Wiki articles	293
Getting views count on blog entries	294
Getting views on Message Boards threads	294
Setting up view counter on the Image Gallery images	295
Setting up view counter on Document Library documents	296
Getting visits on bookmark entries	296
Personalizing user comments	297
Creating user comments model	298
Building the Ext Comment portlet	300
Adding permissions based on user groups	301
Updating the UI tag	302
Setting up email notification	303
Customizing My Account	304
Customizing login view	305
Locating the portlet My Account	306
Overriding login view	307
Creating a customized account on the fly	309
Building personal community—My Street	311
Customizing user model	312
Building the portlet My Street	314
Adding Struts view page	316
Sharing the My Street theme	316
Adding videos, games, and playlists into My Street	317
Using personal community efficiently	318
Extending user account and user preferences	318
Setting My Community	319
Using Control Panel to manage My Account	319
Using dynamic query API	320
Using pop ups	321
Applying Floating DIV pop up	321
Employing window pop up	322
Summary	322

Table of Contents

Chapter 9: Developing Layout Templates and Themes	323
Building layout templates in Ext	324
Constructing custom layout templates	326
Experiencing default layout templates	326
Adding customized layout templates	328
Registering layout templates	330
Developing layout templates in Plugins SDK	332
Building layout templates	334
Creating layout templates	336
Building themes in Plugins SDK	339
Creating a customized theme	340
Setting up the theme project	340
Building differences of themes	343
What's happening after deploying themes?	343
Putting HTML to use	345
Experiencing CSS and images	346
Using jQuery JavaScript library	347
Employ theme settings	350
Adding color schemes	351
Adhering to WAP standard	352
Adding runtime portlets to a theme	353
Using theme, CSS, and JavaScript	353
Making use of themes	353
Applying CSS	354
Employing JavaScript	354
Experiencing the developing and debugging tools	355
Customizing Velocity templates in themes	355
Using default Velocity templates	356
Experiencing default Velocity variables	356
Customizing Velocity variables	358
Adding customized Velocity templates	360
Using Velocity templates in drop-down menu	361
Using Velocity templates in journal article-based navigation	364
Setting up customized themes and layout templates as default	366
Using Plugins SDK more efficiently	367
How does it work?	367
When to use Plugins SDK?	368
Summary	368
Chapter 10: Building My Social Office	369
Experiencing the Control Panel	370
What's Control Panel?	370
How does it work?	373
Using the Control Panel theme	373

Table of Contents

Employing Control Panel settings	374
Configuring portlets for Control Panel	376
How to customize it?	377
Changing theme	377
Updating both edit page and view page	378
Configuring customized portlets	379
Building Inter-Portlet Communication	381
Creating IPC portlet project	381
Constructing IPC portlets	382
Defining portlets	384
Defining events	384
Registering portlets	384
Specifying portlet process actions	386
Specifying portlet views	387
Developing Social Office theme	389
Setting up the theme project	390
Constructing differences of the so-theme	390
Adding mail and chat portlets	391
Setting up the mail portlet	391
Setting up the chat portlet	393
Deploying the chat portlet	393
What's happening behind?	394
Building Social Office with portlets	394
Rearing the Social Office portlets project	396
Assembling social portlets	396
Raising JavaScript functions and friendly URL	399
Erecting social views	400
What's happening?	402
Experiencing social models	402
Experiencing social services	403
Adding social activity tracking	405
Hooking properties and JSP files into Social Office	406
Building hooks	407
Applying portal event handlers	408
Putting model listeners to use	409
Erecting portal properties	410
Employing JSP hooks	412
Using hooks more efficiently	414
General usage	414
WOL—World of Liferay	416
Special usage	416
Document library hooks	416
Auto-login hooks	417

[x]

Table of Contents

Mail hooks	417
Summary	418
Chapter 11: Staging and Publishing	419
Building dynamic navigation and site map	420
Constructing custom navigation and street navigation	421
Build portlets' views	423
Establishing custom site map	424
Constructing the street site map portlet	425
Building up portlet view	425
Customizing event handlers and model listeners	427
Handling events	427
Configuring global startup and shutdown actions	428
Creating a custom cookie on login	429
Building custom model listeners	431
Creating custom model listener	432
What's happening?	433
Undergoing local staging and publishing	434
Activating staging	435
What's happening?	436
How does it work?	437
Staging and publishing locally	438
Copying from live	439
Publishing to live	440
Employing staging workflow and other workflows	442
Activating staging workflow	442
Creating a proposal	443
What's happening?	444
Customizing staging workflow	448
Extending model	448
Building a standalone workflow portlet	449
Employing the journal article workflow	452
Play with the jBPM workflow	452
Using Intalio BPMS	453
Scheduling web content	453
Scheduling pages	454
Scheduling the web content	454
What's happening?	455
Setting a scheduler engine	455
Scheduling layouts	455
Configuring scheduler class	457
Experiencing remote staging and publishing	458
What's remote staging and publishing?	460
How does it work?	461

Table of Contents

Importing and exporting	462
Using tunnel web	462
Setting up tunnel web	463
Using LAR to export and import	464
Defining portlet-data-handler	464
Configuring a portlet with portlet-data-handler	465
Using portlet-data-handler	465
Using SCORM	466
Summary	466
Chapter 12: Using Common API	467
Adding custom attributes	468
Building dynamic table with Velocity Expando template	468
Creating a journal structure	469
Creating a journal template	469
Building Book Title List	470
What's happening?	472
The Expando Velocity template variables	472
Models and services	473
Extending custom attributes	475
Enhancing users and organizations	475
What's happening?	476
Sharing the Expando portlet	478
Building OpenSearch	480
What's happening?	481
Adding the OpenSearch capability on custom portlets	483
Adding search capabilities in portlets	485
Using Solr for enterprise search	486
Overriding the Spring services	487
Overriding method validation	488
Changing model name via ServiceBuilder	490
What's happening?	492
Consuming Liferay services in portlets	494
How does it work?	494
Customizing friendly URL mappings	496
What's happening?	497
Constructing web services	498
Building custom web services	499
Consuming web services in portlets	500
How does it work?	500
What's happening?	502
Enjoying best practices	502
Using JavaScript Portlet URL	503
Customizing user and organization administration	504
Creating a new section	505

Table of Contents

Customizing fields of form section	505
Customizing columns of the list	505
Speeding up the portal	506
Sharing UI Taglibs in portlets	507
How does it work?	507
Consuming WSRP	509
How do we get the WSRP portlets?	510
How does it work?	511
Integrating with SharePoint	512
Integrating with Terracota DSO	513
Summary	513
Index	515



This material is copyright and is licensed for the sole use by Matt Bedsaul on 27th May 2009
1 Microsoft Way, , Redmond, , 98052

Preface

Liferay portal is one of the most mature portal frameworks in the market. It offers many key business benefits that involve personalization, customization, and workflow. If you are a Java developer who wants to build custom web sites and intranet applications using Liferay portal, this is where your search ends.

Liferay Portal provides within a secure, administrated framework, an ability to organize the potential chaos of an unfettered Web 2.0 environment. It empowers users with tools such as blogs, instant emails, message boards, instant messaging, shared calendar, social networking, social office, CMS, WCM, and so on.

This book shows how Java developers can use Liferay as a framework to develop custom intranet systems based on Liferay portal platform, thus, helping you to maximize your productivity gains. Get ready for a rich, friendly, intuitive, and collaborative end user experience.

Using this book, you can customize Liferay into a single point of access to all of an organization's data, content, web content, and other information from both the existing in-house applications (such as HR and CRM) and the external sources (such as Alfresco, FatWire, Magnolia, and Vignette).

What this book covers

In *Chapter 1*, we look at what Liferay portal is and why we should use it. Then we introduce the Liferay portal architecture and framework. Liferay portal can be extendible at three levels—Plugins SDK environment, Extension environment, and Liferay portal source code. Finally, we discuss portal development strategies in detail.

In *Chapter 2*, we cover the experience of Liferay portal and portlets, using JSR-286 portlets, employing portlet configuration, context, request, response, and preferences, extending JSR-286 portlets, serving resources, and coordinating portlets. It helps you to build larger applications and re-use portlets in different scenarios.

In *Chapter 3*, we look at how to set up, build, and deploy Ext by using ServiceBuilder, how to set up Plugins SDK, and how to use development environments in an efficient way.

In *Chapter 4*, we include experiencing Struts portlets in our discussion, where we first discuss how to develop a JSP portlet. Then we introduce how to develop a basic Struts portlet in Ext—defining the portlet, and specifying page action, and page layout. Accordingly, we also introduce how to develop an advanced Struts portlet in Ext—redirecting, adding more actions, setting up permissions, and so on. Finally, we address how to use Struts efficiently.

In *Chapter 5*, we first look at extending the Communities portlet, then we move on how to customize the Manage Pages portlet. We also look at how to customize page management with more features, and use communities and layout pages in an efficient way.

In *Chapter 6*, we focus on customizing the WYSIWYG editor. We first introduce how to configure the WYSIWYG editor, quickly deploy the updates, and upgrade it. Then we introduce how to customize FCKeditor to make images, links, videos, games, video queues, video lists, and playlists a part of web content. Finally, we introduce how to use the WYSIWYG editor FCKeditor.

In *Chapter 7*, we look at one of the most common parts of Liferay portal—CMS and WCM. We first discuss how to manage the terms of use dynamically with a journal article. Then, we present a way to build articles with multiple image icons, rating, comments, polls, related content, recently added content, and so on. Finally, we discuss how to use and extend CMS and WCM. We also discuss relationship among articles, structures, and article templates, CMS extension, and the Asset Publisher portlet extension.

In *Chapter 8*, we look at how to build My Community in general, and how to customize and extend this feature as well. First, we introduce how to share web site, pages, or portlets with friends. Then we introduce how to customize My Account and how to build My Street with personalized preferences. Finally, we address the best practices to use My Community efficiently, including dynamic query API, pop-up JavaScript, My Community settings, My Account Control Panel, user account extension, and user preferences.

In *Chapter 9*, we discuss how to develop layout templates in both Ext and Plugins SDK, and how to build themes in Plugins SDK. It introduces how to build layout templates in Ext first. Then it discusses how to build layout templates and themes in Plugins SDK and how to add Velocity services in themes. Finally, it addresses how to use Plugins SDK in an efficient way.

In *Chapter 10*, we focus on how to build My Social Office in general. We introduce Control Panel first—how it works and how to customize it. Then we address Inter-Portlet Communication (IPC)—how to build IPC portlets. Later, we discuss how to set up Social Office themes and portlets, and how to hook language properties, and portal properties. Finally, we discuss an efficient way to use hooking features.

In *Chapter 11*, we look at staging and publishing both locally and remotely, where we first discuss simple extension—how to build dynamic navigation and how to construct customized site map. Then, we address how to handle events and model listeners. Based on these features, we further introduce local staging and publishing, and staging workflow. A way to schedule pages and assets is also discussed. Finally, we address how to publish the web content remotely, where `portlet-data-handler` (for export and import via LAR) is addressed as well.

In *Chapter 12*, we first cover how to use custom attributes for both journal article templates and custom portlets. Then, we address how to build OpenSearch and how to employ search capabilities. Later, we focus on approaches on how to employ Spring services and how to construct web services. Finally, we discuss the best practices such as using JavaScript portlet URL, customizing the user and organization administration, speeding up portal, sharing UI Taglibs, producing and consuming WSRP, and integrating with SharePoint and Terracotta DSO.

What you need for this book

This book uses Liferay portal version 5.2.3 mainly with the following settings in Windows:

- Eclipse IDE 3.4
- MySQL 5.0
- Java SE 6.0
- Tomcat 6.0

Optionally, you can also work in Windows, MacOS, and Linux with the following settings:

- Liferay portal version 5.2.3 or above
- Eclipse IDE 3.4 or above
- MySQL 5.0 or above
- Java SE 5.0
- Tomcat 5.5

You can use one of the following options for Servlet containers and full Java EE application servers to run the Liferay portal:

- Geronimo + Tomcat
- Glassfish 3
- Glassfish 2 for AIX
- Glassfish 2 for Linux
- Glassfish 2 for OSX
- Glassfish 2 for Solaris
- Glassfish 2 for Solaris (x86)
- Glassfish 2 for Windows
- JBoss + Tomcat 4.2
- JBoss + Tomcat 5.0
- Jetty
- JOnAS + Jetty
- JOnAS + Tomcat
- Resin
- Tomcat 5.5 and 6.0
- Borland ES 6.5
- JRun 4 Updater 3
- Oracle AS 10
- Orion 2.0
- Pramati 5.0
- RexIP 2.5
- SUN JSAS 9.1
- WebLogic 8.1 SP4, 9.2, 10
- WebSphere 5.1, 6.0, 6.1, 7.0

Databases that the Liferay portal can run on include:

- Apache Derby
- IBM DB2
- Firebird
- Hypersonic
- Informix
- InterBase
- JDataStore
- MySQL
- Oracle
- PostgresSQL
- SAP
- SQL Server
- Sybase

Operating systems that the Liferay portal can run on include:

- LINUX (Debian, RedHat, SUSE, Ubuntu, and so on)
- UNIX (AIX, FreeBSD, HP-UX, OS X, Solaris, and so on)
- WINDOWS
- MAC OS X

Who this book is for

This book is for Java developers who want to build custom web sites, portals, and highly customized intranet applications using Liferay as a framework.

Readers need to know the basics of Liferay and be competent Java developers. They should have some knowledge of the "standards" that Liferay adopts, but that's not so essential—we will try to explain the important ones in the book.

Although Liferay portal makes heavy use of open source frameworks such as Spring, Hibernate, Struts, and Lucene, no prior experience in using these is assumed.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: "The above code shows that the `BookReportsPortlet` portlet extends `StrutsPortlet` and the portlet mode `VIEW` is specified."

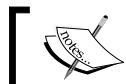
A block of code will be set as follows:

```
<portlet>
  <portlet-name>book_reports</portlet-name>
  <struts-path>ext/book_reports</struts-path>
  <use-default-template>false</use-default-template>
</portlet>
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items will be shown in bold:

```
PortletPreferences prefs = renderRequest.getPreferences() ;
String currentURL = PortalUtil.getCurrentURL(request) ;
%>
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in our text like this: "when the user enters an empty book title and clicks on the **Add Book** button, the error page depicts an error message: **Error! The Book Title is null!**".



Warnings or important notes appear in a box like this.



Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book – what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply drop an email to feedback@packtpub.com, and mention the book title in the subject of your message.

If there is a book that you need and would like to see us publish, please send us a note in the **SUGGEST A TITLE** form on www.packtpub.com or email suggest@packtpub.com.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code for the book

Visit http://www.packtpub.com/files/code/4701_Code.zip to directly download the example code.

The downloadable files contain instructions on how to use them.

Errata

Although we have taken every care to ensure the accuracy of our contents, mistakes do happen. If you find a mistake in one of our books – maybe a mistake in text or code – we would be grateful if you would report this to us. By doing so, you can save other readers from frustration, and help us to improve subsequent versions of this book. If you find any errata, please report them by visiting <http://www.packtpub.com/support>, selecting your book, clicking on the **let us know** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata added to any list of existing errata. Any existing errata can be viewed by selecting your title from <http://www.packtpub.com/support>.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or web site name immediately so that we can pursue a remedy.

Please contact us at copyright@packtpub.com with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at questions@packtpub.com if you are having a problem with any aspect of the book, and we will do our best to address it.

1

Introducing Liferay Portal Architecture and Framework

This book will show you how to build custom systems on top of the Liferay portal. In this chapter, we will look at:

- The features of Liferay portal
- Why Liferay portal is an excellent choice for building custom systems
- The framework and architecture of Liferay portal
- The portal development strategies and how they work

So let's begin by looking at exactly what Liferay portal is.

What's Liferay portal?

As the world's leading open source portal platform, Liferay portal provides a unified web interface to the data and tools scattered across many sources. Within Liferay portal, a portal interface is composed of a number of portlets—self-contained interactive elements that are written to a particular standard. As portlets are developed independent from the portal itself and are loosely coupled with the portal, they are, apparently, **Service Oriented Architecture (SOA)**.

Liferay portal has a wide range of portlets freely available for things such as blogs, calendar, Document Library, Image Gallery, mail, message boards, polls, RSS feeds, Wiki, web content, and so on. Liferay portal also ships with the **Content Management System (CMS)** and **Web Content Management (WCM)** solutions. Liferay CMS provides basic **Enterprise Content Management Systems (ECMS)** features. Liferay portal is the best ECMS for small team collaborations. Event data can be specific to a small group within a company. In any organization, some data will be relevant at a team level and other data that will be relevant across the whole business. Liferay portal supports such things very well.

As the world's leading open source enterprise portal solution, Liferay portal uses the latest Java, J2EE, and Web 2.0 technologies in order to deliver solutions for enterprises across both public and private sectors. Meanwhile, the built-in web content management and a content integration framework allow us to aggregate and publish existing repository content with new content. This helps create web sites and collaborative workspaces, for example, intranets, extranets, team sites, and so on. In addition, the built-in suite of social computing tools allows multiple forums, Wikis, blogs, and Document Libraries to be created and matched to specific user groups or knowledge areas within the same site.

Liferay currently has the following three main functionalities:

1. **Liferay portal**—JSR-168/JSR 286 enterprise portal platform.
2. **Liferay CMS and WCM**—JSR-170 content management system and web content management.
3. **Liferay collaboration and social software**—collaboration software such as blogs, calendar, web mail, message boards, polls, RSS feeds, Wiki, presence (AJAX chat client, dynamic friend list, activity wall, activity tracker), alert and announcement, and so on.

Generally speaking, a web site built by Liferay might consist of CMS and WCM, a portal, and collaboration and social software.

Liferay portal

As the world's leading open source enterprise portal, Liferay portal provides portal solution for both the public and private sectors. The Liferay portal has at least the following features:

- Runs on all major application servers and Servlet containers, databases, and operating systems, and over 700 deployment combinations.
- Uses the latest in Java, J2EE, and Web 2.0 technologies.
- Uses an open SOA framework.
- JSR-168/JSR-286 compliant.
- Out of the box usability over 60 portlets.
- Personalized pages for all users.
- AJAX-enabled user interface.
- Multilanguage support—localizing up to 22 languages.
- Full LDAP synchronization and secure **Single Sign On (SSO)** support.
- Granular role-based authorizations.

- Control Panel—centralized administration for all content, users, organizations, communities, roles, server resources; full customizability with the ability to hide different parts of the form as desired or add custom ones with portlets.
- Single-click configuration, dynamic drag-and-drop, search and tagging capability, and work from desktop, for example WebDAV.
- Built-in CMS, WCM, Collaboration, and Social Networking.

Liferay CMS and WCM

Liferay's built-in CMS and WCM supports portal-based web content publishing and document/content management. Liferay CMS and WCM have at least the following features:

- **Document Library and Image Gallery**—one central place to aggregate and manage all content.
- **Dynamic virtual hosting**—allows using the same installation of Liferay portal to spin off an infinite number of other portals.
- Publishing workflow, versioning, structured content, XSL content, breadcrumb, navigation and Velocity templates, and WYSIWYG editing for end users.
- **The Asset Publisher portlet**—publishes any piece of content in your portal as though it were a Web Content portal, either through a set of publishing rules or by manual selection.
- **The Web Content portlet** (also called Journal, accessible through the Control Panel)—helps create, edit, and publish articles, as well as article templates for one-click changes in layout. It has built-in workflow, article versioning, search, and metadata.
- **The Web Content List** (called Journal Articles)—displaying a dynamic list of all journal articles for a given community.
- **The Web Content Display** (called Journal Content)—publishes any article from the Journal CMS on a portal page.
- **The Web Content Search portlet**—it's powered by the Apache Lucene search engine; search can be restricted to Journal CMS articles.
- **The Nested Portlets portlet**—allows the users to drag-and-drop portlets into other portlets, making complex page layouts possible.

- **Custom attributes**—adds custom attributes to users and organization forms. It provides a framework to add custom attributes to any ServiceBuilder entity at runtime.
- Page staging, scheduling, and publishing, either locally or remotely.
- **Integration with SharePoint**—implementation of the SharePoint protocol allows to save documents to Liferay as if it were a SharePoint server.

Liferay collaboration and social networking software

Liferay collaboration and social networking software take advantage of the benefits of today's virtualized work environment. The portal ties all of the collaboration functions together with the latest social networking features for a truly dynamic work experience. By using this, you share what you know. It has the following features:

- Blogs, Wikis, mail, calendar, enterprise **Instant Message (IM)**, RSS, and more.
- Micro-format support—calendar and user information can be transferred via Web 2.0 standards. Data in micro formats (hCard, hCalendar, and so on) can be easily used by and integrated with third-party applications.
- Dynamic tagging—tagging web content, documents, message board threads, and more, to dynamically share important or interesting content with other portal users.
- Activity tracking—keeping tabs on the most recent activity on blogs, message boards, Wiki, and other tools.
- Announcements and alerts—broadcasting messages to different groups of users.
- Social networking services.
- Capability to build My Social Office.
- Polls—creating multiple choice polls and keeping track of votes.
- WSRP 2.0 consumer/producer and full 1.0/2.0 specification support.
- Integration with Terracotta DSO: setting up large and clustering environments.

Why Liferay portal?

Generally speaking, portals offer basic benefits that involve personalization, customization, and workflow. **Personalization** means that different people with the same role work differently. Different roles require different information via **customization**. People also have direct access to information and applications they need through **workflow**. Further, customization ensures that people do not miss anything. Liferay portal is one of the most mature portal frameworks in the market. It offers these basic benefits and much more.

Besides the basic benefits mentioned above, Liferay portal provides a number of the key business benefits, some of which have been discussed in the following sections:

A rich, friendly, intuitive, and collaborative end user experience

A good user experience is regarded as a key to capture the highest return on an enterprise portal investment. Liferay portal maximizes the productivity gains of portal users and provides a rich, friendly, intuitive, and collaborative end-user experience.

- **Intuitive**—users can drag-and-drop portlets to customize the experience to the unique preferences of a user or community.
- **Rich**—users can use one of the theme plugins from both the out of the box and the community to change the look and feel of the portal without dealing with complex code. There are more than 60 theme plugins available in the community.
- **Friendly**—community members can have their pages with a user-defined friendly URL. This gives the users a better sense of ownership over the technology. Thus, it enhances the user experience, and moreover, generates users' loyalty.
- **Collaborative**—users can create true communities of users via collaborative tools such as instant messaging, message boards, blogs, Wikis, and so on.

A single point of access to all information

As we know, it will become very annoying to collaborate across business units or among distinct business entities if the end users have to stop and log into the applications every single time. Fortunately, Liferay portal provides users a single point of access to all organization's data, content, web content, and other information from existing **in-house applications** (that is, HR, CRM) as well as **external sources** (for example Alfresco, FatWire, Magnolia, and Vignette). That is, in a web site, users can access all organization's data, content, web content, and other information via a single point of access.

By integration with LDAP, information about users and groups are managed in a centralized server. Liferay portal, portlets, and others applications can share the same users' and groups' information directly.

By integration with SSO, users need to log in once to access all their information needs. For example, after the users have logged in once in the Liferay portal, they can automatically log in the portlets such as Alfresco client, Alfresco content, and other systems.

Moreover, the fine-grained permissioning allows the users to customize and control the user access to sensitive information and functionality. Users get an intuitive frontend, whereas behind the scenes, Liferay portal supports **Enterprise Service Bus (ESB)** such as Service-Mix and Mule technologies. Thus, it simplifies the integration, upgrade, and substitution of disparate applications for the developers.

In addition, Control Panel provides centralized administration for all content, users, organizations, communities, roles, server resources, and more. It has full customizability with the ability to hide different parts of the form as desired, or add custom ones with portlets.

Especially, as a social collaboration solution for the enterprise, My Social Office provides full virtual workspace, streamlines communication, saves time, builds group cohesion, and raises productivity. With My Social Office, all you have to do is log in and work the way you want at your own convenience.

High adaptability to the demands of fast-changing market

Liferay portal provides high adaptability to the demands of a fast-changing market. In general, Liferay portal can grow according to your organizations. For instance, Liferay portal allows **clustering** – the addition of hardware to meet growing usage demands. Thus, the capacity for content and applications is boundless.

On the one hand, Liferay portal integrates with workflow engine such as jBPM and Intalio. A **Workflow engine** is an automation of processes. It involves a combination of human- and machine-based activities, and interaction with applications and tools in particular. By this integration, Liferay portal allows organizations of all sizes to be more agile that makes business processes more dynamic, modular, and adaptable to the demands of fast-changing markets.

On the other hand, Liferay portal was benchmarked using **LogicLibrary's Logiscan**. No matter how your processes change, you can always be confident about the security of your data. For this reason, Liferay portal is benchmarked as one of the most secure portal platforms in the market.

Further, Liferay portal accommodates global business environment with multilingual support. For example, after adding a language portlet to any page, you can quickly select a different localization with one simple click.

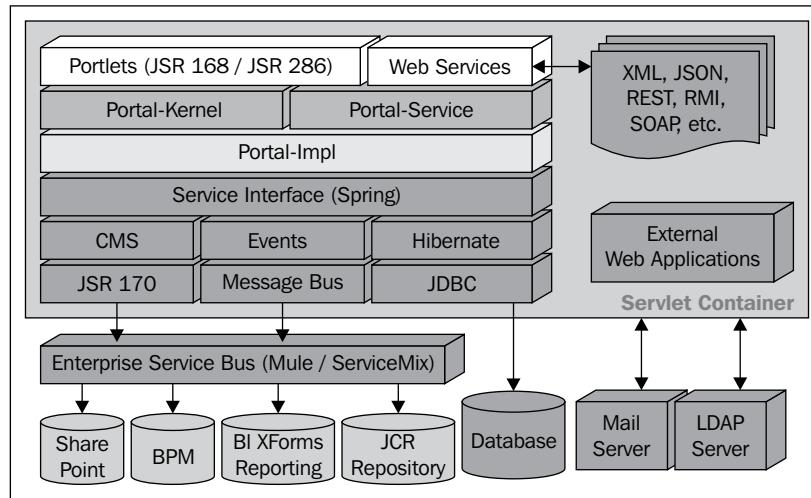
Highest values

Liferay portal also provides the highest value at every level. Liferay portal is based on 100% standards and a set of key technologies. The standards include AJAX and JSON, hCalendar Micro-format and iCalendar, JSR-127 and JSR-314 (compliant JSF), JSR-168 and JSR-286 (compliant portlet), JSR-170 (Content Repository API for CMS), JSR-220 (Hibernate), OpenSearch, WebDAV, WSRP, and so on. The technologies involve ICEFaces, jQuery JavaScript Framework, Ruby on Rails, PHP, Spring and AOP, Struts and Tiles, Velocity, FreeMarker, and so on. More importantly, you have the freedom to integrate with your favorite content repository such as Liferay CMS, Alfresco, FatWire, Magnolia, Vignette, and so on. In short, Liferay portal strengthens the compliance to the standards and reduces the risk of investment.

More interestingly, Liferay portal leverages the existing IT environment and works with any application server, database server, or operating system with over 700 deployment configurations. Surely, your existing technology investments are not discarded. Further, the future changes will not require an overhaul of an existing Liferay portal installation.

Architecture and framework

The most important aspect of any portal is its underlying architecture. Liferay portal architecture supports high availability for mission-critical applications using clustering, fully distributed cache, and replication support across multiple servers. The following figure depicts various architectural layers and functionality of portlets:



Service oriented architecture: SOA

Liferay portal uses **SOA** design principles throughout, and provides the tools and framework to extend SOA to other enterprise applications. Under Liferay enterprise architecture, not only can the users access the portal from traditional and wireless devices, but the developers can also access it from the exposed APIs via REST, SOAP, RMI, XML-RPC, XML, JSON, Hessian, Burlap, and custom tunnel classes.

Liferay portal is designed to deploy portlets that adhere to the portlet API compliant with both JSR-168 and JSR-286. A set of useful portlets are bundled with the portal such as Image Gallery, Document Library, Calendar, Message Boards, Blogs, Wikis, and so on. They can be used as examples for adding custom portlets.

In a word, the key features of Liferay include using SOA design principles throughout, reliable security, integrated with SSO and LDAP, multitier and limitless clustering, high availability, caching pages, dynamic virtual hosting, and so on.

Enterprise service bus: ESB

The **Enterprise Service Bus (ESB)** is a central connection manager that allows applications and services to be added quickly to an enterprise infrastructure. When an application needs to be replaced, it can be easily disconnected from the bus at a single point. Liferay portal uses Mule or ServiceMix as ESB.

Through ESB, the portal could integrate with SharePoint, BPM (such as jBPM workflow engine, Intalio | BPMS engine), BI Xforms reporting, JCR repository, and so on. It supports JSR 170 for content management system with integration of JCR repository, such as Jackrabbit. It also uses Hibernate and JDBC to connect to any databases. Further, it supports events system with asynchronous messaging and lightweight message bus.

Liferay portal uses Spring framework for its business and data services layers. It also uses Spring framework for its transaction management. Based on service interfaces (Spring framework), `Portal-Impl` is implemented and is exposed only for the internal usage for the Extension environment, for example. `Portal-Kernel` and `Portal-Service` are provided for the external usage (and for the internal usage, either) for Plugins SDK environment, for example. Custom portlets, both JSR-168 and JSR-286, and web services could be built based on the `Portal-Kernel` and `Portal-Service`.

In addition, the Web 2.0 Mail portlet and the Web 2.0 Chat portlet are supported as well. More interestingly, scheduled staging and remote staging and publishing are served as a foundation through tunnel web for web content management and publishing.

Liferay portal supports web services to make it easy for different applications in enterprise to communicate with each other. Java, .NET, and proprietary applications can work together easily because web services use XML standards. It also supports REST-style JSON web services for lightweight, maintainable code, and supports AJAX-based user interfaces.

Liferay portal uses industry-standard, government-grade encryption technologies, including advanced algorithms such as DES, MD5, and RSA. Liferay was benchmarked as one of the most secure portal platforms using LogicLibrary's Logiscan suite. Liferay offers customizable Single Sign-On with Yale CAS, JAAS, LDAP, NTLM, Netegrity, Microsoft Exchange, and more. Open ID, Yale CAS, Siteminder, and OpenSSO integration are offered out of the box.

In short, Liferay portal uses the ESB in general to provide an abstraction layer on top of an implementation of an enterprise messaging system. It allows integration architects to exploit the value of messaging without writing code.

Portal development strategies

Liferay portal is extensible at least at the following three levels:

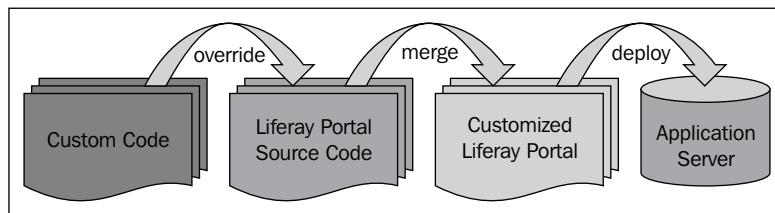
- Plugins SDK environment
- Extension environment
- Liferay portal source code

In general, each level of extensibility offers a different compromise of flexibility with different migration requirements to the future version.

Extension environment

The Extension environment provides capability to customize Liferay portal completely. As it is an environment which extends Liferay portal development environment, it has a name **Extension**, or **Ext**. By the name Ext, we can modify internal portlets or call the out of the box portlets. Moreover, we can override the JSP files of portal and the out of the box portlets. This kind of customization is kept separate from the Liferay portal source code. That is, Liferay portal source code does not have to be modified and a clear upgrade path is available in Ext.

As shown in following figure, *Custom Code* will override *Liferay Portal Source Code* in Ext only. In the deployment process, custom code is merged with *Liferay Portal Source code* in Ext. That is, developers override the Liferay portal source code. Moreover, the custom code and Liferay portal source code will be constructed as *Customized Liferay Portal* in Ext first, and then the customized Liferay portal will be deployed from Ext to the *Application Server*.



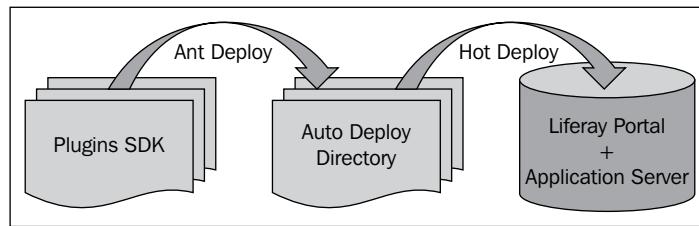
During customization, we could use ServiceBuilder to generate models and services. In general, ServiceBuilder is a code generator that uses an XML descriptor. For a given XML file `service.xml`, it will generate SQL for creating tables, Java Beans, Hibernate configuration, spring configuration, Axis Web Service, and JSON JavaScript Interface, and so on.

JSP files of the portal and the out of the box portlets can be overridden with custom versions in Ext. Note that Ext is used for customizing Liferay portal only, as portlets written in Ext are not hot-deployable. Moreover, Ext is a monolithic environment.

Plugins SDK environment

Plugins SDK is a simple environment for the development of Liferay plugins, including themes, layout templates, portlets, hooks, and webs (that is, web applications). It provides the capability to create hot-deployable portlets, themes, layout templates, hooks, and webs.

How does it work? As shown in following figure, *Plugins SDK* provides environment for developers to first build themes, layout templates, portlets, hooks, or webs. Afterwards, it uses the Ant target *Deploy* to form WAR file and copy it to the *Auto Deploy Directory*. Then, *Liferay Portal* – together with an *Application Server* – will detect any WAR files in the auto hot-deploy folder, and automatically extract the WAR files into the *Application Server* deployment folder.



Portlets go in the `/portlets` folder, themes go in the `/themes` folder, layout templates go in the `/layouttpl` folder, web applications go in the `/webs` folder, and hooks go in the `/hooks` folder. By the way, Ant targets are used to build and deploy plugins to local application server. It is surely possible to deploy plugins directly to remote application server via custom Ant targets.

Especially, the portlets developed in Plugins SDK may only import classes from the portal API (Portal-Kernel and Portal-Service) and other JAR files contained in the specific portlet `/WEB-INF/lib` folder. This forces the portlets to rely completely on the Portal API and not to depend on implementation classes defined in `Portal-Impl`.

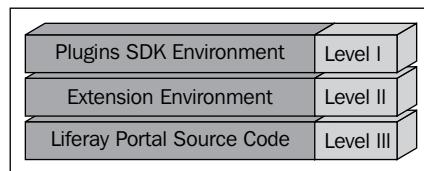
As you can see, portlets can make use of any application framework that Liferay supports – **Model-View-Controller (MVC)** frameworks. Here is a list of a few application frameworks: Struts, Spring, Tapestry, JSF, Wicket, and so on.

In addition, as mentioned above, Liferay portal can also integrate with certain web applications as webs, for instance `solr-web`—search engine integration plugin; `jbpmp-web`—workflow engine integration plugin; `mule-web` and `servicemix-web`—ESB integration plugins.

Development strategies

As mentioned earlier, there are at least two development environments: Ext and Plugins SDK. Thus, you may ask: Which kind of development environment is suitable for our requirements? When should we use Ext and when should we use Plugins SDK, or even Liferay portal source code? Let's take a deep look at the development strategies.

As shown in following figure, Liferay Portal is extensible at least at three levels, for example *Plugins SDK Environment (Level I)*, *Extension Environment (Level II)*, and *Liferay Portal Source Code (Level III)*. As you can see, each level of extensibility offers a different compromise of flexibility with different migration requirements to the future version. Thus, we need to choose the appropriate level for the requirements at hand which allows for easier future maintainability.



In **Level I**, we can develop portlets, themes, layout templates, hooks, and webs as independent software components. Moreover, these plugins can be distributed and deployed as WAR files, and can be organized in plugin repositories. Liferay portal provides Plugins SDK to help us with the development of these plugins.

In addition, portlets developed in Plugins SDK can only import classes from the portal API (Portal-Kernel and Portal-Service), and not `Portal-Impl`. That is, portlet development in Plugins SDK does not even touch portal properties, language properties, and JSP files related to `Portal-Impl`. Fortunately, Hooks provides the capability to hook up portal properties, language properties, and JSP files related to `Portal-Impl`.

In **Level II**, we can manage configuration files, custom source code, custom JSP files, and modified JSP files related to the `Portal-Impl`. That is, Ext provides different sublevels (for example, configuration files, custom source code, custom JSP files, and modified JSP files) of extensibility.

Among the configuration files, `portal-ext.properties` has the main configuration options: layouts, deployment, themes, Hibernate, cache, instance settings, users, groups, language, session, auth, integration, events, and so on. Meanwhile, the `system-ext.properties` file is a convenient way to provide and extend the Java System properties used by Liferay portal. We can also create custom classes for the most common extensibility which needs to be configured through the `portal.properties` file. Examples are authentication chain, upgrade and verification processes, deployment processes, database access and caching, user fields generation and validation, session events, permissions, model listeners, and so on.

For custom source code, we can use Spring-based dependency injection mechanisms configured in the `ext-spring.xml` file; add the Servlet extended in the `web.xml` file; add the Struts action extended in the `struts-config.xml` file; and moreover, create portlets that access `Portal-Impl`, or events extending its models and services.

For custom JSP files and modified JSP files, we can customize any of the JSP files used by the out of the box portlets and management tools. This is a very flexible extension mechanism.

Without a doubt, it is easier to develop portlets in Ext, where, you can easily access and use all Portal APIs, taglibs, JSP files, and almost everything. This is not the case in Plugins SDK at the time of writing.

In **Level III**, we can modify the Liferay portal source code. This approach can only be used for sponsored development. That is, we develop specific features for specific projects first and then contribute back to Liferay portal source code.

In brief, if your requirements are related to customize and/or extend the `Portal-Impl` (for example UI changing, LDAP import algorithms, Document Library lock mechanism, forms for user registration or organization creation, integration, modifying the out of the box portlets, and so on), you should use Ext. Otherwise, it is better to use Plugins SDK. Note that with Hooks you can hook up portal properties, language properties, and JSP files related to the `Portal-Impl`.

Summary

This chapter looked at what Liferay portal is and why we should use it. Then it introduced the Liferay portal architecture and framework. Finally, it discussed portal development strategies in detail. Liferay portal can be extensible at three levels—Plugins SDK environment, Extension environment, and Liferay portal source code. Both Plugins SDK environment and Extension environment will be helpful to build custom system on top of the Liferay portal.

In the next chapter, we're going to work with the JSR-286 portlets using real examples.

2

Working with JSR-286 Portlets

Liferay portal supports Java Portlet Specification 2.0 (that is, JSR-286). Liferay portal, starting from version 5.1, is a 100 % compliant implementation of the JSR-286. In this chapter, we will be discussing what portlets and portlets are, what it means for them to be JSR-286 compliant, how to use and extend them smoothly, and how to coordinate inter-portlet communication. For these purposes, you can assume that we will always be talking about JSR-286 compliant portlets.

The JSR-286 has grown up, and it has evolved. It now allows 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. You may skip this chapter and go to the next chapter directly if you are already familiar with the JSR-286.

By the end of this chapter, you will have learned how to:

- Experience Liferay portal and portlets
- Use JSR-286 portlets
- Employ portlet configuration, context, request, response, and preferences
- Extend JSR-286 portlets
- Serve resources
- Coordinate portlets

Experiencing Liferay portal and portlets

As an administrator at the Enterprise "Palm-Tree Publications", you can first experience the Liferay portal locally. Simply log in to your local Liferay portal. You will see the portal page interface similar to that in the following screenshot. Generally, a portal page is made up of a set of portlets, for example, **Reports**, **Language**, **Sign In**, and so on. Liferay portal is running locally with the URL: <http://localhost:8080/web/admin/home>.



At this point, you probably have some questions: What is a portal? What is a portlet? Why do we need JSR-286 portlets?

What's JSR-286?

 **JSR** means **Java Specification Request**. JSR-286 means Portlet Specification 2.0. For more details about portlet specification 2.0, refer to <http://jcp.org/aboutJava/communityprocess/final/jsr286/index.html>.

What is a portal?

The above screenshot shows, the portal page of the Liferay portal for the user **Admin**. This page is made up of the header, footer, navigation, and a set of portlets with a specific layout template, for example, **Reports**, **Language**, **Sign In**, and so on.

In general, a portal (otherwise known as a web portal) is a web-based application that, typically, provides personalization, authentication, and content aggregation from different sources, and hosts the presentation layer of information systems. Aggregation is the action of integrating content from different sources within a web page. A portal may have sophisticated personalization features to provide customized content to users. Portal pages may have a different set of portlets creating content for different users.

What is a portlet?

As shown in the following screenshot, you will see a set of portlets, for example, **Reports**, **Language**, and **Sign In**.



In brief, a portlet, for example **Reports**, is an application that provides a specific piece of content (in this case, the link **Test123**) to be included as part of a portal page, for example, **Home Welcome Page**. It is managed by a portlet container that processes requests and generates dynamic content. Actually, the portlets are used by portals as pluggable user interface components.

The content generated by a portlet is also called a fragment. A fragment is a piece of markup (for example, HTML, XHTML, WML, and so on) adhering to certain rules and can be aggregated with other fragments to form a complete document. The content of a portlet (for example, **Test123**) for the portlet **Reports** is normally aggregated with the content of the other portlets to form the portal page. The life cycle of a portlet is managed by the portlet container.

Obviously, web clients interact with portlets via a request-response paradigm implemented by the portal. Normally, users interact with the content produced by the portlets (for example, by following links **test123** for the portlet **Reports** or submitting forms) resulting in portlet actions being received by the portal, which are forwarded by it to the portlets targeted by the user's interactions.

The content generated by a portlet may vary from one user to another depending on the user configuration for the portlet. For example, the content generated by the portlet **Reports** was the link **Test123**, while the content generated by the portlet, **Language** was the links—language icons, for example, American English, Chinese, and so on.

What is a portlet container?

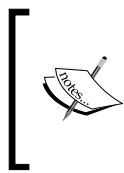
After having portlets in portal pages, we need a portlet container to manage portlets. A portlet container runs portlets, provides them with the required runtime environment, and manages their life cycles. It also provides persistent storage for portlet preferences. A portlet container receives requests from the portal to execute requests on the portlets hosted by it.

Normally, a portlet container is not responsible for aggregating the content produced by the portlets. It is the responsibility of the portal to handle the aggregation. A portal and a portlet container can be built together as a single component of an application suite or as two separate components of a portal application.

The following is a typical sequence of events, initiated when you access the portal page, for example, **Home Welcome Page**.

- A client (for example, the administrator), after being authenticated, makes an HTTP request to the portal
- The request is received by the portal (for example, Liferay portal)
- The portal determines if the request contains an action (for example, process action, render) targeted at the portlets (for example, **Language** associated with the portal page **Home Welcome Page**)
- If there is an action targeted at a portlet, for example, **Language**, the portal requests the portlet container to invoke the portlet to process the action
- A portal invokes portlets (for example, **Language, Reports**) and so on, through the portlet container
- The portal aggregates the output of the portlets in the portal page to the client (for example, the administrator)

Liferay portal has its own portlet container – a logical component for handling the life cycle and modes of the portlets. Interestingly, Liferay portal can also integrate with other portlet containers by replacing its own portlet container, for example, OpenPortal Portlet Container.



OpenPortal Portlet Container is a fully compliant implementation of the Portlet 2.0 (JSR286) specification, implementing many of the optional features such as expiration and validation caching, support for alias in events and public render parameters, and support for wild card in events. Refer to <https://portlet-container.dev.java.net>.

Why JSR-286 portlets?

Why do we need Java Portlet Specification 2.0 (that is, JSR-286)? The objective of JSR-286 is to make the portlet API more mature than that of JSR-168 in order to incorporate essential features. We can use JSR-286 to create view objects after request processing, to set them as request attributes, and to employ the same for display purposes. To summarize, here are the major features of JSR-286:

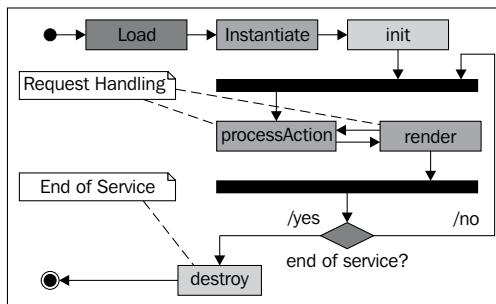
- Coordination (events support, sharing session beyond portlet application, sharing render parameters across portlets, and so on)
- **WSRP (Web Services for Remote Portlets) 2.0 alignment**
- Better support for Web Frameworks (for example, JSF, Struts, Spring, Wicket, and so on)
- Serving dynamically-generated resources directly through portlets
- Serving AJAX or JSON data directly through portlets (while JSR-168 serves the AJAX data using an additional servlet)

Using JSR-286 portlets

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.

Understanding portlet life cycle

As mentioned earlier, a portlet is managed through a life cycle that defines how it is loaded, instantiated, and initialized, how it handles requests from clients, and how it is taken out of service. The life cycle of a portlet is expressed through the `init`, `processAction`, `render`, and `destroy` methods of the `Portlet` interface, as shown in following figure:

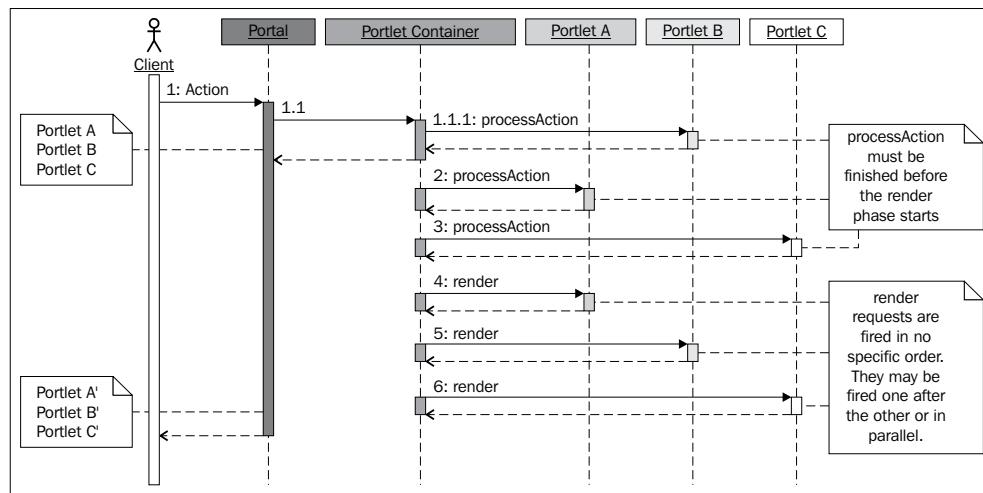


The **Portlet** interface is called by the portlet container to indicate to a portlet that it is being placed into service. The following are the methods specified in the **Portlet** interface:

```
init(PortletConfig);  
processAction(ActionRequest, ActionResponse);  
render(RenderRequest, RenderResponse);  
destroy();
```

Here are the detailed explanations of the phases the portlet goes through when it is rendered:

- In the **Loading and Instantiation**, the portlet container is responsible for loading and instantiating the portlets. The loading and instantiation can occur when the portlet container starts the portlet application, or is delayed until the portlet container determines whether the portlet is needed to service a request.
- In the **Initialization** (the method `init`), after the portlet object is instantiated, the portlet container must initialize the portlet before invoking it to handle requests. Initialization is provided so that portlets can initialize costly resources (for example, backend connections), and perform other one-time activities.
- In the **Request Handling**, after a portlet object is properly initialized, the portlet container may invoke the portlet to handle client requests. As shown in the following figure, the portlet interface defines two methods for handling requests – the `processAction` method and the `render` method.



Normally, a client request triggered by an action URL (related to the method `processAction`) translates into one action request and many render requests – one per portlet in the portal page. While a client request triggered by a render URL (related to the `render` method) translates into many render requests – one per portlet in the portal page.

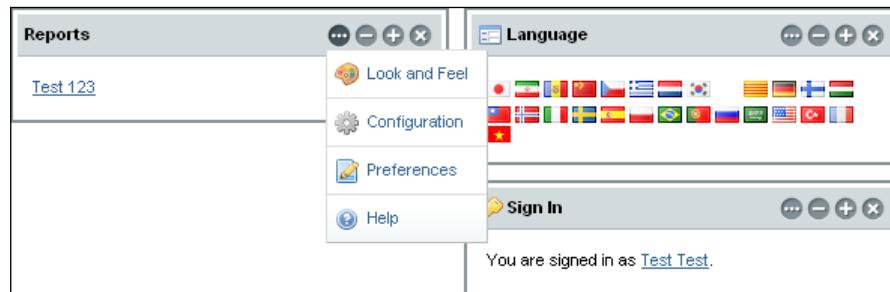
Typically, in response to an action request, a portlet updates the state based on the information sent in the action request parameters. Such request parameters are often contained in the URLs created by portlets, also called **portlet URLs**. Portlet URLs could either be action URLs or render URLs.

Generally, during a render request, portlets generate the content based on their current state.

- In the **end of service**, the portlet container is not required to keep a portlet loaded for any particular period of time. A portlet object may be kept active in a portlet container for a period of a millisecond, for the lifetime of the portlet container (which could be a number of days, months, or years), or any amount of time in between. When the portlet container determines that a portlet should be removed from service, it calls the `destroy` method of the Portlet interface to allow the portlet to release any resources it is using and save any persistent state. For instance, the portal will clean up any resources that it holds (including memory, file handles, and threads) in the implementation of the method `destroy`.

Utilizing portlet modes

A portlet mode advises the portlet what task it should perform and what content it should generate. As shown in the following screenshot, the **Reports** portlet has three modes: view mode which is generated by default; edit mode named **Preferences** with an edit icon; and help mode named **Help** with a help icon. These three are the typical modes that Liferay portlets use.



In general, a portlet mode indicates the function a portlet is performing. Normally, portlets can perform different tasks, and further, create different content depending on the function they are performing at present. When invoking a portlet, the portlet container will provide the current portlet mode to the portlet. Logically, portlets can change their portlet mode programmatically when processing an action request, that is, the `processAction` method.

By default, there are three portlet modes, `VIEW`, `EDIT`, and `HELP`. The `PortletMode` class defines constants for these portlet modes as follows:

```
PortletMode VIEW = new PortletMode ("view");
PortletMode EDIT = new PortletMode ("edit");
PortletMode HELP = new PortletMode ("help");
```

As shown in the code above, the expected functionality for a portlet in the `VIEW` mode is to generate a markup that reflects the current state of the portlet. For example, the `VIEW` mode of a portlet may include one or more screens that the user can navigate and interact with, or it may consist of static content that does not require any user interaction.

Within the `EDIT` portlet mode, a portlet should provide content and logic that lets a user customize the behavior of the portlet. The `EDIT` mode may include one or more screens among which the users can navigate to enter their customization data.

When in the `HELP` portlet mode, a portlet should provide information about the portlet. This information could be a simple help screen explaining the entire portlet in coherent text, or it could be context-sensitive help.

For a portlet, the availability of the portlet modes may be restricted to specific user roles by Liferay portal. For example, an anonymous user such as `guest` could be allowed to use the `VIEW` and `HELP` portlet modes, but only an authenticated user, for example, `Palm Tree`, could use the `EDIT` portlet mode.

The following is the sample code for specifying different modes in `portlet.xml` for **Reports**:

```
<portlet>
  <init-param>
    <name>view-action</name><value>/ext/reports/view</value>
  </init-param>
  <init-param>
    <name>edit-action</name><value>/ext/reports/edit</value>
  </init-param>
  <init-param>
    <name>help-action</name><value>/ext/reports/help</value>
  </init-param>
```

```

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

```

As shown in the code above, it first initiates the parameters available in the portlet `PortletConfig` instance, for example, `view-action`, `edit-action`, and `help-action`. Then it indicates that the portlet supports HTML markup as MIME type value `text/html`. Finally, it shows that the portlet supports the `view`, `edit`, and the `help` modes.

In addition, you can have custom portlet modes. In Liferay portal, you can define custom portlet modes for a specific functionality. The portal will manage these custom modes. However, you may define additional modes for your customized portlets that don't need to be managed by the portal. In this case, you need to provide implementation to manage these modes.

Employing window states

Window states indicate the amount of portal page space that will be assigned to a portlet. As shown in the following screenshot, the **Reports** portlet has three states: normal state which is generated by default, minimized state when **minimize** icon is clicked, and maximized state when **maximize** icon is clicked.



Generally speaking, a window state is an indicator of the amount of portal page space that will be assigned to the content generated by a portlet. When invoking a portlet, the portlet container provides the current window state to the portlet. The portlet may use the window state to decide how much information it should render. Of course, portlets can programmatically change their window states when processing an action request.

By default, there are three window states: NORMAL, MAXIMIZED, and MINIMIZED. The `WindowState` class defines constants for these window states that are as follows:

```
WindowState NORMAL = new WindowState ("normal");
WindowState MAXIMIZED = new WindowState ("maximized");
WindowState MINIMIZED = new WindowState ("minimized");
```

As shown in code above, The NORMAL window state indicates that a portlet may be sharing the page with other portlets. It may also indicate that the target device has limited display capabilities. Therefore, a portlet should restrict the size of its rendered output in this window state.

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 the other portlets in the portal page. A portlet may generate richer content when its window state is MAXIMIZED.

When a portlet is in the MINIMIZED window state, the portlet should render only minimal output, or should render no output.



This screenshot shows a sample URL for the portlet **Reports** with MAXIMIZED state.

```
web/guest/home?p_p_id=EXT_1&p_p_lifecycle=0&p_p_state=maximized&p_p_col_id=column-1&p_p_col_count=1
```

This code shows that the **Reports** portlet is displayed in a MAXIMIZED state. `p_p_id` is the name of the portlet, `p_p_lifecycle` is set to one of the following: 0 = render, 1 = action, 2 = resource, `p_p_state` is the value of window state, `p_p_col_id` is the value of the column related to the layout template, and `p_p_col_count` is the value of the column count.

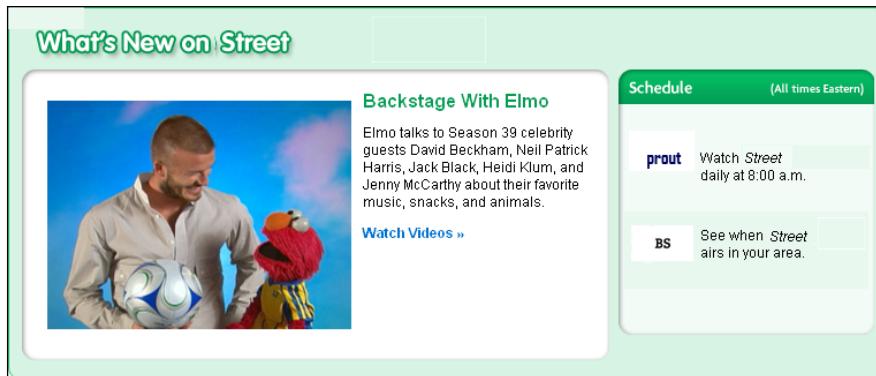
The following is an example URL for the portlet **Reports** with MAXIMIZED state:

```
web/guest/home?p_p_id=EXT_1&p_p_lifecycle=0&p_p_state=normal
```

In addition, you can have custom window states in Liferay portal. Note that the portlets can use only window states that are defined by the Liferay portal.

What's the difference between a portlet and a servlet?

First of all, let's consider a scenario. There is a page with special links as shown in the following screenshot. The first link **Prout** will bring the users to the Prout web site while the second link **BS** will bring users to the BS web site. But before going to third-party web sites, such as Prout or BS, we can ask the users to decide whether to go to third-party web sites or not through a bridge page.



The following are the sample links of bridges pages:

```
/bridgepage?p_p_id=bridgePagePortlet_WAR_bookportlets&p_p_lifecycle=0&pageURL=http://www.sproutonline.com/sprout/home/jump.aspx
```

This code shows a bridge page with the target source (that is, **Watch Book Street daily at 8:00 a.m.** in the previous screenshot) plus a URL: <http://www.sproutonline.com/sprout/home/jump.aspx>;

```
/bridgepage?p_p_id=bridgePagePortlet_WAR_bookportlets&p_p_lifecycle=0&pageURL=http://pbskids.org/tvschedules/localizer.html?dest=/book/index.html&nola=SESA
```

This code shows a bridge page with a target source (that is, **See when Book Street airs in your area** in the preceding screenshot) plus a URL: <http://pbskids.org/tvschedules/localizer.html?dest=/book/index.html&nola=SESA>.

As shown in following screenshot, the bridge page explains to the user that he's leaving the site, and gives the user options to cancel or carry on.

The bridge page is developed as a portlet, originally, `bridge-page-portlet` while the redirecting function is implemented by the Servlet `bridge-street-servlet`. What's the difference between a portlet and a servlet? Let's take a deeper look at the relationship between a portlet and a servlet.



Use cookies, document head section elements, and HTTP headers

In the JSR-286, you can set cookies, document head section elements (for example, HTML elements such as meta, link, or style), and HTTP headers (for example, application-specific Pragma headers). JSR-286 added a two-part render call as follows:

```
String RENDER_HEADERS = "RENDER_HEADERS";  
String RENDER_MARKUP = "RENDER_MARKUP";
```

The `RENDER_HEADERS` part allows the portlet to return content outside the current portlet window such as portlet title, preferred next possible portlet modes, cookies, document head section elements, and HTTP headers while the `RENDER_MARKUP` part allows the portlet to return its normal markup.

You can also set cookies at the response of each life cycle method (`processAction`, `processEvent`, `render`, and `serveResource`) of `PortletResponse` with this code:

```
void addProperty(javax.servlet.http.Cookie cookie);
```

The cookie can then be accessed in all life cycle methods of `PortletRequest` using:

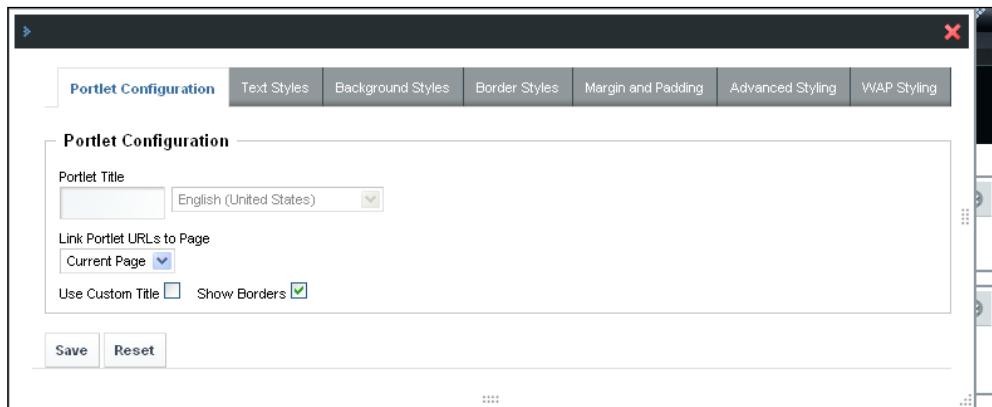
```
javax.servlet.http.Cookie[] getCookies();
```

Using cookies in portlets is different from using cookies in servlets in these ways. You can set cookies in the `render` method. The `renderHeaders` option should be turned on and the cookies should be set in the `RENDER_HEADERS` part by using overriding `doHeaders()` of portlets such as `GenericPortlet`.

Cookies may not be accessible at the client side as they are stored at the portal server, or they are put in a different namespace when the portlet runs as a remote portlet through WSRP. Moreover, cookies are not guaranteed to be shared across different portlets.

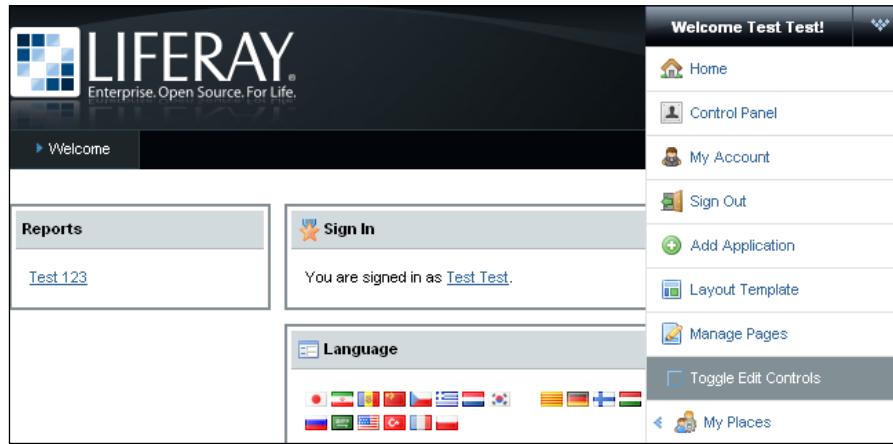
Employing portlet configuration, context, request, response, and preferences

Before employing portlet configuration, context, request, response, and preferences, we are going to see examples for specific portlets. Suppose you are using the **Reports** portlet and you want to customize it by changing the title and hiding the border.



As shown in this screenshot, to change the title of the portlet and hide the border of the portlet, first click on the **Look and Feel** icon that appears in the **Reports** portlet. Then simply enter a title, select a language, select the checkbox **Use Custom Title** (which, by default, is unchecked), and deselect the checkbox **Show Borders** (which, by default, is selected). Finally, click on the **Save** button when you are ready.

In addition, we are planning to hide the edit controls—viewing the **Look and Feel** of portlets without the edit controls. Thus you can turn the **Toggle Edit Controls** on or off. That is, you can choose either to display all edit controls or to hide them. Under the **Toggle Edit Controls**, you can see the portlets as shown in following screenshot:



Liferay portal provides the ability to change the look and feel of portlets dynamically with the following possibilities:

- **Portlet configuration**—using custom title, showing borders, selecting languages for title, and so on
- **Text style**—font, size, color, alignment, text decoration, word spacing, line height, letter spacing, and so on
- **Background style**—background color, and so on;
- **Border style**—border width, border style, border color, and so on
- **Margin and padding**—padding, margin, and so on
- **Advanced styling**—entering in your custom CSS

You can also customize the portlet via programming. For example, normal users do not have any access to edit controls and, in addition, they cannot see the border. But they can configure the portlet, for example, Home-Host (that is, **Elmo Day**), as shown in following screenshot:



The following is the sample code for the portlet Home-Host:

```
portletDisplay.setShowConfigurationIcon(false);
portletDisplay.setShowCloseIcon(false);
portletDisplay.setShowMoveIcon(false);
portletDisplay.setShowPortletCssIcon(false);
```

As you can see, you can configure the portlet—either hide or display the edit controls and the border. What's happening? All these features are related to the portlet configuration, context, request, response, and preferences. Let's take a deeper look at them.

Using portlet configuration

The configuration holds information about the portlet that is valid for all of the users retrieved from the portlet definition in the deployment descriptor. Portlet configuration is provided by the `PortletConfig` interface and the portlet can read only the configuration data.

For example, the **Reports** portlet has a portlet name, a display name, and a portlet class in `portlet.xml`, as shown in the following code:

```
<portlet>
  <portlet-name>EXT_1</portlet-name>
  <display-name>Reports</display-name>
  <portlet-class>
    com.ext.portlet.reports.ReportsPortlet
  </portlet-class>
```

As seen in the previous code, the configuration information contains the portlet name, the portlet initialization parameters, the portlet resource bundle, and the portlet application context. They are defined in `PortletConfig.java` in the following manner:

```
Map<String, String []> getContainerRuntimeOptions();
String getInitParameter(String name);
Enumeration<String> getInitParameterNames();
PortletContext getPortletContext ();
String getPortletName ();
ResourceBundle getResourceBundle(java.util.Locale locale);
Enumeration<Locale> getSupportedLocales();
```

This code shows that the `PortletConfig` object provides the portlet object with information to be used during initialization. It also provides access to the portlet context, default event namespace, public render parameter names, and the resource bundle that provides title-bar resources, and so on.

Employing portlet context

Using the context, a portlet can access the portlet log, and obtain URL references to resources. The `PortletContext` interface defines a portlet view of the portlet container. The `PortletContext` also makes resources available to the portlet. The following is sample code from the `PortletContext` interface:

```
String getServerInfo ();
InputStream getResourceAsStream (String path);
int getMajorVersion ();
int getMinorVersion ();
String getMimeType(String file);
String getRealPath(String path);
Set<String> getResourcePaths(String path);
URL getResource(String path);
ObjectgetAttribute(String name);
```

A portlet, for example **Reports**, obtains a `PortletContext` object from the request object using `getPortletContext` method.

Using portlet request

The `PortletRequest` defines the base interface to provide client request information to a portlet. The portlet container uses two specialized versions of this interface when invoking a portlet— `ActionRequest` and `RenderRequest` (both extend `PortletRequest`). The portlet container creates these objects, and further passes them as arguments to the portlet, and the `processAction` and `render` methods. The following is the same code for the constants from the `PortletRequest` interface:

```
String USER_INFO = "javax.portlet.userinfo";
String LIFECYCLE_PHASE = "javax.portlet.lifecycle_phase";
String RENDER_PART = "javax.portlet.render_part";
String RENDER_HEADERS = "RENDER_HEADERS";
```

As shown in the previous code, the request objects encapsulate all the information about the client request, parameters, request content data, portlet mode, window state, and so on. Request object is passed to the `processAction`, `processEvent`, `serveResource`, and `render` methods of the portlet. The following is the same code for a portlet request of the **Reports** portlet.

```
public ActionForward render(
ActionMapping mapping, ActionForm form, PortletConfig portletConfig,
RenderRequest req, RenderResponse res) {
String title = req.getParameter("title");
}
```

Employing portlet response

The `PortletResponse` defines the base interface to assist a portlet in creating and sending a response to the client. The portlet container uses specialized versions of this interface when invoking a portlet. The portlet container creates these objects. It then passes them as arguments to the `processAction`, `processEvent`, `serveResource`, and the `render` methods of the portlet. Here is the part of methods of the `PortletResponse` interface:

```
void addProperty(Cookie cookie);
void addProperty(String key, Element element);
void addProperty(String key, String value);
Element createElement(String tagName);
String encodeURL(String path);
String getNamespace();
void setProperty(String key, String value);
```

As shown in this code, the response objects encapsulate all the information to be returned from the portlet to the portlet container during a request – a redirection, a portlet mode change, title, content, and so on. The Liferay portal or the portlet container will use this information to construct the response to be returned to the client. A response object is passed to the `processAction`, `processEvent`, `serveResource`, and the `render` methods of the portlet.

Working with portlet preference

The portlet preferences are intended to store basic configuration data for portlets. The `PortletPreferences` interface allows the portlet to store configuration data, not to replace the general-purpose databases. The following is the list of methods from the `PortletPreferences` interface:

```
boolean isReadOnly(String key);
String getValue(String key, String def);
String[] getValues(String key, String[] def);
void setValue(String key, String value);
void setValues(String key, String[] values);
Enumeration<String> getNames();
Map<String, String[]> getMap();
void reset(String key);
void store();
```

As shown in the previous code, changes are persisted when the `store` method is called. Note that the `store` method can be invoked only within the scope of a `processAction` call. Otherwise, changes that are not persisted are discarded when the `processAction` or the `render` method ends. The following is a simple code for the preferences of the **Reports** portlet in `portlet.xml`:

```
<portlet><portlet-preferences>
  <preference><name>test</name><value>123</value>
    <read-only>true</read-only>
  </preference>
</portlet-preferences>
</portlet>
```

This code shows that the portlet Reports has preferences named `test` and value `123`, and a `read-only` tag with a value `false`. Normally, there are two different types of preferences: `modifiable` and `ready-only`. `Modifiable` preferences can be changed by the portlet in any standard portlet mode (for example, `EDIT`, `HELP`, and `VIEW`). By default, preference is `modifiable`. While `read-only` preferences cannot be changed by the portlet in any standard portlet mode, they may be changed by administrative modes. Preferences would be `read-only` if they are defined in the deployment descriptor with `read-only` set to `true`, or if the portlet container restricts write access.

Extending JSR-286 portlets

The JSR-286 is made extensible so that Liferay portal can add features and functionalities in the frameworks on top of JSR-286 in a noninvasive and container-independent manner. Let's take a deeper look at the features for extending the JSR-286.

Using portlet filters

Filter functionality allows us to plug filters around any life cycle call of the portlet. `PortletFilter` is the base interface for all portlet filters such as `ActionFilter`, `EventFilter`, `RenderFilter`, and `ResourceFilter`. It provides the `init` and `destroy` life cycle methods for putting a portlet filter into and out of service as follows:

```
void init(FilterConfig filterConfig);
void destroy();
```

All portlet filters such as `ActionFilter`, `EventFilter`, `RenderFilter`, and `ResourceFilter` extend the `PortletFilter` with one method as follows:

```
void doFilter(EventRequest request, EventResponse response,
  FilterChain chain);
```

The `doFilter` method of the filter is called by the portlet container each time an action request-response pair is passed through the chain due to a client request for a portlet method at the end of the chain. The `FilterChain` passed into this method allows the filter to pass on the action request and response to the next component in the chain.

The portlet filter functionality allows users to plug filters around any life cycle call of the portlet. Not only can the filters do pre-processing or post-processing, but they can also modify or wrap the request and response objects that are passed to the portlet. The following items show typical applications of portlet filters:

- Passing information from additional sources to the portlet as attributes or parameters
- Output filtering for security enforcement or markup compliance
- Collecting diagnostic information
- Bridging among web application frameworks
- Servlet life cycle listener

The portlet filter programming model provides a filter-mapping element where you prescribe the portlets to which the filter should be applied. The order of the filter-mapping elements in the deployment descriptor also defines the order of the filters that are applied to the portlet. For example, in order to integrate CAS SSO, we can use `CAS Filter` and `CAS Filter Mapping` in `web.xml` as follows:

```
// filter declaration
<filter>
    <filter-name>CAS Filter</filter-name>
    <filter-class>
        com.liferay.portal.servlet.filters.sso.cas.CASFilter
    </filter-class>
</filter>
// filter mapping
<filter-mapping>
    <filter-name>CAS Filter</filter-name>
    <url-pattern>/c/portal/login</url-pattern>
</filter-mapping>
```

This code shows a filter—a filter name (`CAS Filter`) with a filter class (`com.liferay.portal.servlet.filters.sso.cas.CASFilter`). Then it shows filter mapping—a filter name (`CAS Filter`) with a URL pattern (`/c/portal/login`).

Using portlet-managed modes

The JSR-286 introduces the portlet-managed modes that are not known to the portal, instead managed by the portlet itself. The portlet can declare such a mode in the portlet deployment descriptor with the code in `portlet.xml` shown as follows:

```
<custom-portlet-mode>
    <description>Look and Feel</description>
    <portlet-mode>lookAndFeel</portlet-mode>
    <portal-managed>false</portal-managed>
</custom-portlet-mode>
```

As shown in this code, the portal-managed mode is set to the value `false`. This setting indicates to the portal that it should treat this mode just like the standard `VIEW` mode. It concerns all aspects, including how it provides the portlet with preferences and from the perspective of access control. Liferay portal provides UI controls that allow the portlet to switch to this mode (`Look and Feel`).

Utilizing container runtime options

The portlet can define additional runtime behavior in the `portlet.xml` file on either the portlet application level or at the portlet level with the `container-runtime-option` element. Runtime options that are defined on the application level will be applied to all portlets in the portlet application. Runtime options that are defined on the portlet level will be applied for this portlet only, and will override any runtime options defined on the application level with the same name. The following is the sample code for the container runtime options, which will be specified in `portlet.xml`:

```
<portlet>
    <container-runtime-option>
        <name>javax.portlet.actionScopedRequestAttributes</name>
        <value>true</value>
        <value>numberOfCachedScopes</value>
        <value>10</value>
    </container-runtime-option>
</portlet>
```

The code above shows container runtime option. The name, `javax.portlet.actionScopedRequestAttributes`, allows web frameworks to pass complex objects from the action or event phase to the render phase through the request.

Serving resources

The JSR-286 introduced a new URL type – Resource URLs. Resource URLs trigger the `serveResource` life cycle method on the `ResourceServingPortlet` interface that you can leverage to create dynamic resources directly in the portlet. In this section, we're going to address how resource URL links can be created, and how the portlet is called to serve the resource.

Using Resource URL

A portlet may need to create URLs that reference the portlet itself. For example, when a user acts on a URL that references the portlet (by clicking a link or submitting a form) the result is a new client request to the portal targeted at the portlet. These URLs are called Portlet URLs. The Portlet API defines the `PortletURL` and the `ResourceURL` interfaces. Portlets must create portlet URLs either by using `PortletURL` or the `ResourceURL` objects. The following are the methods of `PortletURL` that extend `BaseURL`:

```
void write(java.io.Writer out, boolean escapeXML);
void setWindowState (WindowState windowState);
void setPortletMode (PortletMode portletMode);
PortletMode getPortletMode ();
WindowState getWindowState ();
void removePublicRenderParameter(String name);
```

Links created by `ResourceURL`, when clicked by the user, will result in a `serveResource` call of the `ResourceServingPortlet` interface. Resource URLs can be created with the `createResourceURL` method on the `RenderResponse` and `ResourceResponse`. For example:

```
ResourceURL url = renderResponse.createResourceURL();
```

Now, you can set parameters on the URL as you do for the other portlet URLs. You receive these parameters in the `serveResource` call. In order to clearly identify your resource, you can also set an additional resource ID on the resource URL. If you extend `GenericPortlet`, you can forward that resource ID for a `serveResource` call. Moreover, you can set the path to your resource as ID.

```
url.setResourceId ("WEB-INF/portlet-ext.xml");
```

In this case, `GenericPortlet` automatically dispatches to the view, which can then make use of the portlet state information by including the portlet tag library.



ResourceURL cannot set new render parameters, portlet mode, or window state. This restriction occurs because serveResource calls do not generate a full new portal page, but they return the response of serveResource. Thus, the portal does not have a chance to update other parts of the page, where this information may be encoded. Further, static resources such as GIF files packaged in the portlet WAR should normally be referenced with static resource URLs, for example, `response.encodeURL(request.getContextPath() + "/images/my-image.gif")`. This is because serving static resources using the portlet serveResource method will cause unnecessary performance overhead.

Using caching levels of resources

With the `setCacheability` method on the ResourceURL, a portlet can indicate parts of the overall state via the cache level parameter, and thus the portal can create URLs being served from a browser web cache. With the `getCacheability` method on the ResourceURL, the portlet can retrieve the current cache level.

The JSR-286 supports the following scenarios in order to cache `serveResource` calls:

- FULL: Cacheable resources which are independent of the interaction state.
- FULL and SHARED: Cacheable resources such as shared static resources, for example, JavaScript libraries.
- PORTLET: Cacheable resources that depend on portlet interaction state, for example, a dynamically-generated PDF output of the current portlet view.
- PAGE: Cacheable resources such as resources providing action or render URLs, for example, the markup returned for an AJAX call. Page-level cache ability is the default setting in the Liferay portal.

The following is a sample code for the `serveResource` method in `LiferayPortlet`:

```
public void serveResource(ResourceRequest resourceRequest,  
    ResourceResponse resourceResponse) {  
    if (!isProcessResourceRequest(resourceRequest)) { return; }  
    super.serveResource(resourceRequest, resourceResponse);  
}
```

In brief, these cacheability levels give us the ability to provide the runtime with as many hints about the cacheability as possible. The runtime can also enhance the cacheability of resources in other ways, for example, by keeping track of the state changes on the client and re-rendering only those parts affected by the state changes.

Serving the AJAX data directly from the portlet

The JSR-286 provides you with the ability to serve resources directly using the portlet—serving the AJAX directly for the portlet. Thus, you can issue XMLHttpRequests to ResourceURLs and, on the server side, get complete access to the portlet context, for example, render parameter, portlet mode, window state, portlet preferences, and portlet session.

You can also make some state changes in the `serveResource` call, for example, changing the portlet preferences and data in the portlet session scope. The following is sample AJAX code abstracted from `portlet.js`:

```
var xHR = jQuery.ajax( { async: false,
  data: instance._buildrequestData(), type: 'GET',
  url: themeDisplay.getPathContext() + '/c/portal/portlet_url'
}; return xHR.responseText;
```

The code above implements asynchronous updates that make the user interface more responsive. For instance, the AJAX calls go through the portlet URL. Note that you can also leverage resource URLs to provide the links to the JavaScript library.

Utilizing other features

The JSR-286 was designed to avoid breaking binary code compatibility with the JSR-168, and to maintain compatible behavior for all API methods. Here, we only cover various smaller additions that can support use cases, for example, Java 5 features, caching features, runtime IDs, and taglib additions.

Using JAVA 5 features

The JSR-286 leverages some of the Java 5 features, for example, using **Generics** and introducing a class **Enum** for all user profile attributes.

`GenericPortlet` also allows using the annotations to make dispatching life cycle calls to specific handler code easier. `Action` annotates a method that should process a specific action. In order to allow dispatching, the Action URL needs to include the `javax.portlet.action` parameter that is predefined action name in `ActionRequest.ACTION_NAME`. `ProcessEvent` annotates methods that should process specific events, while `RenderMode` annotates methods that should render a specific portlet mode.

The following code shows how to use the `ProcessAction` annotations:

```
// generating the URL  
String name = "getting_category";  
PortletURL url = response.createActionURL();  
url.setParameter(ActionRequest.ACTION_NAME, name);  
url.write(output);  
// method for handling the action  
@ProcessAction (name="getting_category")  
void processMyActionA(ActionRequest req, ActionResponse resp)
```

Employing caching features

The JSR-286 adds new caching features that help making portlets scale better:

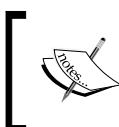
- Public caching scope – marking cache entries shared across users
- Multiple cached views per portlet
- Validation-based caching for validating expired cache entries

In the JSR-286, you can tell the portlet container that the cache entry can be shared using the `cache-scope` element in the deployment descriptor. Thus, you can dramatically reduce the memory footprint for these portlets. For example, here is public caching scope example specified in `portlet.xml`:

```
<portlet>  
  <expiration-cache>30</expiration-cache>  
  <cache-scope>public</cache-scope>  
</portlet>
```

In brief, the validation-based caching is useful in situations where you do not want to re-compute the markup of a portlet often, as it is an expensive operation. It allows you to define a short expiration time so that the portlet gets called often and can check for changes in the backend system.

If nothing has changed in the backend system, validate that the expired cache content is still valid for use by setting the `CacheControl.setUseCachedContent (true)` and setting a new expiration time. If something has changed in the backend system, then you produce a new markup and change the setting to `false`.



Using validation-based caching is beneficial only if the operations for creating the markup are expensive as compared to the operations for checking the backend state.

Sharing runtime ID

You can use the **Runtime ID** to scope the data that the portlet handles to avoid collisions between multiple occurrences of a portlet within the same portal, and maybe even on the same page.

First of all, the JSR-286 extended the lifetime of the namespace runtime ID, using the `response.getNamespace` method, from being valid for only one request to now being stable for the lifetime of the portlet window.

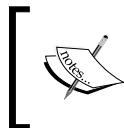
Then, JSR-286 introduced a new API call that allows you to get to a unique ID for the portlet window using the `portletRequest.getWindowID` method, allowing you to use this ID as a key for the data that you want to create namespace per portlet window.

`PortletRequest` returns an array containing all of the `Cookie` properties with the following methods:

```
String getScheme();  
String getServerName();  
String getServerPort();  
String getWindowID();
```

The `getScheme()` method returns the name of the scheme used to make this request; the `getServerName()` method returns the host name of the server that received the request; the `getServerPort()` method returns the port number on which this request was received; and the `getWindowID()` method returns the portlet window ID—it is unique for this portlet window and is constant for the lifetime of the portlet window.

For example, the portlet **Reports** wants to cache data that it received from a backend system per portlet window.



No life cycle calls are defined around the portlet window. Thus, if you used the portlet window ID for specifying namespace entries in a persistent data store, you also need to clean up these entries yourself.

Using taglib

The JSR-286 provides a tag library with its own namespace. When customizing or developing portlets, you need to include the new tag library with the following import:

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

You can use the following additional variables beyond the request, response, and portletConfig variables:

- portletSession – accesses the portlet-scoped session
- portletSessionScope – accesses the portlet-scoped session attribute key / values map
- portletPreferences – used for the portlet preferences
- portletPreferencesValues – used for the portlet preferences key / values map

Each of the URL generation tags has additional attributes.

`copyCurrentRenderParameters` copies the current private render parameters to the URL and `escapeXml` turns the default XML escaping of the URL. By default, all of the URLs are escaped, but you can turn it off using the `escapeXml` attribute.

In addition, JSR-286 adds the `resourceURL` and `propertyTag` tags. The `resourceURL` tag is used for generating resource URLs, while `propertyTag` is used for attaching properties to URLs. For instance, the **Reports** portlet has a view with the following code in `view.jsp`:

```
<portlet:defineObjects />
<% PortletPreferences prefs = renderRequest.getPreferences(); %>
<a href=<portlet:renderURL windowState=<%
   WindowState.MAXIMIZED.toString() %>>">Test <%= 
    prefs.getValue("test", "") %></a>
```

You can find the tags `portlet:defineObjects`, `portlet:renderURL`, and so on. The first one indicates defined objects, while the second one indicates render URL.

Coordinating portlets

In order to build the intranet web site `bookpub.com` of the enterprise "Palm Tree Publications", we need to develop a set of portlets and provide capabilities for coordination between different portlets further.

For example, we have one page called **FAQ**, as shown in following screenshot. It contains at least two portlets: **FAQ** and **Browse-FAQ**. The **Browse-FAQ** portlet contains categories, for example, **Top Questions**, **About Our Other Offerings**, **About the Television Program**, and **About the Website**. You can select the different categories in this page via the portlet, **Browse-FAQ**.

FAQ - Look and Feel - Configuration - Close

Frequently Asked Questions

Top Questions

- ▼ **Can my child appear on Street?**
The children who appear on Street are chosen from schools, agencies, and mail submissions of the tri-state area of New York, New Jersey, and Connecticut.
- ▼ **How do I find out when programs are on BS?**
To find out when shows are on in your area, please visit this page: http://bskids.org/in_your_town/?address=/whatson. If you need more information, contact your local BS station.
- ▼ **What are Playlists?**
Playlists are a collection of fun and educational games and videos organized around an educational theme. Each playlist includes seven games or videos.

Video Player Selection - Look and Feel - Configuration - Close

Select video player.

Browse-FAQ - Look and Feel - Configuration - Close

Browse FAQs by Category

- **Top Questions**
- About Our Other Offerings
- About the Television Program
- About the Website

Journal Content - Look and Feel - Configuration - Close

Select an existing unit or add a unit.

Each category contains a set of questions with answers. For instance, the category **Top Questions** contains three questions: **Can my child appear on Book Street?**, **How do I find out when programs are on BS?**, and **What are Book Playlists?**

Thus, the portlet **FAQ** should provide the main functions. Clicking on the question will show its answer; re-clicking on the question will hide the answer. For instance, if you click on the question **Can my child appear on Book Street?**, you will see the answer: **The children who appear** If you re-click on the question, you will see only the question: **Can my child appear on Book Street?**

As you can see, the **FAQ** page requires the capability for coordination between the **FAQ** and **Browse-FAQ** portlets. When a user clicks on one category, for example, **Top Questions** in the **Browse-FAQ** portlet, it should be highlighted. At the same time, the portlet **FAQ** should display the questions with answers for the selected category, **Top Questions**. In brief, we need inter-portlet communication.

The JSR-286 provides capabilities for coordination between different portlets, that is, inter-portlet communication, with the following mechanisms:

- Sharing data between artifacts in the same web application via **session** in application scope
- Using **public render parameters** in order to share render state between portlets
- Using portlet **events** that a portlet can receive and send

In this section, we're going to address the session, the public render parameters, and the portlet events in detail.

Sharing data via the session

One way to coordinate portlets is to store variables in the session and make them sharable for portlets – called sharing data via the session. By sharing data via the session, you can put stuff into the session and display different things based on the information in the session.

Considering the previous example, we are going to define the sharing attribute, category, of the **Browse-FAQ** portlet, in `portlet.xml` as follows:

```
<shared-application-session-attribute>
    <name>category</name>
</shared-application-session-attribute>
<portlet>
    <portlet-name>Browse-FAQ</portlet-name>
    <shared-portlet-session-attribute>
        <name>category</name>
        <java-class>com.faq.Category</java-class>
    </shared-portlet-session-attribute>
</portlet>
```

This code shows shared application-session attribute and shared portlet-session attribute. Note that the attributes must be serialized and have a valid JAXB 2.0 binding, and `HttpSessionBindingListener` should be supported as well.

In addition, the three approaches related to sharing the data via the session to achieve inter-portlet communication are using: the `PortletSession`, `PortletContext`, and page parameters.

Using PortletSession

`PortletSession` is created for each user, making `PortletSession` useful for communicating all user-related information among different portlets. There are two scopes to set attributes: the `PORTLET` scope and the `APPLICATION` scope. If it is required that a particular attribute be shared among the portlets, we should use `APPLICATION`.

For example, consider **FAQ** and **Browse-FAQ** – two portlets in the same application. In `processAction()` of **Browse-FAQ**, we can set the attribute `category` as follows:

```
portletSession.setAttribute("category", "Top Questions",
                           PortletSession.APPLICATION_SCOPE);
```

This code shows that the attribute `category` has been set as `APPLICATION` scope. In the **FAQ** portlet, we can retrieve the changed category name from the `PortletSession`. In the **FAQ** portlet, we could get the value of the attribute, `category`, first, and then show its value. The following is the sample code:

```
String currentCategoryName = (String)portletSession.getAttribute("category", PortletSession.APPLICATION_SCOPE);  
showFAQForThisCategory(currentCategoryName);
```

Using PortletContext

`PortletContext` (similar to `ServletContext`) is shared among all the users and all the portlets in the portlet application. You can get and set attributes from `PortletContext` as a global point of communication.

For example, if the **FAQ** portlet uses a service which is currently available, we could get the attribute `category` first and then show the value of the attribute `category`. The following is a sample code:

```
If (portletContext.getAttribute("category") != null) {  
    String currentCategoryName = (String) portletContext.getAttribute("category");  
    showFAQForThisCategory(currentCategoryName);  
}
```

Using page parameters

In addition, we can share data among portlets on a specific page by using page parameters. It needs to include the WAR reference if it is in a WAR portlet. For example, a WAR named `ipcportlet`, having a portlet named `faq`, will have a fully qualified name of `faq_WAR_ipcportlet`.

The portlet **Browse-FAQ** contains categories, for example, **Top Questions**, **About Our Other Offerings**, **About the Television Program**, and **About the Website**. Each category contains a link URL with page parameters. For example, the category **Top Questions** has a link URL such as:

```
/faq?p_p_lifecycle=0&p_p_id=faq_WAR_ipcportlet&p_p_category=Top  
Questions
```

As shown in the code above, when users click on the category **Top Questions**. They will be brought to the page `/faq` with page parameters, for example, `p_p_lifecycle` with a value `0`, `p_p_id` with a value `faq_WAR_ipcportlet`, and `p_p_category` with a value `Top Questions`.

Similar to `PortletSession` in the **FAQ** portlet, you can retrieve the changed category name from the `RenderRequest`. The following is a sample code:

```
String currentCategoryName = (String) renderRequest.getAttribute("category");
showFAQForThisCategory(currentCategoryName);
```

The approach, using page parameters, is simple, but it has its own limitations. The parameters are transferred to the specific portlet only (called one-way sharing data) as there is only one specified parameter `p_p_id` for the portlet, `faq_WAR_ipcportlet`.

Using portlet events

Events-based coordination provides support for active notifications with action capabilities to the portlets. This coordination feature is based on a loosely-coupled publish-subscribe model, where the Liferay portal acts as broker between the different portlets, and distributes the events.

In general, portlet events allow portlets to react to actions or state changes not directly related to an interaction of the user with the portlet. Events could be Liferay portal, portlet-container generated, or the result of user interaction with other portlets.

Logically, the portlet must implement the `EventPortlet` interface in the `javax.portlet` package first, in order to receive events. Then the portlet container will call the `processEvent` method for each event targeted at the portlet with `EventRequest` and `EventResponse` objects. Finally, portlet events are targeted by the Liferay portal or portlet container to a specific portlet window in the current client request.

Let's consider the two portlets shown in the previous screenshot: **FAQ** and **Browse-FAQ**. The **Browse-FAQ** portlet may send events for category changes triggered by simple user interactions, for example, clicking on a category of **FAQ**. Another **FAQ** portlet receives the event for category changes and displays the **FAQ** of a selected category. How to implement this feature? Let's have a look at how to implement receiving events and sending events.

Sending events

Considering the example above, the **Browse-FAQ** portlet can publish events via the `StateAwareResponse.setEvent` method, while the `StateAwareReponse` methods are exposed via the `ActionResponse` and `EventResponse` interfaces. You can also call `StateAwareResponse.setEvent` multiple times in the current `processAction` or `processEvent` method.

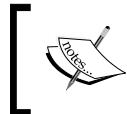
In `portlet.xml`, the following is the portlet specification of the **Browse-FAQ** portlet:

```
<event-definition>
  <qname xmlns:x="http://book.com/events">x:category</qname>
  <value-type>java.lang.String</value-type>
</event-definition>
<portlet>
  <supported-publishing-event>
    <qname xmlns:x="http://book.com/events">x:category</qname>
  </supportedpublishing-event>
</portlet>
```

Here is a sample code for the **Browse-FAQ** portlet:

```
void processEvent(EventRequest req, EventResponse resp) {
    String category = "Top Questions";
    QName name = new QName ("http://book.com/events", "category");
    resp.setEvent(name, category);
}
```

This code shows how to send the event, `category`. In general, events can be published either with their full `QName` (that is, a namespace, to avoid naming conflicts) with the `setEvent(QName, Serializable)`, or by specifying only their local part with the `setEvent(String, Serializable)` method.



If only the local part is specified, the namespace must be the default namespace defined in the portlet deployment descriptor with the `default namespace` element.



Receiving events

Considering the previous example, the **FAQ** portlet can access the event that triggered the current process event call by using the `EventRequest.getEvent` method. Normally, this method returns an object of the type `Event`, encapsulating the current event name and value. The event must always have a name, and may optionally have a value.

In `portlet.xml`, the following is the portlet specification of the **FAQ** portlet:

```
<default-namespace>http://book.com/events</default-namespace>
<event-definition>
  <name>category</name><value-type>java.lang.String</value-type>
</event-definition>
<portlet>
  <supported-processing-event>
    <name>category</name></supported-processing-event>
</portlet>
```

The following is the sample code for the **FAQ** portlet:

```
void processEvent(EventRequest req, EventResponse resp) {  
    Event event = req.getEvent();  
    if (event.getName().equals("category")) {  
        String payload = (String) event.getValue();  
    }  
}
```

This code shows how to consume the event, `category`. You can retrieve the event name either with the `getQName` method that returns the complete `QName` of the event, or with the `getName` method that only returns the local part of the event name.

Employing public render parameters

Public-render-parameters-based coordination provides support for the sharing of view state across portlets. Besides the above portlet events allowing coordination, you can use render parameters with the other portlets within the same portlet application or across portlet applications. The portlets, for example, **FAQ** and **Browse-FAQ**, can declare public render parameters in their deployment descriptors using the `public-render-parameter` element in the portlet application section. Loosely speaking, public render parameters are available in all life cycle methods of the portlet: `processAction`, `processEvent`, `render`, and `serveResource`. Especially, public render parameters can be viewed and changed by the other portlets or components.

In `portlet.xml`, the following is the portlet specification of the **FAQ** and **Browse-FAQ** portlets:

```
<public-render-parameter>  
    <identifier>category_b</identifier>  
    <qname xmlns:x="http://book.com/params">x:category</></qname>  
</public-render-parameter>  
<portlet>  
    <portlet-name>browse-FAQ</portlet-name>  
    <supported-public-render-parameter>  
        category  
    </supported-public-render-parameter>  
</portlet>  
<portlet><portlet-name>FAQ</portlet-name>  
    <supported-public-render-parameter>  
        category  
    </supported-public-render-parameter>  
</portlet>
```

The previous above defines the public render parameter and declares the portlets which support the public parameter, for example, **FAQ** or **Browse-FAQ**. In the portlet section, each portlet can specify the public render parameters it would like to share via the supported-public-render-parameter element. The supported-public-render-parameter element must reference the identifier of a public render parameter defined in the portlet application section in a public-render-parameter element.

The following is the sample code for setting and getting the public render parameter:

```
public void processAction(ActionRequest actionRequest, ActionResponse  
    actionResponse) {  
    String categoryValue = actionRequest.getParameter("category");  
    actionResponse.setRenderParameter("category", categoryValue);  
}
```

This code shows getting and setting the public render parameter, `category`. In short, the portlet (either **FAQ** or **Browse-FAQ**) uses the public render parameter identifier in the code above in order to access the public render parameter.

Summary

This chapter first discussed Liferay portal and portlets. It then explained how to use JSR-286 portlets, for example, portlet life cycle, portlet modes, and windows states and their relationship with Servlets. It showed how to employ portlet actions, for example, configuration, context, request, response, and preferences. Further, it addressed how to extend JSR-286 portlets smoothly, how to serve resources, and how to coordinate portlets to implement inter-portlets communication.

In the next chapter, we're going to introduce the Extension environment in order to customize JSR-286 portlets in the Liferay portal.



This material is copyright and is licensed for the sole use by Matt Bedsaul on 27th May 2009
1 Microsoft Way, , Redmond, , 98052

3

ServiceBuilder and Development Environments

We have discussed how to use JSR-286 portlets in the previous chapter. Now it is time to develop JSR-286 portlets for the intranet web site bookpub.com. Before moving on to develop JSR-286 portlets, we have to set up our development environment properly. As we know, a good development environment will save a lot of developing time. Fortunately, there are two existing developing environments: **Plugins SDK environment (Plugins SDK)** and **Extension environment (Ext)**. Both of them have advantages and disadvantages with regards to the portal customization and development, portlets development, and integration with other systems. Meanwhile, Liferay portal provides ServiceBuilder as a tool to build Java services that can be accessed in a variety of ways. That includes local access from Java code, remote access using web services, and so on.

How do we set up the development environments? How do we use ServiceBuilder to generate models and services automatically? This chapter will give answers for these questions. This chapter will first introduce how to set up Ext, and how to build it. Then it will address how to use ServiceBuilder in Ext. Finally, it will discuss how to set up Plugins SDK and how to use Ext and Plugins SDK efficiently.

By the end of this chapter, you will have learned how to:

- Set up Ext
- Build Ext
- Deploy Ext
- Use ServiceBuilder in Ext
- Set up Plugins SDK
- Use development environments efficiently

Setting up Ext

Ext is a wrapper for Liferay portal core source. Generally, Ext mirrors Liferay portal core source directories (that is, `ext-impl/`, `ext-service/`, and `ext-web/`). It allows the users to develop on top of Liferay portal, like a platform, without having to worry about upgrading in the future. As we know, it is very difficult to upgrade to new versions of Liferay portal if one modifies the source code directly. Fortunately, Ext allows users to extend and modify the source code without breaking the upgrade path to new versions of Liferay portal.

In order to set up the development environments, including Ext and Plugins SDK for development, customization, deployment, and debugging, we need to consider five aspects: required tools, databases, application servers, **IDE (Integrated Development Environment)**, and portal source code. Let's have a deeper look at these aspects.

Required tools

Liferay portal is Java-based portal application, using Ant build tool. Thus, the required tools are JDK and Ant. For both tools, we recommend you to use the latest version.

JDK

First you need to download the latest version of JDK. It is available at <http://java.sun.com> for every OS. The installation instructions can also be found here. When you install it, make a note of the location as you will need it when setting your `JAVA_HOME` variable.

Next, you need to set the `JAVA_HOME` variable. The following steps can be used to set up the `JAVA_HOME` variable in Windows. You can also set it up in Linux, Unix, and Mac operating systems as you can run Liferay portal in any OS.

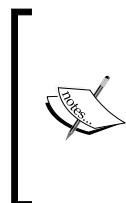
1. Right-click on **My Computer** and go to **Properties**.
2. Go to the **Advanced** tab and click on **Environment Variables** and you will need to add a new system variable.
3. Set **JAVA_HOME** to the location you have noted above.
4. Edit the path and add `%JAVA_HOME%\bin` to the beginning of the path system variable.
5. Click on **OK**.

You can check whether OS recognizes Java, and also if it is the correct version, by typing the command `java -version`. By the way, we use `$JDK_MAJOR_VERSION` to represent the major version of JDK. For example, the value of `$JDK_MAJOR_VERSION` could be 1.5, 1.6, and so on.

Ant

You need to download the latest version of Apache Ant. It is available at <http://ant.apache.org> for every OS. The installation instructions can also be found here. When you install it, make a note of the location as you will need it when setting up your `ANT_HOME` variable.

Then you need to set the `ANT_HOME` variable in a similar manner as `JAVA_HOME` variable was set. You can check whether OS recognizes Ant, and also if it is the correct version, by running the command `ant -version`.



Apache Ant is a Java-based build tool. In theory, it is kind of like Make, but without Make's wrinkles. Instead of a model where it is extended with shell-based commands, Ant is extended using Java classes. Instead of writing shell commands, the configuration files are XML-based, calling out a target tree where various tasks get executed. Refer to <http://ant.apache.org>.



Databases

Liferay portal supports many databases. Databases which the Liferay portal can run on include Apache Derby, IBM DB2, Firebird, Hypersonic, Informix, InterBase, JDataStore, Oracle, PostgreSQL, SAP, SQL Server, Sybase, MySQL, and so on. Eventually, you can use any one of them. For demo purpose, we shall use MySQL.

MySQL

You need to download the latest version of MySQL. It is available at <http://www.mysql.com> for every OS. The installation instructions can also be found here. When you install it, make a note of the location as you will need it when setting up your `MYSQL_HOME` variable.

Then you need to set `MYSQL_HOME` variable in a similar manner as `JAVA_HOME` variable was set. Similarly, you can check whether the OS recognizes MySQL, and also if it is the correct version, by running the command `mysql --version`.

In addition, we need to prepare a database `lportal` and username/password `lportal/lportal`, which has full access to the database, `lportal`. Of course, you can have different database names, usernames, and passwords. We will use `lportal` as values of database, username, and password only for demo purposes. Thus, in the MySQL database command lines, log in as `root`, and create a database, `lportal` and username/password as `lportal/lportal` as follows:

```
drop database if exists lportal;
create database lportal character set utf8;
grant all on lportal.* to 'lportal'@'localhost' identified by
'lportal' with grant option;
grant all on lportal.* to 'lportal'@'localhost.localdomain' identified
by 'lportal' with grant option;
```

If you were interested in the details about SQL scripts of MySQL, you can refer to the online manual at <http://www.mysql.com/>.

Application servers

Liferay portal supports any application servers. The application servers (or the servlet containers) that the Liferay portal can run on include Borland ES, Apache Geronimo, Sun GlassFish, JBoss, JOnAS, JRun, OracleAS, Orion, SUN JSAS, WebLogic, WebSphere, Jetty, Resin, Tomcat, and so on.

Of course, it is up to you to choose one of them. But for demo purposes, we will use Tomcat for testing, debugging, and developing.

Optionally, there are a set of Liferay portal bundles, available at <http://www.liferay.com/web/guest/downloads/portal> as application servers. You can use one of them for servlet containers and full Java EE application servers. It is simple to use Liferay portal bundles—just download one bundle from the above URL and unzip it to the specific folder in your local machine.

Why use Tomcat, and not Liferay portal bundled with Tomcat? Naturally, it is better to use Liferay portal bundled with Tomcat as it is preconfigured. But you will lose an opportunity to configure Tomcat with Liferay portal from a new installation. We choose Tomcat, and will show the approach later on how to configure Tomcat to support the Liferay portal.

Tomcat

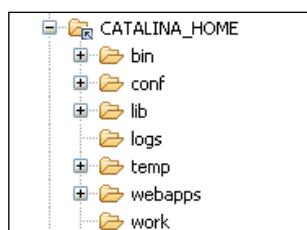
Before installing Tomcat, we need to set the working folder `$LIFERAY_PORTAL` variable. Logically, you can have a different folder name. But in order to be referred simply and easily, we use a folder named `Liferay-Portal`. That is, the folder `$LIFERAY_PORTAL` setting has a value, `Liferay-Portal`. More specifically, you will have a value for `$LIFERAY_PORTAL`—`C:\Liferay-Portal` in Windows, and `/Liferay-Portal` in Linux, Unix, and MacOS. Thus, you will see some specific examples and diagrams in this book with these values related to a specific OS—Windows.

First of all, you need to download the most recent version of Tomcat. It is available at <http://tomcat.apache.org> for every OS. It is a ZIP file named `$TOMCAT_ZIP_FILENAME.zip` (you can download a ZIP file named `$TOMCAT_ZIP_FILENAME.tar.gz` also). Here, the `TOMCAT_ZIP_FILENAME` is the actual ZIP file name, for example, `apache-tomcat-version`. The version is made up of a major version named `$TOMCAT_MAJOR_VERSION`, and a minor version named `$TOMCAT_MINOR_VERSION`.

To install Tomcat, we need to unzip the ZIP file into the folder, `$LIFERAY_PORTAL`, and set the value of the variable `$CATALINA_HOME` to `$LIFERAY_PORTAL/$TOMCAT_ZIP_FILENAME`. In other words, we should use `$CATALINA_HOME` referring to this location where the Tomcat has been installed, actually.

Why do we need to install Tomcat under the `$LIFERAY_PORTAL` folder? In the runtime, Liferay portal will create two folders `data` and `deploy` under the `$LIFERAY_PORTAL` folder sharing the same parent folder `$LIFERAY_PORTAL`, with the folder `$TOMCAT_ZIP_FILENAME`. Thus, if we install Tomcat in the folder, `$LIFERAY_PORTAL`, it would be easy to refer to the `data` and `deploy` folders as `$LIFERAY_PORTAL/data` and `$LIFERAY_PORTAL/deploy`, respectively.

You will see a set of subfolders under the folder `$CATALINA_HOME`—`bin`, `conf`, `logs`, `lib`, `temp`, `work`, and `webapps`, as shown in following screenshot:





Apache Tomcat is an implementation of the Java Servlet and **Java Server Pages (JSP)** technologies. The Java Servlet and JSP specifications are developed under the Java Community Process. Refer to <http://tomcat.apache.org> for further information.

IDE

Why do we need IDE? **IDE** or **Integrated Development Environment** provides comprehensive facilities for software development. An IDE normally consists of a source code editor, a compiler, an interpreter, and a build automation tools debugger. Of course, you can develop portlets in Liferay portal without using any IDE. But an IDE is designed to maximize programmer productivity by providing tightly-knit components with similar user interfaces. Thus we plan to use an IDE.

Optionally, there are a set of IDEs you may choose from—Eclipse IDE, NetBeans IDE, and IntelliJ. You may use IntelliJ IDE—a commercial Java IDE (which includes coverage across languages and technologies), a deep and intelligent editor, code analyzer, and refactor, all of which boost user productivity. Refer to <http://www.jetbrains.com/idea/> for more information.

NetBeans, a free, open source IDE, includes professional desktop, enterprise, web, and mobile applications with the Java, C/C++, and Ruby. Refer to <http://www.netbeans.org> for further information. In addition, you may use the Portal Pack to make portlet development easier in NetBeans. Refer to <http://portalpack.netbeans.org/> for more information.

As an IDE, Eclipse is a software platform comprising extensible application frameworks, tools, and a runtime library for software development and management, written primarily in Java. Refer to <http://www.eclipse.org> for more details.

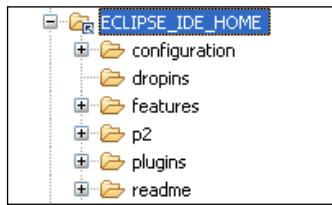
Logically, you could choose any IDE. For our example, we will choose Eclipse IDE. We are going to use Eclipse IDE for development, customization, deployment, and debugging custom code based on the Liferay portal.

Eclipse IDE

Obviously, you can download the most recent version for every OS, available at <http://www.eclipse.org>. There are still a lot of download choices—Eclipse Classic, Eclipse IDE for Java EE Developers, Eclipse IDE for Java Developers, and so on. For our example, we will choose Eclipse Classic.

You can install Eclipse IDE anywhere. When you install it, make a note of the location, as you will need it when setting up `ECLIPSE_IDE_HOME`. For the convenience of reference, we prefer to install Eclipse IDE under the folder, `$LIFERAY_PORTAL`. Thus, the `$ECLIPSE_IDE_HOME` setting has a value, `$LIFERAY_PORTAL/eclipse`.

Later, you may see the following subfolders under the folder `$ECLIPSE_IDE_HOME`—configuration, features, plugins, readme, p2, and dropins, as shown in the following screenshot:

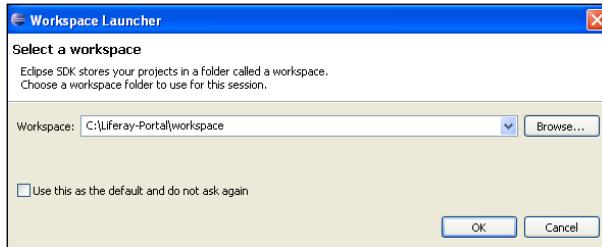


[ You need to set `-XX: maxPermSize` with 256M or **higher**. Otherwise, you may have memory issues in your development. Check the default setting in `$ECLIPSE_IDE_HOME/eclipse.ini`.]

Workspace

Before starting the Eclipse IDE, we need to build a workspace and a folder to store projects. Logically, you could create the folder of the workspace anywhere, and give it any name. For the sake of reference, we use the name, `workspace`, as the folder of the workspace. Moreover, place the folder `workspace` under the folder `$LIFERAY_PORTAL`. Thus, we can refer to the folder of the workspace simply as `$LIFERAY_PORTAL/workspace`.

When starting Eclipse IDE, you are asked to provide the workspace path as \$LIFERAY_PORTAL/workspace, as shown in following screenshot (especially, if it is a sample of workspace in Windows). After that, you will have your own Eclipse IDE.

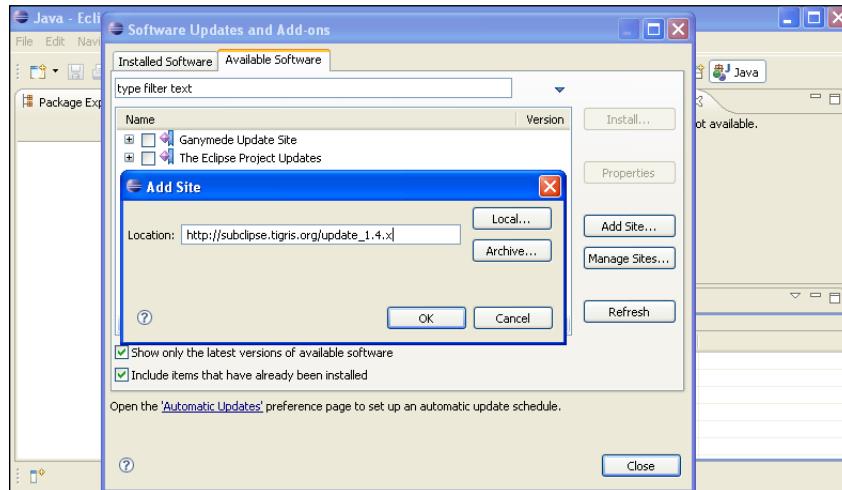


Subclipse

In order to get Liferay portal source code, we have to use Subclipse in the Eclipse IDE. Subclipse is an Eclipse Team Provider plugin providing support for Subversion (an open source version control system) within the Eclipse IDE. Refer to <http://subclipse.tigris.org>.

As you can see, there are a set of version control systems for source management, for example, **Concurrent Versions System (CVS)**, Perforce, **Subversion (SVN)**, IBM Rational ClearCase, and so on. Why do we only need SVN? As Liferay portal source code is managed via Subversion, we have no other choice except SVN. Let's install the Subclipse in the Eclipse IDE.

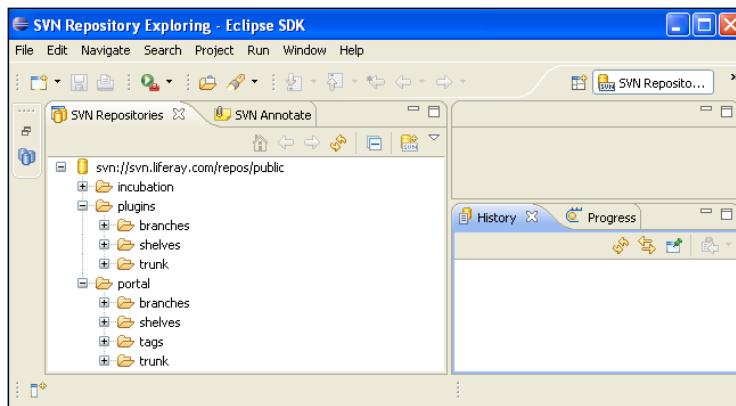
As shown in following screenshot, the installation of the Subclipse is simple. For its installation instructions, refer **Download and Install** at <http://subclipse.tigris.org/>. It would be better to use the most recent version of the Subclipse.



After installation, you can use the Subclipse as follows:

1. From the **Window** menu, select **Open Perspectives**.
2. Click on **Other...**
3. Select **SVN Repository Exploring**, and then click on the button **OK**.
4. Right-click on the **SVN Repositories** view.
5. Select **New** and the **Repository Location...**
6. Provide input as `svn://svn.liferay.com/repos/public` and click on **Finish**.

If you expanded the `plugins` and `portal` folders, you will see possible folders of Liferay portal source code for both `portal` and `plugins`, as shown in following screenshot:



Tomcat plugins

Why do we need Tomcat plugin? As you can see, we have the Eclipse IDE for development, customization, and deployment. But the debugging tool is missing. For this reason, we need a Tomcat plugin. Thus, we could do all the work—development, customization, deployment, and debugging—in the Eclipse IDE.

Optionally, there are at least two Tomcat plugins—MyEclipse and Sysdeo. MyEclipse IDE provides Eclipse plugin development tools for Java, JSP, XML, Struts, HTML, CSS, and EJB. We could particularly use MyEclipse web project as a Debugger in Eclipse. The web project in MyEclipse is an Eclipse Java project that includes metadata defining the project's web nature and a directory structure patterned after the J2EE web archive (WAR) structure. Most interestingly, it provides the most comprehensive applications server connector coverage with over 30 server connectors such as Beiy Tiger, JBoss, Jetty, JOnAS, JRun, Oracle, Orion, Resin, Sun, Tomcat, WebLogic, and WebSphere.



MyEclipse is an Eclipse-based J2EE development platform. It is built upon the Eclipse platform, and integrates both proprietary and open source solutions into the development environment. Refer to <http://www.myeclipseide.com> for more information.

Sysdeo Eclipse Tomcat Launcher plugin provides capability to use a custom Tomcat class loader to load classes in several Java projects at the same class loader level as compared to other classes. Refer to <http://www.eclipsetotale.com> for more details.

Why just Sysdeo? Logically, you could choose any kind of Tomcat plugins. We choose Sysdeo as a debugger, because it is lightweight and, most importantly, simple for demo purposes.

It is easy to install Sysdeo. You just have to download its most recent version from <http://www.eclipsetotale.com> and unzip it in the `$ECLIPSE_IDE_HOME/dropins` folder. That's it! Then restart the Eclipse IDE, and you will see the Tomcat icons (for example, Start Tomcat, Stop Tomcat, and Restart Tomcat), as shown in following screenshot:



By the way, we need to configure Sysdeo plugin, linking to the Tomcat, `$CATALINA_HOME`. First, we need to set the Tomcat version and the Tomcat home: **Window | Preferences**. Select **Tomcat** and set **Tomcat version** (6.x for example) and **Tomcat home** as `$CATALINA_HOME`. Note that the **Tomcat version** and the **Tomcat home** are the only required fields. The other settings are there for advanced configuration.

Then, set a JDK as default JRE for the Eclipse IDE. Open the preference window—**Window | Preferences | Java | Installed JREs**. This JRE must be a JDK (This is a Tomcat prerequisite).

Finally, set JDK and JVM parameters—**Preferences | Tomcat | JVM Settings**. For development, you can set JVM parameters as `-Xms128m -Xmx1024m -XX:MaxPermSize=128m`. This is a minimal setting. You can increase the numbers based on your local machine. You may set Sysdeo plugin class-path and boot-class-path if you need any specific settings.

In addition, you may be interested in Liferay Eclipse Plugin—a tool used for easy creation of portlets in the Eclipse IDE. Normally, you can find the latest version (and installation instructions) at http://sourceforge.net/project/showfiles.php?group_id=49260&package_id=215051

Portal source code

Where do we get the source code for the portal? In general, there are four kinds of portal sources: the officially released version, the tag version, the branch version, and the trunk version. Let's have a deeper look at these options.

The **official** version has been released. You can download the latest release from <http://www.liferay.com/web/guest/downloads/portal>. Note that there is only one version (either the major version, for example 5.3, or the minor version, for example 2). If you need the new version, say 5.3.3 or 5.4.3, you have to download the latest version and install it again.

The **tag** version is functionally the same as the officially released version. You can check out on a specific tag at <svn://svn.liferay.com/repos/public/portal/tags>.

The **branch** version provides portal source code with a fixed major version. You can check out the latest branch from <svn://svn.liferay.com/repos/public/portal/branches>. Note that there is only one major version (for example, 5.3), but you can get the latest minor version, (for example, 2, 3, 4) from SVN update. But if you need a new major version, say 5.4 or 5.5, you have to get another branch and install it again.

The **trunk** version provides the portal source code with the latest version—both major and minor. You can check this out at <svn://svn.liferay.com/repos/public/portal/trunk>. Note that from SVN update, you can get the latest version.

Of course, as a developer, you can choose one of them. For our example, we will use the trunk version.

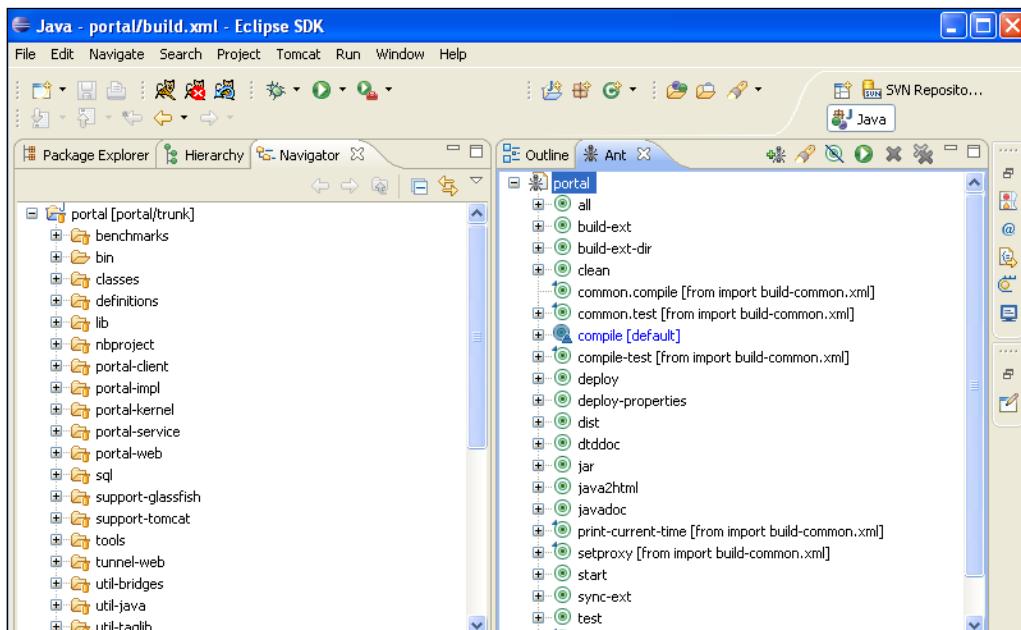
Building Ext

We have prepared a set of tools for Ext development environment. Now, let's build Ext. First of all, let's get Liferay portal source code.

Getting portal source code

Let's consider one scenario. Suppose that you, as an administrator or a developer from an engineering group, expect to base your extensions on top of the latest sources of Liferay portal (for example, `trunk`), you can follow these instructions to set it up:

1. In the Eclipse IDE, switch to the perspective, **SVN Repository Exploring**.
2. Select the root: `svn://svn.liferay.com/repos/public`.
3. Expand the root.
4. Expand the `portal`, and the `trunk`, and check out the `trunk`, using the default project name in the workspace, `portal`. Of course, you can have any name for this project. We are using this name only for ease of reference.



You will have a project named `portal` in the workspace. From now on, we can refer anything under the `portal` project as a value starting with `/portal/`. Here we use the `trunk` version as the portal source code. One big benefit you can get is that you can upgrade from a lower version, for instance 5.3.2 to a higher version such as 5.4.3 (if that was the latest version), simply by clicking on the **SVN Update**.

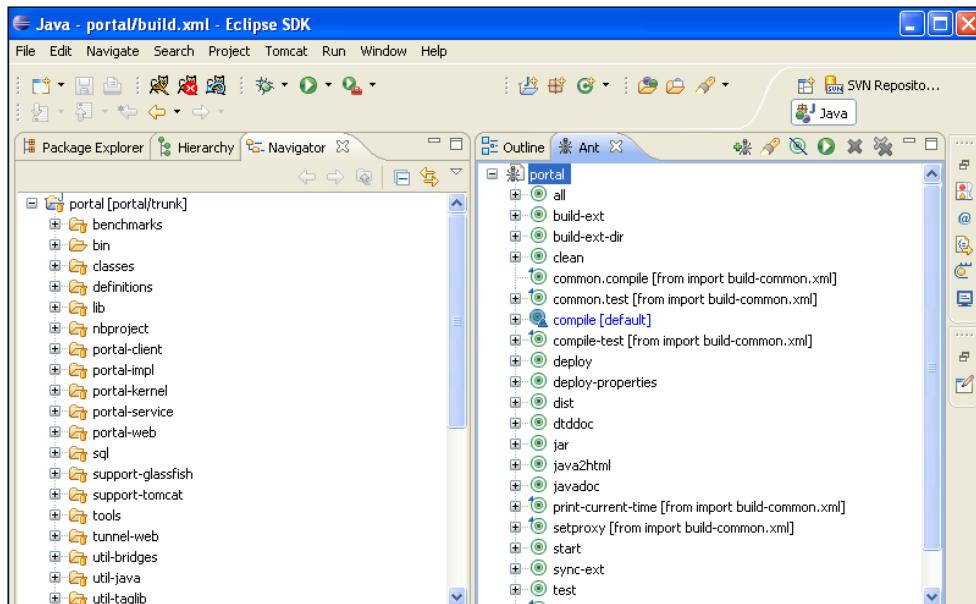
Source structures and Ant targets

Ant targets are just shortcuts for Ant commands. After checking out the Liferay portal source code, you can view the source structures and Ant targets as shown in following screenshot. How do we get this view? You can do it by following these steps:

1. Switch to Java perspective, and add the view, **Navigator**. You will see the project **portal**.
2. Expand the project **portal**, and you will see all the folders.
3. Add the view **Ant** from the menu, **Windows | Show View**, and drop `build.xml` under the `/portal` folder in the **Ant** view. By doing this, you will see all the targets for the portal in the **Ant** view.

The source code includes the following folders: `portal-impl`, `portal-kernel`, `portal-service`, `definitions`, `portal-web`, `support-tomcat`, `util-bridge`, `util-java`, `util-taglib`, and so on. We will refer to the details related to these folders later.

The Ant targets include: `all`, `build-ext`, `build-ext-dir`, `clean`, `start`, `deploy`, `deploy-properties`, `dist`, `jar`, `javadoc`, and so on. For example, the target `clean` will delete classes related to `/portal-impl`, `/portal-kernel`, `/portal-service`, and delete `/tmp`, `/logs`, `/work` and `/webapp/ROOT` in the folder, `$CATALINA_HOME`, and so on. We are going to use some of these targets in the coming sections:

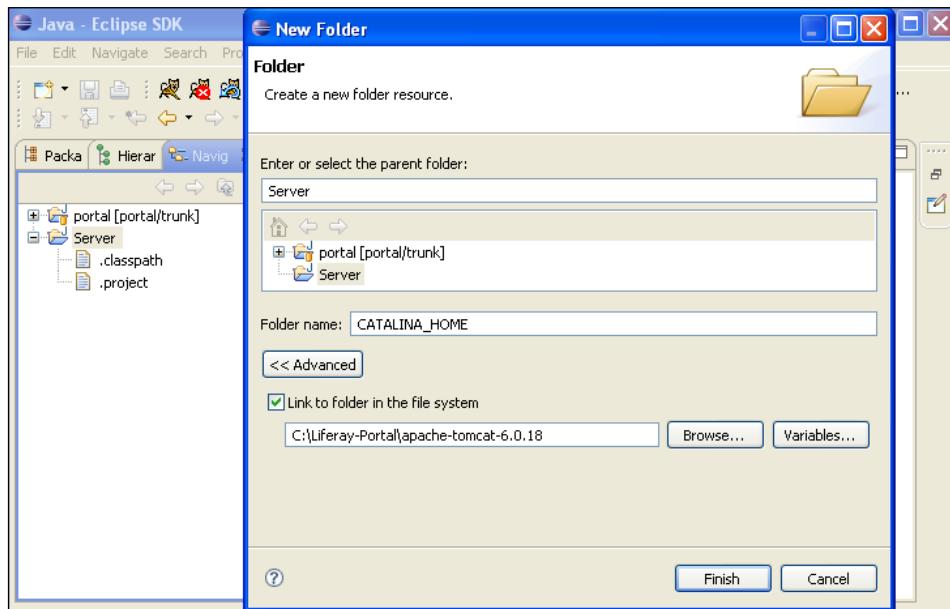


Updating Tomcat to support Ext development

Before building Ext, we need to update Tomcat to support Ext development. It would be a good idea to manage all files of Tomcat in the Eclipse IDE. To do this, follow these steps:

1. Create a Java Project named **Server** (you can use any other name).
2. Right-click on the project **Server** and select **New | Folder**.
3. Enter the **Folder name** as **CATALINA_HOME** (any name can be used).
4. Click on the button, **<<Advanced**.
5. Click on the checkbox, **Link to folder in the file system**, and enter the value, **\$CATALINA_HOME**.
6. Click on the **Finish** button when you are ready.

You can use possible tools (for example, properties editor, text editor, and so on) to change the files of Tomcat in the Eclipse IDE. The following screenshot illustrates the above steps:



From now on, we can work on the linked folder `CATALINA_HOME` in the Eclipse IDE. To update pure Tomcat in order that it supports Ext development, follow these steps:

1. Open the properties file (`/conf/catalina.properties`) in the Eclipse IDE.
2. Update the common loader (`common.loader`).
3. Add `${catalina.home}/lib/ext/*.jar` at the end of `common.loader`.
`common.loader= ${catalina.home}/lib, ${catalina.home}/lib/*.`
`jar, ${catalina.home}/lib/ext/*.jar.`
4. Save the properties file when you are ready.

By doing this, the pure Tomcat is compliant with the Liferay portal bundled with Tomcat. That is, now Tomcat supports Ext development smoothly. Why should we do such an update? In the target `deploy`, Liferay portal will copy a set of JARs (for example, `portal-service.jar`, `portal-kernel.jar`, and so on) into the folder, `/lib/ext`. Thus, we need to let `common.loader` know these JARs as well.

In addition, you can set up Tomcat runtime environment. Thus, you can run Tomcat via command lines. To do so, create the `setenv.sh` and `setenv.bat` files in the `/bin` folder. For Windows, add the following lines in `setenv.bat`:

```
set JAVA_OPTS=%JAVA_OPTS% -Xms128m -Xmx1024m -XX:MaxPermSize=128m -Dfile.encoding=UTF8 -Duser.timezone=GMT
```

For Linux/MacOS, add the following lines in `setenv.sh`:

```
JAVA_OPTS="$JAVA_OPTS -Xms128m -Xmx1024m -XX:MaxPermSize=128m -Dfile.encoding=UTF8 -Duser.timezone=GMT"
```

By the way, you may be interested in the settings of `setenv` in the production servers. For production servers, you should increase the numbers, for example, `-Xms1024m -Xmx2048m -XX:MaxPermSize=256m`, according to specific production server environment. The setting, `-Xms1024m -Xmx2048m -XX:MaxPermSize=256m` is good, normally, for most of the production servers.

Customizing properties

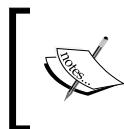
There are three properties files related to the target `build-ext:release`. `properties`, `app.server.properties`, and `build.properties`. You can find these properties in the folder `/portal`.

First of all, we need to tell Liferay portal where we want to build Ext. `lp.ext.dir` is the property that tells Liferay where to build Ext. The default `lp.ext.dir` with the value, `${project.dir}/../ext` in `release.properties` has to be overridden with the actual folder for Ext.

You can put Ext in any folder and with any name. For our convenience, we prefer to call Ext in the Eclipse IDE as ext and put it in the folder, \$LIFERAY_PORTAL/workspace. Thus, the actual value of lp.ext.dir should be \$LIFERAY_PORTAL/workspace/ext.

To update the properties of release.properties, create a separate properties file named release.\${user.name}.properties with the properties to overwrite in the folder /portal:

```
lp.ext.dir=$LIFERAY_PORTAL/workspace/ext
```



{user.name} is your system login name. The name must be simple without special characters such as *, ?, _, @, and so on. For example, the login name Jonas is good, but the login name Jonas.Yuan is not good.



Then, we need to tell the Liferay portal which compiler we want to use for compiling the code via the Ant. The default ant.build.javac.source and ant.build.javac.target is set in build.properties. This value may be different from the value, \$JDK_MAJOR_VERSION. If it is different, we need to override this value. To update the properties of this file, create a separate properties file named build.\${user.name}.properties with the properties to overwrite in the /portal folder.

```
ant.build.javac.source=$JDK_MAJOR_VERSION  
ant.build.javac.target=$JDK_MAJOR_VERSION
```

Finally, we need to tell Liferay which application server we want to use in order to run Liferay portal on top of it. The default application server is set as Tomcat with the default version, as indicated by the app.server.type property in app.server.properties. The default version of Tomcat may be different from the value, \$TOMCAT_MAJOR_VERSION. If it is true, we need to override the setting of Tomcat with the value, \$TOMCAT_MAJOR_VERSION. Normally, we need to select the type of application server, that is, Tomcat, which we have installed previously. Set the corresponding directory and version according to Tomcat that we have installed. To update the properties of this file, create a separate properties file named app.server.\${user.name}.properties with the properties to overwrite in the /portal folder as follows:

Property Name	Property Value
app.server.type	Tomcat
app.server.tomcat.dir	\$CATALINA_HOME
app.server.tomcat.version	\$TOMCAT_MAJOR_VERSION
app.server.tomcat.classes.global.dir	\${app.server.tomcat.dir}/lib

Property Name	Property Value
app.server.tomcat.lib.endorsed.dir	<code> \${app.server.tomcat.dir}/lib/ext</code>
app.server.tomcat.lib.global.dir	<code> \${app.server.tomcat.dir}/lib/ext</code>
app.server.tomcat.lib.support.dir	<code> \${app.server.tomcat.dir}/lib/ext</code>
app.server.tomcat.support.dir	<code> \${app.server.tomcat.dir}/lib/ext</code>

Building via Ant

After installing the application server Tomcat, getting the Liferay portal source code, and customizing properties, we can now build Ext. Sooner or later, you may be interested in getting the latest updates from SVN for Liferay portal source code by following these steps:

1. Right-click on the Java project **portal**, and select **Team**.
2. Then select **Update to HEAD**. You will see update **\$LIFERAY_PORTAL/workspace/portal -r HEAD -force** and **Updates** in **Console** view.
3. If the **Console** view is not open yet, open it in the Eclipse IDE from **Window | Show View**.

Thus, we have the latest code. Now, let's build the Java project **ext** by Ant scripts by following these steps:

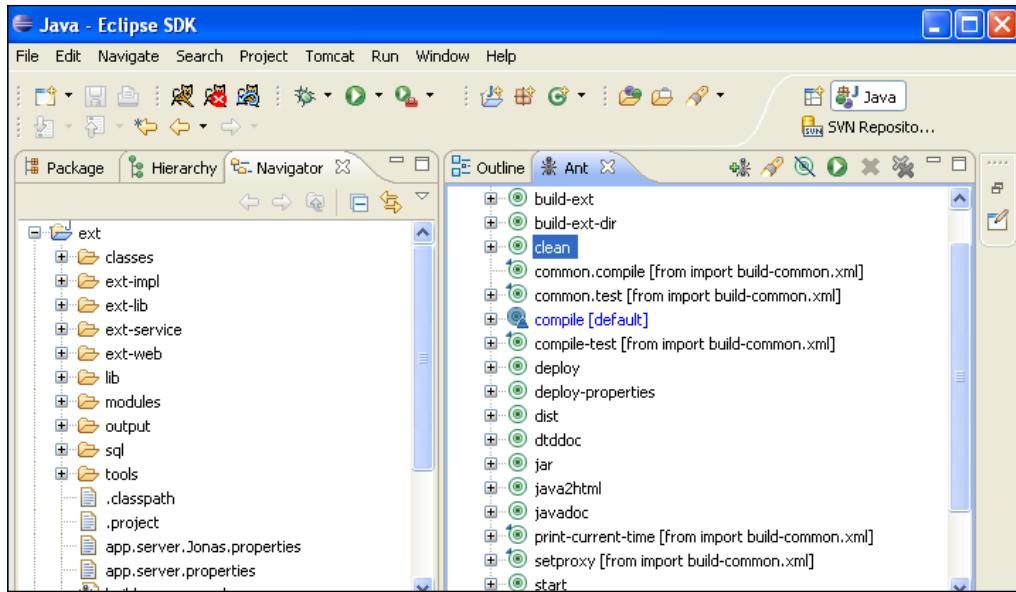
1. Double-click on the target `clean` in the **Ant** view. If the **Ant** view is not open yet, open it in the Eclipse IDE via **Window | Show View** and drag the `build.xml` file under the `/portal` folder to the **Ant** view.
2. Then double-click on the target `start` in the **Ant** view.
3. Double-click on the target `build-ext` in the **Ant** view.

Navigating Ext structures

The Java Project **ext** has been built properly. Now, let's import it into the Eclipse IDE by following these steps:

1. Right-click on the **Navigator** view and select **Import...**
2. Select an import source: **General | Existing Projects into Workspace**.
3. Select root directory: `$LIFERAY_PORTAL/workspace/ext`.
4. Select `ext` (`$LIFERAY_PORTAL/workspace/ext`).
5. Click on the **Finish** button.

Now, you have the Java project ext in the Navigator view, as shown in following screenshot. You can expand the root and folders to view details of the root and the contents under the folders. The source code mainly includes ext-impl, ext-service, ext-web, and so on.



Deploying Ext

We have succeeded in building ext Java project in the Eclipse IDE. Now we can deploy it. Before running the Ant target deploy, we need to set up properties files and configure the databases.

Configuring database

Liferay portal unifies the configuration of the database in a single file for all application servers. Like most of the other configuration options, you can set this up in the `portal-ext.properties` file in the `/ext/ext-impl/src` folder in the following manner:

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

As mentioned earlier, we have created the `lportal` database with `username/password` as `lportal/lportal` in MySQL. The code above shows the configuration of the database MySQL in Liferay portal.

Note that the Liferay portal uses the Apache's Commons Pool for pooling connections to the database. If you prefer to use the pooling mechanism provided by the portal application server, you can still do that through spring. In particular, you can add this to `Ext-spring.xml` file in the `/ext/ext-impl/src/META-INF` folder:

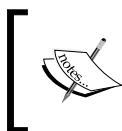
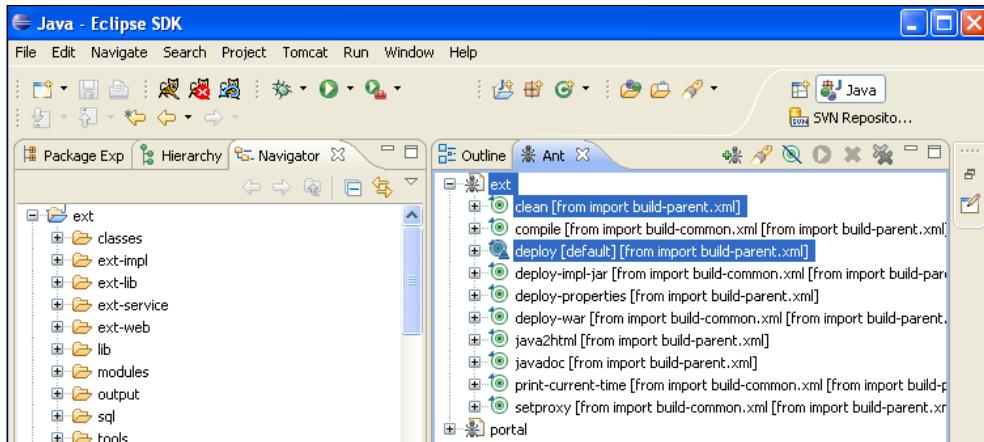
```
// after <beans> add following lines:
<bean id="liferayDataSource"
      class="org.springframework.jdbc.datasource.
      LazyConnectionDataSourceProxy">
  <property name="targetDataSource">
    <bean class="com.liferay.portal.spring.jndi.
      JndiObjectFactoryBean">
      <property name="jndiName" value="jdbc/LiferayPool" />
    </bean>
  </property>
</bean>
```

Using Ant deploy

You have created an `ext` directory—a Java project in the Eclipse IDE. Now, let's set it up for deployment. Similar to the Java project `portal`, we need to set up `build.${user.name}.properties` and `app.server.${user.name}.properties`.

As in both the Java project `ext` and the Java project `portal`, the same JDK environment and application server is used, you can simply copy the two files (`build.${user.name}.properties` and `app.server.${user.name}.properties`) from the `/portal` folder to the `/ext` folder. Then, drag the `build.xml` file in the `/ext` folder to the **Ant** view.

Now, you are ready to run Ant targets: `clean` and `deploy`. After making sure that Tomcat is not running, first double-click on `clean` and then double-click on `deploy` (as shown in following screenshot). This will clean the files currently on Tomcat, and then deploy the custom code in Ext to Tomcat.

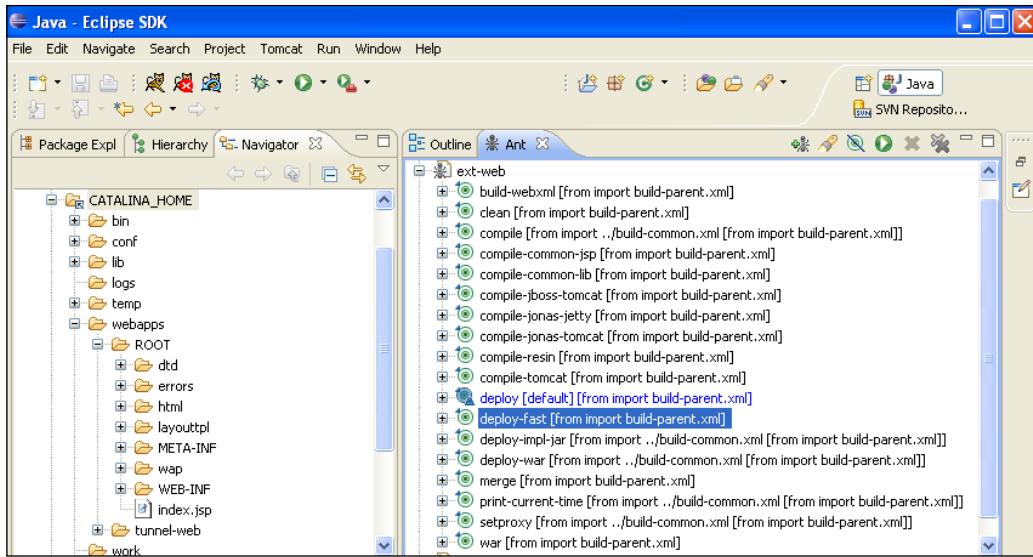


Ext is a fully self-contained version of customized portal core source. The portal source is not needed any more in order to build customized portal. However, it is a good idea to keep it around as a reference.

View portal structures in Tomcat

Now, you can view the structures of the Liferay portal in Tomcat as shown in following screenshot. You can find structures related to Liferay portal as follows:

1. Portal APIs plus JDBC drivers are available in the `/lib/ext` folder.
2. The common loader in the properties file `/conf/catalina.properties` was updated previously—`, ${catalina.home}/lib/ext/*.jar` was added at the end of `common.loader=${catalina.home}/lib, ${catalina.home}/lib/*.jar`
3. Liferay portal plus customization are available at the `/webapps/ROOT` folder. The folders include: `dtd`, `errors`, `html` (common, js, portal, portlet, sound, tablib, and themes), `layouttpl`, `META-INF`, `wap`, and `WEB-INF`.
4. In addition, there is one application by default: `tunnel-web`.



Fast-deploy in Ext

Fast-deploy means that changes in Ext will be ready in the application server immediately. If you change JSP (or JavaScript) files only, then you can use fast-deploy. The following are some simple steps to use the fast-deploy on the JSP files and JavaScript files, as shown in above screenshot:

1. Drag the build.xml file in the /ext/ext-web folder to the Ant view
2. Click on the Ant target: deploy-fast

After the deploy-fast target, you will see changes in the Liferay portal runtime immediately.

Now, you can verify the deployment by following these steps:

1. Click on the Tomcat Start icon. You would see the portal when Tomcat starts properly.
2. After Tomcat starts (the URL is <http://localhost:8080>), sign in as `test@liferay.com/test` via your local browser, as shown in following screenshot.
3. The portal will show you the terms of use; click on **I Agree**.
4. Select a reminder question and enter your answer.

You will see the portal page interface. Congratulations! You now have a running copy of Liferay portal.

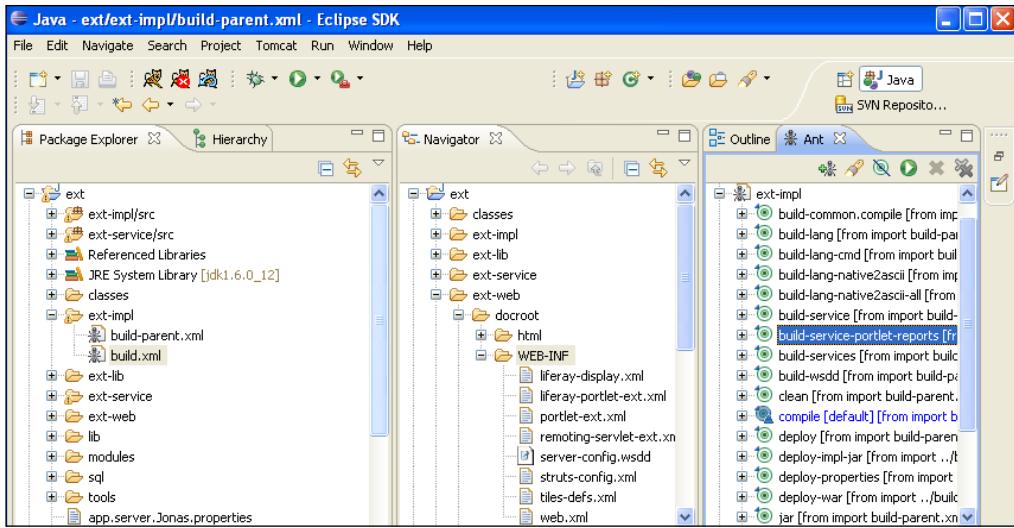


Using ServiceBuilder in Ext

We have set up Ext and updated the latest code successfully. Now, let's use ServiceBuilder with a sample portlet in Ext. In general, ServiceBuilder is a tool included with Liferay portal that can be used to build Java services that can be accessed in a variety of ways, including local access from Java code, remote access using web services, and so on.

Suppose that we want to develop a portlet called Reports in Ext. It has a specific view named Reports. The portlet Reports has the title, Reports, icons (for example, Look and Feel, Configuration, Minimize, Maximize, and Remove), service links (for example, Test123), and persistent data stored in database. All of these features could be developed in Ext environment. Let's have a deeper look at the sample portlet, Reports, and show how to use ServiceBuilder to generate code in Ext as follows:

- View portlet development structures
- Use ServiceBuilder
- View portlet specification



Viewing portlet development structures

By default, we have a portlet called Reports in Ext. Now, let's view the portlet and its development structures. Normally, the portlets are specified (as shown in the above screenshot) in the following folders:

- **ext-service/**: Specifying external services, for example, model, service, and service persistence. There are Java services that can be accessed in a variety of ways such as local access from Java code, remote access using web services, and so on.
- **ext-impl/**: Specifying implementations of the above services (model, service, and service persistence), the portlet definition and view actions. Further, it also includes:
 - portal-ext.properties and system-ext.properties
 - Spring beans, models, and hibernate mappings in the META-INF/ folder, for example, ext-hbm.xml, ext-model-hints.xml, and ext-spring.xml
 - Language-ext.properties in the content/ folder
- **ext-web/docroot/html/**: Describing the portlet view. For example, there are three files for Reports in the portlet/ext/reports/ folder:
 - view.jsp – defining the view of the Reports portlet.

- `init.jsp`—common initiate import for the Reports portlet. This will give us access to the Liferay tag libraries.
- `view_reports.jsp`—for the view when you click on the service links `Test123` for the Reports portlet.
- `ext-web/docroot/WEB-INF/`: Describing the portlet according to the JSR-286 specification. It includes the following items:
 - `liferay-display.xml`—portlets category specification
 - `liferay-portlet-ext.xml`—customized portlets registration
 - `portlet-ext.xml`—customized portlets specification
 - `remote-servlet-ext.xml`—customized service (remote Servlet) specification
 - `server-config.wsdd`—web service specification
 - `struts-config.xml`—actions mappings based on the Struts
 - `tiles-defs.xml`—tiles definition based on the Struts
- `ext-web/tmp/`: A folder—used to integrate the customized code in the `/ext/ext-web/tmp/` folder with Liferay portal source code in the `/portal/portal-web/docroot` folder. This is a temporal folder where the customized Liferay portal exists before it is deployed to Tomcat. It includes portal specifications (for example, `html/common` and `html/portal`), portlet views (for example, `html/portal`), JavaScript (for example, `html/js`), themes, layout templates, portlet specifications (for example, `WEB-INF`), and so on.

Note that you should not change anything in this folder, `/ext/ext-web/tmp/`. All files and subfolders under this folder are generated in the deployment process automatically.

For detailed information about Struts portlet development, you can refer to Chapter 4, *Experiencing Struts Portlets* where it addresses the details related to development of Struts portlet and use the sample portlet as a real example.

Building services

You are not required to write a lot of code in Ext. You can use ServiceBuilder to generate code. Generally speaking, ServiceBuilder is a tool built by Liferay to automate the creation of interfaces and classes that are used by a given portal or portlet. ServiceBuilder is used to build Java services that can be accessed in a variety of ways including local access from Java code, remote access using web services, and so on. Let's use ServiceBuilder to automate the creation of interfaces and classes.

ServiceBuilder, by using an XML descriptor, generates:

- Java Beans
- SQL scripts for database tables creation
- Hibernate Configuration
- Spring Configuration
- Axis Web Services
- JSON JavaScript Interface

Create service XML

First of all, create an XML file in the /ext/ext-impl/src com/ext/portlet/reports folder (the XML file is named `service.xml`) in the following manner:

```
<?xml version="1.0"?>
<!DOCTYPE service-builder PUBLIC "-//Liferay//DTD Service Builder
5.2.0//EN" "http://www.liferay.com/dtd/liferay-service-
builder_5_2_0.dtd">
<service-builder package-path="com.ext.portlet.reports">
    <namespace>Reports</namespace>
    <entity name="ReportsEntry"
        uuid="true"
        local-service="true"
        remote-service="true"
        persistence-class="com.ext.portlet.reports.service.
            ReportsEntryPersistenceImpl">
        <!-- PK fields -->
        <column name="entryId" type="String" primary="true" />
        <!-- Audit fields -->
        <column name="companyId" type="String" />
        <column name="userId" type="String" />
        <column name="userName" type="String" />
        <column name="createDate" type="Date" />
        <column name="modifiedDate" type="Date" />
        <!-- Other fields -->
        <column name="name" type="String" />
        <!-- Order -->
        <order by="asc">
            <order-column name="name" case-sensitive="false" />
        </order>
        <!-- Finder methods -->
        <finder name="CompanyId" return-type="Collection">
            <finder-column name="companyId" />
        
```

```
</finder>
<finder name="UserId" return-type="Collection">
    <finder-column name="userId" />
</finder>
</entity>
<exceptions>
    <exception>EntryName</exception>
</exceptions>
</service-builder>
```

The code above shows ServiceBuilder and the entity reports. It first specifies the version of ServiceBuilder, the package path, com. ext .portlet .reports, and the namespace, Reports. It also specifies the entity, Reports. The XML file specifies an entity ReportsEntry with UUID, local services, and remote-services, which tells ServiceBuilder how to generate the related code. The entity has fields: entryId, companyId (the ID of company where the portal belongs), userId (the ID of the user), username (the name of the user), createDate (date of creation of the reports), modifiedDate (date of modification of the reports), and Name (the name of the reports).

In fact, the XML file service.xml was re-defined in the /ext/ext-impl/src com/ ext/portlet/reports folder, when we did the Ant target build-ext to build Ext in the previous section. Normally, it is better to put the XML file service.xml in the package path, for instance, com.ext.portlet.reports. Moreover, the services related to the portlet Reports have been generated by ServiceBuilder, by default. In order to show how ServiceBuilder works, it is better to remove the existing code. You can do it as follows:

1. Switch to the **Java** perspective, if you are not in the Java perspective yet.
2. Click on the **Package Explorer** view (If the view **Package Explorer** is not open yet, open it in the Eclipse IDE via **Window | Show View**).
3. Delete the packages: com.ext.portlet.reports.model, com.ext.portlet. reports.service, and com.ext.portlet.reports.service.persistence from the folder /ext/ext-service/src, and also delete the packages: com. ext.portlet.reports.model.impl, com.ext.portlet.reports.service. base, com.ext.portlet.reports.service.http, com.ext.portlet. reports.service.impl, and com.ext.portlet.reports.service. persistence from the folder /ext/ext-impl/src.

Now, it is clean and you are ready to use ServiceBuilder for the portlet, Reports.

Build services

service.xml is ready. Now, you can build services by ServiceBuilder as follows:

1. Drag build.xml in the folder /ext/ext-impl to the **Ant** view.
2. Expand Ext-impl in the **Ant** view.
3. Double-click on the Ant target, build-service-portlet-reports.

ServiceBuilder will create models, services, and service persistence first in the folder ext/ext-service, which will form ext-service.jar and it goes to the \$CATALINA_HOME/lib/ext in Ext deployment. Here are the packages of the services for the portlet, Reports:

- com.ext.portlet.reports.model
- com.ext.portlet.reports.service
- com.ext.portlet.reports.service.persistence

Then ServiceBuilder will create implementations of models, services and service persistence in the folder, ext/ext-impl. This will form ext-impl.jar and it goes to \$CATALINA_HOME/webapps/ROOT/WEB-INF/lib in Ext deployment. Here are packages of the implementations of the services for the portlet, Reports:

- com.ext.portlet.reports.model.impl
- com.ext.portlet.reports.service.base
- com.ext.portlet.reports.service.http
- com.ext.portlet.reports.service.impl
- com.ext.portlet.reports.service.persistence

In addition, under the /ext/ext-impl/src/META-INF folder, it also generates hibernate-mapping in the XML file, ext-hbm.xml, model hints ext-model-hints.xml, and Spring beans configuration ext-spring.xml.

In short, the code above generated by ServiceBuilder follows well-defined design patterns and closely matches the most common J2EE Enterprise design patterns.

What's happening?

We have used the Ant target, build-service-portlet-reports, to generate code for Services, Spring, Persistence, and Model. This Ant target is specified in the XML file, build-parent.xml under the /ext/ext-impl/ folder as follows:

```
<target name="build-service-portlet-reports">
    <antcall target="build-service">
        <param name="service.file"
            value="src/com/ext/portlet/reports/service.xml" />
    </antcall>
</target>
```

This code shows the Ant target, `build-service-portlet-reports`. This target calls the Ant target, `build-service` with a parameter, `service.file`. As you can see, the value of the parameter `service.file` is `src/com/ext/portlet/reports/service.xml`.

Similarly, you can set your own Ant target different from that of `build-service-portlet-reports`, or you can use the same target name, but have a different value for the parameter, `service.file`.

By the way, the Ant target `build-service` is specified in the XML file, `build-parent.xml` that is also in the folder, `/ext/ext-impl/`. For more details, you can find the class of ServiceBuilder `com.liferay.portal.tools.servicebuilder.ServiceBuilder` in the `/portal/portal-impl/src` folder. You can also find ServiceBuilder-related objects: `Entity`, `EntityColumn`, `EntityFinder`, `EntityMapping`, and `EntityMapping`.

Navigating portlet specification

Now, you can navigate the sample portlet specification in the folder, `ext-web/docroot/WEB-INF`. You will find the following files to specify the portlet, Reports:

- `portlet-ext.xml`: Defining the custom portlets:

```
<portlet>
    <portlet-name>EXT_1</portlet-name>
    <display-name>Reports</display-name>
    <portlet-class>
        com.ext.portlet.reports.ReportsPortlet
    </portlet-class>
    <init-param>
        <name>view-action</name>
        <value>/ext/reports/view_reports</value>
    </init-param>
    ...
</portlet>
```
- `liferay-portlet-ext.xml`: Defining custom portlet registration in Liferay portal.

```
<portlet>
    <portlet-name>EXT_1</portlet-name>
    <struts-path>ext/reports</struts-path>
    <use-default-template>false</use-default-template>
</portlet>
```

- `liferay-display.xml`: Showing the categories in which the portlet should appear in the Liferay portal, for example, `EXT_1` in the categories, `category.sample`. Besides this customized portlet and its category, you can also find a lot of default categories with the out of the box portlets. Comparing the XML file `liferay-display.xml` with the XML file `/portal/portal-web/docroot/WEB-INF/liferay-display.xml`, you will see the difference. Obviously, the customized portlet and its categories are merged in the XML, `liferay-display.xml` directly. Ideally, it should be merged in the `/ext/ext-web/tmp` only. This would be improved by updating the Ant target `deploy`.

```
<display>
... // other categories
<category name="category.sample">
... // other portlets
<portlet id="EXT_1" />
</category>
... // other categories
</display>
```

- `struts-config.xml`: Describing action mappings based on the Struts.

```
<action path="/ext/reports/view_reports"
       type="com.ext.portlet.reports.action.ViewReportsAction">
    <forward name="portlet.ext.reports.view"
             path="portlet.ext.reports.view" />
    <forward name="portlet.ext.reports.view_reports"
             path="portlet.ext.reports.view_reports" />
</action>
```

- `tiles-defs.xml`: Describing tiles based on the Struts.

```
<definition name="portlet.ext.reports.view" extends="portlet">
    <put name="portlet_content"
         value="/portlet/ext/reports/view.jsp" />
</definition>
<definition name="portlet.ext.reports.view_reports"
            extends="portlet">
    <put name="portlet_content"
         value="/portlet/ext/reports/view_reports.jsp" />
</definition>
```

- In addition, there are `remoting-servlet-ext.xml`—describing remote servlet calls and `server-config.wsdd`—describing web services calls.



For the detailed information about portlet specification, you can refer to Chapter 2, *Working with JSR-286 Portlets*, where it addresses the details related to portlet specification JSR-286 and use the sample portlet as a real example.

Setting up Plugins SDK

Plugins SDK is a simple environment for the development of Liferay plugins, for example, themes, layout templates, portlets, hooks, and webs (web applications). It is completely different from the Liferay portal core services as it uses external services only if required.

How do we set up Plugins SDK? First, let's see where we could find the source code of Plugins SDK. In general, there are three kinds of source code for Plugins SDK: specific version package, branch version, and trunk version.

The **specific** version package is released with the portal. You can get the latest release from <http://www.liferay.com/web/guest/downloads/additional>. Note that there is only one version (either major version, for example, 5.3, or minor version, for example, 2). If you need the new version of Plugins SDK, say 5.3.3 or 5.4.3, you have to get the latest version and install it again.

The **branch** version provides source code of Plugins SDK with a fixed major version. You can check out the latest branch from <svn://svn.liferay.com/repos/public/plugins/branches>. Note that there is only one major version (for example, 5.3), but you can get the latest minor version, for instance, 2, 3, and 4, by SVN Update. But if you need a new major version of Plugins SDK, say 5.4 or 5.5, you have to get another branch and to install it again.

The **trunk** version provides source code of Plugins SDK with the latest version, both the major and the minor versions. You can check this out at <svn://svn.liferay.com/repos/public/plugins/trunk>. Note that by using SVN update, you can get the latest version eventually. By the way, both branch version and trunk version contain a lot of sample themes, layout templates, portlets, hooks, and webs.

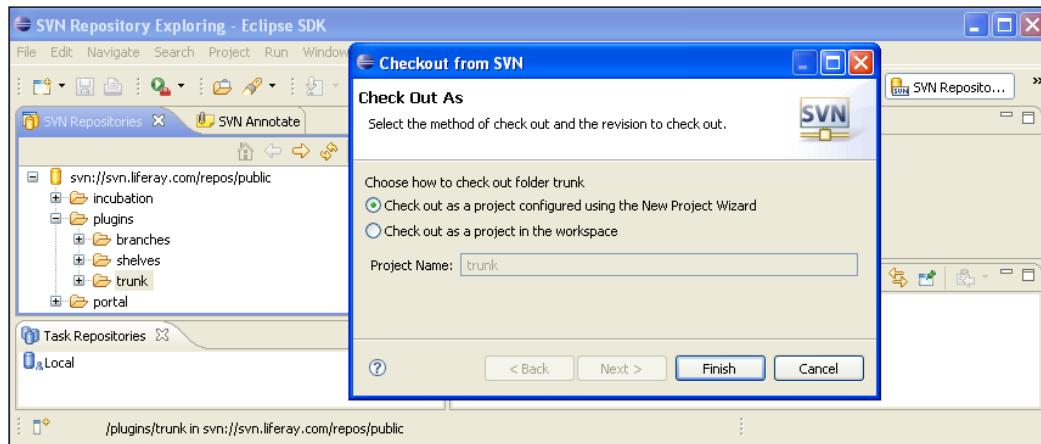
Of course, it is then fine for developers to choose one of them. For demo purposes, we will use the trunk version only.

Building Plugins SDK project

As shown in screenshot, you can use the following instructions to set Plugins SDK project up:

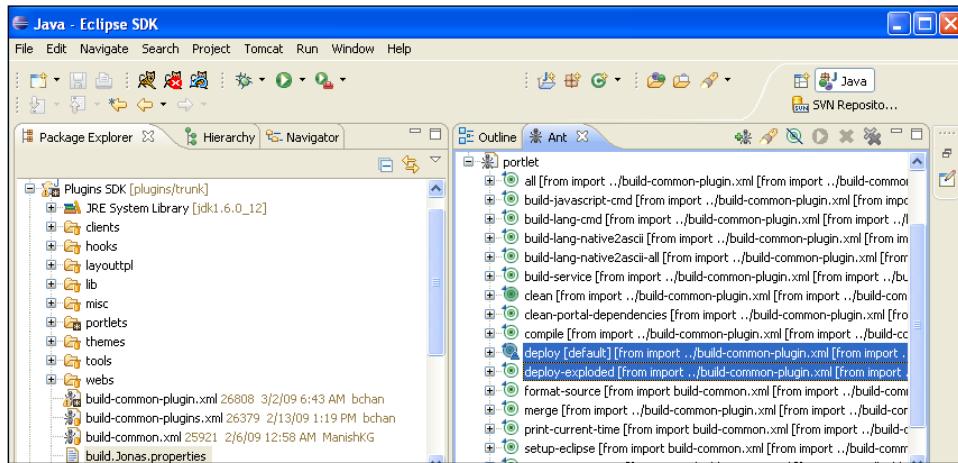
1. In the Eclipse IDE, switch to the perspective, **SVN Repository Exploring**.
2. Select the root: `svn://svn.liferay.com/repos/public`.
3. Expand the root.
4. Expand `plugins` and further `trunk`, and check out `trunk` using the Java project in the workspace.
5. Name it Plugins SDK. Of course, you can have any name for this project; we use the name Plugins SDK only for ease of reference.

As you can see, we should set `$PLUGINS_SDK_HOME` as `$LIFERAY_PORTAL/workspace/Plugins SDK/`.



Deploying plugins

Later, you can view the folders of Plugins SDK. As shown in following screenshot, there are several folders for different plugins: themes, layouttpl, hooks, portlets, and webs. That is, plugins themes go to the folder /themes, plugins layout templates go to the folder /layouttpl, plugins hooks go to the folder /hooks, plugins portlets go to the folder /portlets, and plugins web applications go to the folder /webs.



There are two ways in which we can deploy portlets in Plugins SDK – all portlets or individual portlets. If you want to deploy all portlets in Plugins SDK at the same time, you can drop the XML file `build.xml` (in the folder, `$/PLUGINS_SDK_HOME/portlets`) into the **Ant** view. In the **Ant** view, under `portlets`, you will see the Ant target `deploy` – using this, you can deploy all the portlets into the portal at one go.

If you want to deploy individual portlet in Plugins SDK, drop the XML file `build.xml`, (in the folder, `$/PLUGINS_SDK_HOME/portlets/{plugin.name}-portlet`) into the **Ant** view. In the **Ant** view, under the `portlet`, you will see the Ant target `deploy` as shown in above screenshot. Using this, you can deploy current portlet into the portal.

Before deploying, we need to tell Plugins SDK which version of JDK you are using, which application server the plugins will be running on, where auto-deploy directory is, and so on. These properties are set with default values in the properties file, `build.properties` under the folder, `$/PLUGINS_SDK_HOME`. Of course, your local properties would be different from these default properties. If they are different, you need to override these values. To update the properties of this file, create a separate properties file named `build.${user.name}.properties` with the properties to overwrite in the folder, `$/PLUGINS_SDK_HOME`. The following is a sample code for `build.${user.name}.properties`.

```
## Compiler
ant.build.javac.source=${JDK_MAJOR_VERSION}
ant.build.javac.target=${JDK_MAJOR_VERSION}
## Auto Deploy
auto.deploy.dir=$LIFERAY_PORTAL/deploy
## Application Server
app.server.dir=$CATALINA_HOME
app.server.classes.portal.dir=${app.server.portal.dir}/WEB-INF/classes
app.server.lib.global.dir=${app.server.dir}/lib/ext
app.server.lib.portal.dir=${app.server.portal.dir}/WEB-INF/lib
app.server.portal.dir=${app.server.dir}/webapps ROOT
```

The code above shows different values for the compiler, auto-deploy directory, and the application server.

Fast development of plugins with Tomcat

What's fast development of plugins? Fast development allows the developers to work with exploded plugin WAR, instead of having to package them for deployment. For example, if you change JSP files in a plugin, these JSP files will be modified when you refresh the page in your browser. Further, if you update other files (for example, JSF pages, Java beans, servlets, and so on) besides JSP files, these files will automatically be reloaded by the class loader of the Tomcat. Obviously, it will save a lot of development time.

How do we make it happen? First, we need to tell Tomcat that the context property `reloadable` should be true. To do so, open the `$CATALINA_HOME/conf/context.xml` file, replace the line `<Context>` by `<Context reloadable="true">`, and restart Tomcat.

Then, we need to add a new target `deploy-explored` inside the `$PLUGINS_SDK_HOME/build-common-plugin.xml` file as follows:

```
// after <target name="deploy" depends="war">
// <copy file="${plugin.file}" todir="${auto.deploy.dir}" />
// </target>, add following lines:
<antelope:stringutil string="${basedir}/${plugin.name}.xml"
property="plugin.context.file">
<antelope:replace regex="\\" replacement="/" />
</antelope:stringutil>
<target name="deploy-explored" depends="compile">
<copy file="${plugin.context.file}" todir="${auto.deploy.dir}" />
</target>
```

Finally, we need to create a plugin context file for that specific plugin, pointing to the exploded plugin WAR. The plugin context file is an XML file called in a similar manner a plugin is called. For example, if the plugin is a portlet called `ipc-faq-portlet`, then the plugin context file must be called `ipc-faq-portlet.xml`, and the content will look like this:

```
<Context  
    path="${plugin.name}"  
    docBase="$PLUGINS_SDK_HOME/portlets/${plugin.name}-portlet/docroot"  
/>
```

The code above shows the content of plugin context file. `${plugin.name}` represents the plugin name, for example, `ipc-faq`. `$PLUGINS_SDK_HOME` represents the home of Plugins SDK.

When you are ready, you can click on the Ant target, `deploy-explored`, from `build.xml` plugin in the **Ant** view. From now on, if you change JSP files, they are modified when you refresh the page in your browser. For Java classes and XML files, you need to click on the `compile` target from the plugin, `build.xml` in the **Ant** view. Generally, you can now update JSP file, JSF pages, Java beans, servlets, and so on, and they will automatically be reloaded by the class loader of the Tomcat.

What's happening? Once you click on the `deploy-explored` target, the plugin context file will be copied into the auto-deploy directory first. Then the Liferay portal will copy the plugin context file to the folder, `$CATALINA_HOME/conf/Catalina/localhost`. Later, the plugin will be registered. If the WAR was previously deployed, a new copy of the context file to the auto-deploy directory will cause a re-deploy of the exploded plugin WAR.

Note that if you had deployed your plugin before, it is better to remove the old deployed application and restart Tomcat.

Using development environments efficiently

Ext is a complete development environment that eases customizing Liferay portal to requirements. It integrates several Liferay tools that can optionally be used to develop portlets and the portal, for example, JSP portlets, Struts portlets and even ServiceBuilder.

How does Ext work?

Within the Ext environment, there is a directory called `ext-impl/`. Within this directory, there are JARs and WAR files that encapsulate the entire core source. That is, you could deploy and run Liferay with Ext environment alone. There is another directory called `ext-service/`. Within this directory, you could specify your external models and services.

Ext has `Ext-web/docroot/html/` folder where you could have all JSP files or JavaScript files. Ext also has `web.xml` under `Ext-web/docroot/WEB-INF/` folder that you can manipulate. Under this folder, you could have all your classes, configure files, XML files, and so on.

Last, but not the least, using Ext means that you can develop your portlets as an extension of Liferay portal source code. That is, you could build your own portlets as if you were writing them for Liferay portal core functions. You are not required to build WAR files for your portlets. The only thing you need to do is to leverage all of Liferay portal utility classes and tags as well.

When do we use Ext?

When should we use Ext? There are two kinds of services in the Liferay portal: internal services (that is, Liferay core services, for instance, `portal-impl.jar`) and external services (that is, Portal's API, for instance, `portal-kernel.jar` and `portal-service.jar`). Normally, the Liferay core services will be updated regularly, but External services are relatively stable. On the one hand, Ext may use both core services and external services. Therefore, the upgrade from a lower version to higher version becomes complicated—it depends on how many changes the core services have undergone. On the other hand, Ext no longer supports portlets and themes development, except for Struts portlet and modifying or extending the out of the box portlets. Moreover, customizations and extensions will be kept separate from the Liferay portal source code.

Meanwhile, Plugins SDK is a simple environment for the development of Liferay plugins. It is completely separate from the Liferay portal core services—using external services only if required. This forces the portlets to rely completely on the portal's API `portal-kernel.jar` and `portal-service.jar`, and not to depend on implementation classes defined in `portal-impl.jar`. The upgrade from a lower version to a higher version becomes simple. At the same time, portlets in Plugins SDK can make use of any application framework that Liferay supports, for example, JSF, JSP, Ruby, Lazzlo, PHP, JSON, Hibernate, DAO, Spring MVC, Struts, Tapestry, Wicket, Python, LAR (Liferay Archive), and so on. Therefore, if the work has nothing related to extension and customization of the out of the box portlets, you should use Plugins SDK environment.

Summary

This chapter discussed how to set up, build, and deploy Ext in the Eclipse IDE. Then it discussed how to use ServiceBuilder to generate code in Ext. It also addressed how to set up Plugins SDK environment in the Eclipse IDE, and how to set up fast development of plugins with Tomcat. Finally, it talked about how to use the development environments efficiently.

In the next chapter, we're going to use Ext, mainly to build a Struts portlet with several features: ServiceBuilder, models, services, views, actions, permissions, and so on.

4

Experiencing Struts Portlets

In this chapter , we are going to build dynamic, content-rich, and ad-serving Internet web sites(www.bookpubstreet.com and www.bookpubworkshop.com) in the intranet web site bookpub.com for "Palm Tree Publications" on top of the Liferay portal. Each web site is made up of a set of portlets, involving content management in the backend and public sites and/or **My Street** in the frontend. In the previous chapter, we discussed how to use ServiceBuilder and development environments, especially Ext. Now we are going to use Ext for customization and extension on portlets.

Why **Struts** portlets? Liferay portal provides a set of out of the box portlets, that is, Struts portlets. Thus, we need to know more about Struts portlets in Ext before customizing and extending them. Apache Struts is an elegant, extensible framework for building enterprise-ready Java web applications using **Model-View-Controller (MVC)** architecture. The **Model** represents the business or database code, the **View** represents the page design code, and the **Controller** represents the navigational code. Normally, JSP files are used to build the view of Struts.

This chapter first introduces how to develop a JSP portlet, focusing on the view part and portlet structure. Next, it addresses how to construct basic struts portlets by viewing the title and adding an action only. Further, this chapter discusses how to build advanced struts portlets. Finally, it discusses how to use Struts efficiently.

By the end of this chapter, you will have learned how to:

- Develop a JSP portlet
- Construct a basic Struts portlet
- Build an advanced Struts portlet
- Use Struts efficiently

Developing a JSP portlet

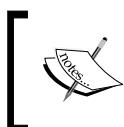
First of all, let's consider an example requiring portlet development in Ext.

Suppose we need to develop a simple portlet to display static content as shown in the following screenshot. It has the message **JSP Portlet for Palm Tree** and the **Palm-Tree Publications** logo. They both have the title **Liferay Portal Enterprise Intranets** and the reference link <http://liferay.cignex.com>.



So how do you develop a portlet with the view shown in the screenshot above?

A JSP portlet would be the best choice on top of Liferay portal. We can develop a JSP portlet with the above view in Ext of Liferay portal. First, we will define a JSP portlet; then we will change the title and category of the portlet.



Java Server Pages (JSP) technology provides a simplified and fast way to create dynamic web content. JSP technology enables rapid development of web-based applications that are server- and platform-independent.
Refer to <http://java.sun.com/products/jsp>.



Defining the JSP portlet

Now let's define a JSP portlet with the name `jsp_portlet`. Here we need to configure the JSP portlet `jsp_portlet` in both `portlet-ext.xml` and `liferay-portlet-ext.xml` first, then create the JSP page `view.jsp`, and finally deploy it to Tomcat.

First, let's configure the JSP portlet `jsp_portlet` in the XML file `portlet-ext.xml` as follows:

1. Locate the `portlet-ext.xml` file in the `/ext/ext-web/docroot/WEB-INF` folder and open it.
2. Add the following lines between `</portlet>` and `</portlet-app>`, and save the file:

```
<portlet>
    <portlet-name>jsp_portlet</portlet-name>
    <display-name>JSP Portlet Introduction</display-name>
    <portlet-class>
        com.liferay.util.bridges.jsp.JSPPortlet
    </portlet-class>
```

```
<init-param><name>view-jsp</name>
    <value>/html/portlet/ext/jsp_portlet/view.jsp</value>
</init-param>
<expiration-cache>0</expiration-cache>
<supports><mime-type>text/html</mime-type></supports>
    <resource-bundle>
        com.liferay.portlet.StrutsResourceBundle
    </resource-bundle>
<security-role-ref>
    <role-name>power-user</role-name>
</security-role-ref>
    <security-role-ref><role-name>user</role-name>
</security-role-ref>
</portlet>
```

This code shows portlet definition. The `portlet-name` element contains the canonical name of the portlet. This name can be anything. We've simply chosen `jsp_portlet` to follow the conventions. Moreover, each portlet name is unique within the portlet application. The `display-name` element contains a short name that is intended to be displayed in Liferay portal. Here, `display-name` has the value `JSP Portlet Introduction`. Again, you can have any value for `display-name`. The `portlet-class` element contains the fully qualified class name of the portlet `com.liferay.util.bridges.jsp.JSPPortlet`. The `init-param` element contains a name-value pair `view-jsp - /html/portlet/ext/jsp_portlet/view.jsp` as an initialization parameter of the portlet.

Further, the `expiration-cache` element defines expiration-based caching for this portlet. The parameter indicates the time in seconds, for example, after 0 the portlet output expires; -1 indicates that the output never expires. The `supports` element contains the supported MIME-type. The `resource-bundle` element contains this resource bundle class, that is, `com.liferay.portlet.StrutsResourceBundle`. Finally, the `security-role-ref` element contains the declaration of a security role reference in the web application's code.

Secondly, let's configure the `jsp_portlet` portlet in the XML file `liferay-portlet-ext.xml` as follows:

1. Locate the `liferay-portlet-ext.xml` file in the `/ext/ext-web/docroot/WEB-INF` folder and open it.
2. Add the following lines immediately after `<!-- Custom Portlets -->` and save the file:

```
<portlet><portlet-name>jsp_portlet</portlet-name>
</portlet>
```

The code above indicates that the `jsp_portlet` portlet is registered in the Liferay portal as well.

Thirdly, let's create the JSP page view.jsp with init.jsp as follows:

1. Locate the /ext folder in the /ext/ext-web/docroot/html/portlet folder.
2. Create a folder named jsp_portlet in the /ext/ext-web/docroot/html/portlet/ext folder.
3. Create a file named init.jsp in the /ext/ext-web/docroot/html/portlet/ext/jsp_portlet folder with the following content and save it:

```
<%@ include file="/html/common/init.jsp" %>
<portlet:defineObjects />
```
4. Create a file named view.jsp in the /ext/ext-web/docroot/html/portlet/ext/jsp_portlet folder. Add the following content and save it:

```
<%@ include file="/html/portlet/ext/jsp_portlet/init.jsp" %>
<a href="http://liferay.cignex.com" title="Liferay Portal
Enterprise Intranets">JSP Portlet for Palm Tree</a>
<a href="http://liferay.cignex.com" title="Liferay Portal
Enterprise Intranets"></a>
```

Finally, we can deploy the files into Tomcat as follows:

1. Drag the build.xml file in the /ext/ext-web folder to the Ant view, if the build.xml file is not in the **Ant** view.
2. Stop Tomcat if it is running.
3. Click on the Ant target deploy.
4. Start Tomcat.
5. Open up a new browser with the URL <http://localhost:8080> and click on **Sign in**.
6. Log in as `test@liferay.com/test`.
7. Click on **Add Application**, and then click on **Undefined**.
8. Click on `javax.portlet.title.jsp_portlet`.

Congratulations! You have developed the JSP portlet jsp_portlet. Click on **Sign out** and you will see the JSP portlet with the title `javax.portlet.title.jsp_portlet` and its content—a message and an image with reference links.

Changing the title and category

As mentioned earlier, the JSP portlet has the title `javax.portlet.title.jsp_portlet` and it is under the `Undefined` category by default. Now, let's change the title and the category. We need to set the title in `Language-ext.properties` and add the JSP portlet `jsp_portlet` to the `Sample` category in `liferay-display.xml`. Then it can be deployed to Tomcat.

First, add the JSP Portlet title in the properties file, `Language-ext.properties` as follows:

1. Locate the `Language-ext.properties` file in the `/ext/ext-impl/src/content` folder and open it.
2. Add the following line of code after the line `javax.portlet.title.EXT_1=Reports` and save it:

```
javax.portlet.title.jsp_portlet=JSP Portlet
```

The code above maps the title from `javax.portlet.title.jsp_portlet` to `JSP Portlet`.

Next, add the `jsp_portlet` portlet to the `Sample` category in the XML file `liferay-display.xml` as follows:

1. Locate the `liferay-display.xml` file in the `/ext/ext-web/docroot/WEB-INF` folder and open it.
2. Add the following line of code after the line `<portlet id="EXT_1" />` and save it:

```
<portlet id="jsp_portlet" />
```

The code above shows that the `jsp_portlet` portlet has been added in the `Sample` category.

Finally, we can deploy the files into Tomcat as follows:

1. Stop Tomcat if it is running.
2. Click on the Ant targets `clean` and `deploy`.
3. Start Tomcat

You will see the `jsp_portlet` portlet with the `JSP Portlet` title and contents – message and an image with reference links. Note that the `jsp_portlet` portlet with the title `JSP Portlet` is under the `Sample` category now.

Using JSP portlet efficiently

We have created the JSP portlet `jsp_portlet`. Now let's consider more details about the view of the portlet. The view of the `jsp_portlet` portlet is changeable—we plan to update the message to **Palm Tree Publications**, centralize the message and the image, and add blank spaces between the message and the image. All this is shown in the following screenshot:



In this case, we only need to update the view page `view.jsp` and deploy fast to Tomcat while it is still running.

Fast deploy

Let's update the view page `view.jsp` and deploy quickly as follows:

1. Locate the `view.jsp` file in the `/ext/ext-web/docroot/html/portlet/ext/jsp_portlet` folder and open it.
2. Rewrite the `view.jsp` file completely with the following content and save it:

```
<%@ include file="/html/portlet/ext/jsp_portlet/init.jsp" %>
<center><a href="http://liferay.cignex.com" title="Liferay Portal
Enterprise Intranets">Palm Tree Publications</a>
</center>
<br/>
<center><a href="http://liferay.cignex.com" title="Liferay Portal
Enterprise Intranets"></a>
</center>
```
3. Click on the Ant target `deploy-fast`.

Reloading the current page in the browser, you will see the updated view of the JSP portlet that was shown in the above screenshot.

In short, you can use the Ant target `deploy-fast` when there are only JSP files and JavaScript files updated in the `/ext/ext-web/docroot/html` folder. You do not need to restart the application server. Just click on the Ant target `deploy-fast` first, and then reload the current page in the browser. You will see the updated view immediately.

Employing JSP portlet

As we know, JSP files are HTML files with special tags containing Java source code that provide dynamic content. It is easy to learn and allow developers to quickly produce web sites and applications in an open and standard way.

Moreover, a JSP portlet is made up of the typical MVC components:

- Model—Java objects put in as request attributes
- View—template, for example, `view.jsp`
- Controller—JSP action

The controller specifies the JSP portlet actions. The view provides a JSP template, which will generate the content of the portlet by pulling out dynamic model information from the request attributes. The standard JSP variables involve `request`, `response`, `session`, and so on. The model is made up of Java objects put in as request attributes. For example, the variable `UID` (the unique identity of assets) is available in the model. It can be retrieved from the request attribute, as in the following example:

```
String uid = (String) request.getAttribute("uid");
```

JSP portlets are basically about dynamic content. If you have static content, there is no real benefit of using a JSP portlet. Of course, you can use a JSP portlet for static content generation. In this case, the basic function of JSP is really very simple. It provides a similar function to an HTML portlet as well as what we have discussed here. The HTML portlet presents HTML—a static content—in a portlet.

A JSP portlet is a good starting point if you are new to Liferay portal. On the one hand, JSP portlets help you to learn the basic file and directory structure that you will need to know in order to develop JSP portlets using `portlet-ext.xml` and `liferay-portlet-ext.xml`. They further allow you to develop generic portlets within Liferay Portal. On the other hand, JSP portlets show how to add a title and category using `Language-ext.properties` and `liferay-display.xml`.

In short, the following are the main steps to build JSP portlets on top of Liferay portal:

1. Define portlets (JSR-286 attributes) in `portlet-ext.xml`.
2. Register portlets (Liferay portal attributes) in `liferay-portlet-ext.xml`.
3. Create JSP pages: `view.jsp`, `init.jsp`.
4. Map the title to a value in `Language-ext.properties`.
5. Add the portlet to a category in `liferay-display.xml`.

Constructing a basic Struts portlet

As shown in the next screenshot, the **Book Reports** portlet lists the reports about books with the report name as a link. Clicking on this link will allow the user to view the report details. So how can we build the `book_reports` portlet? As mentioned earlier, the view is a JSP template, which will generate the content of the portlet by pulling out dynamic model information from the request attributes. We have provided static content from the JSP file. So how can we generate dynamic content in these JSP? Let's construct a basic Struts portlet—using `book_reports` as an example—to see how to build the portlet and how to generate dynamic content in the JSP files of the portlet.



As shown in the following screenshot, we are planning to build a struts portlet `book_reports` with the following view—a link with the message Preference, a title Book Reports, the preference name Liferay Book, and the value Liferay Portal Enterprise Intranets.

Moreover, after the link is clicked, it will bring us to another view: **View Reports**. The view of reports includes the message name-value pair, the static content (a text and an image with a link), preference name and value, and other dynamic content (current window state, portlet mode, portlet session, and so on).



Let's take a deeper look at how to construct a basic Struts portlet in order to generate the above two JSP pages.

Defining a Struts portlet

Now, let's define a Struts portlet with the name `book_reports`. Similar to the JSP portlet, we need to define the Struts portlet, and configure the Struts portlet `book_reports` in both `portlet-ext.xml` and `liferay-portlet-ext.xml`.

First, let's define the Struts portlet `book_reports`. Create a package named `com.ext.portlet.bookreports` in the `ext/ext-impl/src` folder. Then create the portlet class named `BookReportsPortlet`, which extends `StrutsPortlet` under the `com.ext.portlet.bookreports` package in the `/ext/ext-impl/src` folder as follows:

```
public class BookReportsPortlet extends StrutsPortlet {
    public void doView(RenderRequest renderRequest, RenderResponse
        renderResponse) throws IOException, PortletException {
        super.doView(renderRequest, renderResponse);
    }
}
```

The code above shows that the `BookReportsPortlet` portlet extends `StrutsPortlet` and the portlet mode `VIEW` is specified. You can further specify other modes, for example, `HELP` and `EDIT`.

Then let's configure the Struts portlet `book_reports` in `portlet-ext.xml` by following these steps:

1. Locate the `portlet-ext.xml` file in the `ext/ext-web/docroot/WEB-INF` folder and open it
2. Add the following lines between `</portlet>` and `</portlet-app>`, and save them:

```
<portlet>
    <portlet-name>book_reports</portlet-name>
    <display-name>Book Reports</display-name>
    <portlet-class>
        com.ext.portlet.bookreports.BookReportsPortlet
    </portlet-class>
    <init-param><name>view-action</name>
        <value>/ext/book_reports/view_reports</value>
    </init-param>
    <expiration-cache>0</expiration-cache>
    <supports><mime-type>text/html</mime-type></supports>
    <resource-bundle>
        com.liferay.portlet.StrutsResourceBundle
    </resource-bundle>
    <portlet-preferences>
        <preference>
            <name>Liferay Book</name>

```

```
<value>Liferay Portal Enterprise Intranets</value>
</preference>
</portlet-preferences>
<security-role-ref>
    <role-name>power-user</role-name>
</security-role-ref>
<security-role-ref>
    <role-name>user</role-name>
</security-role-ref>
</portlet>
```

The code above shows the definition of the `book_reports` portlet. The `portlet-name` element contains the canonical name of the portlet: `book_reports`. Moreover, each portlet name is unique within the portlet application. The `display-name` element contains a short name that is intended to be displayed in the Liferay portal. Here the `display-name` has the value Book Reports. The `portlet-class` element contains the fully qualified class name of the portlet, `com.ext.portlet.bookreports.BookReportsPortlet`. The `init-param` element contains a name-value pair `view-action - /ext/book_reports/view_reports` as an initialization parameter of the portlet. Similarly, the `preference` element contains the name-value pair `Liferay Book - Liferay Portal Enterprise Intranets` as an initialization parameter for the preference of the portlet.

The `expiration-cache` element defines expiration-based caching for this portlet. The parameter indicates the time in seconds. After `0`, the portlet output expires; `-1` indicates that the output never expires. The `supports` element contains the supported MIME-type and portlet modes: `VIEW`, `HELP`, `EDIT`, and `CONFIG`—the customized portlet mode provided by Liferay portal. The `resource-bundle` element contains resource bundle class: `com.liferay.portlet.StrutsResourceBundle`. Finally, the `security-role-ref` element contains the declaration of a security role reference in the code of the web application.

Finally, let's configure the Struts portlet `book_reports` in `liferay-portlet-ext.xml` as follows:

1. Locate the `liferay-portlet-ext.xml` file in the `/ext/ext-web/docroot/WEB-INF` folder and open it.
2. Add the following lines immediately after `<!-- Custom Portlets -->` and save the file:

```
<portlet>
    <portlet-name>book_reports</portlet-name>
    <struts-path>ext/book_reports</struts-path>
    <use-default-template>false</use-default-template>
</portlet>
```

The preceding code indicates that the `book_reports` portlet is registered in the Liferay portal as well. Liferay will check `struts-path` to check whether a user has the required roles to access the portlet. Here, the `struts-path` value is `ext/book_reports`. This tells the portal that all requests to the `ext/book_reports/*` path are considered to be a part of this portlet scope. Users who request paths that match `ext/book_reports/*` will only be granted access if they also have access to this portlet.

Moreover, the `use-default-template` element has a value set to `false`, allowing the developers to own and maintain the entire outputted content of the portlet. If you want the portlet to use the default template to decorate and wrap content, you can set the `use-default-template` element to the value `true`.

Similarly, the `restore-current-view` element (that you may use later) has the value `false` so that the portlet will reset the current view when toggling between maximized and normal states. If you want the portlet to restore to the current view when toggling between maximized and normal states, you can set the `restore-current-view` element to the value `true`.

Specifying the page flow and page layout

We have defined the `book_reports` portlet. Now we need to define the page flow and page layout, and identify the main differences between a JSP portlet and a Struts portlet. Instead of forwarding directly to a JSP, the Struts portlet uses `struts-config.xml` to define the page flow and uses `tiles-defs.xml` to define the page layout.

First of all, let's define the Struts action. Create the portlet action class named `ViewBookReportsAction`, which extends the class `PortletAction` under the `com.ext.portlet.bookreports.action` package in the `/ext/ext-impl/src` folder.

```
public class ViewBookReportsAction extends PortletAction {
    public ActionForward render(
        ActionMapping mapping, ActionForm form, PortletConfig
        portletConfig, RenderRequest renderRequest, RenderResponse
        renderResponse) throws Exception {
        if (renderRequest.getWindowState() .
            equals(WindowState.NORMAL)) {
            return mapping.findForward
                ("portlet.ext.book_reports.view");
        }
        else {
            List<String> reports = Collections.synchronizedList(new
                ArrayList<String>());
            PortletPreferences prefs = renderRequest.getPreferences();
            ...
        }
    }
}
```

```
        reports.add("1: Preferences book - " + prefs.getValue
                     ("Liferay Book", ""));
        reports.add("2: Window State - " + renderRequest.
                     getWindowState());
        reports.add("3: Portlet Mode - " + renderRequest.
                     getPortletMode().toString());
        reports.add("4: Portlet Session - " +
                     renderRequest.getPortletSession().getId());
        renderRequest.setAttribute("reports", reports);
        return mapping.findForward
            ("portlet.ext.book_reports.view_reports");
    }
}
}
```

The code above shows that the `ViewReportsAction` portlet extends the `PortletAction` class, and the portlet to render is specified too. At the same time, it will return dynamic content as the value of the `reports` attribute for the JSP file, for example preference, window states, portlet mode, portlet session, and so on.

Based on the above Struts action `ViewBookReportsAction`, let's define the page flow in `struts-config.xml` for the `book_reports` portlet:

1. Locate the `struts-config.xml` file in the `/ext/ext-web/docroot/WEB-INF` folder and open it.
2. Add the following lines immediately after `<action-mappings>` and save it:

```
<!-- Book reports -->
<action path="/ext/book_reports/view_reports"
       type="com.ext.portlet.bookreports.action.
             ViewBookReportsAction">
    <forward name="portlet.ext.book_reports.view"
            path="portlet.ext.book_reports.view" />
    <forward name="portlet.ext.book_reports.view_reports"
            path="portlet.ext.book_reports.view_reports" />
</action>
```

What's `/ext/book_reports/view_reports` in the code above? It is the value of the `name` `view-action` in the `init-param` element of `portlet-ext.xml`. What are `portlet.ext.book_reports.view` and `portlet.ext.book_reports.view_reports`? They are the forwards that are used to look up the tiles definition.

Thirdly, based on the page flow and JSP files, let's define the page layout in `tiles-defs.xml` for the `book_reports` portlet.

1. Locate the `tiles-defs.xml` file in the `/ext/ext-web/docroot/WEB-INF` folder and open it.

2. Add the following lines immediately after `<tiles-definitions>` and save them:

```

<!-- Book Reports -->
<definition name="portlet.ext.book_reports.view"
            extends="portlet">
<put name="portlet_content"
      value="/portlet/ext/book_reports/view.jsp" />
</definition>
<definition name="portlet.ext.book_reports.view_reports"
            extends="portlet">
<put name="portlet_content"
      value="/portlet/ext/book_reports/view_reports.jsp" />
</definition>

```

What are `portlet.ext.book_reports.view` and `portlet.ext.book_reports.view_reports` in the code above? They are the forwards. What is `portlet`? The `portlet` is the template that will be used (that is, `/common/themes/portlet.jsp`, refer to the XML file `/portal/portal-web/docroot/WEB-INF/tiles-defs.xml`). Why is the path starting from `/portlet`, and not from `/html`, even though the directory structure is `/docroot/html/portlet`? It's because the `/html` part already added the `/portal/portal-web/docroot/html/common/themes/portlet.jsp` file. In general, there are two pages: `view.jsp` and `view_reports.jsp`.

What are the main differences between a JSP portlet and a Struts portlet? The JSP portlet goes directly to a JSP file, while the Struts portlet has a page flow. The JSP portlet directly points to the JSP file from `portlet-ext.xml`; in Struts portlet, it is done through `tiles-defs.xml`. Where does the page flow and page layout get defined? The `struts-config.xml` file defines the page flow while the `tiles-defs.xml` file defines the page layout.

Creating JSP pages

As stated above, there are two pages which are specified in the page layout. We need to create these two pages: `view.jsp` and `view_reports.jsp`. First, let's create the JSP page `view.jsp` as follows:

1. Create the `/book_reports` folder under the `/ext/ext-web/docroot/html/portlet/ext` folder.
2. Create a file named `init.jsp` under the `/ext/ext-web/docroot/html/portlet/ext/book_reports` folder with the following content:

```

<%@ include file="/html/common/init.jsp" %>
<%@ page import="java.util.List" %>
<portlet:defineObjects />
<% PortletPreferences prefs = renderRequest.getPreferences(); %>

```

3. Create a file named `view.jsp` under the `/ext/ext-web/docroot/html/portlet/ext/book_reports` folder with the following content:

```
<%@ include file="/html/portlet/ext/book_reports/init.jsp" %>
<a href=<portlet:renderURL windowState="<%=
WindowState.MAXIMIZED.toString() %>" />>Preference: Liferay
Book - <%= prefs.getValue("Liferay Book", "") %></a>
```

As shown in the code above, `init.jsp` contains the line `<%@ include file="/html/common/init.jsp" %>`. This will give access to the Liferay tag libraries. Normally, you can add commonly-used variables and declarations, for example, the import `<%@ page import="java.util.List" %>` and the variable `PortletPreferences prefs`.

Similarly, the `view.jsp` file adds the `init.jsp` file, and further specifies a link with the portlet window state as `MAXIMIZED`, the text `Preference: Liferay Book -`, and the preference value for the name `Liferay Book`.

Then, let's create the JSP page `view_reports.jsp` as follows:

1. Locate the `/ext/ext-web/docroot/html/portlet/ext/book_reports` folder.
2. Create a file named `view_reports.jsp` with the following content:

```
<%@ include file="/html/portlet/ext/book_reports/init.jsp" %>
<% List reports = (List)request.getAttribute("reports"); %>
<div> <liferay-ui:message key="view-reports" />
<center>
    <a href="http://liferay.cignex.com" title="Liferay Portal
        Enterprise Intranets">Palm Tree Publications</a>
</center>
<center>
    <a href="http://liferay.cignex.com" title="Liferay Portal
        Enterprise Intranets"></a>
</center>
<% for (int i = 0; i < reports.size(); i++) {
    String reportName = (String)reports.get(i);
    %><%= reportName %><br><%
}>%>
</div>
```

As shown in the code above, the `view_reports.jsp` file include the `init.jsp` file first. Then it specifies the same links as JSP portlet view had specified. At the same time, it displays the dynamic contents by the list of `reports`.

Changing the title and category

As we know, the Struts portlet has the title Book Reports and it is under the Unknown category. Now let's change the title and the category. Suppose that we want to set the title to Reports for Books, the message as view-reports with the value View Reports for Books in `Language-ext.properties`, and add the portlet book_reports to the category Book in `liferay-display.xml`.

First, add the title, for example, Reports for Books in `Language-ext.properties` by following these steps:

1. Locate the `Language-ext.properties` file under the `/ext/ext-impl/src/content` folder and open it.
 2. Replace Book Reports with Reports for Books, and View Reports with View Reports for Books.
 3. Add the following lines at the end, and save it.
- ```
Category titles
category.book=Book
```

As shown in the code above, it maps the category title from the name `category.book` to the value `Book`.

Secondly, add the `book_reports` portlet to the category Book in `liferay-display.xml`, by using the following steps:

1. Locate the `liferay-display.xml` file in the `/ext/ext-web/docroot/WEB-INF` folder and open it.
  2. Add the following lines between `</category>` and `</display>` and save the file:
- ```
<category name="category.book">
<portlet id="book_reports" />
</category>
```

The code above adds the `book_reports` portlet to the category Book. From now on, you will be able to select your portlet from the Book category.

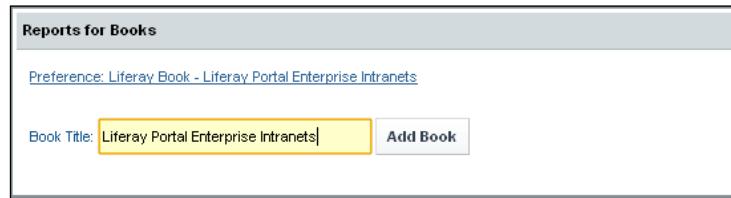
Finally, we can deploy the updates into Tomcat. After restarting Tomcat, you will see the Struts portlet `book_reports` with the title Reports for Books and contents—a message with a reference link. Note that the `book_reports` portlet with the title Reports for Books is under the Book category now. By default, you will see the normal view of the `book_reports` portlet. You will find a link with the text Preference: Liferay Book – Liferay Portal Enterprise Intranets. When you click on the link, it will show the message View Reports for Books, static content, and dynamic content.

Building an advanced Struts portlet

We have discussed the basic Struts portlet `book_reports`. It includes the basic features of a Struts portlet: defining the portlet in both `portlet-ext.xml` and `liferay-portlet-ext.xml`, changing the title and category in `Language-ext.properties` and `liferay-display.xml`, and creating the JSP pages with static and dynamic contents. In this section, we want to add more features to the Struts portlet: adding an action, using services, redirecting the pages, adding more actions, setting up permissions, and so on.

Adding an action

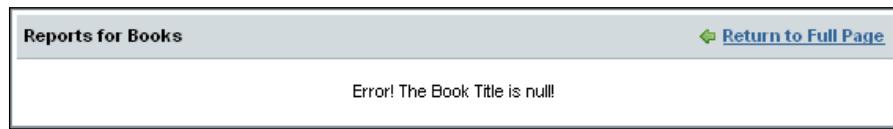
In this part, we will add an `Action` class to the Struts portlet, and further display a success page and an error page. As shown in the following screenshot, there are two items: an action—adding a new book, and a link as a view—viewing dynamic content (as mentioned earlier).



As shown in the following screenshot, when the user enters a book title **Liferay Portal Enterprise Intranets** and clicks on the **Add Book** button, the success page displays a success message: **Success! Book Title: Liferay Portal Enterprise Intranets**. Moreover, a link **Return to Full Page** is also provided to return the view page `view.jsp` as shown in the following screenshot.



As shown in the following screenshot, when the user enters an empty book title and clicks on the **Add Book** button, the error page depicts an error message: **Error! The Book Title is null!** Moreover, a link **Return to Full Page** is also provided to return to the view page.



Now we will learn how the `view.jsp`, `success.jsp`, and `error.jsp` views are implemented in the following sections.

Creating an action

First of all, let's create an action class named `AddBookAction` (this name is used for following the conventions; you can use a different name), extending `PortletAction` under the `com.ext.portlet.bookreports.action` package of `/ext/ext-impl/src` as follows:

```
public class AddBookAction extends PortletAction {
    public void processAction(ActionMapping mapping, ActionForm form,
        PortletConfig config, ActionRequest req, ActionResponse res)
        throws Exception {
        String title = req.getParameter("title");
        if (Validator.isNull(title)) {
            setForward(req, "portlet.ext.book_reports.error");
        }
        else {
            setForward(req, "portlet.ext.book_reports.success");
        }
    }
    public ActionForward render(
        ActionMapping mapping, ActionForm form, PortletConfig
        portletConfig, RenderRequest req, RenderResponse res)
        throws Exception {
        return mapping.findForward(getForward(req,
            "portlet.ext.book_reports.view"));
    }
}
```

The code above specifies two methods: `processAction` and `render`. The `processAction` method is called by the portlet container to allow the portlet to process an action request, for example, adding a book. The `render` method is called by the portlet container to allow the portlet to generate the content of the response based on its current state, for example, `portlet.ext.bookreports.view`.

Where does `title` come from? It comes from the `view.jsp` form. What is the `if / else` statement doing? It is detecting if the book title was submitted. Further, it sets the forward paths according to the detected state, for example `portlet.ext.bookreports.success` and `portlet.ext.bookreports.error`.

Defining the action

After creating the action, we can define the action in both `struts-config.xml` and `tiles-defs.xml`. As stated earlier, `struts-config.xml` defines the page flow; while `tiles-defs.xml` defines the page layout.

Let's add the Struts action path `/ext/book_reports/add_book` to the page flow. We simply add the following lines immediately after `<!-- Book reports -->` in `struts-config.xml`:

```
<action path="/ext/book_reports/add_book"
        type="com.ext.portlet.bookreports.action.AddBookAction">
    <forward name="portlet.ext.book_reports.view"
            path="portlet/ext/book_reports.view" />
    <forward name="portlet.ext.book_reports.error"
            path="portlet/ext/book_reports.error" />
    <forward name="portlet.ext.book_reports.success"
            path="portlet/ext/book_reports.success" />
</action>
```

The code above defines the Struts' action path `/ext/book_reports/add_book`. What is the `type`? It is Struts' defined way of passing control to the `AddBookAction` class. Let's have a deeper look at the forward nodes, for example, the forward name `portlet.ext.book_reports.error` and the forward path `portlet.ext.book_reports.success`. The forward name is the unique identifier for that forward node. The forward path is the link to the `tiles-def.xml` file. For more details, refer to <http://struts.apache.org/1.x/>.

Moving further, add the following lines immediately after `<!-- Book reports -->` in `tiles-def.xml`:

```
<definition name="portlet.ext.book_reports.error" extends="portlet">
    <put name="portlet_content"
        value="/portlet/ext/book_reports/error.jsp" />
</definition>
<definition name="portlet.ext.book_reports.success"
            extends="portlet">
    <put name="portlet_content"
        value="/portlet/ext/book_reports/success.jsp" />
</definition>
```

As shown in the code above, the definition names `portlet.ext.book_reports.error` and `portlet.ext.book_reports.success` are specified in `struts-config.xml`. In addition, there are two JSP files: `success.jsp` and `error.jsp`.

Adding a form in JSP page

The view page `view.jsp` in the `/ext/ext-web/docroot/html/portlet/ext/book_reports` folder needs to be updated. As stated earlier, we need to add a link to view all books and create a form to add a new book by the title. Let's add a form with the following lines at the end of the `view.jsp` view page:

```
<br/><br/>
<form action=<portlet:actionURL windowState="<%
    WindowState.MAXIMIZED.toString() %>">
    <portlet:param name="struts_action"
        value="/ext/book_reports/add_book" />
    </portlet:actionURL>
    method="post"
    name=<portlet:namespace />fm">
    <div style="color: #12558E">Book Title:
        <input name=<portlet:namespace />title" size="35" type="text"
            value="">
        <input onClick="submitForm(document.<portlet:namespace />fm);"
            style="margin-top: 5px;" type="button" value="Add Book">
        <br/>
    </div>
<br/>
</form>
```

As shown in the code above, the `portlet:param` tag specifies the action name as `struts_action` and the link value as `/ext/book_reports/add_book`. Thus, `struts_action` connects us to the `struts-config.xml` file. In addition, the Add Book button is also specified as the `button` type, and the input with the `text` type.

Creating success and error pages

Later, we need to create two JSP files: `success.jsp` and `error.jsp`. As stated earlier, these two JSP files are already used in `tiles-def.xml`. Here we are going to address the process of creating these two JSP files.

First, let's create the JSP file `success.jsp` using the following steps:

1. Locate the `/ext/ext-web/docroot/html/portlet/ext/book_reports` folder.
2. Create a `success.jsp` file with the following content and save it:

```
<%@ include file="/html/portlet/ext/book_reports/init.jsp" %>
<% String title = request.getParameter("title"); %>
<div style="text-align: center">
    Success! Book Title: <i><%= title %></i>
</div>
```

The code above imports `init.jsp`. It also displays the information value:
Success! Book Title: Liferay Portal Enterprise Intranets!.

Now, let's create the JSP file `error.jsp` as follows:

1. Locate the `/ext/ext-web/docroot/html/portlet/ext/book_report` folder.
2. Create a `error.jsp` file with the following content and save it:

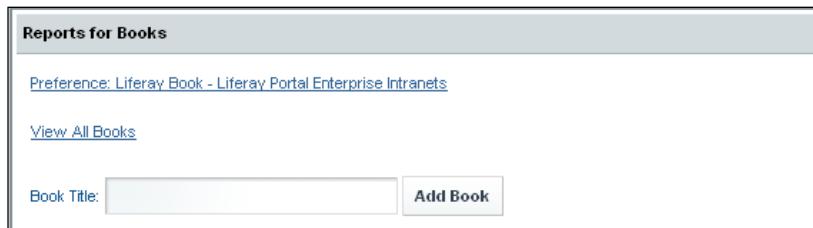
```
<%@ include file="/html/portlet/ext/book_reports/init.jsp" %>
<div style="text-align: center">Error! The Book Title is null!
</div>
```

Similarly, the code above imports `init.jsp`. It also displays the error message as
Error! The Book Title is null!.

Interacting with the database

We have added an action in the `book_report` portlet. But there is no database interaction yet; there is only static data. Now we're going to add database interaction.

First, we expect to add the message **View All Books** with a link, which will bring us to browse all the books as shown in the following screenshot:



Then, as shown in the following screenshot, when the user clicks on the **View All Books** link, the view-books page will display a list of books with **Book ID**, **Group ID**, **Company ID**, **User ID**, **User Name**, and **Title**, and so on. Likewise, the **Return to Full Page** link is also provided to return the view page.

Book ID	Group ID	Company ID	User ID	User Name	Title	Actions
2	10129	10109	10134	Test Test	Liferay Portal Enterprise Intranets	Edit Delete

Let's implement the above features as follows:

- Creating a database structure,
- Creating methods to add and retrieve records,
- Updating existing files, and
- Retrieving records from the database.

Creating a database structure

First, we need to create a database structure for book reports in `service.xml`, which contains a proprietary definition of entities, finder methods required for each entity, and exceptions that the module may throw, and so on. Let's do it as follows:

1. Locate the `com.ext.portlet.bookreports` package in the `/ext/ext-impl/src` folder.
2. Create the `service.xml` file with the following content and save it:

```
<?xml version="1.0"?>
<!DOCTYPE service-builder PUBLIC "-//Liferay//DTD Service Builder
5.2.0//EN" "http://www.liferay.com/dtd/liferay-service-
builder_5_2_0.dtd">
<service-builder package-path="com.ext.portlet.bookreports">
<namespace>BookReports</namespace>
<entity name="BookReportsEntry" uuid="true" local-service="true"
remote-service="true"
persistence-class="com.ext.portlet.bookreports.service.
persistence.BookReportsEntryPersistenceImpl">
<!-- PK fields; entryId is mapped into BookId;
BookReportsEntry is mapped into Book. -->
<column name="entryId" type="Long" primary="true" />
<!-- Group instance -->
<column name="groupId" type="Long" />
<!-- Audit fields -->
<column name="companyId" type="Long" />
<column name="userId" type="Long" />
<column name="userName" type="String" />
<column name="createDate" type="Date" />
<column name="modifiedDate" type="Date" />
<!-- Other fields -->
<column name="name" type="String" />
<column name="title" type="String" />
<!-- Order -->
<order by="asc">
<order-column name="title" case-sensitive="false" />
</order>
</entity>
<exceptions><exception>EntryName</exception></exceptions>
</service-builder>
```

As shown in the preceding code, package-path `com.ext.portlet.bookreports` is the path that the class will generate to the `/ext/ext-impl/src/com/ext/portlet/bookreports` folder. The entity has the `BookReports` namespace with the name `BookReportsEntry`. It will use UUID, local service, and remote service. The entity has the primary key field `entryId`, and other fields: `groupId`, `companyId`, `userId`, `userName`, `createDate`, `modifiedDate`, `name`, `title`, and so on. Meanwhile, the query results will be arranged in an ascending order by the `title` field. Accordingly, an exception with the name `EntryName` is also specified—it will generate a class `EntryNameException` at the `com.ext.portlet.bookreports` package.

We now need to build services with ServiceBuilder. After having `service.xml` ready, we can build services with ServiceBuilder as follows:

1. Locate the XML file `/ext/ext-impl/build-parent.xml` and open it.
2. Add the following lines between `</target>` and `<target name="build-service-portlet-reports">`, and save it:

```
<target name="build-service-portlet-bookreports">
<antcall target="build-service">
<param name="service.file" value="src/com/ext/portlet/bookreports/
service.xml" />
</antcall>
</target>
```
3. Drag the `build.xml` file in the `/ext/ext-impl` folder to the **Ant** view.
4. Expand `ext-impl` in the **Ant** view.
5. Double-click on the target `build-service-portlet-bookreports`.
6. Refresh the `/ext` project.

ServiceBuilder will first create models, services, and service persistence at `/ext/ext-service/src`. The following are packages for the `book_reports` portlet:

- `com.ext.portlet.bookreports`
- `com.ext.portlet.bookreports.model`
- `com.ext.portlet.bookreports.service`
- `com.ext.portlet.bookreports.service.persistence`

Similarly, ServiceBuilder will create implementations of models, services, and service persistence at `/ext/ext-impl/src`. The following are packages of the implementations of the services for the `book_reports` portlet:

- `com.ext.portlet.reports.model.impl`
- `com.ext.portlet.reports.service.base`
- `com.ext.portlet.reports.service.http`

- com.ext.portlet.reports.service.impl
- com.ext.portlet.reports.service.persistence

Meanwhile, under the /ext/ext-impl/src/META-INF folder, ServiceBuilder generated Hibernate mappings in ext-hbm.xml, model-hints in ext-model-hints.xml, and Spring beans in ext-spring.xml.

Finally, we need to create a new table in the database as follows:

1. Log in to the lportal database as mysql -u lportal -plportal lportal.
2. Import the following code:

```
drop table if exists BookReportsEntry;
create table BookReportsEntry (
    uuid_ varchar(75) null,
    entryId bigint not null primary key,
    groupId bigint,
    companyId bigint, userId bigint null,
    createDate datetime null, modifiedDate datetime null,
    name varchar(511) null, title varchar(511) null
);
```

The code above specified a database table called BookReportEntry with the columns entryId, groupId, companyId, name, title, and so on.

Creating methods to add and retrieve records

When the basic services and models are ready, we can add the database insert method to add a book, and retrieve method to get all books. Let's create an action named BookLocalServiceUtil under the com.ext.portlet.bookreports.action package of the /ext/ext-impl/src folder as follows:

```
public class BookLocalServiceUtil {
    public static ReportsEntry addBook(ActionRequest req)
        throws PortalException, SystemException {
        ThemeDisplay themeDisplay = (ThemeDisplay)req.getAttribute(
            (WebKeys.THEME_DISPLAY));
        long userId = themeDisplay.getUserId();
        String title = req.getParameter("title");
        User user = UserLocalServiceUtil.getUser(userId);
        Date now = new Date();
        long bookId = CounterLocalServiceUtil.increment(
            (BookReportsEntry.class.getName()));
        BookReportsEntry book = BookReportsEntryLocalServiceUtil.
            createBookReportsEntry(bookId);
        book.setTitle(title);
        book.setGroupId(themeDisplay.getScopeGroupId());
        book.setCompanyId(themeDisplay.getCompanyId());
        book.setUserId(userId);
```

```
        book.setUserName(user.getFullName()); book.setCreateDate(now);
        book.setModifiedDate(now); book.setName(title);
        BookReportsEntryLocalServiceUtil.updateBookReportsEntry(book);
        return book;
    }
    public static List<BookReportsEntry> getAll()
        throws PortalException, SystemException {
        int end = BookReportsEntryLocalServiceUtil.
            getBookReportsEntriesCount();
        return BookReportsEntryLocalServiceUtil.
            getBookReportsEntries(0, end);
    }
}
```

As shown in the code above, there are two methods: `addBook` and `getAll`. The `addBook` method inserts an entry to the database, while the `getAll` method retrieves all entries. We used the names `addBook` and `getAll` for demo purpose only. You can have different names—they should just have the same functions. Moreover, the service (for example, `BookReportsEntryLocalServiceUtil`) includes **CRUD (Create, Read, Update, and Delete)** operations. We do not want to affect these operations generated by ServiceBuilder—just leave them as they are. We added and customized the `BookLocalServiceUtil` wrapper on top of the service, for example `BookReportsEntryLocalServiceUtil`. Thus, when upgrading ServiceBuilder, we only need to upgrade this wrapper; no other changes are required.

Updating existing files

For the `AddBookAction` action, we need to add the following line before the `setForward(req, "portlet.ext.book_reports.success")`; line inside the `processAction` method:

```
BookLocalServiceUtil.addBook(req);
```

Add the following lines as forward path before the line `return mapping.findForward(getForward(req, "portlet.ext.book_reports.view"))`; inside the `render` method as follows:

```
String title = req.getParameter("title");
if(title == null)setForward(req, "portlet.ext.book_reports.view_
books");
```

For the `struts-config.xml` file, add the following lines immediately after `<!-- Book reports -->`:

```
<action path="/ext/book_reports/view_books"
       type="com.ext.portlet.bookreports.action.AddBookAction">
    <forward name="portlet.ext.book_reports.view"
            path="portlet.ext.book_reports.view" />
```

```
<forward name="portlet.ext.book_reports.view_books"
         path="portlet.ext.book_reports.view_books" />
</action>
```

As shown in the code above, the Struts action has the action path `/ext/book_reports/view_books` with two forward paths: `portlet.ext.book_reports.view` and `portlet.ext.book_reports.view_books`.

Similarly, in the `tiles-defs.xml` file, add the following lines immediately after `<!-- Book reports -->`:

```
<definition name="portlet.ext.book_reports.view_books"
            extends="portlet">
    <put name="portlet_content"
        value="/portlet/ext/book_reports/view_books.jsp" />
</definition>
```

The code above shows that the forward path `portlet.ext.book_reports.view_books` is associated with the JSP file `/portlet/ext/book_reports/view_books.jsp`.

Further, add the following lines immediately after the line `<%@ include file="/html/common/init.jsp" %>` in `init.jsp`:

```
<%@ page import="com.ext.portlet.bookreports.
                    model.BookReportsEntry" %>
<%@ page import="com.ext.portlet.bookreports.
                    action.BookLocalServiceUtil" %>
```

As shown in code above, we add imports `BookReportsEntry` and `BookLocalServiceUtil` for all JSP files.

And finally, we need to add one render URL immediately before the `

` line in `view.jsp` of the `/ext/ext-web/docroot/html/portlet/ext/book_reports` folder.

```
<br/><br/>
<a href=<portlet:renderURL windowState="
                <%=WindowState.MAXIMIZED.toString() %>">
    <portlet:param name="struts_action"
                  value="/ext/book_reports/view_books" />
</portlet:renderURL>">
    View All Books
</a>
```

As shown in the code above, the `<portlet:param>` tag specifies the action name as `struts_action` and the link value as `/ext/book_reports/view_books`. Thus, `struts_action` connects us to the `struts-config.xml` file.

Retrieving records from the database

Finally, we can retrieve records from the database in `view_books.jsp`. Let's create a JSP file `view_books.jsp` by using the following steps:

1. Locate the `/ext/ext-web/docroot/html/portlet/ext/book_reports` folder.
2. Create the `view_books.jsp` file with the following content and save it:

```
<%@ include file="/html/portlet/ext/book_reports/init.jsp" %>
<% List books = (List) BookLocalServiceUtil.getAll();
BookReportsEntry book = null; %>
<table style="border: 1px solid #CCC; width: 100%; padding: 5px;">
<tr>
<td style="color: #12558E;">Book ID</td>
<td style="color: #12558E;">Group ID</td>
<td style="color: #12558E;">Company ID</td>
<td style="color: #12558E;">User ID</td>
<td style="color: #12558E;">User Name</td>
<td style="color: #12558E;">Title</td>
</tr>
<c:if test="<%= books != null %>">
<% for (int i=0; i < books.size(); i++) {
book = (BookReportsEntry) books.get(i); %>
<tr>
<td><%= book.getEntryId() %></td>
<td><%= book.getGroupId() %></td>
<td><%= book.getCompanyId() %></td>
<td><%= book.getUserId() %></td>
<td><%= book.getUserName() %></td>
<td><%= book.getTitle() %></td>
</tr>
<% }%> </c:if>
</table>
```

The code above displays all book titles first. Then, it loops through the book titles and further displays all details such as Book ID, Group ID, User Name, Title, and so on.

Redirecting

We have used forms to submit the book message `Title` in order to create a new record in database. On the one hand, there is only one submit per request. Thus, we have to prevent multiple submits. A redirect mechanism would be the right way to go.

On the other hand, we need to redirect from one view to another view. When entering the value `title` in the correct way, the `book_reports` portlet will create a new record—a book with value `title` and redirect from the view page to the success page.

Consequently, as shown in the following screenshot, the `view-books` page displays a list of books with **Book ID**, **Group ID**, **Company ID**, **User ID**, **User Name**, and **Title**, and so on. A link **Back to View Page** is also provided in order to return to the view page `view.jsp`.

Reports for Books					
Book ID	Group ID	Company ID	User ID	User Name	Title
2	10129	10109	10134	Test Test	Liferay Portal Enterprise Intranets
Back to View Page					

In this part, we are going to discuss the redirect mechanism in the `book_reports` portlet.

Updating the action

First of all, update the `processAction` method of the `AddBookAction` action as follows. Note that the highlighted items are newly added or updated.

```
String title = req.getParameter("title");
String redirect = req.getParameter("redirect")+"&title="+title;
String error = req.getParameter("error");
if (Validator.isNull(title)) {
    res.sendRedirect(error);
}
else {
    req.setAttribute("title", title);
    BookLocalServiceUtil.addBook(req);
    res.sendRedirect(redirect);
}
```

As shown in the code above, the `AddBookAction` action replaces the forward path with a `sendRedirect` path. `sendRedirect` is a portlet method that grabs a request and then redirects to the redirect value. Make sure that the redirect value is a valid URL. By the way, `redirect` and `error` are the names of hidden fields in `view.jsp`. Meanwhile, the parameter `&title=value` is used to provide a value for the `title` key.

Updating action paths

Now let's add page flow paths `/ext/book_reports/success`, `/ext/book_reports/error`, and `/ext/book_reports/view` so that we have landing paths. To do so, just add the following lines immediately after the line `<!-- Book reports -->` in `struts-config.xml`:

```
<action path="/ext/book_reports/success"
       forward="portlet.ext.book_reports.success" />
<action path="/ext/book_reports/error"
       forward="portlet.ext.book_reports.error" />
<action path="/ext/book_reports/view"
       forward="portlet.ext.book_reports.view" />
```

Updating existing JSP files

Then let's update the existing JSP files in order to support redirects. Add two hidden form fields for `redirect` and `error` before the line `<div style="color: #12558E">Book Title: in view.jsp:`

```
<input name="
               <portlet:param name="struts_action"
                             value="/ext/book_reports/success" />
               </portlet:renderURL>">
<input name="
               <portlet:param name="struts_action"
                             value="/ext/book_reports/error" />
               </portlet:renderURL>">
```

As shown in the code above, the hidden fields `redirect` and `error` have Struts action paths `/ext/book_reports/success` and `/ext/book_reports/error`, respectively.

Add the redirect links at the end of the `success.jsp`, `error.jsp`, and `view_books.jsp` files as follows:

```
<br/>
<a href="Back to
View Page</a>
<br/>
```

As shown in the code above, the render URL with the `Back to View Page` value has Struts action path `/ext/book_reports/view`.

Adding more actions

We have discussed how to add a book and display all books as a table. Now let's go further and add more actions – editing a book and deleting a book. As shown in the following screenshot, each book will be displayed with an **Actions** icon (**Edit** and **Delete** icons):

Book ID	Group ID	Company ID	User ID	User Name	Title	
2	10129	10109	10134	Test Test	Liferay Portal Enterprise Intranets	Edit Delete

[Back to View Page](#)

When you click on the **Actions | Edit** icon, it will show the view page as shown in the next screenshot. Suppose you have clicked on the **Actions | Edit** icon next to the book with the title **Liferay Portal Enterprise Intranets**, the view page `view.jsp` will show the **Edit Book** and **Cancel** buttons with the value **Liferay Portal Enterprise Intranets** in the text input box by default. Clicking on the **Edit Book** button will update the current book and you will be redirected to the success page. Meanwhile, clicking on the **Cancel** button will redirect you to the view-books page.

Preference: Liferay Book - Liferay Portal Enterprise Intranets

[View All Books](#)

Book Title:

[Edit Book](#) [Cancel](#)

Similarly, when you click on the **Actions | Delete** icon, a window with two buttons **OK** and **Cancel**, and a message **Are you sure you want to delete this?** will pop up. If you click on the **OK** button, the selected book will be deleted and the current page will be refreshed. Likewise, if you click on the **Cancel** button, the current page will be kept as it is.

In this part, let's implement the two actions **Edit** and **Delete**.

Creating methods to edit and delete records

First of all, we need to add business logic for update and delete methods. Let's add business logic for update, delete, and getBook methods before the last } in BookLocalServiceUtil as follows:

```
public static void deleteBook(ActionRequest req)
throws PortalException, SystemException {
    long entryId = ParamUtil.getLong(req, "bookId");
    BookReportsEntryLocalServiceUtil.deleteBookReportsEntry(entryId);
}
public static BookReportsEntry updateBook(ActionRequest req) throws
PortalException, SystemException {
    String title = ParamUtil.getString(req, "title");
    BookReportsEntry reportsEntry = BookReportsEntryLocalServiceUtil.
        getBookReportsEntry(entryId);
    reportsEntry.setTitle(title);
    reportsEntry.setModifiedDate(new Date());
    BookReportsEntryLocalServiceUtil.updateBookReportsEntry
        (reportsEntry);
    return reportsEntry;
}
public static void getBook(RenderRequest req) throws Exception {
    HttpServletRequest httpReq = PortalUtil.getHttpServletRequest(req);
    getBook(httpReq);
}
public static void getBook(HttpServletRequest req) throws Exception {
    long entryId = ParamUtil.getLong(req, "bookId");
    BookReportsEntry book = null;
    if (entryId > 0) {
        book = BookReportsEntryLocalServiceUtil.
            getBookReportsEntry(entryId);
    }
    req.setAttribute(BOOK, book);
}
```

As shown in the code above, `deleteBook` will remove the book from the databases, while `updateBook` will get the book from the database first and then update the book. In the same way, the `getBook` method will get the book from the database.

Updating the action

Next, we need to add update and delete methods to the AddBookAction action. Let's update the processAction method of the AddBookAction action as follows. Note that the highlighted items are newly added or updated.

```
String error = req.getParameter("error");
String cmd = ParamUtil.getString(req, Constants.CMD);
if (cmd.equals(Constants.EDIT)) {
    BookLocalServiceUtil.updateBook(req);
    res.sendRedirect(redirect);
}
else if (cmd.equals(Constants.DELETE)) {
    BookLocalServiceUtil.deleteBook(req);
}
else if (cmd.equals(Constants.ADD)) {
    if (Validator.isNull(title)) {
        res.sendRedirect(error);
    }
    else {
        BookLocalServiceUtil.addBook(req);
        res.sendRedirect(redirect);
    }
}
```

As shown in the code above, the cmd command is used to determine whether to ADD, EDIT, or DELETE. Under EDIT, it updates the book and then redirects to the success.jsp file, which displays a success message. While under DELETE, it removes the book from the database. In the same way, under ADD, it adds a new book as mentioned earlier.

Creating actions menu JSP file

We need to add the logic for the Actions menu button. Let's create a entry_action.jspf file as follows:

1. Locate the /ext/ext-web/docroot/html/portlet/ext/book_reports folder.
2. Create an entry_action.jspf file with the following content and save it:

```
<liferay-ui:icon-menu>
<portlet:renderURL windowState="<%=
   WindowState.MAXIMIZED.toString() %>" var="editEntryURL">
<portlet:param name="struts_action"
    value="/ext/book_reports/view" />
<portlet:param name="<%= Constants.CMD %>"
    value="<%= Constants.EDIT %>" />
```

```
<portlet:param name="bookId"
               value="<%= String.valueOf(
                       (book.getEntryId()) %)" />
<portlet:param name="redirect" value="<%= currentURL %>" />
</portlet:renderURL>
<liferay-ui:icon image="edit" message="edit"
                 url="<%= editEntryURL %>" />
<portlet:actionURL var="deleteEntryURL">
    <portlet:param name="struts_action"
                  value="/ext/book_reports/view_books" />
    <portlet:param name="<%= Constants.CMD %>"
                  value="<%= Constants.DELETE %>" />
    <portlet:param name="bookId"
                  value="<%= String.valueOf(
                          (book.getEntryId()) %)" />
    <portlet:param name="redirect"
                  value="<%= currentURL %>" />
</portlet:actionURL>
<liferay-ui:icon-delete url="<%= deleteEntryURL %>" />
</liferay-ui:icon-menu>
```

In brief, `portlet:actionURL` creates ActionURL, which, on use, fires ActionRequest and processAction gets invoked for the portlet, while `portlet:renderURL` creates a RenderURL, which triggers RenderRequest and the render method gets executed.

Updating existing JSP files

We have created the actions menu JSP file: `entry_action.jspf`. Now we use this code in order to edit or delete books in `view_books.jsp`. First, we need to update `init.jsp` using the following code. Note that highlighted line is newly added.

```
PortletPreferences prefs = renderRequest.getPreferences();
String currentURL = PortalUtil.getCurrentURL(request);
%>
```

Then, let's update the `view_books.jsp` file to list all books and add the Actions menu button. To do so, just add the line `<td style="color: #12558E;">Actions</td>` between `<td style="color: #12558E;">Title</td>` and `</tr>`. Moreover, and add the line `<td><%@ include file="entry_action.jspf" %></td>` between `<td><%= book.getTitle() %></td>` and `</tr>`.

The code above adds variables for the window state and current URL. It also adds an Actions column with the Action menu buttons Edit and Delete, by including the `entry_action.jspf` file.

Last but not least, we need to update `view.jsp` in order to provide the ability to edit the book record. It covers a previous function (adding a book) and adds a new function (editing a book). First, let's add a command variable that tells the action whether to add or edit the book. To do so, add the following lines immediately after the line `<%@ include file="/html/portlet/ext/book_reports/init.jsp" %>` in `view.jsp`:

```
<%
    BookLocalServiceUtil.getBook(request);
    String redirect = ParamUtil.getString(request, "redirect");
    BookReportsEntry book = (BookReportsEntry)
        request.getAttribute(BookLocalServiceUtil.BOOK);
    long bookId = (book==null?0:book.getEntryId());
    String cmd = ParamUtil.getString(request, Constants.CMD);
    String tabNames = "Add Book";
    if (bookId > 0) {tabNames = "Edit Book";}
%>
```

Now let's update the form by consuming the variables above—for example `bookId`, `tabNames`, and so on—in `view.jsp` using the following code. Note that the highlighted items are newly added or updated.

```
<form action="
    <portlet:param name="struts_action"
        value="/ext/reports/add_book" />
</portlet:actionURL>
method="post"
name="
    <portlet:param name="struts_action"
        value="/ext/book_reports/success" />
</portlet:renderURL>">
<input name="
    <portlet:param name="struts_action"
        value="/ext/book_reports/error" />
```

```
</portlet:renderURL>">
<div style="color: #12558E">Book Title:
    <input name="">
    <input onClick="submitForm(document.<portlet:namespace />fm);"
        style="margin-top: 5px;" type="button" value="<%= tabNames %>">
    <% if(bookId > 0) { %>
        <input type="button"
            value="
```

As shown in the code above, if the book is null, then it sets Constants.CMD to add. And if the book is not null, it sets Constants.CMD to edit. It also adds hidden inputs Contants.CMD and bookId. Moreover, it adds the Cancel button for editing the book with a redirect.

Setting up permissions

We have discussed how to add a book, how to display all books, how to edit a book, and how to delete a book. As an administrator of the enterprise "Palm Tree Publications", it is great to manage books with full functions: add, display, edit, and delete. Unfortunately, these functions are also open to anyone, including the Guest user.

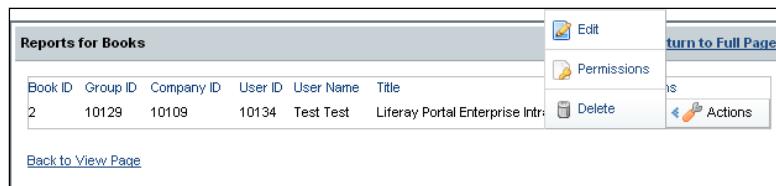
Now we need to set up permissions. As an administrator of the enterprise "Palm Tree Publications", you can see the **Add Book** button and, moreover, you can add a new book and set up the permissions on it for a **Guest** user and **Community** users. As shown in the following screenshot, you can set up permissions – such as **Delete**, **Permissions**, **Update**, and **View** – on the newly-created book.

- **Delete:** Delete a book from the database
- **Update:** Edit a book from the database
- **Permissions:** Give permissions to others so that they can set up permissions
- **View:** View a book

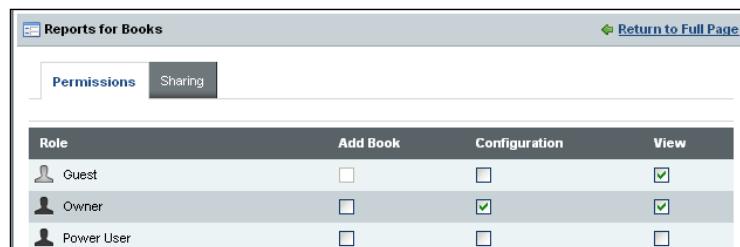
Other users, for example, Guest who does not have permissions, should not see the **Add Book** button and the **Permissions** option. By default, the guest user has no permissions to add a book and to re-assign the permissions.



As shown in the following screenshot, we need to hide the **Edit** and **Delete** options from the **Actions** menu button if the user—for example, David Berger—does not have the required permissions, and to add a new **Permissions** option. For example, the user David Berger has full permissions on the book **Liferay Portal Enterprise Intranets**, and only has View permission on other books. These permissions are called **Entry permissions** because they work on each entity, that is, the book entity.



Further, as an administrator of the enterprise "Palm Tree Publications", you can configure the `book_reports` portlet by clicking on the **Configure** icon and setting up permissions for other roles. For instance, we need to give the user David Berger the **Add Book**, **Configuration**, and **View** permissions on the `book_reports` portlet. These permissions are called as portlet permissions because they work on the portlet, that is, `book_reports`.



Let's implement the above features on the `book_reports` portlet—set up permissions in the backend and the frontend.

Setting up permissions in the backend

When creating a portlet with permissions, we need to specifically define what functionality should be protected by permissions. There are two sets of permissions: portlet permissions (the portlet resources and permissions for the entire portlet such as viewing the portlet) and entry permissions (the model resources and permissions for each of the entries such as updating or deleting an entry). In order to set up permissions in the backend, we need to consider defining permissions for the portlet and model resources in `book.xml`, mapping to the portlet permissions in `default-ext.xml`, hooking permissions in `portal-ext.properties`, and adding permission entries in `Language-ext.properties`.

First, define permissions for both the portlet resources and model resources. Let's create an XML file `book.xml` using the following steps:

1. Create a folder named `resource-actions` in the `/ext/ext-impl/src` folder.
2. Create a `book.xml` file in the `/ext/ext-impl/src/resource-actions` folder with the following content:

```
<?xml version="1.0"?>
<resource-action-mapping>
    <portlet-resource>
        <portlet-name>book_reports</portlet-name>
        <supports>
            <action-key>ADD_BOOK</action-key>
            <action-key>CONFIGURATION</action-key>
            <action-key>VIEW</action-key>
        </supports>
        <community-defaults>
            <action-key>VIEW</action-key>
        </community-defaults>
        <guest-defaults>
            <action-key>VIEW</action-key>
        </guest-defaults>
        <guest-unsupported>
            <action-key>ADD_BOOK</action-key>
        </guest-unsupported>
    </portlet-resource>
    <model-resource>
        <model-name>
            com.ext.portlet.bookreports.model.BookReportsEntry
        </model-name>
        <portlet-ref>
            <portlet-name>book_reports</portlet-name>
        </portlet-ref>
        <supports>
            <action-key>DELETE</action-key>
            <action-key>PERMISSIONS</action-key>
            <action-key>UPDATE</action-key>
        </supports>
    </model-resource>
</resource-action-mapping>
```

```

        <action-key>VIEW</action-key>
    </supports>
    <community-defaults>
        <action-key>VIEW</action-key>
    </community-defaults>
    <guest-defaults><action-key>VIEW</action-key></guest-defaults>
    <guest-unsupported>
        <action-key>UPDATE</action-key>
        <action-key>DELETE</action-key>
        <action-key>PERMISSIONS</action-key>
    </guest-unsupported>
    </model-resource>
</resource-action-mapping>
```

As shown in the code above, resources can represent any object in Liferay. In this case, the resource is the `book_reports` portlet. The portlet name specifies the `book_reports` portlet, where permissions are defined. Similarly, it specifies the tag, `<supports>`, which is supported by this portlet. The action key specifies the actions that can be performed on this portlet, such as `ADD_BOOK`, while the default action keys specify the default permissions given to all portlets in Liferay such as `CONFIGURATION` and `VIEW`. At the same time, community defaults specify the default permissions for the community such as `VIEW`; the guest defaults specify the default permissions for guest users such as `VIEW`; and the guest unsupported specify permissions that cannot be assigned to guest users such as `ADD_BOOK`.

The model name specifies the model that the permission is defined for, such as `com.ext.portlet.reports.model.ReportsEntry`. The portlet reference specifies the portlet, for example `book_reports`, which contains the model object. Meanwhile, the supported permissions for each book are `DELETE`, `PERMISSIONS`, `UPDATE`, and `VIEW`. If a user has the `PERMISSIONS` permission, then the user will be able to assign permissions for other users. At the same time, community defaults specifies the default permissions for the community such as `VIEW`; the guest defaults specifies the default permissions for guest users such as `VIEW`; and the guest unsupported specifies permissions that cannot be assigned to guest users such as `DELETE`, `PERMISSIONS`, and `UPDATE`.

Then, we need to map to the portlet permissions in `default-ext.xml` using the following steps:

1. Create a `default-ext.xml` file in the `/ext/ext-impl/src/resource-actions` folder and open it.
2. Add the following lines in the `default-ext.xml` file and save it:

```

<?xml version="1.0"?>
<resource-action-mapping>
    <resource file="resource-actions/book.xml" />
</resource-action-mapping>
```

As shown in the preceding code, adding the above lines adds mapping to the specific portlet permissions.

Thirdly, let the portal know where to look for permissions written in the Ext environment. Let's hook permissions in `portal-ext.properties` as follows:

1. Locate the `portal-ext.properties` file in the `/ext/ext-impl/src` folder and open it.
2. Add the following lines at the end of the `portal-ext.properties` file and save it:

```
## resource action configurations
resource.actions.configs=resource-actions/default.xml,resource-
actions/default-ext.xml
```

As shown in the code above, it inputs a comma-delimited resource action configuration, for example `resource-actions/default-ext.xml`, which will be read from the class path. It includes the default resource action, for example `resource-actions/default.xml`.

Finally, we need to add permission entries in `Language-ext.properties` using the following steps:

1. Locate the `Language-ext.properties` file in the `/ext/ext-impl/src/content` folder and open it.
2. Add the following lines at the end of the `Language-ext.properties` file and save it:

```
## Model resources
model.resource.com.ext.portlet.bookreports.model.
BookReportsEntry=Book
## Action names
action.ADD_BOOK=Add Book
```

The code above maps the action key from `action.ADD_BOOK` to Add Book and, moreover, it maps the model resource from `model.resource.com.ext.portlet.bookreports.model.BookReportsEntry` to Book.

Setting up permissions in frontend

In order to set up permissions in the frontend, we need to consider updating the action class to pass in the necessary information, add a permission checker in `view.jsp`, and add resource action icons for each book entry in `entity_action.jspf`.

Firstly, we need to add a new method `addEntryResources` and update the `addBook` method of the `BookLocalServiceUtil` class to handle information passed in from the JSP file. To do so, add the following lines at the beginning of the `addBook` method:

```

String[] communityPermissions = req.getParameterValues
("communityPermissions");
String[] guestPermissions = req.getParameterValues
("guestPermissions");

```

Add the following line before the line `return book;` inside the method `addBook:`

```
addEntryResources(book, communityPermissions, guestPermissions);
```

Now, add the following lines before the line `public static List<BookReportsEntry> getAll():`

```

public static void addEntryResources(ReportsEntry book, String[]
communityPermissions, String[] guestPermissions)
throws PortalException, SystemException {
    ResourceLocalServiceUtil.addModelResources(
        book.getCompanyId(), book.getGroupId(), book.getUserId(),
        ReportsEntry.class.getName(), book.getEntryId(),
        communityPermissions, guestPermissions);
}

```

Then, we need to add a permission checker in `view.jsp` as follows. Note that the highlighted items are newly added or updated:

```

<c:if test="<%=" permissionChecker.hasPermission
            (themeDisplay.getPortletGroupId(),
             portletDisplay.getRootPortletId(),
             portletDisplay.getResourcePK(),
             BookLocalServiceUtil.ADD_BOOK) %>" >
<input onClick="submitForm(document.<portlet:namespace />fm);"
       style="margin-top: 5px;" type="button"
       value="<%=" tabNames %>"/>
<c:if test="<%=" bookId > 0 %>" >
    <input type="button" value="

```

As shown in the code above, if the user does not have the `ADD_BOOK` permission, the Add Book button and the permission settings do not get displayed.

Finally, we need to add the permissions resource action icons for each book entry to add permission checker in entity_action.jspf. For displaying more details, add the following line before the line <portlet:renderURL windowState="<%=\nWindowState.MAXIMIZED.toString() %>" var="editEntryURL">:

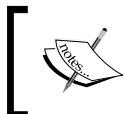
```
<c:if test="<%=\n    permissionChecker.hasPermission(themeDisplay.getPortletGroupId(),\n    BookReportsEntry.class.getName(), book.getEntryId(),\n    ActionKeys.UPDATE) %>">
```

Add the line </c:if> after the line <liferay-ui:icon image="edit" message="edit" url="<%=\n editEntryURL %>" />, and add the following lines immediately after the line </c:if>:

```
<c:if test="<%=\n    permissionChecker.hasPermission\n        (themeDisplay.getPortletGroupId(),\n            BookReportsEntry.class.getName(),\n            book.getEntryId(), ActionKeys.PERMISSIONS) %>">\n<liferay-security:permissionsURL\n    modelResource="<%=\n        BookReportsEntry.class.getName() %>"\n    modelResourceDescription="<%=\n        book.getTitle() %>"\n    resourcePrimKey="<%=\n        String.valueOf(book.getEntryId()) %>"\n    var="permissionsEntryURL" /\>\n<liferay-ui:icon image="permissions"\n    url="<%=\n        permissionsEntryURL %>" /\>\n</c:if>\n<c:if test="<%=\n    permissionChecker.hasPermission\n        (themeDisplay.getPortletGroupId(),\n            BookReportsEntry.class.getName(),\n            book.getEntryId(), ActionKeys.DELETE) %>">
```

Now add the line </c:if> before the line </liferay-ui:icon-menu> in entity_action.jspf.

As shown in the code above, it checks if the user has the corresponding permissions, for example UPDATE, PERMISSIONS, and DELETE. You can see the code <c:if test= ... > ...</c:if>. These are the **JSTL** tags.



The **Java Server Pages Standard Tag Library (JSTL)** encapsulates the core functionalities common to many web applications as simple tags. For more details, refer to <http://java.sun.com/products/jsp/jstl/>.

Deploying

Congratulations! You have developed the advanced struts portlet book_reports. You can now deploy the changes into Tomcat by using the following steps:

1. Stop Tomcat if it is running.
2. Click on the `clean` and `deploy` Ant targets from `ext` in the **Ant** view.
3. Click on the Ant target `deploy-fast` from `ext-web` in the **Ant** view if there are changes only in JSP files.
4. Start Tomcat.

In short, the following are the main processes (either in sequence or in parallel) to build an advanced Struts portlet on top of Liferay portal:

1. Define the portlet (JSR-286 attributes) in `portlet-ext.xml`.
2. Register the portlet (Liferay portal attributes) in `liferay-portlet-ext.xml`.
3. Set the title, map the category to a value, and add permission entries in `Language-ext.properties`.
4. Create JSP pages such as `view.jsp`, `init.jsp`, `view_books.jsp`, `entry-action.jspf`, `error.jsp`, `success.jsp`, and `view_reports.jsp`.
5. Add the portlet to a category in `liferay-display.xml`.
6. Define the action path and configure the tiles plug-in in `struts-config.xml`.
7. Create the tiles definition and set up redirecting in `tiles-defs.xml`.
8. Build services by using ServiceBuilder with `service.xml`.
9. Define permissions for the portlet and model resources in `book.xml`.
10. Map to portlet permissions in `default-ext.xml`.
11. Hook permissions in `portal-ext.properties`.
12. Define the resource bundle and add internalization in `portlet-ext.xml`.

Using Struts efficiently

We have developed the JSP portlet and the Struts portlet. We have also smoothly customized Liferay portal with Struts. As we know, Liferay portal is made up of a Portlet API, Struts and Tiles, Spring and Hibernate, SOAP, RMI and Tunneling, and so on. Struts are one of the core parts of Liferay portal. A better understanding of Struts leads to a better understanding of the Liferay portal web framework functions.



Apache Struts is an open source framework for building Servlet/JSP based web applications that are based on the MVC design paradigm.
Refer to <http://struts.apache.org/>.

Why use Struts?

Apache Struts provides centralized page-flow management in the form of `struts-config.xml`. This makes it highly scalable and allows you to modularize the coding process. By using Struts, you will be using a number of best practices that have been built into the framework.

Why use tiles?

Apache Struts provides centralized page-layout management in the form of `tiles-defs.xml`. This makes it highly scalable and allows you to manage layout information for Liferay portal with customizable templates. As we know, a page layout is typically designed using including statements. If there were 1,000 JSP files and the header and footer needed to be swapped, all 1,000 JSP files would need to be changed. With tiles, a single template can be used to determine the page layout. Only the template needs to be changed, and all the pages will be updated accordingly.

When do we use Struts?

Liferay portlet provides over 60 portlets out of the box. All of these portlets are, by default, developed as Struts portlets. Thus, if you want to modify or extend the out-of-the-box portlets, you have to use Struts.

Liferay portal uses Struts version 1.1 by default. If you want to use a Struts version other than 1.1, the struts portlets in Ext will not be your best choice. If you want to use Struts version 2.x or above, you should not use it in Ext. Fortunately, you can use Struts version 2.x or above in Plugins SDK. In Plugins SDK, you can use any Struts version to develop your portlets and to integrate other systems.

Summary

This chapter first discussed how to develop a JSP portlet. Then it introduced how to develop a basic Struts portlet in Ext—defining the portlet, specifying page action, and page layout. Accordingly, it also introduced how to develop an advanced Struts portlet in Ext—redirecting, adding more actions, setting up permissions, and so on. Finally, it addressed how to use Struts efficiently.

In the next chapter, we are going to customize two Struts portlets: Manage Pages and Communities.

5

Managing Pages

The Internet web sites www.bookpubstreet.com and www.bookpubworkshop.com will be built on top of the Liferay portal. Each site will be represented as a community and each community is made up of a lot of pages, for example, public pages and private pages. A private page in a community can be accessed only by logged-in users—who are a part of the community—whereas, a public page in a community can be accessed by guests. In order to build these web sites, we need to manage communities and, further, manage pages for each community.

The Communities portlet provides the ability to create and manage communities and their users, as well as that of the portlet Manage Pages. The portlet Manage Pages provides the ability to manage a lot of pages of different communities. We have discussed how to customize and extend the portal using Struts. Now, let's use Struts to customize the out of the box portlets—Communities and Manage Pages.

This chapter will first discuss how to extend the Communities portlet with additional fields. Then we will address how to customize the Manage Pages portlet with additional fields and how to apply layout templates in runtime. Further, this chapter will address how to employ features of page management. Finally, it will introduce how to use communities and layout pages efficiently.

By the end of this chapter, you will have learned how to:

- Extend the Communities portlet
- Customize the Manage Pages portlet
- Customize page management with more features
- Use communities and layout pages efficiently

Extending Communities portlet

The Communities portlet provides the ability to create and manage communities and their users. A community is a special group holding a number of users who share common interests. By default, a community is represented by the Group_ table with fields such as groupId, companyId, creatorUserId, name, description, type, typeSettings, friendlyURL, active, and so on. Now let's take an in-depth look at the customization of the community.

As shown in following screenshot, we may want to add one searchable field for each community, which is **Keywords**. For example, suppose we are creating a new community with the name **Book Street**, and the description **a community for website www.bookpubstreet.com**. Now we have a chance to add the new **Keywords** field with the value, for example, **Book; Street; Palm Tree; Publication**. Similarly, when editing the properties of a community—for example **Name**, **Description**, **Type**, and **Active**—we again have a chance to edit **Keywords**.

The screenshot shows the 'Ext Communities' portlet interface. At the top, there are 'View All' and 'Add' buttons. Below them, a form is displayed with the following fields:

- Name:** Book Street
- Keywords:** Book; Street; Palm Tree; Publication
- Description:** a community for website www.bookpubstreet.com
- Type:** Open (selected from a dropdown)
- Active:** checked (indicated by a checked checkbox)

In addition, we expect to have more fields in the customized communities: **Created** (when the community was created), **ModifierUserId** (who modified the community), and **Modified** (when the community was modified).

The screenshot shows the 'Ext Communities' portlet interface with a list of communities. The top navigation bar includes 'Communities I Own', 'Communities I Have Joined', and 'Available Communities'. Below this is a search bar and an 'Add Community' button. The main area displays a table of communities:

Name	Creator	Keywords	Type	Members	Online Now	Active
Book Street	Test Test	Book; Street; Palm Tree; Publication	Open	1	0	Yes

To the right of the table is a sidebar with the following options:

- Edit
- Manage Pages
- Assign User Roles
- Assign Members
- Leave
- Delete
- Actions

As shown in the preceding screenshot, when listing communities, not only should default fields (for example, **Name**, **Type**, **Members**, **Online Now**, **Active**, **Pending Requests**) and **Actions** icons (for example, **Edit**, **Permissions**, **Manage Pages**, **Assign User Roles**, **Assign Members**, **Leave**, and **Delete**) be displayed, but also the customized columns (for example, the username and **Keywords**) should be displayed.

How do we implement these features? In this section, we're going to show how to customize the **Communities** portlet using the above requirements as examples. Obviously, it is open for you to customize this portlet in a number of ways according to your own requirements. In general, the processes for customization of this portlet should be the same.

Building Ext Communities portlet

The **Communities** portlet can be used to create and manage new portal communities and their users. As you can see, a community can be regarded as a separate portal instance; each community gets its own set of pages, content management system, shared calendar, and default permissions. Moreover, a user belonging to multiple communities can navigate among them within the same portal session.

Generally speaking, we do not want to update the **Communities** portlet, but keep it as it is. Our goal is to customize and extend it. In general, this can be done by using the following two steps:

1. Build a customized Ext Communities portlet, which has exactly the same functions, look, and feel as that of the original **Communities** portlet.
2. Extend this customized portlet and let it have an additional model and service, and moreover, its own look and feel.

In this part, let's build the Ext Communities portlet, having exactly same functions, look, and feel as that of the **Communities** portlet.

Constructing the portlet

Now let's define a Struts portlet with the name "Ext Communities". We first need to configure it in both `portlet-ext.xml` and `liferay-portlet-ext.xml`, and then set the title in `Language-ext.properties`, and then add the Ext Communities portlet to the Book category in `liferay-display.xml`.

First, let's configure the Ext Communities portlet in `portlet-ext.xml` as follows:

1. Locate the `portlet-ext.xml` file in the `/ext/ext-web/docroot/WEB-INF` folder and open it.
2. Add the following lines between `</portlet>` and `</portlet-app>` and save the file:

```
<portlet>
    <portlet-name>extCommunities</portlet-name>
    <display-name>Ext Communities</display-name>
    <portlet-class>com.liferay.portlet.StrutsPortlet</portlet-class>
    <init-param><name>view-action</name>
        <value>/ext/communities/view</value></init-param>
    <expiration-cache>0</expiration-cache>
    <supports><mime-type>text/html</mime-type></supports>
    <resource-bundle>
        com.liferay.portlet.StrutsResourceBundle
    </resource-bundle>
    <security-role-ref>
        <role-name>power-user</role-name>
    </security-role-ref>
    <security-role-ref>
        <role-name>user</role-name>
    </security-role-ref>
</portlet>
```

As shown in the code above, the `portlet-name` element contains the canonical name of the portlet (for example, `extCommunities`). The `display-name` element contains a short name that is intended to be displayed in the portal (for example, `Ext Communities`). The `portlet-class` element contains the fully qualified class name of the portlet (for example, `com.liferay.portlet.StrutsPortlet`). The `init-param` element contains a name-value pair, for example `view-action - ext/communities/view`, as an initialization parameter of the portlet. Further, the `expiration-cache` defines expiration-based caching for this portlet. The `supports` element contains the supported MIME-type. The `resource-bundle` element contains a resource bundle class, for example `com.liferay.portlet.StrutsResourceBundle`. Finally, the `security-role-ref` element contains the declaration of a security role reference in the code of the web application.

Secondly, let's register the `extCommunities` portlet in `liferay-portlet-ext.xml` as follows:

1. Locate the `liferay-portlet-ext.xml` file in the `/ext/ext-web/docroot/WEB-INF` folder and open it.
2. Add the following lines immediately after `<!-- Custom Portlets -->` and save it:

```
<portlet>
<portlet-name>extCommunities</portlet-name>
<struts-path>ext/communities</struts-path>
<use-default-template>false</use-default-template>
<restore-current-view>false</restore-current-view>
</portlet>
```

As shown in the code above, the Ext Communities portlet is registered in the portal. The portal will check `struts-path` to see whether a user has the required permissions to access the portlet or not. As you can see, `struts-path` has the value `ext/communities`. It means that all requests to the `ext/communities/*` path are considered a part of this portlet scope. Only those users whose request paths match `ext/communities/*` will be granted access.

Moreover, the `use-default-template` element has the value `false`, so the portlet will not use any user's default template. The `restore-current-view` element has the value `false` so the portlet will reset the current view when toggling between maximized and normal states.

Thirdly, add a title (for example `Ext Communities`), for the Ext Communities portlet at `Language-ext.properties` as follows:

1. Locate the `Language-ext.properties` file in the `/ext/ext-impl/src/content` folder and open it.
2. Add the following line after `javax.portlet.title.book_reports=Reports` for Books and save it:

```
javax.portlet.title.extCommunities=Ext Communities
```

The code above provides mapping for the title of the portlet. If the mapping is not provided, the portal will show the default title `javax.portlet.title.extCommunities`.

Finally, add the `Ext Communities` portlet to the Book category in `liferay-display.xml` as follows:

1. Locate the `liferay-display.xml` file in the `/ext/ext-web/docroot/WEB-INF` folder and open it.
2. Add the following line immediately after the line `<portlet id="book_reports" />` and save it:

```
<portlet id="extCommunities" />
```

As shown in the code above, it adds the Ext Communities portlet to the category Book. From now on, you are able to select this portlet from the Book category directly when adding portlets to pages.

Setting up actions

Now, let's set up all actions required for the Ext Communities portlet. We need to prepare an action class, for example, `ExtEditGroupAction`. So how do we build this action? You can build the actions from scratch, but our purpose is to customize and extend the Communities portlet. In one word, we expect to reuse the out-of-the-box portlet source code as much as possible and to write minimum code.

As mentioned earlier, we have the `portal` project for the portal source code in the Eclipse IDE, which is referred to as the `/portal` prefix. We also have the `ext` project for customized code, which is referred to as the `/ext` prefix. The following is a process flow to build the `ExtEditGroupAction` action class of the Ext Communities portlet.

1. Create a `com.ext.portlet.communities.action` package in the `/ext/ext-impl/src` folder.
2. Create an `ExtEditGroupAction` class in this package and open it.
3. Add the following lines and save it:

```
public class ExtEditGroupAction extends EditGroupAction {  
    public void processAction( ActionMapping mapping, ActionForm  
        form, PortletConfig portletConfig,  
        ActionRequest actionRequest, ActionResponse actionResponse)  
        throws Exception {  
        String cmd = ParamUtil.getString(actionRequest,  
            Constants.CMD);  
        try {  
            if (cmd.equals(Constants.ADD) ||  
                cmd.equals(Constants.UPDATE)) {  
                updateGroup(actionRequest);  
            }  
            else if (cmd.equals(Constants.DELETE)) {  
                deleteGroup(actionRequest);  
            }  
            sendRedirect(actionRequest, actionResponse);  
        }  
        catch (Exception e) {  
            if (e instanceof NoSuchGroupException ||  
                e instanceof PrincipalException) {  
                SessionErrors.add(actionRequest, e.getClass().getName());  
                setForward(actionRequest, "portlet.ext.communities.error");  
            }  
            else if (e instanceof DuplicateGroupException ||  
                    e instanceof GroupFriendlyURLException ||  
                    e instanceof GroupNameException ||  
                    e instanceof RequiredGroupException) {  
                SessionErrors.add(actionRequest, e.getClass().getName(), e);  
                if (cmd.equals(Constants.DELETE)) {  
                    actionResponse.sendRedirect(  
                        "http://  
                            " + portletConfig.getPortletName() + ".  
                            " + portletConfig.getPortletId() + ".  
                            " + portletConfig.getPortletName() + ".do?  
                            " + Constants.DELETE + "&group=" +  
                            group);  
                }  
            }  
        }  
    }  
}
```

```

        ParamUtil.getString(actionRequest, "redirect"));
    }
} else { throw e;}
}
}

public ActionForward render(ActionMapping mapping, ActionForm
form, PortletConfig portlonfig,
RenderRequest renderRequest, RenderResponse renderResponse)
throws Exception {
try { ActionUtil.getGroup(renderRequest); }
catch (Exception e) {
if (e instanceof NoSuchGroupException ||
e instanceof PrincipalException) {
SessionErrors.add(renderRequest,
e.getClass().getName());
return mapping.findForward
("portlet.ext.communities.error");
}
else {throw e;}
}
}

return mapping.findForward(getForward(renderRequest,
"portlet.ext.communities.edit_community"));
}
}

```

As shown in the code above, `ExtEditGroupAction` extends `EditGroupAction` from the `com.liferay.portlet.communities.action` package in the `/portal/portal-impl/src` folder. It overrides two methods (`render` and `processAction`) of `EditGroupAction`.

Setting up page flow and page layout

We have set up the action. We have also updated the forward path as the portlet.
ext.communities.* value. In order to get the page flow working, we need to set up
an action path and a page flow.

First, let's set up the action path and page flow in struts-config.xml as follows:

1. Locate the struts-config.xml file in the /ext/ext-web/docroot/WEB-INF folder and open it.
 2. Add the following lines after <struts-config> <action-mappings> and save it:

```
<!-- Ext Communities -->  
<action path="/ext/communities/edit_community"  
       type="com.ext.portlet.communities.action.  
             ExtEditGroupAction">
```

```
<forward name="portlet.ext.communities.edit_community"
         path="portlet.ext.communities.edit_community" />
<forward name="portlet.ext.communities.error"
         path="portlet.ext.communities.error" />
</action>
<action path="/ext/communities/view"
        forward="portlet.ext.communities.view" />
```

The code above defines a set of action paths associated with the action and forward paths, as well as those mentioned earlier. For example, the action path `/ext/communities/edit_community` is associated with the `com.ext.portlet.communities.action.ExtEditGroupAction` action and the forward path names `portlet.ext.communities.edit_community` and `portlet.ext.communities.error`.

Then based on the page flow and JSP files, let's define the page layout in `tiles-defs.xml`:

1. Locate the `tiles-defs.xml` file in the `ext/ext-web/docroot/WEB-INF` folder and open it.
2. Add the following lines after `<tiles-definitions>` and save it:

```
<!-- Ext Communities -->
<definition name="portlet.ext.communities" extends="portlet" />
<definition name="portlet.ext.communities.edit_community"
            extends="portlet.ext.communities">
    <put name="portlet_content"
         value="/portlet/ext/communities/edit_community.jsp" />
</definition>
<definition name="portlet.ext.communities.view" extends="portlet">
    <put name="portlet_content"
         value="/portlet/ext/communities/view.jsp" />
</definition>
<definition name="portlet.ext.communities.error"
            extends="portlet">
    <put name="portlet_content"
         value="/portlet/communities/error.jsp" />
</definition>
```

The code above defines the page layout for the Ext Communities portlet. For example, `portlet.ext.communities.edit_community` is associated with the JSP file `/portlet/ext/communities/edit_community.jsp`. In addition, it specifies that the community view page layout (for example, `portlet.ext.communities.view`) is associated with the JSP page file `/portlet/ext/communities/view.jsp`.

Preparing JSP files

We have now set up the actions. We have also set up page flow and page layout. Now let's set up the JSP files that are required for the Ext Communities portlet. We need to prepare JSP files such as `view.jsp`, `edit_community.jsp`, `group_search.jsp`, and so on. So how do we build this? You can build them from scratch. However, here we will copy and modify JSP files of the Communities portlet. In this section we expect to reuse the source code, including JSP files, as much as possible.

First, let's create the `view.jsp` JSP file as follows:

1. Create a `communities` folder within the `/ext/ext-web/docroot/html/portlet/ext/` folder.
2. Locate the `view.jsp` JSP file in the `/portal/portal-web/docroot/html/portlet/communities` folder, and copy it to the `/ext/ext-web/docroot/html/portlet/ext/communities` folder.
3. Open `view.jsp` in the `/ext/ext-web/docroot/html/portlet/ext/communities` folder, update `/communities/edit_community` with `/ext/communities/edit_community` as shown in the following two lines, and save it:

```
portletURL.setParameter("struts_action", "/ext/communities/view");
<liferay-ui:search-form
    page="/html/portlet/ext/communities/group_search.jsp"
    searchContainer="<% = searchContainer %>"
    showAddButton="<% = showTabs1 %>" />
```

Next, we need to create the JSP file `edit_community.jsp` as follows:

1. Locate the JSP file `edit_community.jsp` in the `/portal/portal-web/docroot/html/portlet/communities` folder, and copy it to the `/ext/ext-web/docroot/html/portlet/ext/communities` folder.
2. Open `edit_community.jsp` in the `/ext/ext-web/docroot/html/portlet/ext/communities` folder, update `/communities/edit_community` with `/ext/communities/edit_community` as shown in following line, and save it:

```
<form action="

```

In addition, we need to make the button Add Community available, in the following manner:

1. Locate JSP file group_search.jsp in the /portal/portal-web/docroot/html/portlet/enterprise_admin folder.
2. Copy the JSP file group_search.jsp from /portal/portal-web/docroot/html/portlet/_ enterprise_admin to /ext/ext-web/docroot/html/portlet/ext/communities, and open it.
3. Update /communities/edit_community with /ext/communities/edit_community as shown in the following lines, and save it:

```
submitForm(document.<portlet:namespace />fm, '<portlet:renderURL  
windowState="<%WindowState.MAXIMIZED.toString()%>">  
<portlet:param name="struts_action"  
value="/ext/communities/edit_community" />  
<portlet:param name="redirect"  
value="<%currentURL%>" />  
</portlet:renderURL>');
```

Congratulations! You have cloned the Communities portlet. Finally, we can deploy updates into Tomcat as follows:

1. Stop Tomcat if it is running.
2. Click on the Ant target: deploy at the Ant view ext.
3. Start Tomcat.
4. Open up a new browser with the URL <http://localhost:8080>.
5. Click on **Sign in** and enter **test@liferay.com / test**.
6. Click on **Add Application | Book**.

Then you will see the Ext Communities portlet with the **Ext Communities** title and its contents. As you can see, this portlet is under the **Book** category now. Note that the implementation of the **Actions** icons—for example **Edit, Permissions, Manage Pages, Assign User Roles, Assign Members, Leave, and Delete**—is not included yet. But you can use the above processes to include them.

Setting up the Ext Communities portlet in the backend

We have cloned the Communities portlet as well. Now let's see how to implement new features in the backend.

Creating database structure

First, we need to add new fields for the Ext Communities portlet. These fields could be represented in `service.xml`. In order to generate services, let's build the fields as follows:

1. Create a `com.ext.portlet.group` package in the `/ext/ext-impl/src` folder.
2. Create a `service.xml` file in this package and open it.
3. Add the following lines and save it:

```
<?xml version="1.0"?><!DOCTYPE service-builder PUBLIC "-//Liferay//DTD Service Builder 5.2.0//EN" "http://www.liferay.com/dtd/liferay-service-builder_5_2_0.dtd">
<service-builder package-path="com.ext.portlet.group">
    <namespace>ExtGroup</namespace>
    <entity name="ExtGroup" uuid="false"
        local-service="true"
        remote-service="true"
        persistence-class="com.ext.portlet.group.service.persistence.ExtGroupPersistenceImpl">
        <column name="groupId" type="long" primary="true" />
        <column name="keywords" type="String" />
        <column name="creator" type="String" />
        <column name="modifierUserId" type="String" />
        <column name="created" type="Date" />
        <column name="modified" type="Date" /></entity>
        <exceptions><exception>ExtGroup</exception></exceptions>
    </service-builder>
```

As shown in the code above, `package-path com.ext.portlet.group` is the path that the class will generate to the `/ext/ext-impl/src/com/ext/portlet/group` folder. The entity has the `ExtGroup` namespace with an `ExtGroup` name. It will use local service, remote service, and so on. The entity has the primary key field `groupId`, and other fields such as `keywords`, `created`, `modifierUserId`, `modified`, and so on. Accordingly, an exception with the `ExtGroup` name is also specified – it will generate a class `ExtGroupException` at the `com.ext.portlet.group` package.

Then, we need to build a service with ServiceBuilder. After preparing `service.xml`, you can build the services as follows:

1. Locate the XML file `/ext/ext-impl/build-parent.xml` and open it.
2. Add the following lines between `</target>` and `<target name="build-service-portlet-reports">` and save it:

```
<target name="build-service-portlet-extCommunities">
    <antcall target="build-service">
```

```
<param name="service.file"
       value="src/com/ext/portlet/group/service.xml" />
</antcall>
</target>
```

3. Expand `ext-impl` in the **Ant** view.
4. Double-click on the target: `build-service-portlet-extCommunities`.
5. Refresh the `/ext` project.

ServiceBuilder will first generate models, services, and service persistence in the `ext/ext-service/src` folder. The following are the packages that contain services (SOAP-based web services, REST style web services, or JSON access via JavaScript, and so on):

- `com.ext.portlet.group`
- `com.ext.portlet.group.model`
- `com.ext.portlet.group.service`
- `com.ext.portlet.group.service.persistence`

Similarly, ServiceBuilder will generate implementations of models, services, and service persistence in the `/ext/ext-impl/src` folder. The following are the packages that contain the implementation of the services (SOAP-based web services, REST style web services, or JSON access via JavaScript, and so on):

- `com.ext.portlet.group.model.impl`
- `com.ext.portlet.group.service.base`
- `com.ext.portlet.group.service.http`
- `com.ext.portlet.group.service.impl`
- `com.ext.portlet.group.service.persistence`

Finally, we need to create a new table `ExtGroup` in the database in the following manner:

```
drop table if exists ExtGroup;
create table ExtGroup (
    groupId bigint not null primary key,
    keywords varchar(511), modifierUserId bigint,
    created datetime null, modified datetime null
);
```

As shown in code above, it drops the `ExtGroup` table if it exists. Then it creates an `ExtGroup` table with columns `groupId`, `keywords`, and so on.

Creating methods to update, delete, and retrieve

As stated above, the basic services and models are automatically generated by ServiceBuilder. Now we can add the database insert method (to add the community column Keywords) and the database retrieve method (to get all of the additional information about the community). Let's create the action named AddGroupLocalServiceUtil under the com.ext.portlet.group.action of /ext/ext-impl/src package.

```
public class AddGroupLocalServiceUtil {
    public static ExtGroup getGroup(long groupId) {
        ExtGroup group = null;
        try {
            group = ExtGroupLocalServiceUtil.getExtGroup(groupId);
        }
        catch (Exception e){}
        if(group == null)
            group = ExtGroupLocalServiceUtil.createExtGroup(groupId);
        return group;
    }
    public static void deleteGroup(ActionRequest actionRequest) {
        long groupId = ParamUtil.getLong(actionRequest, "groupId");
        try{ ExtGroupLocalServiceUtil.deleteExtGroup(groupId); }
        catch (Exception e){}
    }
    public static void updateGroup(ActionRequest actionRequest) {
        ThemeDisplay themeDisplay = (ThemeDisplay)actionRequest.
            getAttribute(WebKeys.THEME_DISPLAY);
        String name = ParamUtil.getString(actionRequest, "name");
        String keywords = ParamUtil.getString(actionRequest, "keywords");
        long groupId = ParamUtil.getLong(actionRequest, "groupId");
        Date now = new Date(); ExtGroup group = null;
        try{
            Group o = GroupServiceUtil.getGroup
                (themeDisplay.getCompanyId(), name);
            if(groupId <= 0) groupId = o.getGroupId();
            group = ExtGroupLocalServiceUtil.getExtGroup(groupId);
        }
        catch (Exception e){}
        if (group == null) {
            group = ExtGroupLocalServiceUtil.createExtGroup(groupId);
            group.setCreated(now);
        }
        else { group.setModified(now);
            group.setModifierUserId(themeDisplay.getUser().getUserId());
            group.setKeywords(keywords);
            try{ ExtGroupLocalServiceUtil.updateExtGroup(group); }
            catch (Exception e){}
        }
    }
}
```

As shown in the code above, there are three methods: `updateGroup`, `getGroup`, and `deleteGroup`. The `updateGroup` method inserts or updates `extGroup` into the database, the `getGroup` method retrieves `extGroup`, and the `deleteGroup` method deletes `extGroup` by group ID. We use the name `AddGroupLocalServiceUtil` for demo purpose only. You can use a different name. Moreover, the service generated by the `ServiceBuilder`, for example, `ExtGroupLocalServiceUtil`, includes CRUD operations and handling transaction. We do not expect to modify this service. We simply add a customized wrapper `AddGroupLocalServiceUtil` on top of the service, for example `ExtGroupLocalServiceUtil`. Thus, when upgrading `ServiceBuilder`, we only need to upgrade this wrapper if any changes are involved.

Updating the action classes

After creating the methods to update, delete, and retrieve the community's additional information, we need to associate these methods with the corresponding methods in the action class.

First, let's associate the `deleteGroup` method in `AddGroupLocalServiceUtil` with the `deleteGroup` method in `ExtEditGroupAction`, using the following steps:

1. Locate the `ExtEditGroupAction.java` file under the `com.ext.portlet.communities.action` package in the `/ext/ext-impl/src` folder and open it.
2. Add following line before the line `deleteGroup(actionRequest);` and save it:

```
AddGroupLocalServiceUtil.deleteGroup(actionRequest);
```

The code above specifies a function to delete `ExtGroup`.

Similarly, we can associate the `updateGroup` method in `AddGroupLocalServiceUtil` with the `updateGroup` method in `ExtEditGroupAction`.

3. Open the `ExtEditGroupAction.java` file.
4. Add following line after the line `updateGroup(actionRequest);` and save it.

```
AddGroupLocalServiceUtil.updateGroup(actionRequest);
```

The code above shows a function to update `ExtGroup` after updating `Group_`.

Setting up the Ext Communities portlet in the frontend

We have set up the Ext Communities portlet in the backend. Now let's set it up in the frontend.

Updating and deleting Community customized columns

First, we need to add the service and model imports in the `init.jsp` file as follows:

1. Create an `init.jsp` file in the `/ext/ext-web/docroot/portlet/ext/communities` folder and open it.
2. Add the following lines at the beginning of this file and save it:

```
<%@ include file="/html/portlet/communities/init.jsp" %>
<%@ page import="com.ext.portlet.group.action.
AddGroupLocalServiceUtil" %>
<%@ page import="com.ext.portlet.group.model.ExtGroup" %>
```

Then, we need to update the `edit_community.jsp` file in the `/ext/ext-web/docroot/portlet/ext/communities` folder in order to support the updates of community-customized fields. To do so, update `<%@ include file="/html/portlet/communities/init.jsp" %>` with `<%@ include file="/html/portlet/ext/communities/init.jsp" %>`, and add the following lines after the line `long groupId = BeanParamUtil.getLong(group, request, "groupId");` in `edit_community.jsp` file:

```
ExtGroup extGroup = null;
if(group != null)
    extGroup = AddGroupLocalServiceUtil.getGroup(group.getGroupId());
```

Further, add the following lines after the lines `<liferay-ui:input-field model="<% Group.class %>" bean="<% group %>" field="name" /></td></tr>`:

```
<tr>
    <td class="lfr-label"><liferay-ui:message key="keywords" /></td>
    <td>
        <liferay-ui:input-field model="<% ExtGroup.class %>"
            bean="<% extGroup %>" field="keywords" />
    </td>
</tr>
```

As shown in the code above, the JSP gets `ExtGroup` by the group ID as import. Then it adds a message `Keywords`, an input with model `ExtGroup`, bean `extGroup`, and field `Keywords`.

Retrieving community-customized columns

Last but not the least, we need to update the `view.jsp` file in the `/ext/ext-web/docroot/portlet/ext/communities` folder to properly display community columns `Keywords` and `Creator`. To do so, first update `<%@ include file="/html/portlet/communities/init.jsp" %>` with `<%@ include file="/html/portlet/ext/communities/init.jsp" %>`. Then add the following lines after the line `headerNames.add("name");` in the `view.jsp` file:

```
headerNames.add("Creator"); //Ext  
headerNames.add("Keywords"); //Ext
```

Finally, add the following lines between `row.addText(sb.toString());` and `// Type:`

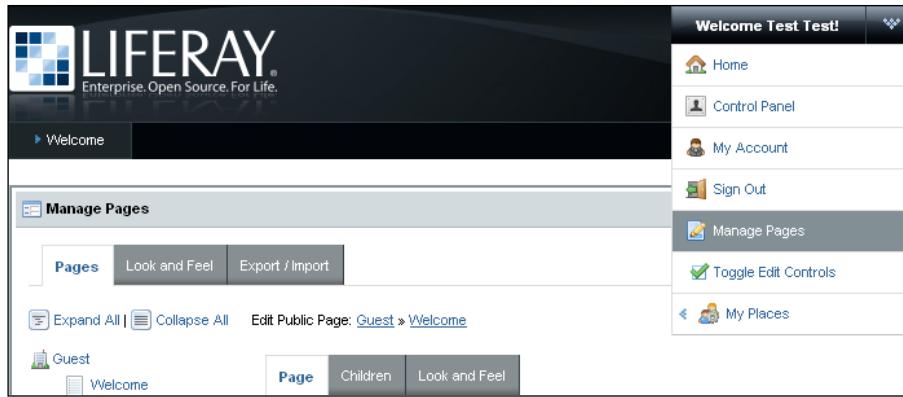
```
// EXT  
User creator = UserLocalServiceUtil.getUser  
    (group.getCreatorUserId());  
row.addText(creator.getFullName());  
ExtGroup extGroup = AddGroupLocalServiceUtil.getGroup  
    (group.getGroupId());  
row.addText(extGroup.getKeywords());
```

As shown in the code above, when retrieving a normal group message, it tries to get the community additional information via the `ExtGroup` object. Then, it lists the headers for the community columns `Creator` and `Keywords`. Finally, it adds the value for customized rows, for example `creator.getFullName()` and `extGroup.getKeywords()`.

Customizing the Manage Pages portlet

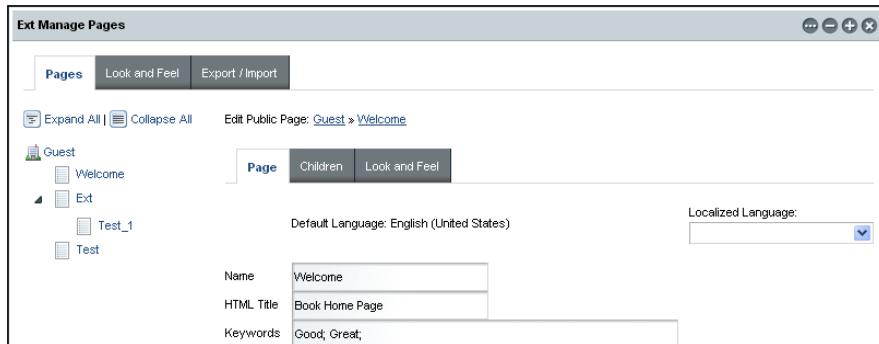
As stated earlier, a community may consist of a set of pages, including public pages and private pages. The `Communities` portlet also provides the ability to manage pages for a given community. In addition, Liferay portal also provides a portlet named `Manage Pages` (known as `Page Settings` previously) to manage the pages—either public or private—for a given group or community. Unfortunately, you cannot use it independently. That is, you cannot find it by clicking on **Add Applications** when you log in as an admin. A link is provided only in the theme at `/portal/portal-web/docroot/html/themes/classic/templates/dock.vm` as follows:

```
#if ($show_page_settings)  
<li class="page-settings">  
  <a href="$page_settings_url">$page_settings_text</a>  
</li>  
#end
```



As shown in this screenshot, you can click on the link **Manage Pages** to update page settings. Let us suppose that you are in the **Guest** community, in the public pages by default and the **Welcome** page is under editing.

As shown in the following screenshot, here we want to customize the **Manage Pages** portlet. First, clone this portlet and make it standalone. Then, call it as **Ext Manage Pages**, and dynamically track the community to which the current page belongs. For example, if the portlet named **Ext Manage Pages** was added in the page of the **Guest** community, it should dynamically get the **Guest** community as a group name. Most importantly, we expect to extend the model of the page layout so that it supports additional messages, for example keywords, created(when it was created), creatorUserId (who created it), modifierUserId (who modified it), modified (when it was modified), and so on.



How to implement the above requirements? As was shown for the Ext Communities portlet, we will show how to customize the **Manage Pages** portlet. We will use the above requirements as examples to show what will be customized. However, it is up to you to make any decisions about what should be customized according to your requirements.

Building a standalone layout management portlet

Here we are not going to update the Manage Pages portlet directly. Instead, we are going to build a new portlet named Ext Manage Pages by customizing the Manage Pages portlet.

Constructing the portlet

Now let's specify a portlet with the name Ext Manage Pages. Thus, we are required to first configure the Ext Manage Pages portlet in both `portlet-ext.xml` and `liferay-portlet-ext.xml`, set the title mapping in `Language-ext.properties`, and add the Ext Manage Pages portlet to the Book category in `liferay-display.xml`.

To do so, first add the following lines between `</portlet>` and `</portlet-app>` in `portlet-ext.xml`:

```
<portlet>
  <portlet-name>extLayoutManagement</portlet-name>
  <display-name>Ext Manage Pages</display-name>
  <portlet-class>com.liferay.portlet.StrutsPortlet</portlet-class>
  <init-param><name>view-action</name>
    <value>/ext/layout_management/edit_pages</value>
  </init-param><expiration-cache>0</expiration-cache>
  <supports><mime-type>text/html</mime-type></supports>
  <resource-bundle>
    com.liferay.portlet.StrutsResourceBundle
  </resource-bundle>
  <security-role-ref>
    <role-name>power-user</role-name>
  </security-role-ref>
  <security-role-ref>
    <role-name>user</role-name>
  </security-role-ref>
</portlet>
```

Then add the following lines after the line `<!-- Custom Portlets -->` in `liferay-portlet-ext.xml`:

```
<portlet>
  <portlet-name>extLayoutManagement</portlet-name>
  <struts-path>ext/layout_management</struts-path>
  <use-default-template>false</use-default-template>
  <restore-current-view>false</restore-current-view>
</portlet>
```

Moreover, add the following line after the line `javax.portlet.title.extCommunities=Ext Communities` in `Language-ext.properties`:

```
javax.portlet.title.extLayoutManagement=Ext Manage Pages
```

Finally, add the following line after the line `<portlet id="extCommunities" />` in `liferay-display.xml`:

```
<portlet id="extLayoutManagement" />
```

Setting up the action

Let's set up the `ExtEditPagesAction` action required for the Ext Manage Pages portlet as follows:

1. Create an `ExtEditPagesAction` class in the `com.ext.portlet.communities.action` package under the `/ext/ext-impl/src` folder and open it.
2. Add the following lines in this class and save it:

```
public class ExtEditPagesAction extends EditPagesAction {
    public void processAction(ActionMapping mapping, ActionForm
        form, PortletConfig portletConfig,
        ActionRequest actionRequest, ActionResponse actionResponse)
    throws Exception {
        String cmd = ParamUtil.getString(actionRequest,
            Constants.CMD);
        try {
            if (cmd.equals(Constants.ADD) ||
                cmd.equals(Constants.UPDATE)) {
                updateLayout(actionRequest, actionResponse);
            }
            else if (cmd.equals(Constants.DELETE)) {
                CommunitiesUtil.deleteLayout(actionRequest,
                    actionResponse);
            }
            String redirect = ParamUtil.getString(
                actionRequest, "pagesRedirect");
            sendRedirect(actionRequest, actionResponse, redirect);
        }
        catch (Exception e) {
            if (e instanceof NoSuchLayoutException ||
                e instanceof NoSuchProposalException ||
                e instanceof PrincipalException) {
                SessionErrors.add(actionRequest, e.getClass().getName());
                setForward(actionRequest,
                    "portlet.ext.communities.error");
            }
        }
    }
}
```

```
        }
        else if (e instanceof RemoteExportException) {
            SessionErrors.add(actionRequest,
                e.getClass().getName(), e);
            String redirect = ParamUtil.getString(
                actionRequest, "pagesRedirect");
            sendRedirect(actionRequest, actionResponse, redirect);
        }
        else if (e instanceof LayoutFriendlyURLException ||
            e instanceof LayoutHiddenException ||
            e instanceof LayoutNameException ||
            e instanceof LayoutParentLayoutIdException ||
            e instanceof LayoutSetVirtualHostException ||
            e instanceof LayoutTypeException ||
            e instanceof RequiredLayoutException ||
            e instanceof UploadException) {
            if (e instanceof LayoutFriendlyURLException) {
                SessionErrors.add(actionRequest,
                    LayoutFriendlyURLException.class.getName(), e);
            }
            else {
                SessionErrors.add(actionRequest, e.getClass().
                    getName(), e);
            }
        }
        else { throw e; }
    }
}

public ActionForward render(ActionMapping mapping, ActionForm
    form, PortletConfig portletConfig,
    RenderRequest renderRequest, RenderResponse renderResponse)
throws Exception {
    ThemeDisplay themeDisplay = (ThemeDisplay)renderRequest.
        getAttribute(WebKeys.THEME_DISPLAY);
    long groupId = themeDisplay.getScopeGroupId();
    renderRequest.setAttribute("GROUP",
        GroupLocalServiceUtil.getGroup(groupId));
    return mapping.findForward( getForward(renderRequest,
        "portlet.ext.communities.edit_pages"));
}
```

Setting up page flow and page layout

We have set up the action. So how do we get the page flow working? We need to set up an action path and page flow. To do so, first add the following lines after the lines `<struts-config> <action-mappings>` in `struts-config.xml` under the `/ext/ext-web/docroot/WEB-INF` folder:

```
<!-- Ext Layout Management -->
<action path="/ext/layout_management/edit_pages"
        type="com.ext.portlet.communities.action.ExtEditPagesAction">
    <forward name="portlet.ext.communities.edit_pages"
              path="portlet.ext.layout.edit_pages" />
    <forward name="portlet.ext.communities.error"
              path="portlet.ext.communities.error" />
</action>
```

Then add the following lines after `<tiles-definitions>` in `tiles-defs.xml` under the `/ext/ext-web/docroot/WEB-INF` folder:

```
<!-- Ext Layout Management -->
<definition name="portlet.ext.layout.edit_pages" extends="portlet">
    <put name="portlet_content"
          value="/portlet/ext/communities/edit_pages.jsp" />
</definition>
```

Preparing JSP files

As mentioned above, we used the `edit_pages.jsp` file in `tiles-defs.xml`. That is, we need to build the `edit_pages.jsp` file:

1. Locate the `edit_pages.jsp` file in the `/portal/portal-web/docroot/html/portlet/communities` folder and copy it to the `/ext/ext-web/docroot/html/portlet/ext/communities` folder.
2. Open this file and first update `<%@ include file="/html/portlet/communities/init.jsp" %>` with `<%@ include file="/html/portlet/ext/communities/init.jsp" %>`.
3. Then update `"/communities/edit_pages"` with `"/ext/layout_management/edit_pages"`, as shown in following lines:

```
portletURL.setParameter("struts_action",
        "/ext/layout_management/edit_pages");
<form action="
    <portlet:param name="struts_action"
```

```
        value="/ext/layout_management  
        /edit_pages" />  
    </portlet:actionURL>  
    method="post"  
    name="<portlet:namespace />fm"  
    onSubmit="<portlet:namespace />savePage(); return false;">
```

Cool! You have cloned the Manage Pages portlet successfully. After re-deploying these changes in Tomcat, you will now see the **Ext Manage Page** portlet with the **Ext Manage Page** title and contents under the **Book** category.

Setting up the Ext Layout Management portlet in the backend

After cloning the Manage Pages portlet, we will implement new features in the backend. By the way, the implementation flow is close to that of the Ext Communities portlet.

Creating a database structure

First of all, a database structure for new features is represented in `service.xml`. Let's create a new XML file `service.xml` as follows:

1. Create a `com.ext.portlet.layout` package in the `/ext/ext-impl/src` folder.
2. Create a `service.xml` file in the `com.ext.portlet.layout` package and open it.
3. Add the following lines in it and save it:

```
<?xml version="1.0"?><!DOCTYPE service-builder PUBLIC "-//  
Liferay//DTD Service Builder 5.2.0//EN" "http://www.liferay.com/  
dtd/liferay-service-builder_5_2_0.dtd">  
<service-builder package-path="com.ext.portlet.layout">  
    <namespace>ExtLayout</namespace>  
    <entity name="ExtLayout"  
        uuid="false"  
        local-service="true"  
        remote-service="true"  
        persistence-class="com.ext.portlet.layout.service.  
                        persistence.ExtLayoutPersistenceImpl">  
        <column name="plid" type="long" primary="true" />  
        <column name="keywords" type="String" />  
        <column name="creatorUserId" type="long" />  
        <column name="modifierUserId" type="long" />  
        <column name="created" type="Date" />
```

```
<column name="modified" type="Date" /></entity>
<exceptions><exception>ExtLayout</exception></exceptions>
</service-builder>
```

As shown in the code above, the entity has the ExtLayout namespace with a name ExtLayout. The entity has the primary key field plid and other fields such as keywords, creatorUserId, created, modifierUserId, and modified.

Then, we need to build the services using ServiceBuilder. This process is the same as that of building services of Ext Communities, which is as follows:

1. Locate the XML file /ext/ext-impl/build-parent.xml and open it.
2. Add the following lines between </target> and <target name="build-service-portlet-reports"> and save it:

```
<target name="build-service-portlet-extLayout">
    <antcall target="build-service">
        <param name="service.file"
            value="src/com/ext/portlet/layout/service.xml" />
    </antcall>
</target>
```
3. Expand ext-impl in the **Ant** view.
4. Double-click on the build-service-portlet-extLayout target.
5. Refresh the /ext project.

Finally, we should create a new table named ExtLayout in the database as follows:

```
drop table if exists ExtLayout;
create table ExtLayout (
    plid bigint not null primary key,
    keywords varchar(511),
    creatorUserId bigint, modifierUserId bigint,
    created datetime null,modified datetime null
) ;
```

As shown in the code above, it first drops the ExtLayout table (if it exists). Then it creates an ExtLayout table with a set of fields such as plid and keywords.

Creating methods to update, delete, and retrieve

The basic services and models are automatically generated by ServiceBuilder. Now we can add the database insert method (to add page columns Keywords) and retrieve method (to get layout page additional information). Let's create the action named `AddLayoutLocalServiceUtil` under the `com.ext.portlet.layout.action` package of the `/ext/ext-impl/src` folder with the following contents:

```
public class AddLayoutLocalServiceUtil {  
    public static ExtLayout getLayout(long plid) {  
        ExtLayout extLayout = null;  
        try{  
            extLayout = ExtLayoutLocalServiceUtil.getExtLayout(plid);  
        }  
        catch (Exception e){}  
        if(extLayout == null)  
            extLayout = ExtLayoutLocalServiceUtil.createExtLayout(plid);  
        return extLayout;  
    }  
    public static void deleteLayout(ActionRequest actionRequest) {  
        ThemeDisplay themeDisplay = (ThemeDisplay)actionRequest.  
           getAttribute(WebKeys.THEME_DISPLAY);  
        long layoutId = ParamUtil.getLong(actionRequest, "layoutId");  
        long groupId = themeDisplay.getScopeGroupId();  
        boolean privateLayout = ParamUtil.getBoolean  
            (actionRequest, "privateLayout");  
        long plid = ParamUtil.getLong(actionRequest, "plid");  
        try{  
            if(plid <= 0 )  
                plid = LayoutLocalServiceUtil.getLayout(groupId,  
                    privateLayout, layoutId).getPlid();  
            ExtLayoutLocalServiceUtil.deleteExtLayout(plid);  
        }  
        catch (Exception e){}  
    }  
    public static void updateLayout(ActionRequest actionRequest) {  
        ThemeDisplay themeDisplay = (ThemeDisplay)actionRequest.  
           getAttribute(WebKeys.THEME_DISPLAY);  
        String cmd = ParamUtil.getString(actionRequest, Constants.CMD);  
        long layoutId = ParamUtil.getLong(actionRequest, "layoutId");  
        String keywords = ParamUtil.getString(actionRequest, "keywords");  
        long groupId = themeDisplay.getScopeGroupId();  
        boolean privateLayout = ParamUtil.getBoolean(  
            actionRequest, "privateLayout");  
        Date now = new Date();ExtLayout extLayout = null;
```

```

long plid = 0;
try{
    if (cmd.equals(Constants.ADD))
        layoutId = getLayoutId(groupId, privateLayout);
    Layout layout = LayoutLocalServiceUtil.getLayout(groupId,
        privateLayout, layoutId);
    plid = layout.getPlid();
}
catch (Exception e){}
try{
    extLayout = ExtLayoutLocalServiceUtil.getExtLayout(plid);
}
catch (Exception e){}
if(extLayout == null){
    extLayout = ExtLayoutLocalServiceUtil.createExtLayout(plid);
    extLayout.setCreatorUserId(themeDisplay.getUser().getUserId());
    extLayout.setCreated(now);
}
else {
    extLayout.setModified(now);
    extLayout.setModifierUserId(themeDisplay.getUser().getUserId());
}
extLayout.setKeywords(keywords);
try{
    ExtLayoutLocalServiceUtil.updateExtLayout(extLayout);
}
catch (Exception e){}
}

public static long getLayoutId(long groupId, boolean privateLayout)
throws SystemException {
    long layoutId = 0;
    List<Layout> layouts = LayoutLocalServiceUtil.getLayouts
        (groupId, privateLayout);
    for (Layout curLayout : layouts) {
        long curLayoutId = curLayout.getLayoutId();
        if (curLayoutId > layoutId) {layoutId = curLayoutId; } }
    return layoutId;
}
}

```

As shown in code above, there are three methods: updateLayout, getLayout, and deleteLayout. The updateLayout method updates extLayout into the database, the getLayout method retrieves extLayout by page layout ID, and the deleteLayout method deletes extLayout.

Updating the action class

After creating the methods to update and delete, and getting the page additional information, we need to associate these methods with the corresponding methods in the action class.

First, let's associate the `deleteLayout` method in `AddLayoutLocalServiceUtil.java` with the `deleteLayout` method in `ExtEditPagesAction.java`. To do so, add the following line before the line `CommunitiesUtil.deleteLayout(actionRequest, actionResponse);` in `ExtEditPagesAction.java`:

```
AddLayoutLocalServiceUtil.deleteLayout(actionRequest);
```

The code above indicates that when using original method `deleteLayout`, applies for the extended method `deleteLayout` of `AddLayoutLocalServiceUtil`.

In the same way, we need to associate the `updateLayout` method in `AddLayoutLocalServiceUtil.java` with the `updateLayout` method in `ExtEditLayoutAction.java`. To do so, add the following line after the line `updateLayout(actionRequest, actionResponse);` in `ExtEditLayoutAction.java`:

```
AddLayoutLocalServiceUtil.updateLayout(actionRequest);
```

Setting up the layout management portlet in the frontend

We have set up the Ext Manage Pages portlet in the backend. It is time to construct the Ext Manage pages portlet in the frontend.

First, we need to add the service and model imports in the `init.jsp` file under the `/ext/ext-web/docroot/portlet/ext/communities` folder, similar to that of Ext Communities. To do so, simply add the following lines at the end of `init.jsp`:

```
<%@ page import="com.ext.portlet.layout.action.  
AddLayoutLocalServiceUtil" %>  
<%@ page import="com.ext.portlet.layout.model.ExtLayout" %>
```

Then, we need to update the `/ext/ext-web/docroot/portlet/ext/communities/edit_pages.jsp` file in order to use the Ext Manage Pages portlet (the portlet name is `extLayoutManagement`) instead of the original portlet name `PortletKeys.LAYOUT_MANAGEMENT`. Simply add a line `String extPortlet = "extLayoutManagement";` after the line `String tabs4 = ParamUtil.getString(request, "tabs4");` in the `edit_pages.jsp`. Now update `PortletKeys.LAYOUT_MANAGEMENT` with `extPortlet` as follows:

```
String extPortlet = "extLayoutManagement";
if (portletName.equals(extPortlet) ||
portletName.equals(PortletKeys.MY_ACCOUNT)) {
    portletURL.setParameter("backURL", backURL);
}
```

The code above tries to use the Ext Manage Pages portlet as well. Further, we need to update the /html/portlet/communities/edit_pages_public_and_private.jspf string with the /html/portlet/ext/communities/edit_pages_public_and_private.jspf string.

As you can see, we involve the JSP files edit_pages_public_and_private.jspf directly and edit_pages_page.jsp indirectly. Let's build these JSP files as follows:

1. Copy the file edit_pages_public_and_private.jspf in the /portal/portal-web/docroot/portlet/communities folder to the /ext/ext-web/docroot/portlet/ext/communities folder and open it.
2. Update the /html/portlet/communities/edit_pages_page.jsp string with the /html/portlet/ext/communities/edit_pages_page.jsp string and save it.
3. Copy the file edit_pages_page.jsp in the /portal/portal-web/docroot/portlet/communities folder to the /ext/ext-web/docroot/portlet/ext/communities folder.

Finally, we need to update the JSP file edit_pages_page.jsp in the /ext/ext-web/docroot/portlet/ext/communities/ folder in order to support the updates of the customized information. To do so, first update `<%@ include file="/html/portlet/communities/init.jsp" %>` with `<%@ include file="/html/portlet/ext/communities/init.jsp" %>` in edit_pages_page.jsp, and then add the following lines after the line `Layout selLayout = (Layout) request.getAttribute("edit_pages.jsp-selLayout");`.

```
ExtLayout extLayout = null;
if(selLayout != null)
    OextLayout = AddLayoutLocalServiceUtil.getLayout
        (selLayout.getPlid());
```

Moreover, add the following lines after the lines `<tr><td colspan="3">
</td></tr><tr><td><liferay-ui:message key="type" />:`

```
<tr>
<td><liferay-ui:message key="keywords" /></td>
<td>
<liferay-ui:input-field model="<%=" ExtLayout.class %>" bean="<%=" extLayout %>"
```

```
        field="keywords" />
</td>
<td></td>
</tr>
```

As shown in the code above, the JSP receives ExtLayout by the page layout ID. It also adds a Keywords message and an input with the ExtLayout model, the extLayout bean, and the keywords field. Interestingly, you can make the Keywords localized too. How to reach it? Refer to the following section.

Customizing page management with more features

We have successfully customized and extended the Manage Pages portlet. The Ext Manage Pages portlet not only clones the out-of-the-box Manage Pages portlet, but it also extends the model and service – supporting customized data, for example, Keywords. As mentioned earlier, we can make the Keywords localized too.

Adding localized feature

Liferay portal is designed to handle as many languages as you want to support. By default, it supports up to 22 languages. When a page is loading, the portal will detect the language, pull up the corresponding language file, and display the text in the correct language.

In this section we want the Keywords to be localized too. For example, the default language is English (United States) and the localized language is Deutsch (Deutschland). Thus, you have the ability to enter not only the Name and HTML Title in German, but also the Keywords in German.

As shown in the following screenshot, when you change the language of the page in German using the language portlet, you will see the entire web site changed to German, including the portlet title and input fields. For example, the title of the portlet now has the **Ext Seiteneinstellungen** value and the **Keywords** now become **Schlüsselwörter**.

How do we implement this feature? In other words, how do we customize the language display in the page management? Let's add the localized feature for the Ext Manage Pages portlet.

Extending model for locale

First of all, we need to extend the model and to implement that model in order to support the localized feature. For the `ExtLayout` model, let's add the `getLocale` method first.

1. Locate the `ExtLayout.java` file from the `com.ext.portlet.layout.model` package in the `/ext/ext-service/src` folder, and open it.
2. Add the following lines before the line `}` in `ExtLayout.java` and save it:

```
public String getKeywords(Locale locale);
public String getKeywords(String localeLanguageId);
public String getKeywords(Locale locale,
                         boolean useDefault);
public String getKeywords(String localeLanguageId,
                         boolean useDefault);
public void setKeywords(String keywords, Locale locale);
```

As shown in the code above, it adds getting and setting methods for the `Keywords` field with locale features. Now let's add the implementation for the `ExtLayout` model:

1. Locate the `ExtLayoutImpl.java` file from the `com.ext.portlet.layout.model.impl` package in the `/ext/ext-impl/src` folder and open it.
2. Add the following lines before the last `}` in `ExtLayoutImpl.java` file and save it:

```
public String getKeywords(Locale locale) {
    String localeLanguageId = LocaleUtil.toLanguageId(locale);
    return getKeywords(localeLanguageId);
}
public String getKeywords(String localeLanguageId) {
    return LocalizationUtil.getLocalization(getKeywords(),
                                             localeLanguageId);
}
```

```
public String getKeywords(Locale locale, boolean useDefault) {  
    String localeLanguageId = LocaleUtil.toLanguageId(locale);  
    return getKeywords(localeLanguageId, useDefault);  
}  
public String getKeywords(String localeLanguageId, boolean  
    useDefault) {  
    return LocalizationUtil.getLocalization( getKeywords(),  
        localeLanguageId, useDefault);  
}  
public void setKeywords(String keywords, Locale locale) {  
    String localeLanguageId = LocaleUtil.toLanguageId(locale);  
    if (Validator.isNotNull(keywords)) {  
        setKeywords(LocalizationUtil.updateLocalization(  
            getKeywords(), "keywords", keywords, localeLanguageId));  
    }  
    else {  
        setKeywords(LocalizationUtil.removeLocalization(  
            getKeywords(), "keywords", localeLanguageId));  
    }  
}
```

As shown in the code above, it adds implementation for getting and setting methods of the ExtLayout model.

Customizing language properties

Language files have locale-specific definitions. By default, `Language.properties` (at `/portal/portal-impl/src/content`) contains English phrase variations further defined for United States, while `Language_de.properties` (at `/portal/portal-impl/src/content`) contains German phrase variations further defined for Germany. In Ext, `Language-ext.properties` (available at `/ext/ext-impl/src/content`) contains English phrase variations further defined for United States, while `Language-ext_de.properties` (should be available at `/ext/ext-impl/src/content`) contains German phrase variations further defined for Germany.

First, let's add a message in `Language-ext.properties`, by using the following steps:

1. Locate the `Language-ext.properties` file in the `/ext/ext-impl/src/content` folder and open it.
2. Add the following line after the line `view-reports=View Reports for Books` and save it.

```
keywords=Keywords
```

This code specifies the `keywords` message key with a `Keywords` value in English:

Then we need to add German language feature in `Language-ext_de.properties` as follows:

1. Create a language file `Language-ext_de.properties` in the `/ext/ext-impl/src/content` folder and open it.
2. Add the following lines at the beginning and save it:

```
## Portlet names
javax.portlet.title.EXT_1=Berichte
javax.portlet.title.jsp_portlet=JSP Portlet
javax.portlet.title.book_reports=Berichte für das Buch
javax.portlet.title.extLayoutManagement=Ext Seiteneinstellungen
javax.portlet.title.extCommunities=Ext Communities
## Messages
view-reports=Ansicht-Berichte für Bücher
keywords=Schlüsselwörter
## Category titles
category.book=Buch
## Model resources
model.resource.com.ext.portlet.reports.model.ReportsEntry=Buch
## Action names
action.ADD_BOOK=Fügen Sie Buch hinzu
```

As shown in the code above, it specifies the same keys as that of `Language-ext.properties`. But all the keys' values were specified in German instead of English. For example, the message `keywords` has a `Schlüsselwörter` value in German.

In addition, you can set German as the default language and Germany as the default country if it is required. Here are the simple steps to do so:

1. Locate the `system-ext.properties` file in the `/ext/ext-impl/src` folder and open it.
2. Add the following lines at the end of `system-ext.properties` and save it:

```
user.country=DE
user.language=de
```

The code above sets the default locale—the language German (Deutsch) and the country Germany (Deutschland). In general, there are many language files, for example `Language-ext.properties` and `Language-ext_de.properties`, and some language files would overwrite others in runtime loading. For example, `Language-ext_de.properties` will overwrite `Language-ext.properties` when the language is set as German.

These are the three simple rules which indicate the priorities of these language files:

1. The ext versions take precedence over the non-ext versions.
2. The language-specific versions, for example _de, take precedence over the non language-specific versions.
3. The location-specific versions, such as -ext_de, take precedence over the non location-specific versions.

For instance, the following is a ranking from bottom to top for the German language:

1. Language-ext_de.properties
2. Language_de.properties
3. Language-ext.properties
4. Language.properties

Displaying multiple languages

In order to support multiple languages, we have to update the updateLayout method at AddLayoutLocalServiceUtil under the com.ext.portlet.layout.action package of ext/ext-impl/src. To do so, add the following lines before the last }:

```
protected static void setLocalizedAttributes(ExtLayout extLayout,
    Map<Locale, String> localeKeywordsMap) {
    Locale[] locales = LanguageUtil.getAvailableLocales();
    for (Locale locale : locales) {
        String keywords = localeKeywordsMap.get(locale);
        extLayout.setKeywords(keywords, locale);
    }
}
```

As shown in the code above, it added the locale feature method setLocalizedAttributes. Now, update the line String keywords = ParamUtil.getString(actionRequest, "keywords"); with the following lines in the updateLayout method of AddLayoutLocalServiceUtil:

```
Locale[] locales = LanguageUtil.getAvailableLocales();
Map<Locale, String> localeKeywordsMap = new HashMap<Locale,
    String>();
for (Locale locale : locales) {
    String languageId = LocaleUtil.toLanguageId(locale);
    localeKeywordsMap.put( locale,
        ParamUtil.getString(actionRequest, "keywords_" + languageId));
}
```

Further, update the line `extLayout.setKeywords(keywords);` with the line `setLocalizedAttributes(extLayout, localeKeywordsMap);` in the `updateLayout` method of `AddLayoutLocalServiceUtil`.

Finally, we need to update the JSP file `edit_pages_page.jsp` for the localized feature. To do so, first update the lines `<td><liferay-ui:input-field model="<%= ExtLayout.class %>" bean="<% extLayout %>" field="keywords" /></td><td></td>` with the following lines:

```

<td>
    <input id=<portlet:namespace />keywords_<%= defaultLanguageId %>
        name=<portlet:namespace />keywords_<%= defaultLanguageId %>
        size="30" type="text"
        value="<% extLayout.getKeywords(defaultLocale) %>" />
</td>
<td>
    <% for (int i = 0; i < locales.length; i++) {
        if (locales[i].equals(defaultLocale)) {
            continue;
        } %>
    <input id=<portlet:namespace />keywords_<%
        LocaleUtil.toLanguageId(locales[i]) %>
        name=<portlet:namespace />keywords_<%
            LocaleUtil.toLanguageId(locales[i]) %>
        type="hidden" value="<% extLayout.getKeywords(locales[i], false) %>" />
    <% } %>
    <input id=<portlet:namespace />keywords_temp"
        size="30"
        type="text"
        <%= currentLocale.equals(defaultLocale) ? "style='display: none'" : "" %>
        onChange=<portlet:namespace />onKeywordsChanged() />
</td>

```

Then add the following lines after the line `var lastLanguageId = "<%= currentLanguageId %>";` in `edit_pages_page.jsp`:

```

var keywordsChanged = false;
function <portlet:namespace />onKeywordsChanged() {
    keywordsChanged = true;
}

```

As shown in the code above, it adds the locale feature for the `keywords` field. It also updates JavaScript for the `onKeywordsChanged()` method. From now on, the Ext Manage Pages portlet has a locale feature (German, for example) on the customized field, `keywords`.

Moreover, add the following lines after the lines `jQuery("#<portlet:namespace />title_" + lastLanguageId).attr("value", titleValue);
titleChanged = false;` } of the JavaScript method function `<portlet:namespace />updateLanguage()` in the `edit_pages_page.jsp` file:

```
if (keywordsChanged) {  
    var keywordsValue = jQuery("#<portlet:namespace />  
        keywords_temp").attr("value");  
    if (keywordsValue == null) {keywordsValue = "";}  
    jQuery("#<portlet:namespace />keywords_" +  
        lastLanguageId).attr("value", keywordsValue);  
    keywordsChanged = false;  
}
```

Add the line `jQuery("#<portlet:namespace />keywords_temp").show();` after the line `jQuery("#<portlet:namespace />title_temp").show();`. Add the line `jQuery("#<portlet:namespace />title_temp").hide();` after the line `jQuery("#<portlet:namespace />keywords_temp").hide();` of the function `<portlet:namespace />updateLanguage()` method in `edit_pages_page.jsp`:

Last but not the least, we need to update the function `<portlet:namespace />updateLanguageTemps(lang)` method in `edit_pages_page.jsp`. To do so, first add the following lines after the line `var defaultTitleValue = jQuery("#<portlet:namespace />title_<%= defaultLanguageId %>").attr("value");`:

```
var keywordsValue = jQuery("#<portlet:namespace />keywords_" +  
    lang).attr("value");  
var defaultKeywordsValue = jQuery("#<portlet:namespace />keywords_<%=  
    defaultLanguageId %>").attr("value");  
if (defaultKeywordsValue == null) { defaultKeywordsValue = ""; }
```

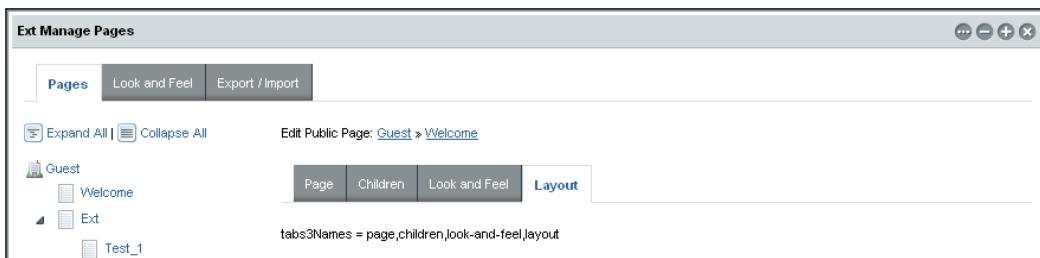
Now, add the following lines after the line `else { jQuery("#<portlet:namespace />title_temp").attr("value", titleValue); }`:

```
if ((keywordsValue == null) || (keywordsValue == "")) {  
    jQuery("#<portlet:namespace />keywords_temp").attr("value",  
        defaultKeywordsValue);  
}  
else {  
    jQuery("#<portlet:namespace />keywords_temp").attr("value",  
        keywordsValue); }
```

As shown in the code above, it specifies variables `keywordsValue` and `defaultKeywordsValue`. It then uses JQuery to update the states of the attribute, for example, `#<portlet:namespace />keywords_temp`.

Employing tabs

As shown in the following screenshot, we expect to add one customized **Layout** tab beside the **Look and Feel** tab. When this tab is selected, it will print the current value, for example, `tabs3Names = page,children,look-and-feel,layout`.



Let's employ the tabs in the **Ext Manage Pages** portlet. First, update the filter of JSP file `edit_pages.jsp` at `/ext/ext-web/docroot/html/portlet/ext/communities`. To do so, add `&& !tabs3.equals("layout")` after `if (tabs2.equals("pages")) && (!tabs3.equals("children")) && !tabs3.equals("look-and-feel")` as follows:

```
if (tabs2.equals("pages")) && (!tabs3.equals("children")) && !tabs3.equals("look-and-feel") && !tabs3.equals("layout")) || ((selLayout != null) && !PortalUtil.isLayoutParentable(selLayout))) {
```

As shown in the code above, it allows the tab value `layout` available for `tabs3` as well as for the values `children` and `look-and-feel`.

Then, add the `Layout` tab and a value in the JSP file `edit_pages_public_and_private.jspf` at `/ext/ext-web/docroot/html/portlet/ext/communities`. To do so, first replace `tabs3Names += ",look-and-feel";` with `tabs3Names += ",look-and-feel,layout";` as follows:

```
if ((selLayout != null) && (permissionChecker.isOmniadmin() || PropsValues.LOOK_AND_FEEL_MODIFIABLE)) {
    tabs3Names += ",look-and-feel,layout";
}
```

Then, add the following lines before the last `</c:choose>` tag:

```
<c:when test='<%= tabs3.equals("layout") %>'>
    tabs3Names = <%= tabs3Names %></c:when>
```

As shown in the code above, it adds a **Layout** tab when the **Look and Feel** tab is added. If the value was `layout`, it prints the current value of `tabs3Names`.

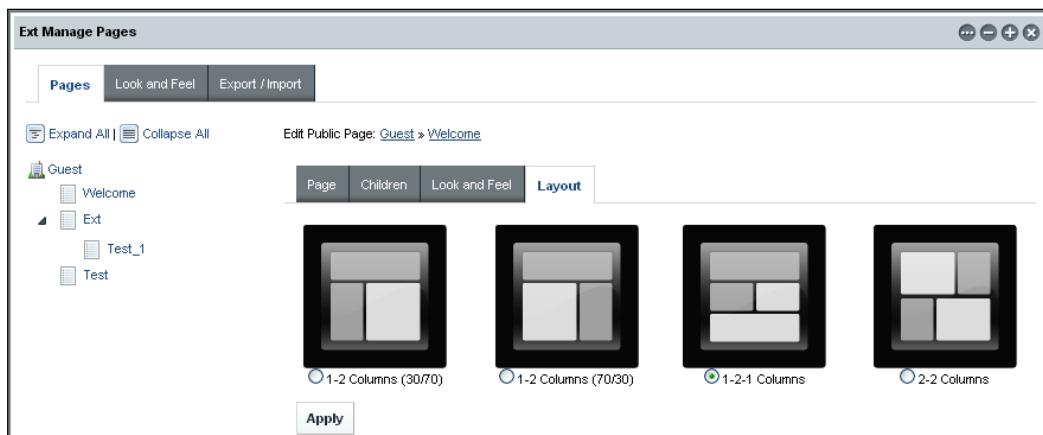
In short, the portal can include custom tabs that allow you to create the same look and feel as that of the out of the box tabs in your own portlets. These tabs would essentially be just other JSP pages, and be displayed based on the selection of custom tabs.

Applying layout templates dynamically

As stated above, we have added the **Layout** tab in the Ext Manage Pages portlet. Now let's apply the layout templates dynamically under this tab. Further, the portal comes with several built-in layout templates. Layout templates are the ways of choosing how your portlets will be arranged on a page. They make up the body of the page—the large area where you can drag and drop your portlets to create your pages.

Why do we need this feature? It is great for developers or CMS administrators to manually create pages using the drag-and-drop feature. But for some content creators and content producers, the drag-and-drop functions would be amazing but difficult to use. Thus they expect the layout templates to be created by the CMS admin only. Then they only view the possible layout templates for pages, apply layout templates dynamically, and populate the content of pages if applicable.

As shown in the following screenshot, when the content creators and content producers are updating the `Welcome` page, they want to view the possible layout templates first—how the page looks. Then they want to apply the layout template, for example 1-2-1 columns. For more details, populate the `book_reports` (Reports for Books) portlet in column-1; populate the `jsp_portlet` (JSP Portlet) portlet in column-2; and populate the Web Content Display (with portlet ID 56) portlet in column-3 and column-4.



In this section, we are going to discuss a generic solution to implement this feature with the above example. By the way, we assume that the **Guest** community is used for the demo of this feature. Of course, you can configure it to any community that you have in the portal.

Setting up pages, layout templates, and portlets mappings

First of all, we need to set up mappings among pages, templates, and portlets. Normally, we can specify these mappings in `portal-ext.properties`. Let's add the following lines at the end of `portal-ext.properties`:

```
## Portlets for layouts
Guest_Welcome_1_2_columns_i=book_reports,jsp_portlet,56
Guest_Welcome_1_2_columns_ii=56,56,56
Guest_Welcome_1_2_1_columns=book_reports,jsp_portlet,56,56
Guest_Welcome_2_2_columns=56,56,56,56
Guest_Ext_1_2_1_columns=56,56,56,56
Guest_Ext_2_2_columns=56,56,book_reports, jsp_portlet
```

As shown in the code above, it specifies the mappings among community, pages, layout templates, and portlets. For the `Welcome` page and the children pages of the `Guest` community, four layout templates are available: `1_2_columns_i`, `1_2_columns_ii`, `1_2_1_columns`, and `2_2_columns`. For example, the `Guest_Welcome_1_2_1_columns` key contains four portlets: `book_reports`, `jsp_portlet`, and two `Web Content Display` portlets.

The layout template `1_2_1_columns` is reused by at least two pages: `Welcome` and `Ext`. Further, the `1_2_1_columns` layout can be applied flexibly for different pages with different portlets associated. Most importantly, you can configure the mappings among the community, pages, layout templates, and portlets. Obviously, you can also use customized layout templates with associated customized portlets.

Adding layout templates

Now we need to add an action in order to add the layout templates for pages. To do so, first add the following lines before the line `String redirect = ParamUtil.getString(actionRequest, "pagesRedirect");` in `ExtEditPagesAction.java` under the `com.ext.communities.action` package:

```
else if (cmd.equals("template")) { addTemplate(actionRequest); }
```

Then add the following method before the last } in `ExtEditPagesAction.java`:

```
protected String addTemplate(ActionRequest actionRequest) throws
Exception {
```

```
String prefixColumn = "column-";
String layoutTemplateId = ParamUtil.getString(actionRequest,
                                                "layoutTemplateId");
String groupSectionName = ParamUtil.getString(actionRequest,
                                                "groupSectionName");
String redirect = "";
ThemeDisplay themeDisplay = (ThemeDisplay) actionRequest.
    getAttribute(WebKeys.THEME_DISPLAY);
long userId = themeDisplay.getUserId();
long groupId = themeDisplay.getScopeGroupId();
boolean privateLayout = ParamUtil.getBoolean(actionRequest, "
    privateLayout");
long layoutId = ParamUtil.getLong(actionRequest, "layoutId");
Layout layout = LayoutLocalServiceUtil.getLayout(groupId,
    privateLayout, layoutId);
LayoutTypePortlet layoutTypePortlet = (LayoutTypePortlet)
    layout.getLayoutType();
layoutTypePortlet.setLayoutTemplateId(userId, layoutTemplateId);
layoutTypePortlet.resetStates();
LayoutServiceUtil.updateLayout(layout.getGroupId(),
    layout.isPrivateLayout(), layout.getLayoutId(),
    layout.getTypeSettings());
redirect = PortalUtil.getLayoutURL(layout, themeDisplay);
String plIds[] = PropsUtil.getArray(groupSectionName +
    layoutTemplateId);
List<String> list = layoutTypePortlet.getPortletIds();
for( int i=0; i < list.size(); i++ ){
    String obj = (String) list.get(i);
    PortletPreferencesLocalServiceUtil.deletePortletPreferences
        (0, 3, layout.getPlid(), obj);
    layoutTypePortlet.removePortletId(userId, obj);
}
if(plIds != null) { layoutTypePortlet.resetStates();
    for(int i=0;i < plIds.length; i++){
        layoutTypePortlet.addPortletId(userId, plIds[i], prefixColumn +
            (i+1), 0);
    }
    LayoutServiceUtil.updateLayout(layout.getGroupId(),
        layout.isPrivateLayout(), layout.getLayoutId(),
        layout.getTypeSettings());
}
return redirect;
}
```

As shown in the code above, it first adds an action-handling command template. Then it gets page, layout template, and associated portlets. Finally, it populates the associated portlets within the selected layout for a given page.

Displaying layout templates by sections

We have added the mappings among community, pages, layout templates, and portlets. We have also added an action to handle the layout template updates. Now let's update JSP files in order to display and apply the layout templates by sections – first-level pages from root, for example, the community name Guest.

First, let's update the JSP file `edit_pages_public_and_private.jspf` in order to include possible layout templates instead of printing a static message. To do so, simply replace the line `tabs3Names = <%= tabs3Names %>` with the following line:

```
<liferay-util:include page="/html/portlet/ext/communities/
    edit_pages_layout_templates.jsp" />
```

Then, let's create a JSP file `edit_pages_layout_templates.jsp` as follows:

1. Create a JSP file `edit_pages_layout_templates.jsp` in the `/ext/ext-web/docroot/html/portlet/ext/communities` folder and open it.
2. Add the following lines at the beginning of `edit_pages_layout_templates.jsp` and save it:

```
<%@ include file="/html/portlet/ext/communities/init.jsp" %>
<% String redirect = ParamUtil.getString(request, "redirect");
   Layout selLayout = (Layout)request.getAttribute("edit_pages.jsp-
   selLayout");
%>
<c:if test="<%=
   themeDisplay.isSignedIn() && (selLayout != null)
   && selLayout.getType().equals(LayoutConstants.TYPE_PORTLET) %>">
   <input name="doAsUserId" type="hidden"
          value="<%=
   themeDisplay.getDoAsUserId() %>" />
   <input name="<%=
   Constants.CMD %>" type="hidden"
          value="template" />
   <input name="<%=
   WebKeys.REFERER %>" type="hidden"
          value="<%=
   HtmlUtil.escape(redirect) %>" />
   <input name="refresh" type="hidden" value="true" />
   <table border="0" cellpadding="0"
          cellspacing="10" style="margin-top: 10px; width=100%">
      <% int CELLS_PER_ROW = 4; List layoutTemplates =
         LayoutTemplateLocalServiceUtil.
         getLayoutTemplates(theme.getThemeId());
         layoutTemplates = PluginUtil.restrictPlugins
             (layoutTemplates, user);
         LayoutTypePortlet selLayoutTypePortlet = (LayoutTypePortlet)
             selLayout.getLayoutType();
         Group group = selLayout.getGroup();
         String groupSectionName = group.getName() + "_" +
             selLayout.getName(locale)+"_" ;
```

```
if(selLayout.getParentLayoutId() != LayoutConstants.  
    DEFAULT_PARENT_LAYOUT_ID) {  
    List parents = selLayout.getAncestors();  
    String parentName = ((Layout)parents.get(parents.size() -  
        1)).getName(locale);  
    groupSectionName = group.getName() + "_" + parentName + "_";  
}  
%>  
<input name="groupSectionName" type="hidden"  
       value="<% groupSectionName %>" />  
<%  
    int index = 0;  
    for (int i = 0; i < layoutTemplates.size(); i++) {  
        LayoutTemplate layoutTemplate = (LayoutTemplate)  
            layoutTemplates.get(i);  
        String key = PropsUtil.get(groupSectionName +  
            layoutTemplate.getLayoutTemplateId());  
        if(key != null){  
%>  
<c:if test="<%=(index % CELLS_PER_ROW) == 0 %>">  
<tr></c:if>  
<td align="center" width="<%=(100 / CELLS_PER_ROW)%>">  
    <img onclick="document.getElementById('layoutTemplateId  
        <%= i %>').checked = true;"  
        src="<% layoutTemplate.getContextPath() %>"  
        <%= layoutTemplate.getThumbnailPath() %>" /><br />  
    <input <%= selLayoutTypePortlet.getLayoutTemplateId().  
        equals(layoutTemplate.getLayoutTemplateId()) ? "checked" :  
        "" %> id="layoutTemplateId<%= i %>"  
        name="layoutTemplateId" type="radio"  
        value="<% layoutTemplate.getLayoutTemplateId() %>" />  
    <label for="layoutTemplateId<%= i %>">  
        <%= layoutTemplate.getName() %>  
    </label>  
</td>  
<c:if test="<%=(index % CELLS_PER_ROW) == (CELLS_PER_ROW - 1)  
    %>"> </tr></c:if> <% index++; } %>  
</table>  
<input class="form-button" type="submit"  
      value="      style="margin: 10px" />  
</c:if>
```

As shown in the code above, it uses the first-level navigations as sections.
The children pages of a section will share the same group of layout templates.

In short, a layout is an instance of a single page, which is composed of one or more portlets arranged inside various columns. Layout templates define the ways in which portlets will be arranged on a page. You, as a developer, can group the layout templates flexibly and, further, dynamically apply the layout templates for layout pages.

Tracking pages

We have built a lot of pages as well. Suppose we want to track all pages using **Google Analytics (GA)**, for example, in order to show how to extend Liferay portal's out of the box tracking ability. Configuring Liferay portal to work with Google Analytics is pretty simple. The following are the main steps to set up Google Analytics:



Google Analytics generates detailed statistics about the visitors to a web site. Refer to <http://www.google.com/analytics>.



First, get an account from Google Analytics—UA-5808951-1, for example. You may have your own account number. If you want to use a different one, simply replace the account number.

Then, let's customize the `bottom-ext.jsp` file in the `/ext/ext-web` folder as follows:

1. Create a `/common/themes` folder page in the `/ext/ext-web/docroot/html` folder.
2. Create a JSP file `bottom-ext.jsp` in the `/ext/ext-web/docroot/html/common/themes` folder and open it.
3. Add the following lines at the end of this file and save it:

```
<script type="text/javascript">
    var gaJsHost = (("https:" == document.location.protocol) ?
                    "https://ssl." : "http://www.");
    document.write(unescape("%3Cscript src='" + gaJsHost + "google-
        analytics.com/ga.js' type='text/javascript'%3E%3C/script%3E"));
</script>
<script type="text/javascript">
    var pageTracker = _gat._getTracker("UA-5808951-1");
    pageTracker._trackPageview();
</script>
```

As shown in the code above, it uses a single Google Analytics account UA-5808951-1 to track the entire portal.

Using communities and layout page efficiently

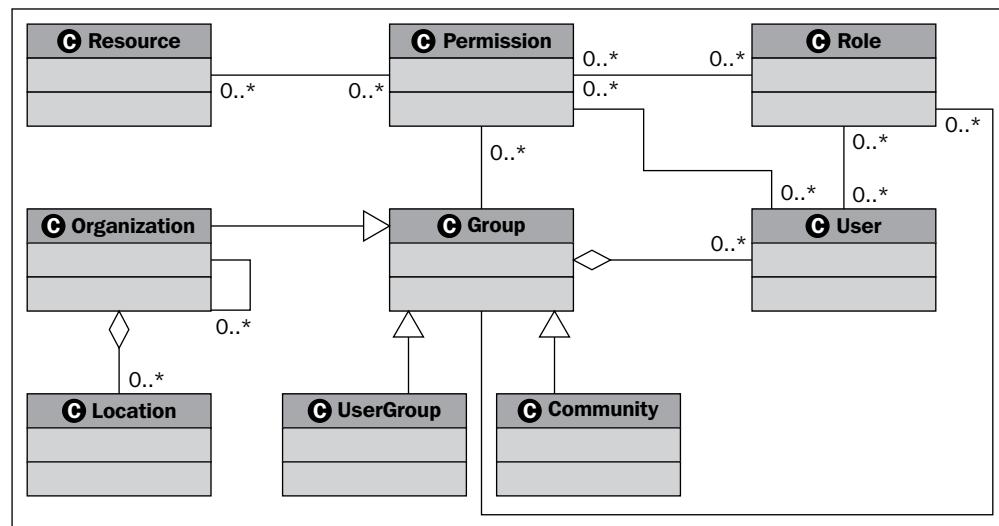
We have discussed how to customize and extend the community and layout pages. Each community has a set of pages, either public or private. You can assign users, roles, permissions, and user groups for a given community. Here we will discuss how to use the community and layout pages efficiently.

Liferay portal extends the security model by resources, users, organizations, locations, user groups, and communities, roles, permissions, and so on. Liferay portal provides a fine-grained permission security model, which is a full-access control security model. At the same time, Liferay also provides a set of administrative tools that we have discussed above to configure and control the membership.

Employing group, community, and permissions

As shown in the following figure, a resource is a base object and permission is an action on a resource. A role is a collection of permissions. A user is an individual. Depending on what permissions and roles have been assigned, the user either does or does not have the permission to perform certain tasks.

Organizations represent the enterprise and departments hierarchy. Organizations can contain other organizations. Moreover, an organization acting as a child organization of a top-level organization can also represent departments of a parent corporation:



A *Location* is a special organization with one, and only one, associated parent organization and without an associated child organization. Organizations can have any number of locations and suborganizations. Both roles and individual permissions can be assigned to organizations (locations or suborganizations). By default, locations and suborganizations inherit permissions from their parent organization.

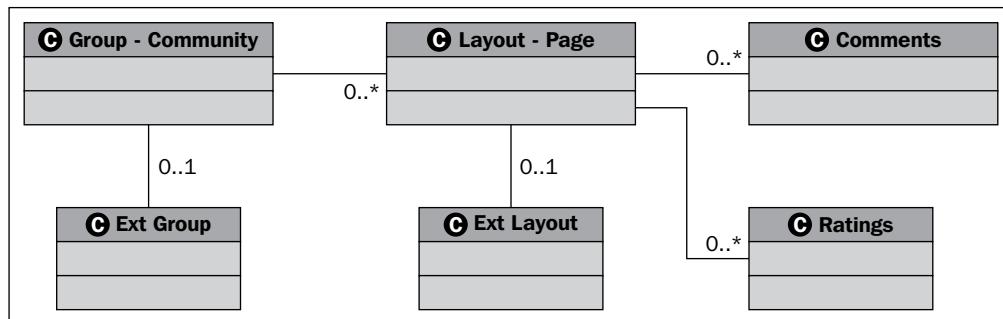
A community is a special group. It may hold a number of users who share common interests. Both roles and individual permissions can be assigned to communities.

Finally, a user group is a special group with no context, which may hold a number of users. Both roles and individual permissions can be assigned to user groups, and every user who belongs to that user group will receive the role or permission.

Using communities, layout pages, comments, and ratings

As shown in the following figure, a community is made up of a set of pages, including private pages and public pages. You can use a community (for example Book Street) to represent a web site (for example www.bookpubstreet.com), where the public pages represent the public site for public end users in the frontend, and the private pages are employed to present web content management for content creators, content publishers, and content editors in the backend.

A layout page consists of a couple of portlets, including the Journal Content portlet. More interestingly, a layout page may have page comments and associated page ratings. You can specify the comments and ratings in the journal articles.



Extending the community and layout pages

We have customized and extended the communities and layout pages in the Ext. At the same time, we did not add columns to Liferay Group_ table and Layout table. As shown in the preceding figure, we have just created additional tables (for example ExtGroup and ExtLayout) with a 1:1 relationship with the Group_ table and Layout table, respectively. With this, we will prevent the upgrade risk properly and make the extension reusable.

Users can belong to any number of communities and inherit permissions and roles from them. Notice that in the Group_ table, there is class-Name and class-PK column. If these columns are blank, then the record is a normal community. If class-Name is com.liferay.portal.model.User, then the record represents a private user community. This is used only for Power Users. If the class-Name is com.liferay.portal.model.Organization, then the record represents an organization or location. If the class-Name is com.liferay.portal.model.UserGroup, then the record represents a user group. For instance, if both class-Name and class-PK columns are blank in Group_, it means that this is a normal community, for example Book Street.

Summary

This chapter first discussed how to customize and extend the Communities portlet. Then it introduced how to customize the Manage Pages portlet. Accordingly, it also introduced how to customize page management with more features such as adding localized features, adding tags, employing layout templates dynamically, tracking pages, and so on. Finally, it addressed how to use the Communities and Layout Pages efficiently.

In the next chapter, we're going to customize the WYSIWYG editor.

6

Customizing the WYSIWYG Editor

The WYSIWYG editor will be helpful in building content on top of Liferay portal, for example, blog entries, forum topics, articles, journal articles, and so on. Within the portal, content involves not only pure HTML text, but also images and links. The images can be managed in Image Gallery, and documents in Document Library. More interestingly, content can be stored and managed in a third-party system and used as web content via a WYSIWYG editor. The content may involve images, links, videos, games, and so on. In order to build content-rich web content, we should be able to enter the content in a WYSIWYG editor, as well as pure HTML text.

This chapter begins by introducing how to configure the WYSIWYG editor, how to quickly deploy the updates, and how to upgrade the WYSIWYG editor. Then it addresses how to change the templates and formats based on different web sites such as bookpubstreet.com or bookpubworkshop.com. Moreover, it introduces how to customize the WYSIWYG editor to include images, links, videos, games, and so on. Finally, it shows how to use the WYSIWYG editor efficiently.

By the end of this chapter, you will have learned how to:

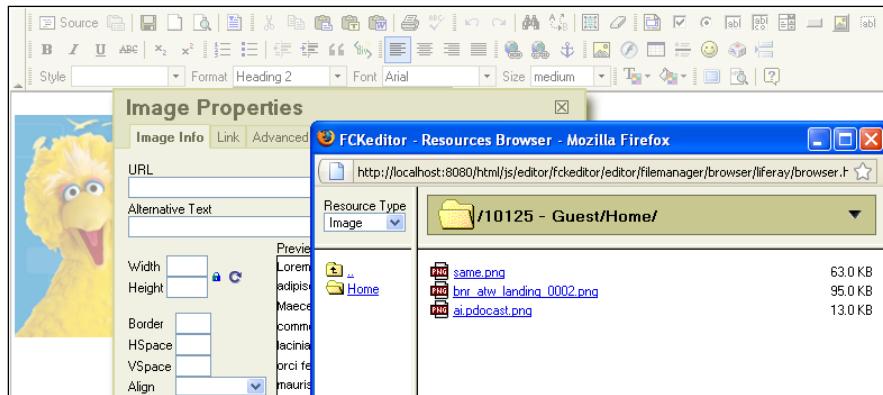
- Configure the WYSIWYG editor
- Add templates and styles in FCKeditor
- Insert images and links from different services
- Insert content-rich flashes into web content
- Use the WYSIWYG editor FCKeditor efficiently

Configuring the WYSIWYG editor

Liferay portal is integrated with the WYSIWYG HTML text editors. Thus, content creation and publishing in the portal is simple and straightforward. The WYSIWYG text editor has spell check and text styling, which enables anyone to create content without knowing how to do the coding. Those who know how to code can edit in the same editor, in the source code mode.

As shown in the following screenshot, we plan to use the latest version of **FCKeditor**. At the same time, you can keep the same functions when changing versions, for example inserting images from Image Gallery, and inserting links to images, pages, and documents from Document Library.

[ **FCKeditor** is an HTML text editor. It brings to the Web much of the power of desktop editors such as MS Word. Refer to <http://www.FCKeditor.net>.]



In this section, we're first going to discuss how to extend the Ant target in order to support `deploy-fast` for HTML file updates. Next, we will discuss how to upgrade the text editor, that is, FCKeditor. Finally, we'll show how to configure FCKeditor.

Extending the Ant target for fast deployment

FCKeditor is made up of JavaScript files and HTML files. That is, FCKeditor is implemented mainly by JavaScript as services, and HTML as frontend UI. Thus, when we customize the FCKeditor, it would be better to use the Ant target `deploy-fast`. This Ant target is good as it adds the updated files to Tomcat when it is still running. By using this, we can save a lot of verification time and test the changes immediately – there is no need to shut Tomcat down, deploy the changes, and restart Tomcat repeatedly.

By default, the portal does not support fast deployment of HTML files. That is, the Ant target `deploy-fast` does not include HTML files. It supports XML, VM, Properties, PNG, JSP, JSPF, CSS, DTD, and so on. Obviously, the HTML files are not included in the Ant target `deploy-fast`. You can find the `deploy-fast` at `/ext/ext-web/build-parent.xml`.

Fortunately, we are able to update the Ant target `deploy-fast` in order to include the HTML files `.html` and `.htm`. To do so, simply add the following lines after the line `<include name="**/*.xml" />` in the `/ext/ext-web/build-parent.xml` file:

```
<include name="**/*.htm" />
<include name="**/*.html" />
```

As shown in the code above, we add two types of files—`.html` and `.htm`—to `deploy-fast`. From now on, you can use `deploy-fast` to immediately deploy the files either with the `.html` extension or with the `.htm` extension under the `/ext/ext-web/docroot` folder to Tomcat. Of course, you can add different files for the Ant target `deploy-fast`, for example, the extended HTML files—XHTML.

Upgrading the WYSIWYG editor: FCKeditor

We have modified `deploy-fast` to support fast deployment of the HTML files. Now let's upgrade the WYSIWYG Text Editor—FCKeditor—to the latest version. By default, Liferay portal has bundled FCKeditor with a specific version—it may not be the latest version. Let's upgrade the FCKeditor to the latest version as follows:

1. Download the latest version of FCKeditor from <http://www.FCKeditor.net>.
2. Create a folder called `editor` under the `/ext/ext-web/docroot/html/js` folder.
3. Unzip the ZIP file to the folder `/ext/ext-web/docroot/html/js/editor`.

Then, let's add a customized configuration to the JSP file `fckconfig.jsp`. To do so, copy the JSP file `fckconfig.jsp` from the folder `/portal/portal-web/docroot/html/js/editor/_FCKeditor/` to the folder `/ext/ext-web/docroot/html/js/editor/FCKeditor/`.

What's happening? The `fckconfig.jsp` file specifies the toolbar sets `liferay-article`, `liferay`, `edit-in-place`, and `email`. By the way, the default toolbar sets `Default` and `Basic` in the `/fckconfig.js` folder besides the `fckconfig.jsp` file. Then, it specifies `FCKConfig.LinkBrowserURL` and `FCKConfig.ImageBrowseURL`. Thus, you can browse the links to images and documents via FCKeditor from Image Gallery and Document Library, respectively. Note that `FCKConfig.FlashBrowser` is set with a `false` value, that is, the ability to browse flashes is disabled by default.

Finally, we need to generate the portal browser folder based in the `/ext/ext-web/docroot/html/js/editor/FCKeditor/editor/filemanager/browser/default` folder in the following manner:

1. Locate the folder `/ext/ext-web/docroot/html/js/editor/FCKeditor/editor/filemanager/browser`.
2. Create a folder `/liferay` and copy all of the files in `/default` to `/liferay`.

To see the above changes, you can click on the `deploy-fast` target of `ext-web` of the **Ant** view. In addition, add the following lines after the lines `if (oConnector.ShowAllTypes) oConnector.ResourceType = 'File' ;` in `browser.html` file under the folder `/ext/ext-web/docroot/html/js/editor/FCKeditor/editor/filemanager/browser/liferay`:

```
if ( oConnector.ShowAllTypes )
oConnector.ResourceType = 'Document' ;
```

The code above resets `oConnector.ResourceType` with a `Document` value.

Setting up the FCKeditor

As mentioned earlier, we have updated the WYSIWYG HTML editor, FCKeditor, successfully. Now we're going to set up FCKeditor so that it works with the portal.

As shown in the following screenshot, we expect to customize the toolbar set `liferay-article` (for example, for Web Content articles) with a set of icons (for example, font name and font size), and to remove a lot of icons from the default settings (for example, spell check, print, rule, about, and so on). Moreover, we will add a normal toolbar set `liferay` (for example, for Blog entries) with the same icons as that of `liferay-article` and a brief toolbar set `edit-in-place` (for example, for a simple editor in portlets) with a small set of icons. At the same time, we plan to add a customized plugin called `Liferay page break` at the end of the toolbar set.



Adding customized icons

In General, you can add any kind of icons provided in the `Default` toolbar set from the `fckconfig.js` file. For instance, FCKeditor provides two sample toolbar sets in the `fckconfig.js` file by default—`Default` and `Basic`. You can use either `Default` or `Basic` toolbar sets, or a part of icons in `Default` toolbar set.

It's quite easy to customize the toolbar buttons according to your needs. Just edit the configuration file `FCKeditor.jsp` and modify or add new items to the `FCKConfig.ToolbarSets` configuration entry, or create this entry in the `FCKeditor.jsp` file. Let's update the content of this file as follows:

1. Locate the JSP file `FCKeditor.jsp` in the folder `/ext/ext-web/docroot/html/js/editor/FCKeditor` and open it.
2. Locate `FCKConfig.ToolbarSets["liferay-article"]` in this file, update it, and save it:

```
FCKConfig.ToolbarSets["liferay-article"] = [
    ['Style','FontFormat','FontName','FontSize'],
    '/',
    ['TextColor','BGColor'],
    ['Bold','Italic','Underline','StrikeThrough'],
    ['Subscript','Superscript'],
    '/',
    ['Undo','Redo','-', 'Cut','Copy','Paste','PasteText',
        'PasteWord','-', 'SelectAll','RemoveFormat'],
    ['Find','Replace','SpellCheck'],
    ['OrderedList','UnorderedList','-', 'Outdent','Indent'],
    ['JustifyLeft','JustifyCenter','JustifyRight','JustifyFull'],
    '/',
    ['Source'],
    ['Link','Unlink','Anchor'],
    ['Image','Flash','Table','-', 'Smiley','SpecialChar',
        'LiferayPageBreak']] ;
```

The code above specifies the toolbar set `liferay-article`. It has a set of icons, for example, font name and font size. In addition, you will see the `LiferayPageBreak` icon. Double-click on **deploy-fast** and refresh your journal article editing view. You will get the message: **Unknown toolbar item LiferayPageBreak**. This is because the `LiferayPageBreak` plugin is not ready yet. Let's add this plugin in FCKeditor.

Employing default configuration

You can configure individual JSP pages to use a specific implementation of available WYSIWYG editors: FCKeditor or **TinyMCE**.


TinyMCE is a platform-independent web-based JavaScript HTML WYSIWYG editor. For more details, refer to <http://tinymce.moxiecode.com>.

For example, you can find the default configuration of the portal with FCKeditor in /portal/portal-impl/src/portal.properties.

```
editor.wysiwyg.default=FCKeditor
editor.wysiwyg.portal-web.docroot.html.portlet.blogs.edit_entry.
jsp=FCKeditor
editor.wysiwyg.portal-web.docroot.html.portlet.journal.edit_article_
content.jsp=FCKeditor
editor.wysiwyg.portal-web.docroot.html.portlet.message_boards.edit_
configuration.jsp=FCKeditor
editor.wysiwyg.portal-web.docroot.html.portlet.wiki.edit.html.
jsp=FCKeditor
```

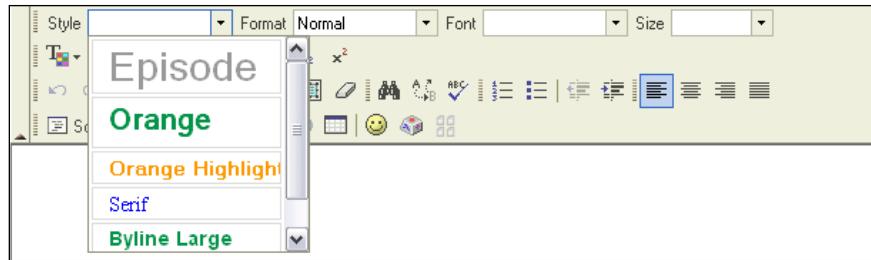
Suppose that you're going to use TinyMCE as the default WYSIWYG editor. Then you can configure it in the portal-ext.properties file in the folder ext/ext-impl/src as follows:

```
editor.wysiwyg.default=tinymce
editor.wysiwyg.portal-web.docroot.html.portlet.blogs.edit_entry.
jsp=tinymce
editor.wysiwyg.portal-web.docroot.html.portlet.journal.edit_article_
content.jsp=tinymce
editor.wysiwyg.portal-web.docroot.html.portlet.message_boards.edit_
configuration.jsp=tinymce
editor.wysiwyg.portal-web.docroot.html.portlet.wiki.edit.html.
jsp=tinymce
```

Adding templates and styles in FCKeditor

FCKeditor offers complete and powerful support for separating text formatting definitions from the text. It also offers a complete set of predefined formatting definitions to the end user (for example, content creator, content producer, and so on.) so that the text can be well designed without messing up the HTML source.

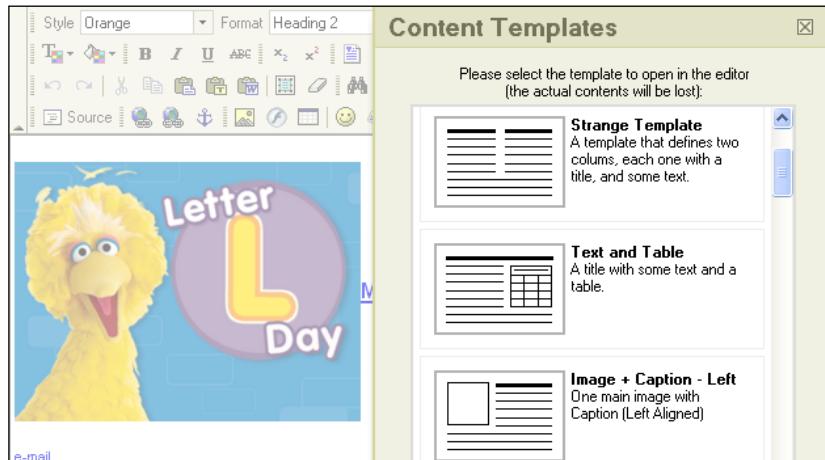
As shown in the following screenshot, the **Style** toolbar command would be useful for styles that show a complete list of available styles with a preview for text styles. When in the text editing mode, if you click on the **Styles** toolbar icon, it will show the customized CSS styles. In short, we're going to use the customized CSS styles from the current theme. Further, if you click on the **Styles** toolbar icon when editing an image, it shows the customized styles **Image on Left** and **Image on Right**.



Similarly, the **Format** toolbar command would be useful for formats that show a complete list of available formats with a preview for text styles. When you click on the **Format** toolbar icon, it shows customized formats for CSS styles. We're going to use the customized format CSS styles from the current theme.

In addition, we're going to use dynamic CSS styles, too. As mentioned earlier, there are two web sites: bookpubstreet.com and bookpubworkshop.com. Each web site has its own CSS style in a theme, for example `book-street-theme` and `book-workshop-theme`. If these two web sites shared the same Liferay portal instance, they would share the same editor, but use different themes. The portal will dynamically identify which web site is using the editor and then display that web site's CSS styles.

Moreover, you can select a template from a list by clicking on the **Templates** icon in the toolbar. As shown in the following screenshot, when clicking on the **Templates** icon, it shows customized templates, for example **Image + Caption - Left**. So, we're going to configure and customize the FCKeditor to use customized templates:



We have mentioned our requirements based on styles and templates. Now let's implement the above requirements.

Constructing styles and formats

Styles combine all of the formatting functions for a given web site. They give us fast access to the most commonly used text formats. For instance, you don't have to change the font, its color, its background, or its size. Simply pick a style you prefer from the **Style** menu and start typing.

Preparing CSS styles in themes

First of all, let's prepare CSS styles in the Book Street and Book Workshop themes. Here we will use Book Street as an example:

1. Create a CSS style file `global_style_headings.css` in the folder `book-street-theme/docroot/_diffs/css` and open it.
2. Add the following lines at the beginning of this file and save it:

```
H1.flush,H2.flush,H3.flush,H4.flush,H5.flush{ margin-top: 0; }
H1{font-size: 28px;
   font-weight: bold;
   color: #ff6600;
   margin: 0px
   0px 10px 0px;
}
H2{font-size: 20px;
   font-weight: bold;
   color: #029449;
   margin-bottom: 5px;
   padding:0px 0px 5px 0px;
}
.episode-guide{font-size: 28px;
   color: #999;
   font-weight: normal;
   margin-top: 0px;
}
.orange {font-weight: bold;
   font-size: 20px;
   color: #029449;
}
.orange-highlight-text{font-weight: bold;
   font-size: 14px;
   color: #ff9900;
}
.serif { font-family: serif;
   font-size: 14px;
   color: blue;
   font-weight: normal;
```

```

}
/byline-large { font-family: arial;
    font-size: 14px;
    color: #029449;
    font-weight: bold;
}
/byline { font-family: arial;
    font-size: 12px;
    color: #009933;
    font-weight: normal;
}

```

The code above first specifies the CSS styles for the formats H1, H2, and so on. Then it specifies the CSS styles for episode-guide, orange, byline, and so on.

Accordingly, import this CSS styles file in main.css from the folder book-street-theme/docroot/_diffs/css. To do so, add the following line at the end of the main.css file:

```
@import url(global_style_headings.css);
```

Similarly, we can specify CSS styles files for the Book Workshop theme and the other themes.

Employing customized CSS styles from themes

Then, let's set the Editor-Area-CSS property in the configuration file, for example using CSS styles defined inside the main.css file of the theme as follows:

1. Locate the configuration file fckconfig.jsp in the folder /ext/ext-web/docroot/html/js/editor/FCKeditor and open it.
2. Update the line FCKConfig.EditorAreaCSS = '<%= HtmlUtil.escape(cssPath) %>/main.css'; with the following lines in the fckconfig.jsp file and save it.


```
FCKConfig.TemplatesXmlPath = FCKConfig.EditorPath +
        'fcktemplates.xml';
FCKConfig.EditorAreaCSS = '<%= cssPath %>/main.css' ;
```

The code above specifies the Editor-Area-CSS property with the main.css value of the theme. For instance, the value for the Book Street theme would be /book-street-theme/css/main.css, whereas the value for the Book Workshop theme would be /book-workshop-theme/css/main.css. Note that the style XML path value is specified as fckstyles.xml. At the same time, the style XML path value is specified as fcktemplates.xml. Let's customize them.

Customizing styles

The list of available styles based on an XML file can be customized completely for the current or future needs.

1. Locate the configuration file `fckstyles.xml` in the folder `/ext/ext-web/docroot/html/js/editor/FCKeditor` and open it.
2. Remove the existing content from this file, add the following lines in it, and save it:

```
<?xml version="1.0" encoding="utf-8" ?>
<Styles>
    <Style name="Image on Left" element="img">
        <Attribute name="style"
            value="padding: 5px; margin-right: 5px" />
        <Attribute name="border" value="2" />
        <Attribute name="align" value="left" />
    </Style>
    <Style name="Image on Right" element="img">
        <Attribute name="style"
            value="padding: 5px; margin-left: 5px" />
        <Attribute name="border" value="2" />
        <Attribute name="align" value="right" />
    </Style>
    <Style name="Episode" element="h1">
        <Attribute name="class" value="episode-guide"/>
    </Style>
    <Style name="Orange" element="h2">
        <Attribute name="class" value="orange"/>
    </Style>
</Styles>
```

The code above shows how to define four styles: one for an `Object` element (in this case, for images) and three for text. The editor will show the styles in a context-sensitive fashion. So when an image is selected, only the `Image on Left` style will be available in the list; and when the text is selected, `Episode` and `Orange` will be shown.

In addition, the `style` nodes have two mandatory attributes. The `name` attribute defines the text shown in the styles list, whereas the `element` is used to apply style on the text selection or the object element to which the style can be applied. The object elements supported by the editor are HTML tags: `img`, `table`, `tr`, `td`, `input`, `select`, `textarea`, `hr`, and `object`.

In general, it is quite easy to make the text look ugly or difficult to read by applying a lot of different formatting to it. Styles help us to make all formatting in the text uniform. Each style option contains a predefined combination of formatting features.

Building templates

A template is a piece of HTML placed inside FCKeditor. It is quite easy to configure and customize FCKeditor in order to use customized templates. Let's update the template file `fcktemplates.xml` as follows:

1. Locate the `fcktemplates.xml` file in the `/ext/ext-web/docroot/html/js/editor/FCKeditor` folder and open it.
2. Add the following lines before the line `</Templates>` and save the file:

```
<Template title="Image + Caption - Left" image="image_left.gif">
    <Description>
        One main image with Caption (Left Aligned)
    </Description>
    <Html>
        <! [CDATA[ <table class="image-box align-left">
            <tr>
                <td class="image">
                    <img src="" width="264" height="198" alt="" />
                </td>
            </tr>
            <tr>
                <td class="image-caption">
                    Caption (100 ch max)
                </td>
            </tr>
        </table>]]>
    </Html>
</Template>
```

As shown in the code above, a root `Templates` element may have many `Template` elements inside it. It specifies the images base path as `fck_template/images/`. `imagesBasePath` sets the base path used to build the full path of the preview images. `Template` defines a single template, `title` defines the title to show in the templates list, and `Image` defines the name of the image file to show as the preview of the template. `Description` defines the textual description of the template. `Html` defines the HTML to be set in the editor when the user selects a template. The HTML must be placed inside a `CDATA` section (**Unparsed Character Data**).

We have used the `image_left.gif` image as the preview of the `Image + Caption - Left` template. So where should we put this image file? Simply copy the `image_left.gif` image to `/ext/ext-web/docroot/html/js/editor/FCKeditor/editor/dialog/fck_template/images/`. It will be concatenated with the `imagesBasePath` attribute value of the `Templates` root element if set. If the `image` attribute is not set, no preview of images will be shown. By the way, there is no fixed size for the image file.

More interestingly, you can create a new separate template file (for example, an XML file) outside the editor's directory and configure the editor to use it. Suppose that you have your own template file `templates.xml`, just set the following configuration in the `fckconfig.jsp` file:

```
FCKConfig.TemplatesXmlPath = '/templates.xml' ;
```

Generally, when working with a template, you don't need to start writing it from scratch. However, a designer (for example, a content administrator) can prepare well-designed templates and avoid errors before they happen.

Inserting images and links from different services

In the previous section, we upgraded FCKeditor successfully. At the same time, we added the `Insert/Edit Image`, `Insert/Edit Link`, and `Insert/Edit Flash` icons in the toolbar set. Now let's take a deeper look at how to insert images and links from different services. To do so, we need to develop server-side integration for FCKeditor.

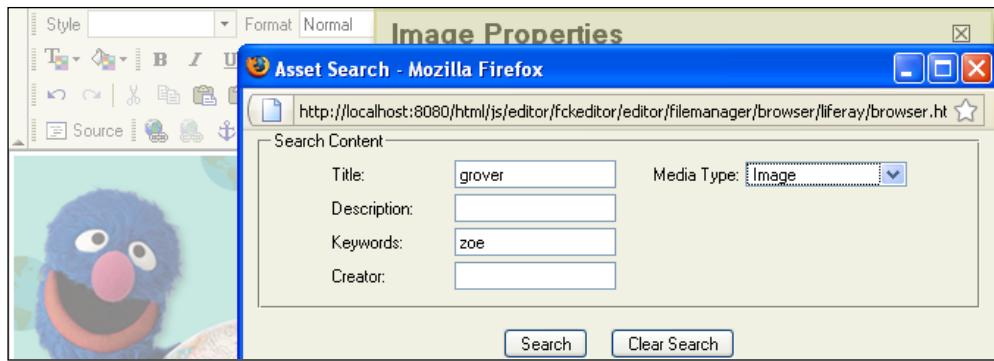
By default, Liferay portal services are provided for two purposes:

1. Browse images by folders and insert an image.
2. Browse images, documents, and pages by folders, and insert them as links.

For the `Insert/Edit Image` icon, one resource type is supported: `Image`. However, for the `Insert/Edit Link` icon, three resource types are supported: `Image`, `Document`, and `Page`.

It would be a good idea to find the contents, images, and documents via the above browsing function when the volume of content is small (that is, if the content amount is less than 1 gigabit, referring to Book Street's requirements). But a huge volume of content (if the content amount is greater than 1 gigabit) should be found using search (either basic search or advanced search). Let us suppose that we have a huge volume of content, and we expect to insert images and links from third-party services. As shown in the following screenshot, the image and link are coming from third-party services:

- Search images by multiple fields – **Title**, **Description**, **Keywords**, and **Creator** – and insert the image.
- Search contents such as TEXT and COMBO by multiple fields – **Title**, **Description**, **Keywords**, and **Creator** – and insert them as links.



For the Insert/Edit Image icon, one resource type is supported: **Image**. But for the Insert/Edit Link icon, two resource types are supported: TEXT (simple format documents such as XML files and text files) and COMBO (complex format files such as DOC, PDF, and ZIP). In addition, Images, TEXT, and COMBO are managed in a third-party repository (for example, Alfresco).

Configuring a File Browser Connector with Liferay portal services

FCKeditor provides an ability to integrate third-party services. Its **File Browser Connector** offers a unique interface that can be used by all server-side languages that are developed completely on JavaScript DHTML, and the integration is available by XML. Let's have a deeper look on FCKeditor's File Browser Connector with Liferay portal services.

Configuring the services for images, documents, and pages

First of all, you can find the services for images, documents, and pages in the fckconfig.jsp file as follows:

```
long plid = ParamUtil.getLong(request, "p_l_id");
String mainPath = ParamUtil.getString(request, "p_main_path");
String doAsUserId = ParamUtil.getString(request, "doAsUserId");
String cssPath = ParamUtil.getString(request, "cssPath");
String cssClasses = ParamUtil.getString(request, "cssClasses");
String connectorURL = HttpUtil.encodeURL(mainPath
    + "/portal/FCKeditor?p_l_id="
    + plid + "&doAsUserId=" +
    HttpUtil.encodeURL(doAsUserId));
response.setContentType(ContentTypes.TEXT_JAVASCRIPT);
```

The code above specifies the connector URL with specific parameters such as `mainPath`, `p_p_id`, and `doAsuserid`. You can also find the specification of the browser URLs in the `fckconfig.jsp` file as follows:

```
FCKConfig.LinkBrowserURL = FCKConfig.BasePath  
+ "filemanager/browser/liferay/browser  
.html?Connector=<%= connectorURL %>";  
FCKConfig.ImageBrowserURL = FCKConfig.BasePath  
+ "filemanager/browser/liferay/browser  
.html?Type=Image&Connector=  
<%= connectorURL %>";  
FCKConfig.FlashBrowser = false ;
```

The code above configures `LinkBrowserURL` with the `Link` type and the connector URL, and builds `ImageBrowserURL` with the `Image` type and the connector URL.

Generally speaking, all requests are simply made by the File Browser via the normal HTTP channel. The request information is always passed by the query string in the URL.

Browsing images and links

We need to update the browsing feature for images and links. As mentioned earlier, there are three subtypes (for example, `Document`, `Image`, and `Page`) for resource type `Link`; and only one subtype (for example, `Image`) for resource type `Image`.

Let's update the DHTML file `frmresourcetype.html` for this request.

Moreover, add the following lines before the line `window.onload = function()` in the HTML file `frmresourcetype.html` under the folder `/ext/ext-web/docroot/html/js/editor/FCKeditor/editor/filemanager/browser/liferay`.

```
aTypes = [['Document', 'Document'],  
          ['Image', 'Image'], ['Page', 'Page']] ;
```

The code above shows that `aTypes` has three types: `Document`, `Image`, and `Page`.

At the same time, we need to set up the default value to the resource type `Link`, as it has three subtypes: `Document`, `Image`, and `Page`. Let's update the DHTML file `browser.html` for this request.

1. Locate the `browser.html` file in the folder `/ext/ext-web/docroot/html/js/editor/FCKeditor/editor/filemanager/browser/liferay` and open it.
2. Add the following line before the line `) oConnector.ResourceType = 'Document' ;` in this file and save it:
`|| oConnector.ResourceType == 'Link'`

This code shows the `Type` parameter and then specifies the default value for the `Link` type, for example `Document`.

When you are ready, simply click on the Ant target `deploy-fast`. Then you can view the updates immediately.

Preparing Liferay portal services

The `Connector` is the main file needed to integrate the server side with the File Browser. By default, Liferay provides services in use for the FCKeditor Connector. These services are focusing on the types `Image`, `Document`, and `Page` as follows:

```
// for resource type: Images  
/c/portal/FCKeditor?p_1_id=10302&doAsUserId=d2kh%2BNVWHCg%3D&Command=G  
etFoldersAndFiles&Type=Image&CurrentFolder=%2F  
  
// for resource type: Links  
// for a link of Document  
/c/portal/FCKeditor?p_1_id=10302&doAsUserId=d2kh%2BNVWHCg%3D&Command=G  
etFoldersAndFiles&Type=Document&CurrentFolder=%2F  
// for a link of Image  
/c/portal/FCKeditor?p_1_id=10302&doAsUserId=d2kh%2BNVWHCg%3D&Command=G  
etFoldersAndFiles&Type=Image&CurrentFolder=%2F  
// for a link of Page  
/c/portal/FCKeditor?p_1_id=10302&doAsUserId=d2kh%2BNVWHCg%3D&Command=G  
etFoldersAndFiles&Type=Page&CurrentFolder=%2F
```

The code above shows the service path as `/c/portal/FCKeditor`. It further specifies a set of parameters such as `p_1_id` (the current page layout ID), `doAsUserId` (the default user ID), `command` (such as `GetFoldersAndFiles`), `type` (for example, possible values `Page`, `Document`, and `Image`), and `current folder` (such as `/`).

In general, the request info is always passed by the query string in the URL using the following format:

```
/c/portal/FCKeditor?p_1_id=plid&doAsUserId=doAsUserId &Command=Command  
Name&Type=ResourceType&CurrentFolder=FolderPath
```

Where `CommandName` is the command that the Connector must execute (such as `GetFoldersAndFiles`), the `ResourceType` is either `Link` or `Image`. `FolderPath` represents the path of the current folder visible in File Browser.

All Connector responses have the same base XML structure like this:

```
<Connector command="GetFoldersAndFiles" resourceType="Page">  
  <CurrentFolder path="/" url="" />  
  <Folders>
```

```
<Folder name="10126 - Guest"/>
<Folder name="10704 - Book Street"/>
<Folder name="10707 - Book Workshop"/>
</Folders>
</Connector>
```

This code shows the list of the children folders and files of a folder. For instance, there are folders 10704 - Book Street and 10707 - Book Workshop. The GetFoldersAndFiles command was in use. Accordingly, you can use other commands: CreateFolder (for creating a child folder) and FileUpload (adding a file in a folder). In addition, resourceType had a value Page. The other values would be Image and Document.

The following is sample code of service response when inserting documents as links:

```
<Connector command="GetFoldersAndFiles" resourceType="Document">
<CurrentFolder path="/10126 - Guest/Home/" url="" />
<Folders/>
<Files>
<File desc="4701_05_1stDraft.doc"
      name="4701_05_1stDraft.doc"
      size="804.0"
      url="/c/document_library/get_file?uuid=
          82210b6c-a938-4010-bf26-0971c84629a0&amp;
          groupId=10126" />
</Files>
</Connector>
```

The code above shows the Document type and current folder path /10126 - Guest/Home/. Moreover, it shows the files with desc (description), name (such as 4701_05_1stDraft.doc), size, and url.

FCKeditor gives the end user a high flexibility to create a custom file browser that can be integrated with it. This is a powerful feature with which specific problems unique to every case can be solved. In any case, FCKeditor offers a default implementation of the File Browser, so you are ready to use the editor without having to develop anything.

Customizing the File Browser Connector with RESTful services

As mentioned earlier, it would be a good idea to browse the contents (for example, images and documents) when the volume of content is small. But for a huge volume of content, we have to find the content by searching—using either basic search or advanced search. We're going to consider a huge volume of content and we expect to insert images and links from third-party services, for example the **RESTful** services (implemented with **Restlet**).



Representational State Transfer (REST) is a style of software architecture for distributed hypermedia systems such as the **World Wide Web (WWW)**. Restlet is a lightweight REST framework for Java. For more details, refer to <http://www.restlet.org>.

Adding advanced search view features

First of all, we need to provide an advanced search view for both images and links (for example, TEXT and COMBO). The entry point is the `file browser.html`. Let's update the `file browser.html` in order to support advanced search as follows:

1. Locate the `file browser.html` in the folder `/ext/ext-web/docroot/html/js/editor/FCKeditor/editor/filemanager/browser/liferay` and open it.
2. Remove the existing content, add the following lines at the beginning of this file, and save it:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Frameset//EN"
"http://www.w3.org/TR/html4/frameset.dtd">
<html>
  <head>
    <title>Asset Search</title>
    <link href="browser.css"
      type="text/css"
      rel="stylesheet"/>
  </head>
  <frameset rows="30%,70%" framespacing="0">
    <frame name="frmActualFolder"
      src="frmactualfolder.html"
      scrolling="auto"
      frameborder="0">
    <frame name="frmResourcesList"
      src="frmresourceslist.html"
      scrolling="auto"
      frameborder="0">
  </frameset>
</html>
```

As shown in the code above, there were two areas involved: `frmActualFolder` and `frmResourcesList`. The first one would cover the advanced search view, whereas the second one would list search results with a thumbnail image.

Then, we need to update the file `frmActualFolder.html` in order to provide the advanced search view. We can update the content of the file `frmActualFolder.html` in the folder `/ext/ext-web/docroot/html/js/editor/FCKeditor/editor/filemanager/browser/liferay`.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
  <head>
    <meta http-equiv="Content-Type"
          content="text/html; charset=utf-8" />
    <link href="browser.css" type="text/css" rel="stylesheet" />
    <link href="/themeStreet/css/main.css" rel="stylesheet"/>
    <script type="text/javascript" src="js/common.js"></script>
    <script type="text/javascript" src="js/fckxml.js"></script>
    <script type="text/javascript" src="js/browse.js"></script>
    <script type="text/javascript" src="js/search.js"></script>
  </head>
  <body bottomMargin="0" topMargin="0" >
    <form name="searchForm" id="searchForm">
      <input type="hidden"
            id="character"
            name="character"
            value="" />
      <div id="search">
        <fieldset>
          <legend>
            <font face="MS Sans Serif" size="1">Search Content</font>
          </legend>
          <table class="liferay-table" align="center" >
            <tr style="font-family: 'MS Sans Serif';">
              <td width="80" nowrap="nowrap">Title:</td>
              <td><input name="cmTitle"
                        id="cmTitle"
                        style="width: 10em;"
                        type="text"
                        value="" />
                </td>
              <td width="80"
                  nowrap="nowrap"
                  align="right">
                Media Type:</td>
              <td><select id="cmbType"
                        style="width: 11em; ">
                </select>
              </td>
            </tr>
          </table>
        </fieldset>
      </div>
    </form>
  </body>
</html>
```

```
</tr>
<!-- omit other items <tr></tr>, get them in attachments -->
</table>
</fieldset><br/>
<center>
    <input type="button"
        value="Search"
        onclick="hiddenParamSetter();"/>
    &nbsp;&nbsp;&nbsp;&nbsp;
    <input id="clearSearch"
        name="clearSearch"
        type="reset"
        value="Clear Search" />
</center>
</div>
</form>
</body>
</html>
```

As shown in the code above, you would find a set of included JavaScript files, for example `browse.js` and `search.js`. Moreover, a search form was specified with two buttons: `Search` and `Clear Search`.

Last but not the least, we need to update the file `frmResourcesList.html` in order to display advanced search results with thumbnail images. Similarly, we can update the content of the DHTML file `frmResourcesList.html` in the folder `/ext/ext-web/docroot/html/js/editor/FCKeditor/editor/filemanager/browser/liferay`.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Frameset//EN"
          "http://www.w3.org/TR/html4/frameset.dtd">
<html>
    <head>
        <link href="browser.css" type="text/css" rel="stylesheet" />
        <link href="alternatives.css" type="text/css" rel="stylesheet" />
        <script type="text/javascript" src="js/common.js"></script>
        <script type="text/javascript" src="js/fckxml.js"></script>
        <script type="text/javascript" src="js/browse.js"></script>
        <script type="text/javascript" src="js/resourcelist.js"></script>
    </head>
    <body class="FileArea"
          bottommargin="1"
          leftmargin="1"
          topmargin="1"
          rightmargin="1">
    </body>
</html>
```

As shown in the code above, you might find a set of included JavaScript files, such as `browse.js` and `resourcelist.js`.

Adding advanced search functions to links and images

We have updated the view of advanced search and search results. The advanced search functions are specified in the JavaScript files. Here we will build these advanced search functions in JavaScript files.

First, we need to specify the functions for the advanced search view as follows:

1. Create a JavaScript file `search.js` in the folder `/ext/ext-web/docroot/html/js/editor/FCKeditor/editor/filemanager/browser/liferay/js` and open it.
2. Add the following lines in this file and save it:

```
var mediaImageTypes = [ ['IMAGE', 'Image'] ];  
var mediaLinkTypes = [ ['TEXT', 'TEXT'], ['COMBO', 'COMBO'] ];  
  
function hiddenParamSetter(){  
    SetTitle(document.searchForm.cmTitle.value);  
    SetKeyword(document.searchForm.cmKeyword.value);  
    SetCreator(document.searchForm.cmCreator.value);  
    setDescription(document.searchForm.cmDescription.value);  
    SetAssetType(document.searchForm.cmbType.value);  
    advancedSearch();}  
  
var adsearch = new Object();  
adsearch.type = "";  
adsearch.creator = "";  
adsearch.keywords = "";  
adsearch.title = "";  
adsearch.description="";  
  
function SetAssetType( type ){  
    adsearch.type = type;  
};  
function SetTitle( title ){  
    adsearch.title = title;  
};  
function setDescription( description ){  
    adsearch.description = description;  
};  
function SetCreator( creator ){  
    adsearch.creator = creator;  
};  
function SetKeyword( keywords ){  
    adsearch.keywords = keywords;  
};
```

```

window.onload = function() {
    if(contentObject.ResourceType == 'Image')
        for ( var i = 0 ; i < mediaImageTypes.length ; i++ )
            AddSelectOption( document.getElementById('cmbType') ,
                mediaImageTypes[i] [1],
                mediaImageTypes[i] [0] ) ;
    else
        for ( var i = 0 ; i < mediaLinkTypes.length ; i++ )
            AddSelectOption( document.getElementById('cmbType') ,
                mediaLinkTypes[i] [1],
                mediaLinkTypes[i] [0] ) ;
    };
    function advancedSearch() {
        var obj = "assetType=" +
            adsearch.type +
            "&startIndex=0&blockSize=50&";
        if(adsearch.title.length !== 0)
            obj += "title=" + adsearch.title + "&";
        if(adsearch.description.length !== 0)
            obj += "description=" + adsearch.description + "&";
        if(adsearch.keywords.length !== 0)
            obj += "keywords=" + adsearch.keywords + "&";
        if(adsearch.creator.length !== 0)
            obj += "creator=" + adsearch.creator + "&";
        window.parent.frames['frmResourcesList']
            .search( obj, adsearch.type ) ;
    };
}

```

The code above shows media types for Image and Link (IMAGE for Image, and TEXT and COMBO for Link). Then it specifies the values of the media types. Finally, it provides a function for advanced search.

Now we need to specify the functions to display advanced search results as follows:

1. Create a JavaScript file `resourcelist.js` in the `/ext/ext-web/docroot/html/js/editor/FCKeditor/editor/filemanager/browser/liferay/js` folder and open it.
2. Add the following lines in this file and save it:

```

var oListManager = new Object() ;
oListManager.Clear = function() {document.body.innerHTML = '' ;
};
oListManager.GetFileRowHtml = function( fileName, fileTitle,
    fileDesc, fileUrl, fileSize, sFileThumbnail, bigImageUid,
    sImageUrl) {
    var sLink = '<a href="" onclick="OpenFile(\'' +

```

```
        fileUrl.replace( '/\g, '\\\\') +
        '\\', '\\' +
        bigImageUid.replace( '/\g, '\\\\') +
        '\\', '\\' +
        sFileUrl.replace( '/\g, '\\\\') +
        '\\');return false;">' ;

var sIcon = oIcons.GetIcon( fileName ) ;
var url = '';
url = '';

return '<tr> +
        '<td>&ampnbsp' +
        sLink +
        url +
        '</a>' +
        '</td><td align="right" nowrap>&ampnbsp' +
        fileTitle +
        '</td><td align="right" nowrap>&ampnbsp' +
        fileDesc +
        '</td><td align="right" nowrap>&ampnbsp' +
        fileSize +
        '</td><td align="right" nowrap>&ampnbsp' +
        '</td></tr>' ;
};

function OpenFile( fileUrl, bigImageUid, sFileUrl){
    window.top.opener.SetUrl( encodeURI( fileUrl ) ) ;
    window.top.close() ;
    window.top.opener.focus() ;
};

function search(param, mediaType) {
    contentObject.mediaType = mediaType ; oListManager.Clear() ;
    if(contentObject.ResourceType == 'Image' )
        contentObject.SendCommand( 'GetFoldersAndFiles',
                                    searchADSUrl(param),
                                    GetFoldersAndFilesCallBack) ;
    else contentObject.SendCommand( 'GetFoldersAndFiles',
                                    searchLINKUrl(param),
                                    GetFoldersAndFilesCallBack) ;

    var tHtml = new StringBuilder( '<p>' ) ;
    tHtml.Append('<b>' + 'Searching...' + '</b></p>');
    document.body.innerHTML = tHtml.ToString() ;
};

function GetFoldersAndFilesCallBack( fckXml ){
```

```

if ( contentObject.CheckError( fckXml ) != 0 )
    return ;
var currentDate = new Date();
var ts = currentDate.getYear() + "" +
        currentDate.getMonth() + "" +
        currentDate.getDate() + "" +
        currentDate.getHours() + "" +
        currentDate.getMinutes() + "" +
        currentDate.getSeconds() + "" +
        currentDate.getMilliseconds();

var ts_url = "t=" + ts + "&";
var oHtml = new StringBuilder( '<table id="tableFiles"
                                class="its">' ) ;
oHtml.Append("<tr><th style=\"color: #333333;
                           font-weight: bold;
                           font-size: 11px;\">Image</th>
<th style=\"color: #333333;
                           font-weight: bold;
                           font-size: 11px;\">Title</th>
<th style=\"color: #333333;
                           font-weight: bold;
                           font-size: 11px;\">
                           Description</th>
<th style=\"color: #333333;
                           font-weight: bold;
                           font-size: 11px;\">Modified</th>
</tr> ");
var oNodes = fckXml.SelectNodes( 'Connector/Files/File' ) ;
if(oNodes.length == 0) {
    var tHtml = new StringBuilder( '<p>' ) ;
    tHtml.Append('<b>' + 'No results found' + '</b></p>');
    document.body.innerHTML = tHtml.ToString() ;
    return;
};
ar url = location.protocol + '//' + location.host + '/';
for ( var j = 0 ; j < oNodes.length ; j++ ){
    var oNode = oNodes[j] ;
    var sFileName = oNode.attributes.getNamedItem('name').value ;
    var sFileDesc = oNode.attributes.getNamedItem
                    ('description').value ;
    var sFileTitle = oNode.attributes.getNamedItem('title').value ;
    var sFileUrl = oNode.attributes.getNamedItem('url').value ;
    var sFileGuid = oNode.attributes.getNamedItem('guid').value ;
    var sFileModified = oNode.attributes.getNamedItem
                       ('modified').value ;
    var sFileThumbnail = oNode.attributes.getNamedItem
                       ('thumbnail').value;

```

```
if(sFileThumbnail.length > "a9bcdb3b-0b2f-11dd-8fd0-
cd36a59c8e0c".length)
    sFileThumbnail = url + sFileThumbnail;
else if (sFileThumbnail.length == "a9bcdb3b-0b2f-11dd-8fd0-
cd36a59c8e0c".length)
    sFileThumbnail = sFileUrl +
                    "=download&uid=" +
                    sFileThumbnail +
                    "&" +
                    ts_url;
else sFileThumbnail = sFileUrl +
                    "=download&uid=" +
                    sFileGuid +
                    "&" +
                    ts_url;
var bigImageUid = "";
var link = sFileUrl +
           "=download&uid=" +
           sFileGuid +
           "&" +
           ts_url;
oHtml.Append( oListManager.GetFileRowHtml( sFileName,
                                           sFileTitle, sFileDesc, link, sFileModified,
                                           sFileThumbnail, bigImageUid, sFileUrl) ) ;
};

oHtml.Append( '</table>' ) ;
document.body.innerHTML = oHtml.ToString() ; };
```

The code above specifies a set of advanced search functions, `GetFileRowHtml`, `openFile`, `search`, and `GetFoldersAndFilesCallBack`.

Finally, we need to specify the functions to link the advanced search services by using the following steps:

1. Create a JavaScript file `browse.js` in the folder `/ext/ext-web/docroot/html/js/editor/FCKeditor/editor/filemanager/browser/liferay/js` and open it.
2. Add the following lines in this file and save it:

```
var searchADSPath = "cms_services/services?action
                     =navigateAssetADS&";
var searchLINKPath = "cms_services/services?action
                     =navigateAssetLINK&";

function searchADSUrl(param) {
    var location = document.location;
    var url = location.protocol + '//' +
              location.host + '/';
    url += searchADSPath + param;
```

```

        return url;
    };

    function searchLINKUrl(param) {
        var location = document.location;
        var url = location.protocol + '//' +
            location.host + '/';
        url += searchLINKPath + param; return url;
    };

    var contentObject = new Object();
    contentObject.ResourceType= GetUrlParam( 'Type' ) ;
    contentObject.SendCommand = function( command,
                                         params, callBackFunction ) {

        var sUrl = params ;
        var oXML = new FCKXml() ;

        if ( callBackFunction ){
            oXML.LoadUrl( sUrl, callBackFunction ) ;// Asynchronous load.
        }
        else {
            return oXML.LoadUrl( sUrl ) ;
        }

        return null ;
    };

    contentObject.CheckError = function( responseXml ) {
        var iErrorNumber = 0 ;
        var oErrorNode = responseXml.SelectSingleNode(
            'Connector/Error' ) ;

        if ( oErrorNode ) {
            iErrorNumber = parseInt( oErrorNode.attributes
                .getNamedItem('number').value, 10 ) ;
            switch ( iErrorNumber ) {
                case 0 : break ;
                case 1 :
                    // Custom error. Message placed in the "text" attribute.
                    alert( oErrorNode.attributes.getNamedItem('text').value );
                    break ;
                case 101 : alert( 'Folder already exists' ) ;
                    break ;
                case 102 : alert( 'Invalid folder name' ) ;
                    break ;
                case 103 : alert( 'You have no permissions to create the
                    folder' ) ;
                    break ;
            }
        }
    };
}

```

```
        case 110 : alert( 'Unknown error creating folder' ) ;
            break ;
        default : alert( 'Error on your request. Error number: '
                        + iErrorNumber ) ;
            break ;
    }
}

return iErrorNumber ;
};

function GetUrlParam( paramName ){
    var oRegex = new RegExp( '[\?&]' +
        paramName + '=([^\&]+)', 'i' ) ;
    var location = window.top.location;
    var oMatch = oRegex.exec( location.search ) ;
    if ( oMatch && oMatch.length > 1 ){
        return decodeURIComponent( oMatch[1] ) ;
    }
    else { return '' ; }
};

var oIcons = new Object() ;
oIcons.AvailableIconsArray = [ 'ai', 'avi', 'bmp', 'cs', 'dll', 'doc',
    'exe', 'fla', 'gif', 'htm', 'html',
    'jpg', 'js', 'mdb', 'mp3', 'pdf',
    'png', 'ppt', 'rdp', 'swf', 'swt',
    'txt', 'vsd', 'xls', 'xml', 'zip' ] ;
oIcons.AvailableIcons = new Object() ;

for ( var i = 0 ; i < oIcons.AvailableIconsArray.length ; i++ ) {
    oIcons.AvailableIcons[ oIcons.AvailableIconsArray[i] ] = true;
}

oIcons.GetIcon = function( fileName ){
    var sExtension = fileName.substr( fileName.lastIndexOf('.') +
        1 ).toLowerCase() ;
    if ( this.AvailableIcons[ sExtension ] === true ) {
        return sExtension ;
    }
    else { return 'default.icon' ; }
};
```

The preceding code shows a set of advanced search paths such as `searchADSPath` and `searchLINKPath`. It also defines advanced search URLs for images and links, for example `searchADSUrl` and `searchLINKUrl`. Lastly, it reuses the `SendCommand` function with a callback ability `callBackFunction`.

After that, you can simply click on the Ant target `deploy-fast`. Now, you should be able to view the updates immediately.

Preparing RESTful services

We have seen the advanced search on the images and links. Now let's look at the RESTful services in detail. The following are the sample RESTful services for images and links (TEXT and COMBO):

```
// Download images and Links  
/cms_services/services?action=download&uid=uid  
  
// Search Images  
/cms_services/services?action=navigateAssetADS&assetType=IMAGE&title=v  
alue&description=value&keywords=value&creator=value&  
  
// Search Links  
// TEXT  
/cms_services/services?action=navigateAssetLINK&assetType=TEXT&title=v  
alue&description=value&keyword89s=value&creator=value&  
  
// COMBO  
/cms_services/services?action=navigateAssetLINK&assetType=COMBO&title=v  
alue&description=value&keywords=value&creator=value&
```

The code above begins by displaying the download service by the UID. Then it creates output content as stream with MIME-types. It also shows search services for images and links with multiple fields such as `title`, `description`, `keywords`, and `creator`. The action for images was `navigateAssetADS`, whereas the action for links was `navigateAssetLINK`. The asset type could be `IMAGE`, `TEXT`, and `COMBO`.

The following is the service XML response when inserting images. When preparing the REST services for images and links, the same XML pattern should be followed. The following is a sample code:

```
<Connector>
  <CurrentFolder name="type" guid="keyword" url="http://type"/>
  <ParentFolder name="type" guid="type" url="http://type"/>
  <Files>
    <File guid="0152b610-31cc-11dd-a782-335275be4d09"
          name="CHARACTER_ARTWORK_A1001001A08C03B60539C13473.xml"
          title="CHARACTER_ARTWORK_A1001001A08C03B60539C13473.xml"
          bigImageUid=""
          description=""
          thumbnail="015cef4a-31cc-11dd-a782-335275be4d09"
          modified="Wed Jun 04 00:20:16 GMT 2008"
          size=" 110288"
          url="/cms_services/services?action" />
    <File guid="0167c4c4-31cc-11dd-a782-335275be4d09"
          name="CHARACTER_ARTWORK_A1001001A08C03B60539C13473
          .xml 264x198"
          title="CHARACTER_ARTWORK_A1001001A08C03B60539C13473
          .xml 264x198"
          bigImageUid=""
          description=""
          thumbnail=""
          modified="Wed Jun 04 00:20:16 GMT 2008"
          size=" 110288"
          url="/cms_services/services?action" />
  </Files>
</Connector>
```

The code above does not only cover the default XML attributes `name` and `size`, but it also extends the XML attributes `GUID`, `title`, `description`, `thumbnail`, `big image UID`, `modified` (the modification date), and (service) URL.

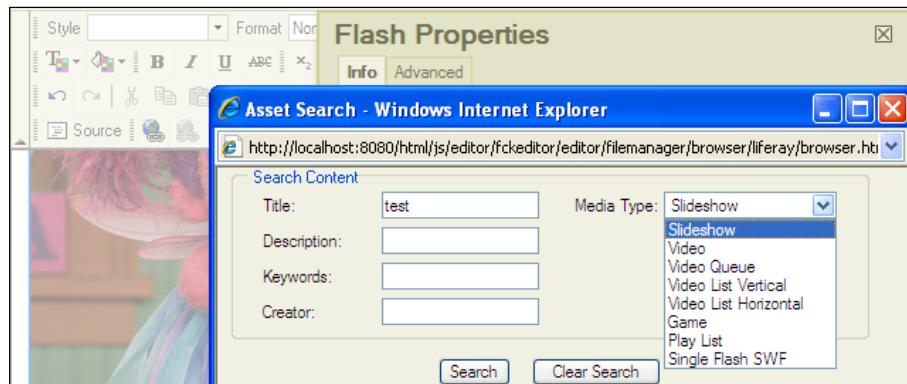
Obviously, FCKeditor's File Browser Connector supports XML responses in the `GetFoldersAndFiles` command (XML). Thus, the XML responses can be extended smoothly in order to support additional attributes within RESTful services (for example, `/cms_services/services?action`).

Inserting content-rich flashes into Web Content

We have successfully inserted images and links into web content (known as journal articles). We also discussed how to browse images and links from Liferay portal services and how to search images and links from third-party services (that is, REST services). Normally, images and links would be helpful to build journal articles. But this is not enough. When building journal articles, you may want to insert content-rich flashes with text, images, and links. These content-rich flashes could be a single flash **SWF (Shockwave Flash)**, slideshow (an ordered set of images), video, game, video queue (an ordered set of videos), video list (an unordered set of videos), and playlist (an ordered set of games and/or videos).

As shown in the following screenshot, we first need to extend the resource type **Flash** and specify **Media Type** for it. The media types here include: **Single Flash SWF**, **Slideshow**, **Video**, **Video Queue**, **Video List Vertical**, **Video List Horizontal**, **Game**, and **Play List**.

Suppose that the volume of content flashes is huge and so we could not find a specific content using the browsing function. Here, an advanced search would be helpful. The advanced search fields should cover **Title**, **Description**, **Keywords**, and **Creator**. In this case, you may have more fields for advanced search. Therefore, the framework for advanced search should be flexible. It should be easy to extend the service and UI framework.



In this section, we're going to focus on how to insert flashes into journal articles smoothly.

Querying flashes

First of all, we need to update the configuration file `fckconfig.jsp` in order to make the flash browser URL available. To do so, replace the line `FCKConfig FlashBrowser = false ;` with the following line in `fckconfig.jsp` under the folder `/ext/ext-web/docroot/html/js/editor/FCKeditor:`

```
FCKConfig.FlashBrowserURL = FCKConfig.BasePath +
    "filemanager/browser/liferay/browser.html?
    Type=Flash&Connector=<%= connectorURL %>";
```

The code above configures `FlashBrowserURL` with the `Flash` type and the connector URL.

Then, we need to update the user interface to preview Flash with one more parameter: `Align`. This feature is specified in the `fck_flash.html` file. Let's update the content of this file as follows:

1. Locate the file `fck_flash.html` in the folder `/ext/ext-web/docroot/html/js/editor/FCKeditor/editor/dialog` and open it.
2. Locate the following lines in the this file:

```
<TD nowrap>
    <span fckLang="DlgImgWidth">Width</span><br>
    <input id="txtWidth"
        onkeypress="return IsDigit(event);"
        type="text"
        size="3">
</TD>
<TD>&nbsp;</TD>
<TD>
    <span fckLang="DlgImgHeight">Height</span><br>
    <input id="txtHeight"
        onkeypress="return IsDigit(event);"
        type="text"
        size="3">
</TD>
```

3. Replace the code above with the following lines in this file and save it:

```
<TD width="40" align="right" nowrap>
    <span fckLang="DlgImgWidth">Width</span>&nbsp;</TD>
<TD width="60" nowrap>
    <input id="txtWidth"
        onkeypress="return IsDigit(event);"
        type="text"
        size="3">
</TD>
<TD width="40" align="right">
```

```

<span fckLang="DlgImgHeight">Height</span>&nbsp;</TD>
<TD width="60" >
    <input id="txtHeight"
        onkeypress="return IsDigit(event);"
        type="text"
        size="3">
</TD>
<TD width="40" align="right" nowrap="nowrap">
    <span fckLang="DlgImgAlign">Align</span>&nbsp;</TD>
<TD width="80">
    <select id="cmbAlign" onchange="UpdatePreview();">
        <option value="" selected="selected"></option>
        <option fckLang="DlgImgAlignLeft" value="left">Left</option>
        <option fckLang="DlgImgAlignAbsBottom"
            value="absBottom">Abs Bottom</option>
        <option fckLang="DlgImgAlignAbsMiddle"
            value="absMiddle">Abs Middle</option>
        <option fckLang="DlgImgAlignBaseline"
            value="baseline">Baseline</option>
        <option fckLang="DlgImgAlignBottom"
            value="bottom">Bottom</option>
        <option fckLang="DlgImgAlignMiddle"
            value="middle">Middle</option>
        <option fckLang="DlgImgAlignRight"
            value="right">Right</option>
        <option fckLang="DlgImgAlignTextTop"
            value="textTop">Text Top</option>
        <option fckLang="DlgImgAlignTop" value="top">Top</option>
    </select>
</TD>

```

The code above adds an additional item `cmAlign` for previewing flashes—positional relationship between the flashes and text. The `select` options covered a set of values, for example `Left`, `Abs Bottom`, `Abs Middle`, `Top`, and so on.

Consequently, we need to specify the functions for advanced search media type of Flash as follows:

1. Locate the JavaScript file `search.js` in the folder `/ext/ext-web/docroot/html/js/editor/FCKeditor/editor/filemanager/browser/liferay/js` and open it.
2. Add the following lines in it before the line function

```

hiddenParamSetter() {
    var mediaFlashTypes = [ ['SLIDESHOW', 'Slideshow'],
        ['VIDEO', 'Video'],
        ['VIDEOQUEUE', 'Video Queue'],

```

```
[ 'VIDEOLISTV', 'Video List Vertical'],
[ 'VIDEOLISTH', 'Video List Horizontal'],
[ 'GAME', 'Game' ], [ 'PLAYLIST', 'Play List' ],
[ 'SWF', 'Single Flash SWF' ] ];
```

3. Add the following lines before the line `else` of the `window.onload = function()` in `search.js` method and save the file:

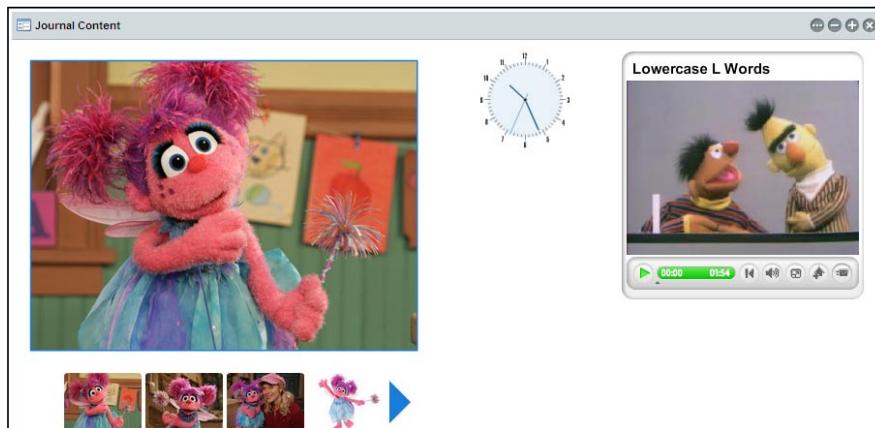
```
else if(contentObject.ResourceType == 'Flash')
    for ( var i = 0 ; i < mediaFlashTypes.length ; i++ )
        AddSelectOption( document.getElementById('cmbType'),
                         mediaFlashTypes[i][1], mediaFlashTypes[i][0] );
```

The code above specifies the media types first, and then it specifies the values of the media types when the window was loaded.

Adding single flash SWF, videos, and slideshows to journal articles

We have added the flash browsing capability on the one hand. On the other hand, we have extended the flash view function by adding the Align feature and advanced search function to support multiple media types. Now let's see how to add single flash SWF, videos, and slideshows as part of HTML text in journal articles.

As shown in the following screenshot, the journal article (web content) includes a slideshow, SWF, and video as its content. Here, slideshow, SWF, and video are flash objects. You should be able to insert them easily into journal articles, like you had done for images and links in the previous sections.



Let's look in detail at how to implement these features – inserting slideshow, SWF, and video into the journal articles via FCKeditor. The following are the main steps used to implement these features, which are discussed in the sections that follow:

- Adding advanced search views
- Adding advanced search functions
- Adding flash objects

Adding advanced search views

First, let's add advanced search views for slideshow, SWF, and video in the `search.js` file as follows:

1. Locate the JavaScript file `search.js` in the folder `/ext/ext-web/docroot/html/js/editor/FCKeditor/editor/filemanager/browser/liferay/js` and open it.
2. Add the following lines in it and save it.

```
if (contentObject.ResourceType == 'Flash') {
    if(adsearch.type == "VIDEO" || adsearch.type == "GAME")
        obj = "assetType=" +
            adsearch.type +
            "&startIndex=0&blockSize=50&" ;
    else obj = "assetType=" + adsearch.type + "&approved=1&" ;
}
```

Adding advanced search functions

After adding advanced search views, let's add the advanced search functions for slideshow, SWF, and video in the `resourcelist.js` file as follows:

1. Locate the JavaScript file `resourcelist.js` in the folder `/ext/ext-web/docroot/html/js/editor/FCKeditor/editor/filemanager/browser/liferay/js` and open it.
2. In this file, add the following lines before the line `window.top.opener.`

```
SetUrl( encodeURI( fileUrl ) ) ; of the function OpenFile( fileUrl,
bigImageUid, sImageUrl) method:
```

```
var location = document.location;
var url = location.protocol + '//' + location.host + '/';
var width = "420";
var height = "410";
if(contentObject.ResourceType == 'Flash'){
    window.top.opener.SetActionType( contentObject.mediaType ) ;
```

```
switch ( contentObject.mediaType ) {
    case "SLIDESHOW" : width = "420";
                        height = "410";
                        window.top.opener.SetUrl( encodeURI( url
                            + "cms_services/jsp/slideshowPreview.
                            jsp?" + "&uid=" + fileUrl ),
                            width, height ) ;
                        break;
    case "VIDEO" : width = "250";
                    height = "256";
                    window.top.opener.SetUrl( encodeURI(
                        url + "cms_services/jsp/videoPreview.
                        jsp?" + "&uid=" + fileUrl ),
                        width, height ) ;
                    break;
    case "SWF" : width = "400";
                  height = "300";
                  window.top.opener.SetUrl( encodeURI(
                      location.protocol + '//' + location.host +
                      fileUrl ), width, height ) ;
                  break;
    default: alert("!nknown Media Type!");
              break;
}
} else
```

3. Then add the following lines before the line `else` of the function `search(param, mediaType)` method:

```
else if(contentObject.ResourceType == 'Flash'){
    if (contentObject.mediaType == "VIDEO") {
        contentObject.SendCommand( 'GetFoldersAndFiles',
            searchADSUrl(param), GetFoldersAndFilesCallBack) ;
    }
    else if( mediaType == "SLIDESHOW" || mediaType == "SWF" ) {
        contentObject.SendCommand( 'GetFoldersAndFiles',
            searchFlashUrl(param), GetFoldersAndFilesCallBack) ;
    }
}
```

4. Finally, add the following lines after the line `var link = sFileUrl + "=download&uid=" + sFileGuid + "&" + ts_url;` of the function `GetFoldersAndFilesCallBack(fckXml)` method and save the file:

```
if(contentObject.ResourceType == 'Flash') link = sFileGuid;
```

The preceding code specifies the media type values and a set of advanced search functions, `openFile`, `search`, and `GetFoldersAndFilesCallBack`.

After that, we need to update the advanced search functions for slideshow, SWF, and video in the `browse.js` file as follows:

1. Locate the JavaScript file `browse.js` in the folder `/ext/ext-web/docroot/html/js/editor/FCKeditor/editor/filemanager/browser/liferay/js` and open it.
2. Add the following lines after the line `var searchLINKPath = "cms_services/services?action=navigateAssetLINK&"`; and save it:


```
var searchFlashPath = "cms_services/services?action=navigateAssetFlash&";
function searchFlashUrl(param) {
    var location = document.location;
    var url = location.protocol + '//' + location.host + '/';
    url += searchFlashPath + param;
    return url;
};
```

The code above specifies the `searchFlashPath` variable and the function `searchFlashUrl(param)` method.

Adding flash objects

In addition, we need to add the flash objects for view and preview of slideshow, SWF, and video in the following manner:

1. Locate the JavaScript file `fck_flash.js` in the folder `/ext/ext-web/docroot/html/js/editor/FCKeditor/editor/dialog/fck_flash` and open it.
2. Add the following lines after the line `var FCKTools = oEditor.FCKTools;` in it:


```
// Ext
var obj = new Object();
obj.actionType = "";
function SetActionType( fSActionType ) {
    obj.actionType = fSActionType;
}
```
3. Insert the following line after the line `GetE('txtHeight').value = GetAttribute(oEmbed, 'height', '');`

```
GetE('cmbAlign').value      = GetAttribute( oImage, 'align', '' );
//Ext
```

4. Insert the following lines before the line `oEmbed = FCK.EditorDocument.`

```
createElement( 'EMBED' ) ;  
if(obj.actionType != "SWF")  
    oEmbed = FCK.EditorDocument.createElement( 'IFRAME' ) ; // Ext  
else
```

5. Then add the following lines after the line `function UpdateEmbed(e):`

```
{  
    if(obj.actionType == "SWF")  
        UpdateSWF( e, true );  
    else UpdateFlash( e, true );  
}; //Ext  
  
function UpdateFlash( e , skipId ) {  
    SetAttribute( e, 'src', GetE('txtUrl').value ) ;  
    SetAttribute( e, "width" , GetE('txtWidth').value ) ;  
    SetAttribute( e, "height", GetE('txtHeight').value ) ;  
    SetAttribute( e, "align" , GetE('cmbAlign').value ) ;  
    SetAttribute( e, "frameborder" , "0" ) ;  
    SetAttribute( e, "scrolling", "no" ) ;  
    SetAttribute( e, "marginwidth" , "0" ) ;  
    SetAttribute( e, "marginheight", "0" ) ;  
    // Advances Attributes  
    if ( ! skipId )  
        SetAttribute( e, 'id', GetE('txtAttId').value ) ;  
    SetAttribute( e, 'title', GetE('txtAttTitle').value ) ;  
    if ( oEditor.FCKBrowserInfo.IsIE ) {  
        SetAttribute( e, 'className', GetE('txtAttClasses').value ) ;  
        e.style.cssText = GetE('txtAttStyle').value ;  
    }  
    else {  
        SetAttribute( e, 'class', GetE('txtAttClasses').value ) ;  
        SetAttribute( e, 'style', GetE('txtAttStyle').value ) ;  
    }  
};  
function UpdateSWF( e , skipId )
```

6. Finally, add the following lines after the line `SetAttribute(e, 'height', '100%') ;` and save the file:

```
if(obj.actionType != "SWF") { // EXT  
    e = oDoc.createElement( 'IFRAME' ) ;  
    SetAttribute( e, 'src', GetE('txtUrl').value ) ;  
    SetAttribute( e, 'width', '100%' ) ;
```

```

SetAttribute( e, 'height', '100%' ) ;
SetAttribute( e, "frameborder" , "1" ) ;
SetAttribute( e, "scrolling", "yes" ) ;
SetAttribute( e, "marginwidth" , "0" ) ;
SetAttribute( e, "marginheight", "0" ) ;
}

```

The code above specifies an object to handle the media type. Then it updates the function to update the HTML objects: `EMBED` and `IFRAME`. A similar update is applied to the preview function.

Now, you can simply click on the Ant target `deploy-fast` and view the above updates immediately.

Adding video queue and video list as part of journal articles

As shown in the following screenshot, we want to insert a video list (vertical view) as well as images via FCKeditor. On the left side, there is normal text with a title and body. An image with a caption is also added in the middle of the body. At the same time, a video list is added as a part of this journal article on the right side.

A Lion and a Little Girl Speak to India's Children

It's early morning on the set of Workshop's locally produced version of *Street*. The production team stands together quietly as one crew member breaks a coconut. The group shares pieces of the fruit, and as they do so, they pray that their day's work will be productive and fruitful.

This quintessential Indian ceremony of performing *pūja* is just one of the many ways in which Indian culture influences the localized production. Producers took great care to weave the diversity of Indian culture into a show that represents people from all different religions and socioeconomic levels.

Identifying Indian children's educational needs

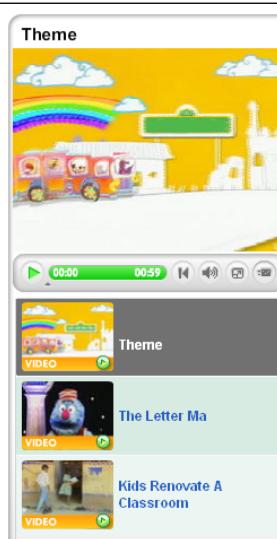
One carefully researched character is Boombah, a gregarious, cuddly lion who loves eating vegetables and dancing to Bhangra music.



Boombah is one of four Muppet characters developed during a curriculum seminar in 2004, when the production team gathered a diverse group together to discuss the program's educational goals. The participants included educators, media specialists, educational consultants, heads of NGOs, partners from Turner Broadcasting, and the Workshop team; there was even a children's toymaker and a city planner in attendance.

Workshop international producer Nadine Zylstra was present during the seminar. She describes the experience as an open dialogue that helped develop the show's direction.

"The experience contained real discussion and depth about the program's objectives, and it really added to my understanding of Indian culture," Zylstra says. "It was a privilege to be at the table."



The screenshot shows the FCKeditor interface with a video player on the left and a video list on the right. The video player has a play button, a progress bar showing 00:59, and various control buttons. The video list contains three items: "Theme" (with a thumbnail of a colorful cityscape), "The Letter Ma" (with a thumbnail of a cartoon character), and "Kids Renovate A Classroom" (with a thumbnail of two children in a room).

As mentioned earlier, we have successfully implemented the solution: inserting a slideshow, SWF, and video into journal articles (web content) via FCKeditor. Here we expect to use video queue and video list as HTML objects as well as slideshows, SWF, and videos.

As shown in the following screenshot, we can insert a video list (horizontal view) as well as images. When creating an article, we could search the video lists first, find the expected video list, and finally insert the video list as a part of content of journal articles.

Further, we expect the ability to add video queue as HTML objects into journal articles as shown in the following screenshot. In other words, when creating an article, we could search the video queues first, then find specific video queue, and finally insert the video queue as part of content of articles.

A video queue is a collection of videos. These videos would be the most popular videos (for example, limit to first ten), related videos, recently added videos, and so on. At the same time, a list of features and tags are displayed as well.



In this section, we're going to take an in-depth look at how to search video queues and video lists, and how to embed video queues and video lists into journal articles via FCKeditor.

Putting a video list into journal articles

Similar to a slideshow, you can put a video list into journal articles via FCKeditor. First, check the advanced search function `advancedSearch()` in the JavaScript file `search.js` for video list. You would see the advanced search functions such as:

```
if (contentObject.ResourceType == 'Flash'){
    if(adsearch.type == "VIDEO" || adsearch.type == "GAME")
        obj = "assetType=" +
              adsearch.type +
              "&startIndex=0&blockSize=50&" ;
    else obj = "assetType=" + adsearch.type + "&approved=1&" ;
}
```

Let's update the advanced search functions in the JavaScript file `resourcelist.js` for the video list as follows:

1. Locate the JavaScript file `resourcelist.js` in the folder `/ext/ext-web/docroot/html/js/editor/FCKeditor/editor/filemanager/browser/liferay/js` and open it.
2. Add the following lines before the line `default:` of the function `OpenFile(fileUrl, bigImageUid, sImageUrl)` method:

```

case "VIDEOLISTV" : width = "250";
                     height = "480";
                     window.top.opener.SetUrl( encodeURI( url +
                     "cms_services/jsp/videolist_v_Preview.jsp?" +
                     "&uid=" + fileUrl ), width, height ) ;
                     break;
case "VIDEOLISTH" : width = "670";
                     height = "380";
                     window.top.opener.SetUrl( encodeURI( url +
                     "cms_services/jsp/videolistPreview.jsp?" +
                     "&uid=" + fileUrl ), width, height ) ;
                     break;
```
3. Add the following after `else if (mediaType == "SLIDEESHOW" || mediaType == "SWF"` of the function `search(param, mediaType)` method and save the file:

```

|| mediaType == "VIDEOLISTH" || mediaType == "VIDEOLISTV"
```

The code above specifies media type values for `VIDEOLISTV` and `VIDEOLISTH`. Moreover, it specifies a set of advanced search functions, `openFile`, `search`, and `GetFoldersAndFilesCallBack`.

Just like you did for a slideshow, you can add media type values for video list as well. Similarly, you can add an HTML flash object for the view and preview of video list such as `IFRAME`.

Setting up video queue in journal articles

Interestingly, we can also add video queue into journal articles via FCKeditor. We can follow the same pattern we used for video list to make video queue available for the `Insert / Edit Flash` function of FCKeditor.

First, check the advanced search views in the JavaScript file `search.js` for video queue. Refer to the above process for video list.

Then update the advanced search functions in the JavaScript file `resourcelist.js` for video queue in the following manner:

1. Locate the JavaScript file `resourcelist.js` in the folder `/ext/ext-web/docroot/html/js/editor/FCKeditor/editor/filemanager/browser/liferay/js` and open it.
2. Add the following lines before the line `default:` of the function `OpenFile(fileUrl, bigImageUid, sImageUrl)` method:

```
case "VIDEOQUEUE" : width = "918";
                     height = "400";
                     window.top.opener.SetUrl( encodeURI( url +
                     "cms_services/jsp/videoqueuePreview.jsp?" +
                     "&uid=" + fileUrl ), width, height ) ;
                     break;
```

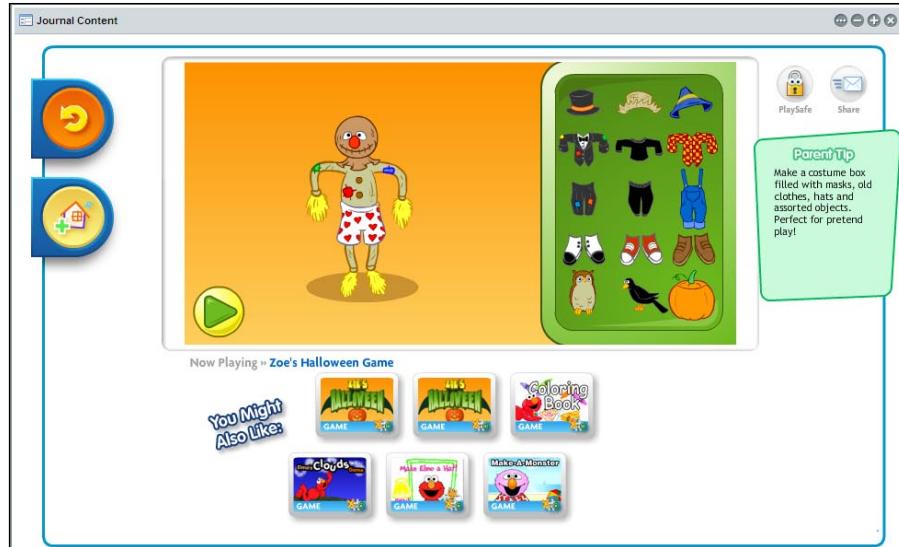
3. Add the following line after `else if(mediaType == "SLIDESHOW" || mediaType == "SWF"` of the method `function search(param, mediaType)` and save the file:
`|| mediaType == "VIDEOQUEUE"`

The code above specifies the media type values for video queue. It also specifies a set of advanced search functions—`openFile`, `search`, and `GetFoldersAndFilesCallBack`. Just like you did for a slideshow, you can add media type values for video queue as well. Similarly, you can add HTML flash object for the view and preview of video queue, for example `IFRAME`.

Adding games and playlists as part of journal articles

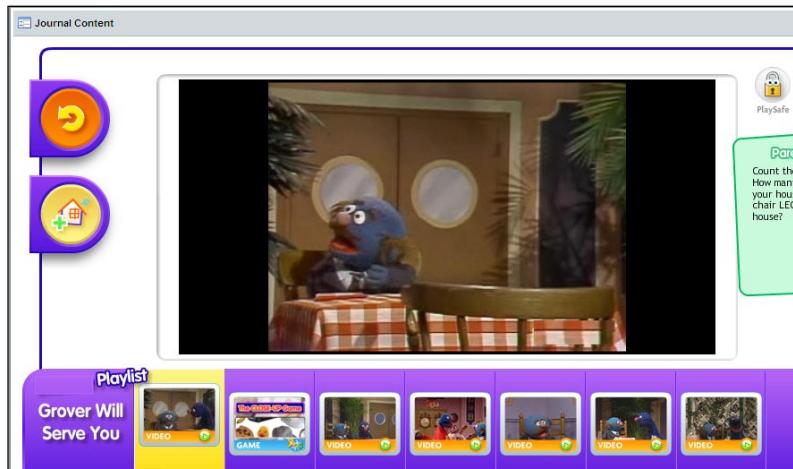
We may have journal articles that talk about games and/or playlists. Thus, it would be perfect if we could put real games and/or playlists as part of journal articles via FCKeditor. So, when users read the message about games and/or playlists, they can play the example game and/or playlist as well. Of course, they will be impressed with both, what they are seeing and what they are experiencing.

As shown in the following screenshot, a game with a player is added in the journal content. At the same time, the related games are attached as well. The number of the related games is seven by default, that is, there is one game and six related games. Moreover, it has a message for parents and a set of buttons, for example **Start-over**, **Add to my street**, **PlaySafe**, and **Share**.



A playlist is a collection of games and/or videos. The number of games and/or videos is seven by default. As shown in the following screenshot, this playlist consists of one game and six videos. Similar to games, the playlist with player has message for parents and a set of buttons, for example **Start-over**, **Add to my street**, **PlaySafe**, and **Share**.

It would be nice if we could add a playlist as a part of journal articles. So when users view the article with a playlist, they can play with the playlist as well. This is *what you see is what you get*. For instance, there is an article talking about a playlist with text. It would be great if the text (for example, introduction) could be coming with the real playlist.



In this section we're look at an approach to insert the game and/or playlist into the journal articles.

Playing games beside text message

In order to play games while reading the text of journal articles, we should provide the ability to insert GAME as an HTML object into FCKeditor. First, check the advanced search views in the JavaScript file `search.js` for games. Refer to the above process for video. Now let's update the advanced search functions for games in the JavaScript file `resourcelist.js` as follows:

1. Locate the JavaScript file `resourcelist.js` in the folder `/ext/ext-web/docroot/html/js/editor/FCKeditor/editor/filemanager/browser/liferay/js` and open it.
2. Add the following lines before the line `default:` of the function `OpenFile(fileUrl, bigImageUid, sImageUrl)` method:

```
case "GAME" : width = "960";
height = "550";
window.top.opener.SetUrl( encodeURI( url +
"cms_services/jsp/gamePreview.jsp?" +
"&uid=" + fileUrl ), width, height );
break;
```
3. Then add the following line after `if (contentObject.mediaType == "VIDEO"` of the function `search(param, mediaType)` method and save the file:
`|| contentObject.mediaType == "GAME"`

The code above specifies the media type values for GAME. It also specifies a set of advanced search functions such as `openFile`, `search`, and `GetFoldersAndFilesCallBack`. Just like you did for video, add media type values for game. Similarly, reuse the same HTML flash object to view and preview a video, that is, `IFRAME`.

Employing playlist as visualization of text information

We have successfully provided a solution to insert VIDEO, GAME, SLIDE SHOW, VIDEOLIST, and VIDEOQUEUE as HTML objects into the web content via FCKeditor. Here we can insert playlist as an HTML object into journal articles via FCKeditor, sharing the same pattern.

First, check the advanced search views in the JavaScript file `search.js` for playlist. Refer to the above process for video. Next, update the advanced search functions in the JavaScript file `resourcelist.js` for playlist as follows:

1. Locate the JavaScript file `resourcelist.js` in the folder `/ext/ext-web/docroot/html/js/editor/FCKeditor/editor/filemanager/browser/liferay/js` and open it.
2. Add the following lines before the line `default:` of the function `OpenFile(fileUrl, bigImageUid, sImageUrl)` method:

```
case "PLAYLIST" : width = "958";
                    height = "536";
                    window.top.opener.SetUrl( encodeURI( url +
                        "cms_services/jsp/playlistPreview.jsp?" +
                        "&uid=" + fileUrl ), width, height ) ;
                    break;
```

3. Then add the following line after `else if(mediaType == "SLIDEESHOW" || mediaType == "SWF"` of the function `search(param, mediaType)` method and save the file:

```
|| mediaType == "PLAYLIST"
```

This code specifies a media type with the `PLAYLIST` value. Moreover, it also specifies a set of advanced search functions such as `openFile`, `search`, and `GetFoldersAndFilesCallBack`. Just like you did for `SLIDEESHOW`, add a media type value `PLAYLIST`. Similarly, reuse the same HTML flash object—`IFRAME`—for the view and preview of `PLAYLIST`.

Preparing RESTful services

We have mentioned the advanced search functions for `SWF`, `VIDEO`, `GAME`, `SLIDEESHOW`, `VIDEOQUEUE`, `VIDEOLISTH`, `VIDEOLISTV`, and `PLAYLIST` in different parts. Now let's look in detail at the RESTful services. The following are sample RESTful services for these objects:

```
// search VIDEO | GAME
/cms_services/services?action=navigateAssetADS&assetType=type &title=v
alue&description=value&keywords=value&creator=value
type=VIDEO|GAME

// Search SWF, SLIDEESHOW, VIDEOQUEUE, VIDEOLISTH, VIDEOLISTV, and
PLAYLIST
/cms_services/services?action=navigateAssetFlash&assetType=Type&title=v
alue&description=value&keywords=value&creator=value
Type =SWF|SLIDEESHOW|VIDEOQUEUE|VIDEOLISTH|VIDEOLISTV|PLAYLIST
```

The code above shows the search services for videos and games with multiple fields `title`, `description`, `keywords`, and `creator`. The action for images is `navigateAssetADS`, whereas the action for `SWF`, `SLIDEESHOW`, `VIDEOQUEUE`, `VIDEOLISTH`, `VIDEOLISTV`, and `PLAYLIST` is `navigateAssetFlash`.

We also reuse a set of services for the HTML object `IFRAME`. The following is sample list of these services:

```
// for Slideshow  
cms_services/jsp/slideshowPreview.jsp?uid=value  
// for Video  
cms_services/jsp/videoPreview.jsp?uid=value  
// for GAME  
cms_services/jsp/gamePreview.jsp?uid=value  
// for Videolist horinzatal view  
cms_services/jsp/videolistPreview.jsp?uid=value  
// for Videolist vertical view  
cms_services/jsp/videolist-v-Preview.jsp?uid=value  
// for Video queue  
cms_services/jsp/videoqueuePreview.jsp?uid=value  
// for Playlist  
cms_services/jsp/playlistPreview.jsp?uid=value
```

As shown in the code above, the main service is called `cms_services`.

 You can play with this service at <http://liferay.cignex.com> in your local machine. Moreover, as a do-it-yourselfer, you can download the `cms_services.war` service and the related example data `bookpub.sql` from http://liferay.cignex.com/palm_tree/book/cms_services.war and `bookpub.sql`. Note that the default database name is `bookpub` and the default account on this database is `lportal/lportal`. MySQL is the default database.

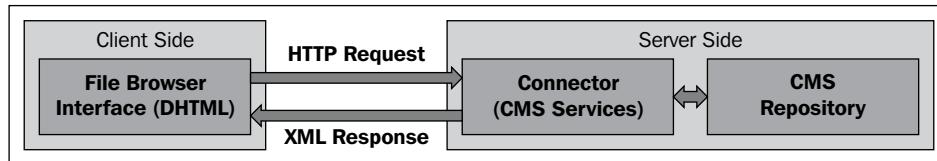
Using the WYSIWYG editor FCKeditor efficiently

FCKeditor can be fully adjusted according to different needs and can be used in many ways. Liferay portal integrated with FCKeditor and others for blog entries, forum topics, WIKI articles, journal articles, and so on. Of course, you can use it as a WYSIWYG editor in portlets too.

Extending the file browser connector

FCKeditor offers a unique interface, which can be used by all server-side languages. The interface was implemented completely on JavaScript DHTML and the integration is available via XML. Thus, the users who want to integrate with it do not have to be worried about its presentation layer.

The following figure depicts how the **File Browser Integration** works when extended to a real scenario. The *File Browser Interface* sends an HTTP request to the *Connector*. The *Connector* is working with *CMS services*, and further, retrieving the content from the *CMS repository*. When the content is ready (either by searching or by browsing), the *Connector* sends the XML response back to the *File Browser Interface*.



In this case, *Connector* is the main file to be implemented regarding the server-side integration with the *File Browser*. The *Connector* should consider the following tasks:

- Receiving the file manager's requests
- Executing operations such as creating and listing folders and files
- Building the XML response in the right format and syntax
- Receiving and handle file uploads from the *File Browser*

Employing the WYSIWYG editor in portlets

We can employ the WYSIWYG HTML text editor in portlets, where the content inputs are in need. By default, the Liferay portal uses the WYSIWYG editor to build content such as blog entries, forum topics, Wiki articles, journal articles, and so on.

Employing the WYSIWYG editor in the Web Content portlet

Here is an example that shows how to add the WYSIWYG editor to build content. Refer to the code in `/portal/portal-web/docroot/html/portlet/journal/edit_article_content_xsd_el.jsp`:

```

<c:if test='<%= elType.equals("text_area") %>'>
<script type="text/javascript">
    function <portlet:namespace />
        initEditor <%= count.getValue() %>() {
            return "<%= UnicodeFormatter.toString(elContent) %>";
        }
</script>

```

```
<liferay-ui:input-editor name='<%= renderResponse.getNamespace() +  
    "structure_el" + count.getValue() +  
    "_content" %>'  
editorImpl='<%= EDITOR_WYSIWYG_IMPL_KEY %>'  
toolbarSet="liferay-article"  
initMethod='<%= renderResponse.getNamespace() +  
    "initEditor" + count.getValue() %>'  
onChangeMethod='<%= renderResponse.getNamespace() +  
    "editorContentChanged" %>'  
height="250"  
width="600" />  
</c:if>
```

The code above specifies the `liferay-ui:input-editor` tag and toolbar set `liferay-article` for the content inputs. It also provides the JavaScript function `initEditor`.

Using Liferay display tag

You can find the tag definition in `/portal/util-taglib/src/META-INF/liferay-ui.tld`. You can also find implementation of the `input-editor` tag in `/portal/portal-web/docroot/html/taglib/ui/input_editor/page.jsp`.

Adding the WYSIWYG editor in a custom portlet

Here is an example for the Annual Reports portlet—a custom portlet. It has a title in the top left, an image (or a video) and text in the middle, and a link at the bottom left. Thus, we can use the FCKeditor to build the content: a title, an image or a video, a body and a link, and so on. The following is a sample code that shows how to add the WYSIWYG editor for the portlet:

```
<form method="post" name="    <fieldset>  
        <input type="hidden" name="description" />  
        <table border=0  
            width="80%"  
            cellpadding=5%  
            cellspacing=10%  
            style="margin-left:15px">  
            <tr>  
                <td width="20%" valign="top">Description</td>  
                <td width="80%"><liferay-ui:input-editor width="100%" /></td>  
            </tr>  
            <tr>  
                <td width="20%"></td>  
                <td width="80%">
```

```

<input type="button" value="Save" onClick="create()"/>
<input type="button" value="Back" onClick="back()"/>
</td>
</tr>
</table>
</fieldset>
</form>

```

The code above uses the `liferay-ui:input-editor` tag and JavaScript functions, for example `create` and `back`.

When do we use the WYSIWYG editor?

It is pretty good that we could edit the HTML text with content-rich flashes. When we build articles, we need to embed not only text, images, and links to documents, but also need content-rich flashes such as SWF, slideshow, video, game, video list, video queue, playlist, and so on. This solution provides robust ability and high flexibility to create media-rich articles.

But normally, content-rich flashes are static in the articles. Once the content-rich flashes are inserted into the HTML text, they become static HTML objects. That is, only specific content-rich flash instances exist in the article. For example, suppose that a content creator added a game Elmo's Birthday into the About Elmo article. Now, the end user can only play the Elmo's Birthday game (optionally, plus its related games) in the About Elmo article and has no chance to play the other games, for example zoe sings a Song (which does not belong to the related games of Elmo's Birthday).

We need to implement the content-rich flashes as standalone portlets when required. For instance, the game player will take any game as an input. That is, once the game player is added in one page, it can play any game regardless of what the end user selects. Similarly, the playlist player will take any playlist as an input. These portlets could be developed in Plugins SDK.

Summary

This chapter began by introducing how to configure the WYSIWYG editor, quickly deploy the updates, and upgrade the WYSIWYG editor. Then it addressed how to change the templates and formats based on different web sites such as bookpubstreet.com and bookpubworkshopcom. Moreover, it introduced how to customize FCKeditor to make images, links, videos, games, video queues, video lists, and playlists a part of content (for example, web content). Finally, it introduced how to use the WYSIWYG editor efficiently.

In the next chapter, we're going to customize CMS and WCM.



This material is copyright and is licensed for the sole use by Matt Bedsaul on 27th May 2009
1 Microsoft Way, , Redmond, , 98052

7

Customizing CMS and WCM

The web sites www.bookpubstreet.com and www.bookpubworkshop.com are made up of many pages. Each page contains a bunch of content—either the basic assets (for example, images, documents, videos, games, slideshows, video queues, and playlists), or complex articles (for example, journal articles, blog entries, forum topics, and Wiki articles).

CMS includes a set of portlets—Document Library and Image Gallery—to aggregate and manage images and documents. WCM includes the Web Content portlet to create and publish articles, as well as article templates and structures; the Web Content Display portlet to publish an article; the Web Content List portlet to display a dynamic list of all journal articles for a given community; the Asset Publisher portlet to publish any piece of content; nested portlets to drag and drop portlets into other portlets; XSL content, and much more.

This chapter will first discuss how to dynamically manage the terms of use with a journal article. Next, it will address how to construct featured content with article structure and article template. Accordingly, the extension of the Web Content List portlet will be introduced to display articles that have a different look and feel. This chapter will especially present the way to build articles with multiple image icons, polls, related content, recently added content, and so on. Finally, it will discuss how to use and extend CMS and WCM.

By the end of this chapter, you will have learned how to:

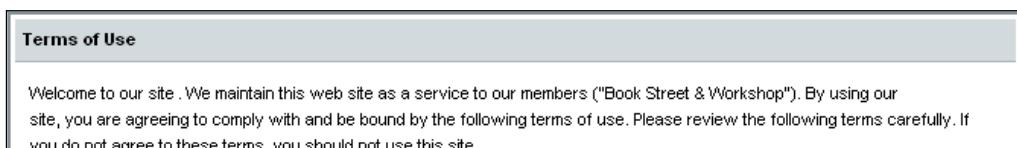
- Manage the Terms of Use portal dynamically
- Construct featured content
- Customize the Web Content List portlet
- Customize the Asset Publisher portlet
- Build dynamic articles with recently added content and related content
- Build dynamic articles with polls
- Extend CMS and WCM

Managing Terms of Use dynamically

First of all, let's consider simple customization of Terms of Use. Liferay portal provides the ability to force all users to accept some terms of use text before using the portal for the first time. Default text is included within the portal. But in most of the installations where this feature is used, this text will need to be customized. First, we will discuss how to customize the **Terms of Use** page statically—change the JSP file directly. Then we will address how to customize it dynamically—represent this page as an article (web content) and change it in runtime.

Customizing static Terms of Use

Let's do a simple customization on top of the portal. When a user logs in to the portal for the first time, the **Terms of Use** page is displayed as shown in the following screenshot. The user may not log in until the **I Agree** button is clicked. Suppose that you want to change the **Terms of Use** page with the content: **Welcome to our site . We maintain this web site as a service to our members ("Book Street & Workshop")**.



In order to customize the **Terms of Use** page, we need to override it in Ext in the following manner:

1. Create a folder portal under the /ext/ext-web/docroot/html/ portal.
2. Copy the terms_of_use.jsp file from the /portal/portal-web/docroot/html/portal/ folder to the /ext/ext-web/docroot/html/portal folder.
3. Modify this file between <c:otherwise> and </c:otherwise> to Welcome to our site. We maintain this web site as a service to our members ("Book Street & Workshop"), and save it.

You have modified the **Terms of Use** page successfully. Similarly, you can change a lot of pages in the portal, for example change_password.jsp, error.jsp, login_create_account.jsp, login_forgot_password.jsp, login.jsp, portlet_error.jsp, and so on. You can find these possible pages that you can modify under /portal/portal-web/docroot/html/portal.

Obviously, the portal allows us to force all users to accept the terms of use before using the portal for the first time. Default text is included with the portal, and this text may need to be customized, which is differs from case to case. A simple way to change this text is to modify the JSP file as stated above. Note that this is probably good only for simple sites. For more complex sites, it is required to allow changing the text of the **Terms of Use** page dynamically. Using a journal article would be the best choice when the terms of use change frequently, and moreover, there is a need to have its text translated to several languages.

Building dynamic Terms of Use

It would be useful to allow the text of the **Terms of Use** page to change dynamically. For example, using a journal article is helpful when the terms of use change frequently and there is a need to have its text translated to several languages.

Here we're going to use a journal article to manage the **Terms of Use** page. First, let's create a journal article named TERMS-OF-USE with the modified content (between <c:otherwise> and </c:otherwise>) as mentioned in the previous section. When creating the article, you need to give a name that is easy to remember, for example TERMS-OF-USE.

Now check the group ID \$GROUP_ID when editing the current community, and the article ID \$ARTICLE_ID of journal article TERMS-OF-USE when editing it. Add the following lines at the end of /ext/ext-impl/src/portal-ext.properties:

```
terms.of.use.journal.article.group.id=$GROUP_ID  
terms.of.use.journal.article.id=$ARTICLE_ID
```

The preceding code shows the default group ID and article ID of the **Terms of Use** page. Note that the default text will be used if no journal article is specified. That's it. Once the article was created, you could use the journal article TERMS-OF-USE. In the `terms_of_use.jsp` file in the `/ext/ext-web/docroot/html/portal/` folder, you can find the following code, which uses the above properties:

```
<c:when test="<%=(PropsValues.TERMS_OF_USE_JOURNAL_ARTICLE_GROUP_ID > 0) && Validator.isNotNull(PropsValues.TERMS_OF_USE_JOURNAL_ARTICLE_ID)%>">
    <liferay-ui:journal-article groupId="<%=(PropsValues.TERMS_OF_USE_JOURNAL_ARTICLE_GROUP_ID %)" articleId="<%=(PropsValues.TERMS_OF_USE_JOURNAL_ARTICLE_ID %)" />
</c:when>
```

As shown in the code above, `groupId` should be the identifier of the community when the article was created. `articleId` should be the value of article ID when searched by TERMS-OF-USE and should show search results in the Web Content portlet.

By default, web sites such as `bookpubstreet.com` and `bookpubworkshop.com` need this page. This feature can easily be turned off by adding the following line at the end of the `/ext/ext-impl/src/portal-ext.properties` file:

```
terms.of.use.required=false
```

Constructing featured content

How do we construct featured content? Featured content is made up of a set of features. Each feature consists of an image, a title, an abstract, a link, and a set of icons. Only one feature is selected in front at a time. As shown in the following screenshot, the first feature with highlighted icon 1 has an image **Sim**, a title **Sim Wins Awards**, an abstract **The Indian ...**, and a link **Read More >>**:



If you click on the icon **2**, it will show the second feature. You can imagine the look and feel of the second feature. Similarly, if you click on the icon **3**, it will display the third feature. As shown in the following screenshot, the third feature with highlighted icon **3** has an image of **Cookie Monster**, a title **Cookie Visits Colbert**, an abstract **Cookie Monster ...**, and a link **Read More >>**:



Of course, you can have any number of features for the featured content—that is, the number is configurable. At the same time, featured content must be a general journal article—end users could update it in runtime easily. The entire look and feel, including the number of features, can be modified directly through the article template.

How to implement featured content? Here we're going to customize the Web Content Display portlet and use the framework journal article (that is, Web Content), structure, and article template. The purpose of using article templates is to make the update of featured content both configurable and dynamic.

Customizing the Web Content Display portlet

Liferay portal provides a portlet named Web Content Display (journal content) to publish any article from the Journal CMS on a portal page. It can be arranged on a page with the convenient drag-and-drop feature. It also has an ability to configure an article to be displayed. Unfortunately, it cannot dynamically accept any article as a parameter and further, there are no customized session attributes to be transferred from one portlet to another. In this section, we're going to customize the Web Content Display portlet named **Ext Web Content Display** that accepts parameters and supports a set of customized session attributes. Let's build the Ext Web Content Display portlet.

Creating the Ext Web Content Display portlet

Similar to building the Ext Communities and Ext Manage Pages portlets , we could clone the Web Content Display portlet as Ext Web Content Display. Thus, we need to configure the Ext Web Content Display portlet in both `portlet-ext.xml` and `liferay-portlet-ext.xml` files first. Then we can set title mapping in the `Language-ext.properties` file and moreover, add the portlet to the Book category in the `liferay-display.xml` file. Finally, specify the Struts actions and forward paths in the `struts-config.xml` and `tiles-defs.xml` files, respectively.

The following is the main process to build the Ext Web Content Display portlet:

1. Locate the `portlet-ext.xml` file in the `/ext/ext-web/docroot/WEB-INF/` folder and open it.
2. Add the following lines between `</portlet>` and `</portlet-app>` and save it:

```
<portlet>
    <portlet-name>extJournalContent</portlet-name>
    <display-name>Ext Journal Content</display-name>
    <portlet-class>
        com.liferay.portlet.StrutsPortlet
    </portlet-class>
    <init-param>
        <name>view-action</name>
        <value>/ext/journal_content/view</value>
    </init-param>
    <expiration-cache>0</expiration-cache>
    <supports>
        <mime-type>text/html</mime-type>
    </supports>
    <supports>
        <mime-type>application/vnd.wap.xhtml+xml</mime-type>
    </supports>
    <resource-bundle>
        com.liferay.portlet.StrutsResourceBundle
    </resource-bundle>
    <security-role-ref>
        <role-name>power-user</role-name>
    </security-role-ref>
    <security-role-ref>
        <role-name>user</role-name>
    </security-role-ref>
</portlet>
```

3. Locate the `liferay-portlet-ext.xml` file in the `/ext/ext-web/docroot/WEB-INF/` folder and open it.

4. Add the following lines after the `<!-- Custom Portlets -->` line and save it:

```
<portlet>
    <portlet-name>extJournalContent</portlet-name>
    <struts-path>ext/journal_content</struts-path>
    <configuration-action-class>
        com.portlet.journalcontent.action.ConfigurationActionImpl
    </configuration-action-class>
    <use-default-template>false</use-default-template>
    <restore-current-view>false</restore-current-view>
</portlet>
```

5. Locate the `liferay-display.xml` file at `/ext/ext-web/docroot/WEB-INF/` and open it.

6. Add the following line after the `<portlet id="extLayoutManagement" />` line and save it:

```
<portlet id="extJournalContent" />
```

7. Similarly, add the following line before the line `javax.portlet.title.EXT_1=Reports` in `/ext/ext-impl/src/content/Language-ext.properties`:
- ```
javax.portlet.title.extJournalContent=Ext Web Content Display
```

8. Locate `struts-config.xml` at `/ext/ext-web/docroot/WEB-INF/` and open it.

9. Add the following lines after the `<action-mappings>` line and save it:

```
<!-- Ext Web Content Display -->
<action path="/ext/journal_content/view"
 type="com.ext.portlet.journalcontent.action.ViewAction">
 <forward name="portlet.ext.journal_content.view"
 path="portlet.ext.journal_content.view" />
</action>
```

10. Locate the `tiles-defs.xml` file in the `/ext/ext-web/docroot/WEB-INF/` folder and open it.

11. Add the following lines after the line `<tiles-definitions>` and save it:

```
<!-- Ext Web Content Display -->
<definition name="portlet.ext.journal_content.view"
 extends="portlet"> <put name="portlet_content"
 value="/portlet/journal_content/view.jsp" />
</definition>
```

Very well! You have created an Ext Web Content Display portlet in Ext. Now, let's go further and add a view action for this portlet.

## Building a view action

Let's build a view action using the following steps:

1. Create a package named `com.ext.portlet.journalcontent.action` in the `/ext/ext-impl/src` folder.
2. Create a Java file `ViewAction.java` in the `com.ext.portlet.journalcontent.action` package and open it.
3. Add the following lines in this file and save it:

```
<public class ViewAction extends WebContentAction {
 public ActionForward render(ActionMapping mapping, ActionForm
 form, PortletConfig portletConfig, RenderRequest renderRequest,
 RenderResponse renderResponse) throws Exception {
 PortletPreferences preferences = renderRequest.
 getPreferences();
 ThemeDisplay themeDisplay = (ThemeDisplay) renderRequest.
 getAttribute(WebKeys.THEME_DISPLAY);
 long groupId = ParamUtil.getLong(renderRequest, "groupId");
 if (groupId < 1) {
 groupId = GetterUtil.getLong(
 preferences.getValue("group-id", StringPool.BLANK));
 }
 String articleId = ParamUtil.getString(renderRequest,
 "articleId");
 String templateId = ParamUtil.getString(renderRequest,
 "templateId");
 if (Validator.isNull(articleId)) {
 articleId = GetterUtil.getString(preferences.getValue
 ("article-id", StringPool.BLANK));
 templateId = GetterUtil.getString(preferences.getValue
 ("template-id", StringPool.BLANK));
 }
 String featuredContentPosition = ParamUtil.getString
 (renderRequest, "featuredContentPosition");
 if (featuredContentPosition != null &&
 featuredContentPosition.length() == 1)
 renderRequest.setAttribute("featuredContentPosition",
 featuredContentPosition);
 else
 renderRequest.setAttribute("featuredContentPosition", "1");
 String viewMode = ParamUtil.getString
 (renderRequest, "viewMode");
 String languageId = LanguageUtil.getLanguageId(renderRequest);
 int page = ParamUtil.getInteger(renderRequest, "page", 1);
 String xmlRequest = PortletRequestUtil.toXML
 (renderRequest, renderResponse);
```

```

JournalArticleDisplay articleDisplay = null;
if ((groupId > 0) && Validator.isNotNull(articleId)) {
 articleDisplay = JournalContentUtil.getDisplay
 (groupId, articleId, templateId, viewMode,
 languageId, themeDisplay, page, xmlRequest);
}
if (articleDisplay != null) {
 renderRequest.setAttribute(WebKeys.JOURNAL_ARTICLE_DISPLAY,
 articleDisplay);
} else {
 renderRequest.removeAttribute(WebKeys.JOURNAL_
 ARTICLE_DISPLAY);
}
return mapping.findForward
 ("portlet.ext.journal_content.view");
}
}

```

As shown in the code above, the `ViewAction` class specifies a parameter and a session attribute as `featuredContentPosition`. If the `featuredContentPosition` parameter is null or invalid, set the session attribute `featuredContentPosition` with the default value 1. Otherwise, transfer the value of the `featuredContentPosition` parameter to the value of the `featuredContentPosition` session attribute. Note that if the importing classes are ignored, you need to add them in the Eclipse IDE as well.

While deploying the Ext Web Content Display portlet, you should see it under the Book category.

## Setting up structure and template

Now, we should set up a structure and template for our featured content. **Structure** is an **XML (Extensible Mark-up Language)** definition of the dynamic parts of journal articles. These parts may be a text, a text box, a text area (HTML), an image, an Image Gallery, a Document Library, a Boolean flag (true or false), a selection list, a multiple selection list, or a link to a page. Actually, the structure is a specific XML schema.

**Template** (or Web Template) is a pattern to rapidly generate and mass-produce web pages that are associated to a structure. A template defines the layout of journal articles and determines how the content items will be arranged. Let's build a structure and template for the featured content.

## Building a structure

According to the use case mentioned earlier, you should build an article structure named `FEATURE_CONTENT`. Just create a structure with `ID`, `Name`, and `Description` with the value `FEATURE_CONTENT`. Then you need to click on **Launch Editor**, add the following lines, and save it:

```
<root>
 <dynamic-element name='image_1'
 type='image_gallery'
 repeatable='false' >
 </dynamic-element>
 <dynamic-element name='text_1'
 type='text_area'>
 </dynamic-element>
 <dynamic-element name='image_2'
 type='image_gallery'
 repeatable='false'>
 </dynamic-element>
 <dynamic-element name='text_2'
 type='text_area'>
 </dynamic-element>
 <dynamic-element name='image_3'
 type='image_gallery'
 repeatable='false'>
 </dynamic-element>
 <dynamic-element name='text_3'
 type='text_area'>
 </dynamic-element>
</root>
```

The code above specifies three pairs of image and text. The image has type `image_gallery` and text has type `text_area`.

## Preparing the icon images

As mentioned earlier, a set of images are used for navigation buttons: 1, 2, and 3. Each button needs two icon images, both selected and unselected. You can upload all related images into the Image Gallery, and then find `$SELECTED_ID` and `$UNSELECTED_ID` for these images. The following are sample image URLs:

```
/image/image_gallery?img_id=$SELECTED_ID
/image/image_gallery?img_id=$UNSELECTED_ID
```

An image URL, which has a pattern like `/image/image_gallery?img_id=ID`, provides the ability to download an image dynamically. Note that the IDs are assigned dynamically. Thus, you may have different IDs for the same image when you upload them into your Image Gallery. But you can use the same pattern to refer any images.

## Building a template

When the `FEATURE_CONTENT` structure is ready, you can create a template with `ID`, `Name`, and `Description` having the value `FEATURE_CONTENT`. You need to select the `FEATURE_CONTENT` structure, uncheck the **Cacheable** checkbox, and select **Language Type VM**—that is, **Velocity**. Then you need to click on **Launch Editor**, add the following lines in this template, and save it:

```
#set ($currentURL = $request.render-url.substring(0, $request.render-
url.indexOf("?")))
#set ($pos = $request.attributes.featuredContentPosition)
#set ($path = $currentURL + "?p_p_id=extJournalContent&p_p_
lifecycle=0&_extJournalContent_featuredContentPosition=""")
#set ($currentPATH_1 = $path + 1)
#set ($currentPATH_2 = $path + 2)
#set ($currentPATH_3 = $path + 3)
#if($pos == 1)

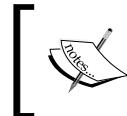
| | | | | |
|---|---|---------------|--|---|
| | <table cellpadding="0" cellspacing="0" width="100%"> <tr> <td colspan="2"> \$text_1.data </td> </tr> <tr> <td width="200px"></td> <td> </td> </tr> </table> | \$text_1.data | | </td> <td> |
| \$text_1.data | | | | |
| </td> <td> | | | | |


```

```

</td>
</tr>
</table>
</td>
</tr>
</table>
#end
#if($pos == 2) /* see details in attached code */# #end
#if($pos == 3) /* see details in attached code */#
#end
```

As shown in the code above, the `FEATURE_CONTENT` template gets the current URL, attribute value of `featuredContentPosition`, and the path first. Then it displays the image and text based on where it is—that is, the attribute value of `featuredContentPosition`. In addition, it specifies buttons icons and links too.



Velocity is a Java-based template engine. It permits web page designers to refer to methods defined in the Java code. Refer to <http://velocity.apache.org>.



## Building featured content articles

We have built a structure and a template for our featured content. Now we are ready to build featured content articles. As we know, an article is the actual content of the web page that is associated with a structure and a template. Each piece of content is populated with actual texts and images. Each article is integrated with two status of workflow, either approved or not approved, and has an abstract involving a description and a small image. Meanwhile, each article may have a set of associated dates, for example, display date, expiration date, review date, and so on.

We have built a structure and template for the featured content. Now let's build an example article for the featured content.

## Preparing images

According to the `FEATURED_CONTENT` structure, three images are required: `Sim`, `Elmo in Military Families`, and `Cookie Monster`. You can upload these images into Image Gallery, and then find `$IMAGE_ID` for these images. For example, the following are sample image URLs:

```
/image/image_gallery?img_id=$IMAGE_ID
```

Similarly, an image URL, which has a pattern like `/image/image_gallery?img_id=$IMAGE_ID`, provides the ability to download an image dynamically.

## Building an article with images and text

We have built a structure and template. We have also uploaded images for featured content articles. Now let's build an example featured content article. You can enter Name as FEATURED\_CONTENT, and select the FEATURED\_CONTENT structure and the FEATURED\_CONTENT template.

Following the FEATURED\_CONTENT structure, you can provide input text for Text 1, Text 2, and Text 3, respectively. Note that you need to add text styles in the WYSIWYG editor. Then just input the images for Image 1, Image 2, and Image 3, respectively. When you are ready, click on the **Save and Approve** button.

Congratulations! You have successfully built featured content articles.

## Customizing the Web Content List portlet

Liferay portal provides the Web Content List portlet to display a dynamic list of all articles for a given community. Using this portlet, we can have a list of articles that include the top articles by creation date, publication date, title, or other criteria. This list of articles will be updated dynamically when new articles are added. Before displaying a list of articles, you may configure the Web Content List portlet by selecting Community, Article Type, Display URL, Display per page, Order by Column, Order by Type, and so on.

Obviously, this portlet has its own limitations. On the one hand, this portlet provides only one view that may not satisfy customized requirement. As shown in the following screenshot, we want to show articles belonging to article type **Press Release** in the **Press Room** page. At the same time, we want to show the date and title of article from article type: **Press Release**. Further, we want to show these articles by date grouping, **Past 12 Months**, **2008**, **2007**, **2006**, and **View All**. Only **11** articles get displayed on each page, and pagination is supported as well.

The screenshot shows the 'Press Room' portlet interface. At the top, it says 'Press Room'. Below that is a dropdown menu labeled 'Show:' with the value '2008'. A dropdown arrow is open, revealing options: 'Past 12 Months', '2008' (which is selected and highlighted in blue), '2007', '2006', '2005', '2004', 'View All', and 'Sep 29, 2008'. To the right of the dropdown, there is a list of articles. The first article is 'Bird's Adventure' (date: Oct 15, 2008). The second is 'Bird's Adventure (in Spanish)' (date: Oct 15, 2007). The third is 'to RTVE In Spain' (date: Oct 14, 2006). The fourth is 'ing Gets Interactive' (date: Oct 13, 2005). The fifth is 'ce President, Global Production' (date: Sep 29, 2004). The sixth is 'Abby in Wonderland and Street on Treehouse' (date: Sep 25, 2008). At the bottom of the portlet, there is a pagination control: 'Page 1 of 2' with a dropdown arrow, and navigation links: 'First', 'Previous', 'Next', and 'Last'.

On the other hand, this portlet can't accept parameters (for example, article type) dynamically. There is only one article type that you can select by defaulted or you may not select any article type (showing all articles belonging to any types). But in fact, it should provide an ability to display articles dynamically for any article types, for example News and Updates. That is, the portlet should dynamically accept article type as a parameter.

In the following sections we're going to customize the Web Content List portlet. At the same time, we'll address how to provide a different view and how to add parameters dynamically.

## Constructing the Ext Web Content List portlet

Similar to the Ext Web Content Display portlet, we're going to build a portlet named Ext Web Content List. We will configure it in both `portlet-ext.xml` and `liferay-portlet-ext.xml` files, set title mapping in the `Language-ext.properties` file, add it to the Book category in the `liferay-display.xml` file, and specify Struts actions and forward paths in `struts-config.xml` and `tiles-defs.xml` files, respectively. First, let's build the Ext Web Content Display portlet as follows:

1. Locate `portlet-ext.xml` file in the `/ext/ext-web/docroot/WEB-INF/` folder and open it.
2. Add the following lines between `</portlet>` and `</portlet-app>` and save it:

```
<portlet>
 <portlet-name>extJournalArticles</portlet-name>
 <display-name>Ext Journal Articles</display-name>
 <portlet-class>
 com.liferay.portlet.StrutsPortlet
 </portlet-class>
 <init-param>
 <name>view-action</name>
 <value>/ext/journal_articles/view</value>
 </init-param>
 <expiration-cache>0</expiration-cache>
 <supports><mime-type>text/html</mime-type></supports>
 <resource-bundle>
 com.liferay.portlet.StrutsResourceBundle
 </resource-bundle>
 <security-role-ref>
 <role-name>power-user</role-name>
 </security-role-ref>
 <security-role-ref>
 <role-name>user</role-name>
 </security-role-ref>
</portlet>
```

3. Locate the `liferay-portlet-ext.xml` file in the `/ext/ext-web/docroot/WEB-INF/` folder and open it.

4. Add the following lines after the line `<!-- Custom Portlets -->` and save it:

```
<portlet>
 <portlet-name>extJournalArticles</portlet-name>
 <struts-path>ext/journal_articles</struts-path>
 <configuration-action-class>
 com.liferay.portlet.journalarticles.action.
 ConfigurationActionImpl
 </configuration-action-class>
 <use-default-template>false</use-default-template>
 <restore-current-view>false</restore-current-view>
</portlet>
```

5. Locate the `liferay-display.xml` file in the `/ext/ext-web/docroot/WEB-INF/` folder and open it.

6. Add the following line after the line `<portlet id="extJournalContent" />` and save it:

```
<portlet id="extJournalArticles" />
```

7. Similarly, add the following line before the line `javax.portlet.title.EXT_1=Reports` in `/ext/ext-impl/src/content/Language-ext.properties`:  
`javax.portlet.title.extJournalArticles=Ext Web Content List`

8. For struts path, add the following lines after the line `<action-mappings>` of `/ext/ext-web/docroot/WEB-INF/struts-config.xml`:

```
<!-- Ext Web Content List -->
<action path="/ext/journal_articles/view"
 type="com.ext.portlet.journalarticles.action.ViewAction">
 <forward name="portlet.ext.journal_articles.view"
 path="portlet.ext.journal_articles.view" />
</action>
```

9. For forward path, add the following lines after the line `<tiles-definitions>` of `/ext/ext-web/docroot/WEB-INF/tiles-defs.xml`:

```
<!-- Ext Web Content List -->
<definition name="portlet.ext.journal_articles.view"
 extends="portlet">
 <put name="portlet_content"
 value="/portlet/ext/journal_articles/view.jsp" />
</definition>
```

Excellent! You have successfully constructed the Ext Web Content List portlet.

## Building a view action

We have created the Ext Web Content List portlet successfully. Now let's build a view action with a feature so that the portlet can accept parameters and the transfer session attribute. This can be done by using the following steps:

1. Create a `com.ext.portlet.journalarticles.action` package in the `/ext/ext-impl/src` folder.
2. Create a view action `ViewAction.java` file in this package and open it.
3. Add the following lines at the beginning of the `ViewAction.java` file and save it:

```
public class ViewAction extends WebContentAction {
 public ActionForward render(ActionMapping mapping, ActionForm
 form, PortletConfig portletConfig, RenderRequest renderRequest,
 RenderResponse renderResponse)
 throws Exception {
 try {
 PortletPreferences preferences = renderRequest.
 getPreferences();
 long groupId = GetterUtil.getLong(
 preferences.getValue("group-id",
 StringPool.BLANK));
 String articleType = ParamUtil.getString(renderRequest,
 "articleType");
 if(articleType != null)
 renderRequest.setAttribute("articleType", articleType);
 else
 renderRequest.setAttribute("articleType",
 PropsUtil.get("ext.journal_artilces.article_type"));
 GroupLocalServiceUtil.getGroup(groupId);
 return mapping.findForward
 ("portlet.ext.journal_articles.view");
 }
 catch (NoSuchGroupException nsge) {
 return mapping.findForward("/portal/portlet_not_setup");
 }
 }
}
```

As shown in the code above, the `ViewAction` class specifies both, the parameter and the session attribute, as the name `articleType`. If the `articleType` parameter is null or invalid, it sets the session attribute `articleType` with a default value. Otherwise, it transfers the value of the `articleType` parameter to the value of the session attribute `articleType`.

## Setting up the view page

Finally, we need to set up the view of the Ext Web Content List portlet.

### Adding custom article types

In order to display a list of article types and add a default article type, we need to add the following lines at the end of the `portal-ext.properties` file:

```
journal.article.types=announcements,blogs,general,news,press-
release,updates,article-content,article-tout
ext.journal_articles.article_type=press-release
```

The code above tells the portal that there is a list of article types—announcements, blogs, general, news, press-release, updates, article-content, and article-tout. It also sets the default article type as press-release.

In order to display article types properly, we need to set a message for article types updates and article-content at the end of the `/ext/ext-impl/src/content/language-ext.properties` file.

```
article-content=Article Content
article-tout=Article Tout
updates=Updates
```

The code above specifies that the article type article-content will be shown as Article Content, and the article type updates will be shown as Updates.

### Consuming custom article types

Finally, we should prepare JSP files for the Ext Web Content List portlet. To do so, first create a folder named `journal_articles` in the `/ext/ext-web/docroot/html/portlet/ext` folder.

Next, create a JSP file `init.jsp` in the `/ext/ext-web/docroot/html/portlet/ext/journal_articles` folder and add the following lines at its beginning:

```
<%@ include file="/html/portlet/journal_articles/init.jsp" %>
if(type == null || type.length() == 0){
 type = (String) renderRequest.getAttribute("articleType");
}
```

The code above adds default values for article type.

Now we need to create the JSP file `view.jsp` in the `/ext/ext-web/docroot/html/portlet/ext/journal_articles` folder, and add the following lines in it:

```
<%@ include file="/html/portlet/ext/journal_articles/init.jsp" %>
<% String articleId = ParamUtil.getString(request, "articleId");
 double version = ParamUtil.getDouble(request, "version"); %> <c:
choose>
<c:when test="<% Validator.isNull(articleId) %>">
<% PortletURL portletURL = renderResponse.createRenderURL();
 if (pageURL.equals("normal")) {
 portletURL.setWindowState(WindowState.NORMAL);
 }
 else { portletURL.setWindowState(WindowState.MAXIMIZED); }
 portletURL.setParameter("struts_action",
 "/ext/journal_articles/view");
 PortletURL articleURL = PortletURLUtil.clone(portletURL,
 renderResponse);
 ArticleSearch searchContainer = new ArticleSearch
 (renderRequest, portletURL);
 searchContainer.setDelta(pageDelta);
 searchContainer.setOrderByCol(orderByCol);
 searchContainer.setOrderByType(orderByType);
 searchContainer.setOrderByComparator(orderByComparator);
 List headerNames = searchContainer.getHeaderNames();
 headerNames.clear();
 searchContainer.setOrderableHeaders(null);
 ArticleSearchTerms searchTerms = (ArticleSearchTerms)
 searchContainer.getSearchTerms();
 searchTerms.setGroupId(groupId);
 searchTerms.setType(type);
 if (Validator.isNotNull(structureId)) {
 searchTerms.setStructureId(structureId);
 }
 searchTerms.setStatus("approved");
%>
<h1>Press Room</h1>
<p>Press Contact:
 press@bookpubworkshop.com
</p>
<%@ include file="/html/portlet/journal/
 article_search_results.jspf" %>
<% List resultRows = searchContainer.getResultRows();
 DateFormatDateTime = new SimpleDateFormat("MMM dd, YYYY");
 for (int i = 0; i < results.size(); i++) {
 JournalArticle article = (JournalArticle)results.get(i);
 article = article.toEscapedModel();
```

```

ResultRow row = new ResultRow(article, article.getArticleId()
 + EditArticleAction.VERSION_SEPARATOR +
 article.getVersion(), i);
String rowHREF = null; articleURL.setParameter("groupId",
 String.valueOf(article.getGroupId()));
articleURL.setParameter("articleId", article.getArticleId());
articleURL.setParameter("version", String.valueOf
 (article.getVersion()));
rowHREF = articleURL.toString();
String target = null;
TextSearchEntry rowTextEntry = new TextSearchEntry
 (SearchEntry.DEFAULT_ALIGN,
 SearchEntry.DEFAULT_VALIGN,
 article.getArticleId(),
 rowHREF, target, null);

// Display date
rowTextEntry = (TextSearchEntry) rowTextEntry.clone();
rowTextEntry.setName(dateFormatDateTime.format
 (article.getDisplayDate()));
row.addText(rowTextEntry);

// Title
rowTextEntry = (TextSearchEntry) rowTextEntry.clone();
rowTextEntry.setName(article.getTitle());
row.addText(rowTextEntry);

// Add result row
resultRows.add(row); } %>
<liferay-ui:search-iterator searchContainer="<%=
 searchContainer %>" />

</c:when>
<c:otherwise>
 <!-- display article, see attachment -->
</c:otherwise>
</c:choose>

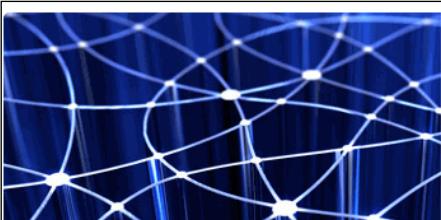
```

As shown in the code above, the JSP file `view.jsp` adds the header first. Then it adds a customized view with the `MMM dd, yyyy` date-time format. That's simple! You have customized the Ext Journal Articles portlet. When you deploy the portlet, you should see it under the Book category. In brief, we have implemented a view for the Ext Web Content List portlet, except the search feature. Only simple search was provided, but you can extend the code and implement the search feature as well.

## Customizing the Asset Publisher portlet

The Asset Publisher portlet allows us to publish any piece of content in a portal, either through a set of publishing rules or by manual selection. These pieces of content could be Bookmark entries, blog entries, Wiki articles, Document Library documents, Image Gallery images, journal articles (web content), Message Board threads, and so on.

The Asset Publisher portlet has its own limitation. On the one hand, it only provides a fixed view, which may not satisfy customized requirements. As shown in the following screenshot, we're going to display product-related articles by tags and current date. That is, there are only four articles related to both the EDI tag and the current date with a descending order. For each page, there is only one article with a large-size image, title, and a description displayed in the lefthand column, and three related articles with title and a description in the righthand column. Moreover, only a limited description will be displayed with a shortcut for reading more (>>).



**Starting EDI with Asian Partners**  
Asia is a region that no business can afford to ignore, but getting started doing business in Asia, particularly setting up EDI with suppliers, can seem >>

**Latest Analyst Research Reveals the True Costs**  
New analyst research puts a dollar figure on the price of traditional document processing methods. You won't want to go back any time soon. As a Sterling >>

**My EDI Product III**  
New analyst research puts a dollar figure on the price of traditional document processing methods. You won't want to go back any time soon. As a Sterling >>

**My EDI Product IV**  
My EDI Product IV

On the other hand, the Asset Publisher portlet can't dynamically accept parameters such as tags. In the following sections, we're going to customize the Asset Publisher portlet with above feature as well as accept dynamic tags, and customize its view.

## Adding a large-size image and a medium-size image in Web Content

By default, a journal article will have a set of basic metadata: title, text, abstract, small-size image, type, versions, tags, Display Date, Expiration Date, and Review Date. These metadata may be sufficient for general purpose. But for requirements mentioned above, we're going to add additional metadata: large-size image, medium size image, original author, original author job title, and original created date. Further, this additional metadata is required only by some type of articles, for example Article Content. For other type of articles, such as Article Tout and General, the basic metadata would be sufficient.

One simple but powerful approach is to represent the additional metadata with an article structure. First of all, let's create a structure for these additional metadata as follows:

1. In the Web Content portlet, click on the **Structure** tab and the **Add Structure** button.
2. Enter **ID**, **name**, and **description** as a value POLL.
3. Click on the **Launch Editor** button, add the following lines, and save it:

```
<root>
 <dynamic-element name='author_article_created'
 type='text'>
 </dynamic-element>
 <dynamic-element name='author_job_title'
 type='text'>
 </dynamic-element>
 <dynamic-element name='author'
 type='text'>
 </dynamic-element>
 <dynamic-element name='large_image'
 type='image'>
 </dynamic-element>
 <dynamic-element name='medium_image'
 type='image'>
 </dynamic-element>
 <dynamic-element name='text'
 type='text_area'>
 </dynamic-element>
</root>
```

As shown in this code, the structure POLL specifies original author, original author job title, and original created date as text. It also defines large-size image and medium size image as image. By default, it keeps original article content with a name **text** and type **text\_area**.

Now, let's create an article template for the additional metadata as follows:

1. In the Journal portlet, click on the **Template** tab and the **Add Template** button. Enter the **ID**, **name**, and **description** as ARTICLE\_CONTENT.
2. Uncheck the **Cacheable** checkbox and select the **POLL** structure.
3. Click on the **Launch Editor** button, add the following lines, and save it:

```
<table width="100%" cellpadding="0" cellspacing="0">
 <tr>
 <td style="width:500px; ;height: 100%; padding: 0px 0px;
vertical-align:top;">
```

```
<table style="border: 0px solid #cccccc;
 border-collapse: separate;
 height: 100%; width: 100%;">
<tr>
 <td style="text-align:left; vertical-align:top;
 color: #00001F; font-size: 12pt;
 font-weight: bold; ">
 $reserved-article-title.data
 </td>
</tr>
<tr>
 <td style="text-align:left; vertical-align:top;">
 <h4>$author.Data, $author_job_title.Data <h4>
 </td>
</tr>
<tr>
 <td style="text-align:left; vertical-align:top;">
 <h4>$author_article_created.Data<h4>
 </td>
</tr>
<tr>
 <td style="text-align:left;
 vertical-align:top;padding:0px;">
 $text.Data
 </td>
</tr>
</table>
</td>
<td style="width:15;" > </td>
<td style="background-color: #ffffff;text-align:center;
 vertical-align:top; width:200px; padding:1px;">
 #if ($medium_image.Data && $medium_image.Data != "")

 #end
 </td>
</tr>
</table>
```

As shown in the code above, the ARTICLE\_CONTENT template first gets reserved-article-title, original author, original author job title, and original created date by Velocity variables and displays them. Then it displays the article content as \$text.data and medium-size image as \$medium\_image.Data.

You must have noticed that there are a lot of CSS pieces and VM (velocity) variables. You can simply change the look and feel in order to display the article content in different ways by updating these CSS pieces and VM variables dynamically.

## Building the Ext Asset Publisher portlet

Similar to build the Ext Web Content List portlet, we can build the Ext Asset Publisher portlet based on the Asset Publisher portlet. This can be done by configuring it in both `portlet-ext.xml` and `liferay-portlet-ext.xml` files, setting title mapping in the `Language-ext.properties` file, adding the portlet to the Book category in the `liferay-display.xml` file, and specifying struts action and forward paths in the `struts-config.xml` and `tiles-defs.xml` files, respectively.

The following is the main process to build the Ext Asset Publisher portlet:

1. Locate the `portlet-ext.xml` file in the `/ext/ext-web/docroot/WEB-INF/` folder.
2. Add the following lines between `</portlet>` and `</portlet-app>` and save it:

```
<portlet>
 <portlet-name>extAssetPublisher</portlet-name>
 <display-name>Ext Asset Publisher</display-name>
 <portlet-class>
 com.liferay.portlet.StrutsPortlet
 </portlet-class>
 <init-param><name>view-action</name>
 <value>/ext/asset_publisher/view</value></init-param>
 <expiration-cache>0</expiration-cache>
 <supports><mime-type>text/html</mime-type></supports>
 <resource-bundle>
 com.liferay.portlet.StrutsResourceBundle
 </resource-bundle>
 <security-role-ref>
 <role-name>power-user</role-name>
 </security-role-ref>
 <security-role-ref>
 <role-name>user</role-name>
 </security-role-ref>
</portlet>
```

3. Locate the `liferay-portlet-ext.xml` file in the `/ext/ext-web/docroot/WEB-INF` folder.
4. Add the following lines after the line `<!-- Custom Portlets -->` and save it:

```
<portlet>
 <portlet-name>extAssetPublisher</portlet-name>
 <struts-path>ext/asset_publisher</struts-path>
 <configuration-action-class>
 com.liferay.portlet.assetpublisher.
 action.ConfigurationActionImpl
 </configuration-action-class>
```

```
</configuration-action-class>
<use-default-template>false</use-default-template>
<restore-current-view>false</restore-current-view>
</portlet>
```

5. Locate the `liferay-display.xml` file in the `/ext/ext-web/docroot/WEB-INF/` folder and open it.
6. Add the following line after the line `<portlet id="extJournalArticles" />` and save it:  
`<portlet id="extAssetPublisher" />`
7. Similarly, add the following line before the line `javax.portlet.title.EXT_1=Reports` in `/ext/ext-impl/src/content/Language-ext.properties`:  
`javax.portlet.title.extAssetPublisher=Ext Asset Publisher`
8. Locate the `struts-config.xml` file in the `/ext/ext-web/docroot/WEB-INF/` folder and open it.
9. Add the following lines after the line `<action-mappings>` and save it:  
`<!-- Ext Asset Publisher -->
<action path="/ext/asset_publisher/view"
 type="com.ext.portlet.assetpublisher.action.ViewAction">
 <forward name="portlet.ext.asset_publisher.view"
 path="portlet.ext.asset_publisher.view" />
</action>
<action path="/ext/asset_publisher/view_content"
 forward="portlet.asset_publisher.view_content" />`
10. Locate the `tiles-defs.xml` file in the `/ext/ext-web/docroot/WEB-INF/` folder and open it.
11. Add the following lines after the line `<tiles-definitions>` and save it:  
`<!-- Ext Asset Publisher -->
<definition name="portlet.ext.asset_publisher.view"
 extends="portlet">
 <put name="portlet_content"
 value="/portlet/ext/asset_publisher/view.jsp" />
</definition>`

Cool! You have built the Ext Asset Publisher portlet in the Ext portlet. Let's go further and add a view action for the Ext Asset Publisher portlet.

Let's build the view action as follows:

1. Create a package named `com.ext.portlet.assetpublisher.action` in the `/ext/ext-impl/src` folder. Then create a Java file `ViewAction.java` in this newly created package and open it.

2. Add the following lines at the beginning of this Java file and save it:

```
public class ViewAction extends WebContentAction {
 public ActionForward render(ActionMapping mapping, ActionForm
 form, PortletConfig portletConfig, RenderRequest
 renderRequest, RenderResponse renderResponse)
 throws Exception {
 return mapping.findForward
 ("portlet.ext.asset_publisher.view");
 }
}
```

3. Finally, create an `asset_publisher` folder in the `/ext/ext-web/docroot/html/portlet/ext` folder. Next, create a JSP file `init.js` in `/ext/ext-web/docroot/html/portlet/ext/asset_publisher`, add the following line, and save it:

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

Let's copy the JSP files `view.jsp`, `view_dynamic_list.jspf`, `view_dynamic_list_asset.jsp`, and `view_display.jsp` from `/portal/portal-web/docroot/html/portlet/asset_publisher` to `/ext/ext-web/docroot/html/portlet/ext/asset_publisher`. To do so, create a folder `display` under the `/ext/ext-web/docroot/html/portlet/ext/asset_publisher` folder and copy the JSP file `abstracts.jsp` from `/portal/portal-web/docroot/html/portlet/asset_publisher/display` to `/ext/ext-web/docroot/html/portlet/ext/asset_publisher/display`.

In order to customize the view of the Asset Publisher portlet, we need to update the above JSP files. In the `view.jsp` file, update the `/html/portlet/asset_publisher/init.jsp` string with the `/html/portlet/ext/asset_publisher/init.jsp` string, and update the `/html/portlet/asset_publisher/view_dynamic_list.jspf` string with the `/html/portlet/ext/asset_publisher/view_dynamic_list.jspf` string. In `view_dynamic_list.jspf`, update the `/html/portlet/asset_publisher/view_dynamic_list_asset.jspf` string with the `/html/portlet/ext/asset_publisher/view_dynamic_list_asset.jspf` string.

Further, in the `view_dynamic_list_asset.jsp` file, update the `/html/portlet/asset_publisher/view_display.jspf` string with the `/html/portlet/ext/asset_publisher/view_display.jspf` string. In the `view_display.jspf` file, replace the `/html/portlet/ext/asset_publisher/display` string `/`. In the `abstracts.jsp` file, update the `/html/portlet/asset_publisher/init.jsp` string with a string `/html/portlet/ext/asset_publisher/init.jsp`, and update the `/asset_publisher/view_content` string with a `/ext/asset_publisher/view_content` string.

Cool! You have successfully built the Ext Asset Publisher portlet based on the Asset Publisher portlet.

## Extending view with tags

Finally, let's extend the view in order to input tags as parameters. Suppose that four tags are supported: tag1, tag2, tag3, and tag4.

## Configuring tags

Before using tags, we need to specify the default tags in the `portal-ext.properties` file. To do so, simply add the following line at the end of this file:

```
default tag
ext.default.tag.value=edi
```

## Setting up default tags

Accordingly, we need to add the code to retrieve tags, and moreover, to consume tags in the `/ext/ext-web/docroot/html/portlet/ext/asset_publisher` folder.

```
%@page import="com.liferay.portlet.journal.service.
 JournalArticleImageLocalServiceUtil"%>
<% ClassName cn = ClassNameLocalServiceUtil.getClassName
 (JournalArticle.class.getName());
 classNameId = cn.getClassNameId();
 if (classNameId > 0) {
 classNameIds = new long[] {classNameId};
 }
 else {
 classNameIds = new long[0];
 }
 String tagValue1 = ParamUtil.getString(request, "tag1");
 String tagValue2 = ParamUtil.getString(request, "tag2");
 String tagValue3 = ParamUtil.getString(request, "tag3");
 String tagValue4 = ParamUtil.getString(request, "tag4");
 List<String> tags = Collections.synchronizedList
 (new ArrayList<String>());
 if(tagValue1!=null && tagValue1.length()>0)
 tags.add(tagValue1);
 if(tagValue2!=null && tagValue2.length()>0)
 tags.add(tagValue2);
 if(tagValue3!=null && tagValue3.length()>0)
 tags.add(tagValue3);
 if(tagValue4!=null && tagValue4.length()>0)
 tags.add(tagValue4);
 entries = new String[tags.size()];
 for(int i=0; i< tags.size(); i++) entries[i]=tags.get(i);
 String defaultTag = PropsUtil.get("ext.default.tag.value");
 if(entries.length == 0){
```

```

 entries = new String[1]; entries[0]= defaultTag;
 }
 delta = GetterUtil.getInteger(preferences.getValue
 ("delta", StringPool.BLANK), 4);
%>
<%! // method for large-size image and medium-size image
public String getData(String image, JournalArticle art, String
languageId) throws Exception {
 long imgId = JournalArticleImageLocalServiceUtil.
 getArticleImageId(art.getGroupId(),
 art.getArticleId(), art.getVersion(),"",image,"");
 String imgURL = "/image/journal/article?img_id=" + imgId;
 return imgURL;
};
%>

```

As shown in the code above, the JSP file `init.jsp` gets tags as parameters and sets them as default tags if applicable. It also sets the default value 4 for `delta`. At the same time, a method to find the image URL of a large-size image and a medium-size image is also specified.

## Updating views

Now let's update views. First, just add a comment on the `Add Asset ...` form in the `view.jsp` file, which is in the `/ext/ext-web/docroot/html/portlet/ext/asset_publisher` folder as follows:

```
<!-- <%@ include file="/html/portlet/ext/asset_publisher/add_asset.
jspf" %> -->
```

Now we need to update the table-based view for the requirement we have mentioned earlier, which is in the `view_dynamic_list_asset.jsp` file in the `/ext/ext-web/docroot/html/portlet/ext/asset_publisher` folder. To do so, add the following lines before the first `<%` of the `view_dynamic_list_asset.jsp` file:

```

<table cellspacing="0" cellpadding="0" border="0" width="100%">
<% if(entryIds.length == 0) { %>
<tr valign="top">
<td style="padding-top: 0px; padding-left: 10px;
 valign: top; height: 250px; ">
 No results! The tag inputs are invalid!
</td>
</tr>
<%

```

Add } else after the first <% of the `view_dynamic_list_asset.jsp` file and add `</table>` at its end. Moreover, replace the line `<%@ include file="/html/portlet/ext/asset_publisher/view_display.jspf" %>` with the following lines:

```
<%
 if(assetIndex==0) { %>
 <tr valign="top">
 <td style="border-right: 1px solid #ABABAC; padding-top: 5px; "
 rowspan="<% results.size()==1 ? "2" :results.size()%>"
 width="50%" valign="top">
 <liferay-util:include page="/html/portlet/ext/asset_publisher
 /view_display.jspf" />
 </td>
 </tr>
 <% }
 else { %>
 <tr valign="top">
 <td style="padding-top: 0px; padding-left: 10px; valign: top; ">
 <liferay-util:include page="/html/portlet/ext/asset_publisher
 /view_display.jspf" />
 </td>
 </tr>
 <% }
%>
```

As shown in the code above, the JSP file `view_dynamic_list_asset.jsp` specifies a table first. Then it specifies an article in the first column with a specific view, and other articles in the second column sharing a different view.

Now we need to update the table-based view in the `bstract.jsp` file, which is in the `/ext/ext-web/docroot/html/portlet/ext/asset_publisher/display` folder. To do so, replace the code between (included) if (`articleDisplay.isSmallImage()`) { and (not included) `sb.append(articleDisplay.getDescription());` with the following lines:

```
viewFullContentURL.setParameter("assetId",
 articleDisplay.getArticleId());
viewFullContentURL.setParameter("type",
 AssetPublisherUtil.TYPE_CONTENT);
viewURL = viewFullContentURL.toString();
if(assetIndex==0){
 sb.append("<div>");
 sb.append("<img src=\"\"");
 sb.append(getData("large_image", articleDisplay, languageId)
 + "\" /></div>");
 sb.append("");
}
```

As shown in the preceding code, the JSP file `abstract.jsp` uses `getData` method to get an image URL of a large-size image. That's it. You have successfully implemented the requirement as mentioned in the beginning of this section.

## Building dynamic articles with recently added content and related content

As mentioned earlier, the Ext Web Content Display portlet provides an ability to publish any article from the WCM on a portal page. It can comprise of most of the content on `bookpub.com`, which can be arranged on a page with the convenient drag-and-drop feature of the portal. You can configure an article that will be displayed in this portlet. Moreover, it can accept a customized parameter and the session attribute `featuredContentPosition`. It would be useful if this portlet could dynamically display journal articles through its asset ID.

As shown in the following screenshot, for the use case *Display Articles by Asset ID*, we're going to dynamically display journal articles by their asset IDs. There are two scenarios that require the asset ID. On the one hand, the asset ID is used directly in the Ext Asset Publisher and Asset Publisher portlets, as these portlets are popular for publishing any piece of content. On the other hand, the asset ID is used indirectly (forming asset ID with group ID and article ID) in the Ext Web Content List and Web Content List portlets, as these portlets are popular for publishing web content (journal articles). Thus, we could extend the Ext Web Content Display portlet and let it have the ability to dynamically publish journal articles by asset ID. In addition, it would be useful to associate a journal article with a medium-size image, ratings, and comments as well.

Moreover, we're going to publish any piece of journal articles via VM services. Suppose that we have a lot of journal articles for the types **Article Tout** and **Press Releases**. For the use cases *Article Tout in Random* and *Recent Press Release*, we're going to publish these articles either recently or in random. Refer to the next screenshot.

For the use case *Article Tout in Random*, the content editor just selects journal articles belonging to the **Article Tout** type, which will be displayed. After that, the article template will dynamically display selected articles with a small-size image, title, and an abstract. In order to display content of the selected journal article, a link on both the small-size image and the title is also specified with the Ext Web Content Display portlet and an asset ID.

The use case *Recent Press Release* dynamically publishes recently added journal articles belonging to the Article Tout type, with created date and title. These press releases should especially be displayed in a descending order. Further, only the article template should be used for this use case. That is, content editor can change the look and feel of recently added journal articles with the **Press Releases** type in runtime.

	<b>Our Mission</b>	<b>Press Releases</b>
	We are committed to the principle that all children deserve a chance to learn and grow; to be prepared for school; to better understand the world and each other; to think, dream and discover; to reach their highest potential.	Oct 15, 2008 <a href="#">One World One Sky - Big Bird's Adventure</a>
		Oct 15, 2008 <a href="#">One World One Sky - Big Bird's Adventure (in Spanish)</a>

Especially when displaying one journal article, we expect to display related content as well. Let's see the following screenshot for the **Related Content** use case. When a journal article, for example **Latest Analyst Research ...**, is displayed, three related articles (**On-Demand Primer**, **Starting EDI with Asian Partners**, and **Liferay Book Review:...**) are also displayed. Related content means that these four journal articles at least contain the **EDI** tag. At the same time, the first three related articles are found through advanced search based on tag **EDI** and current date with descending order.

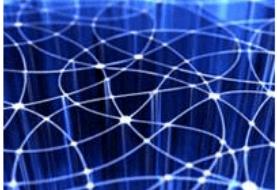
**Latest Analyst Research Reveals the True Costs**  
David Berger, Manager Global, Sept. 19, 2008  
**Find Out How Much You Are Saving With Electronic Trading. And How Much More You Can Save.**

As a Sterling Collaboration Network (SCN) customer, you already know your company saves money by processing transactions electronically over the network as opposed to using the traditional methods of phone, fax and email. But exactly how much savings are you creating? How much profit can you drive? To answer these questions, and to fully articulate the value you are creating for your organization, you need to understand the cost of manually processing an order and an invoice at your company.

To help you, we've collected the latest research on manual processing costs and also included a methodology you can use to break down your company's processing costs (see "Try This Yourself").

**Step 1 – Determining manual processing costs: What the research says**  
According to research conducted by Forrester Consulting for Sterling Commerce, the median industry cost of manual processing of a purchase order rings in at \$10.10, based on a range with a low of \$1.88, and a high of \$18.31. Aberdeen Group's new research places the cost at just over \$20.00, while eC-BP.org's estimate for manual order processing is in the middle at \$17.00.

Invoicing and payment remittance cost more than purchase orders, on average \$12.35



How about the IA product?

a. Good  
 b. OK

**Submit**

**Related Content:**  
[Starting EDI with Asian Partners](#)  
[On-Demand Primer](#)  
[Liferay Book Review: Liferay Portal Enterprise Intranets](#)

The content of the journal article **Latest Analyst Research ...** is displayed within the Ext Web Content Display portlet plus an asset ID. Further, related articles are displayed with a title and a link of the portlet and the asset ID. When you click on the link, for example **On-Demand Primer**, the selected journal article **On-Demand Primer** along with its related content will be displayed. By the way, we could put all of this together as one view. This would include name, content, author, original author job title, original created date, medium-size image, polls, and related content. For more information on how to implement this feature, refer to the next section.

## Displaying journal articles through asset ID

As mentioned earlier, the Ext Web Content Display portlet has the ability to publish any article from the WCM on a portal page. You can configure an article that will be displayed in this portlet. More importantly, it can accept parameters and support a set of customized session attribute such as `featuredContentPosition`.

Now we expect that the Ext Web Content Display portlet can dynamically display journal articles through an asset ID. Let's build this feature as follows:

1. Locate the Java file `ViewAction.java` under the `com.ext.portlet.journalcontent.action` package in the `/ext/ext-impl/src/` folder, and open it.
2. Add the following lines before the line `if (articleDisplay != null) {` in this file and save it:

```
String assetId = ParamUtil.getString(renderRequest, "assetId");
if(assetId != null && assetId.length() > 0){
 TagsAsset asset = TagsAssetLocalServiceUtil.getAsset(new
 Long(assetId).longValue());
 long classPK = asset.getClassPK();
 JournalArticleResource articleResource = JournalArticleResource
 LocalServiceUtil.getArticleResource(classPK);
 articleDisplay = JournalContentUtil.getDisplay(
 articleResource.getGroupId(),
 articleResource.getArticleId(),
 null, null, languageId, themeDisplay);
 renderRequest.setAttribute(WebKeys.JOURNAL_ARTICLE_DISPLAY,
 articleDisplay);
}
else
```

As shown in the preceding code, `ViewAction` receives a value for the `assetId` parameter. If it is valid, `ViewAction` receives `JournalArticleResource` and `ArticleDisplay` by `assetId`. Finally, it sets the value of `ArticleDisplay` as the default value of the `WebKeys.JOURNAL_ARTICLE_DISPLAY` attribute. That's it. After you deploy the portlet `Ext Web Content Display`, you can display journal articles through an asset ID in this portlet.

## Showing touts with article ID

Let's consider the use case *Article Tout in Random*. A tout is a collection of a journal article's partial metadata. It is made up of a small-size image, title, abstract description, and a link to a journal article. Normally, a set of touts are displayed as a group. For example, there are three journal articles: **Our Mission**, **Our Process**, and **Who We Are**. This use case needs to show the three touts together. For the first one, it shows a small-size image, title, abstract description of the journal article **Our Mission**, and a link on both title and small-size image to the journal article **Our Mission**. Obviously, it would be better to represent touts through an article template. Thus, content editors can update touts in runtime. The following is the main process to implement this feature:

- Add a Velocity service to get the journal article through a group ID and an article ID
- Build a structure and template of touts to consume above services
- Build a general article that contains touts

## Adding Velocity services

In order to get the journal article and asset ID through a group ID and an article ID, we need to add Velocity services. First, let's prepare `ExtVelocityToolUtil` as follows:

1. Create a package named `com.ext.portal.service` in the `/ext/ext-impl/src` folder; create an interface `ExtVelocityToolService`, and add service interface methods.

```
public JournalArticle getJournalArticle(String groupId, String
 articleId);
public String getAssetId(String groupId, String articleId);
```

2. Create a package named `com.ext.portal.service.impl` in the `/ext/ext-impl/src` folder; create a class `ExtVelocityToolServiceImpl`, and then add service methods implementation as follows:

```
public String getAssetId(String groupId, String articleId) {
 String assetId = "";
```

```

try{
 JournalArticle journalArticle = JournalArticleLocal
 ServiceUtil.getArticle(new
 Long(groupId), articleId);
 TagsAsset asset = TagsAssetLocalServiceUtil.
 getAsset(JournalArticle.class.getName(),
 journalArticle.getResourcePrimKey());
 assetId = String.valueOf(asset.getAssetId());
}
catch (Exception e){ return assetId; }
return assetId;
}
public JournalArticle getJournalArticle(String groupId, String
articleId) {
 JournalArticle journalArticle = null;
 try{
 journalArticle = JournalArticleLocalServiceUtil.getArticle(new
 Long(groupId).longValue(), articleId);
 }
 catch (Exception e){ return journalArticle; }
 return journalArticle;
}
public String getAssetId(String groupId, String articleId) {
 String assetId = "";
 try{
 JournalArticle journalArticle = JournalArticleLocalServiceUtil
 .getArticle(new
 Long(groupId).longValue(),
 articleId);
 TagsAsset asset = TagsAssetLocalServiceUtil.getAsset
 (JournalArticle.class.getName(),
 journalArticle.getResourcePrimKey());
 assetId = String.valueOf(asset.getAssetId());
 }
 catch (Exception e){ return assetId; }
 return assetId;
}

```

3. Create a package named `com.ext.portal.util` in the `/ext/ext-impl/src` folder. Next create a `ExtVelocityToolUtil` class and consume the above service and method as follows:

```

public String getAssetId(String groupId, String articleId){
 return _extVelocityToolService.getAssetId(groupId, articleId);
}
public JournalArticle getJournalArticle(String groupId,
String articleId){

```

```
 return _extVelocityToolService.getJournalArticle
 (groupId,articleId);
 }
 public static ExtVelocityToolService
 getExtVelocityToolService(){
 return _extVelocityToolService;
 }
 public void setExtVelocityToolService(ExtVelocityToolService
 extVelocityToolService){
 _extVelocityToolService = extVelocityToolService;
 }
 private static ExtVelocityToolService _extVelocityToolService;
```

As shown in the code above, `ExtVelocityToolUtil` specifies the methods `getJournalArticle` (getting journal article through group ID and article ID) and `getAssetId` (getting asset ID through group ID and article ID).

Accordingly, we need to wire them all together. To do that, add the following lines after the line `<beans>` in the `/ext/ext-impl/src/META-INF/ext-spring.xml` file:

```
<bean id="com.ext.portal.service.ExtVelocityToolService"
 class="com.ext.portal.service.impl.ExtVelocityToolServiceImpl" />
<bean id="com.ext.portal.util.ExtVelocityToolUtil"
 class="com.ext.portal.util.ExtVelocityToolUtil">
 <property name="extVelocityToolService"
 ref="com.ext.portal.service.ExtVelocityToolService" />
</bean>
<bean id="com.ext.portal.util.ExtVelocityToolUtil.velocity"
 class="org.springframework.aop.framework.ProxyFactoryBean"
 parent="baseVelocityUtil">
 <property name="target"
 ref="com.ext.portal.service.ExtVelocityToolService" />
</bean>
```

As shown in the code above, the XML file specifies beans `ExtVelocityToolService`, `ExtVelocityToolUtil`, and `ExtVelocityToolUtil.velocity`.

## Building touts structure and template

We have successfully added Velocity services. Now let's create touts structure and template. Accordingly, create three articles with the Article Tout type and the names Our Mission, Our Process, and Who-We-Are, respectively. Then find articles IDs for these journal articles; for example, 13201, 13206, and 13211. Note that in runtime, these articles IDs would be different. Further, let's create a structure TOUT as follows:

1. In the Journal portlet, click on the **Structure** tab.
2. Click on the **Add Structure** button and provide input for ID, name, and description as TOUT.

- Click on the **Launch Editor** button, add the following lines, and save it:

```
<root>
 <dynamic-element name='Article_Touts' type='multi-list'>
 <dynamic-element name='13201' type='Our-Mission'>
 </dynamic-element>
 <dynamic-element name='13206' type='Our-Process'>
 </dynamic-element>
 <dynamic-element name='13211' type='Who-We-Are'>
 </dynamic-element>
 </dynamic-element>
 </root>
```

The code above specifies a multiple list with journal articles: Our Mission, Our Process, and Who-We-Are. By the way, this is a static solution—specifying a multiple list in a structure. That is, if you want to change this multiple list, you have to manually update these journal articles IDs and titles in the structure. A dynamic solution is also available if you specify this multiple list as Template Node. For more details about dynamic solution, for example polls, refer to the next section.

```
#set ($extVelocityToolUtil = $utilLocator.findUtil('com.ext.portal.util.ExtVelocityToolUtil')) #set ($currentURL = $request.render-url.substring(0, $request.render-url.indexOf("?")))
#set ($touts = $Article_Touts.options)


```

```
<tr>
 <td style="border-bottom: 0.05em dotted #ADADAF;
 padding-top: 2px; height: 10px; " colspan="2">
 </td>
</tr>
<tr>
 <td style="height: 10px; " colspan="2"></td>
</tr>
#end
</table>
```

As shown in the code above, the `TOUT` template consumes Velocity services: `extVelocityToolUtil getJournalArticle`, and `getAssetId`. It also provides a simple look and feel. Of course, you can update the look and feel any time.

## Building article tout

We have created a tout structure and template successfully. Now we can create an example article **Tout Test I** as follows:

1. In the Journal portlet, click on the **Articles** tab and then click on the **Add Article** button.
2. Provide the input name as **Tout Test I** and select both the structure and the template as **TOUT**.
3. Select multiple values of **Article Touts** and click on the **Save and Approve** button.

That's it. If you preview the **Tout Test I** article, you will see same view as shown in the use case *Article Tout in Random*. If you want to show only two journal articles, you just have to change the values of multiple selections by editing the **Tout Test I** article. If you want to change the small-size image, title, and abstract description of the **Our Process** tout, just update the same metadata as of the related article **Our Process**.

## Listing recently added content

We have implemented the use case *Article Tout in Random*. Now, let's consider the use case *Recent Press Releases*. First, we need to add the following method in the `ExtVelocityToolService` interface:

```
public List<JournalArticle> getRecentArticles(String companyId,
 String groupId, String type, int limit);
```

Now, add an implementation of the `getRecentArticles` method in the `ExtVelocityToolServiceImpl` class as follows:

```

public List<JournalArticle> getRecentArticles(String companyId, String
groupId, String type, int limit) {
 List<JournalArticle> results = Collections.synchronizedList(new
 ArrayList<JournalArticle>());
 String articleId = null; Double version = null;
 String title = null; String description = null;
 String content = null; String[] structureIds = null;
 String[] templateIds = null; Date displayDateGT = null;
 Date displayDateLT = null; Boolean approved = Boolean.TRUE;
 Boolean expired = Boolean.FALSE; Date reviewDate = null;
 boolean andOperator = true; int start = 0;
 int end = limit; String orderByCol = "";
 String orderByType = "";
 boolean orderByAsc = orderByType.equals("asc");
 OrderByComparator obc = new ArticleModifiedDateComparator
 (orderByAsc);
 if (orderByCol.equals("display-date")) {
 obc = new ArticleDisplayDateComparator(orderByAsc);
 }
 try{
 results = JournalArticleLocalServiceUtil.search(
 new Long(companyId).longValue(),
 new Long(groupId).longValue(), articleId, version,
 title, description, content, type, structureIds,
 templateIds, displayDateGT, displayDateLT, approved,
 expired, reviewDate, andOperator, start, end, obc);
 }
 catch (Exception e){ return new ArrayList<JournalArticle>();}
 return results;
}

```

Add the following method to register above method in `ExtVelocityToolUtil`:

```

public List<TagsAsset> getRelatedArticles(String companyId, String
groupId, String articleId, int limit){
 return _extVelocityToolService.getRelatedArticles(companyId,
 groupId, articleId, limit); }

```

The code above gets recent articles by company ID, group ID, article type, and limited articles that need to be returned.

Let's create a structure named `PRESS_RELEASES` with the following content:

```

<root>
 <dynamic-element name='name_' type='text'></dynamic-element>
 <dynamic-element name='url_' type='text'></dynamic-element>
</root>

```

The code above specifies the title as `name_` and the link `URL_` for viewing all press releases.

Now let's create a template named `PRESS_RELEASES` with the following content:

```
#set ($extVelocityToolUtil = $utilLocator.findUtil('com.ext.portal.util.ExtVelocityToolUtil'))
#set ($results = $extVelocityToolUtil.getRecentArticles($companyId, $groupId, "press-release", 10))
<div class="my-background" style=" width: 300px;
 border: 1px solid #cccccc; padding: 4px; text-align: left; ">
 <div class="press-background" style="color: #00004F;
 font-size: 14pt; font-weight: bold; text-align: left; ">
 $name_.data
 <div>

 <table>
 #foreach ($result in $results)
 <tr>
 <td style="color: #00008F; font-size: 8pt; font-weight: normal;
 text-align: left;">
 $result.getModifiedDate()
 </td>
 <td style="width:10px; "></td>
 <td style="color: #00008F; font-size: 8pt;
 font-weight: normal;text-align: left;">
 $result.Title
 </td>
 </tr>
 #end
 </table>
</div>

<div style="color: #00008F; font-size: 9pt; font-weight: normal;
 text-align: center; ">
 View all press releases
</div>
```

As shown in code above, it consumes the velocity service `extVelocityToolUtil` `getRecentArticles`. It also provides a simple look and feel. Definitely, you can update the look and feel any time. Finally, you can create an article `PRESS RELEASES`, which consumes the above structure and template.

## Exhibiting related content

Similarly, we can exhibit related content as well. For the use case related content, first we need to add the following method in the `ExtVelocityToolService` interface:

```
public List<TagsAsset> getRelatedArticles(String companyId, String
 groupId, String articleId, int limit);
```

Now add an implementation of the method `getRelatedArticles` in the class `ExtVelocityToolServiceImpl` as follows:

```
public List<TagsAsset> getRelatedArticles(String companyId, String
groupId, String articleId, int limit) {
 Date now = new Date();
 List<TagsAsset> results = Collections.synchronizedList(new
 ArrayList<TagsAsset>());
 try {
 long resourcePrimaryKey = JournalArticleResourceLocal
 ServiceUtil.getArticleResourcePrimKey
 (GetterUtil.getLong(groupId),
 articleId);
 String[] entries = TagsEntryLocalServiceUtil.getEntryNames
 (JournalArticle.class.getName(),
 resourcePrimaryKey);
 long[] entryIds = TagsEntryLocalServiceUtil.getEntryIds
 (GetterUtil.getLong(companyId), entries);
 long[] notEntryIds = new long[0];
 long[] classNameIds = {PortalUtil.getClassNameId
 (JournalArticle.class.getName())};
 boolean andOperator = false;
 boolean excludeZeroViewCount = false;
 String orderByColumn1 = "modifiedDate";
 String orderByColumn2 = "title";
 String orderByType1 = "DESC";
 String orderByType2 = "ASC";
 results = TagsAssetLocalServiceUtil.getAssets(new Long(groupId),
 classNameIds, entryIds, notEntryIds, andOperator,
 orderByColumn1, orderByColumn2, orderByType1,
 orderByType2, excludeZeroViewCount, now, now, 0, limit + 1);
 }
 catch (Exception e) { return new ArrayList<TagsAsset>(); }
 return results;
}
```

Add the following method to register the method above in `ExtVelocityToolUtil`:

```
public List<TagsAsset> getRelatedArticles(String companyId, String
groupId, String articleId, int limit){
 return _extVelocityToolService.getRelatedArticles(companyId,
 groupId, articleId, limit); }
```

As shown in the code above, `ExtVelocityToolUtil` gets related articles by company ID, group ID, article ID, and limited articles that needed to be returned.

Then, let's create a template named `RELATED_CONTENT` with the following content:

```
#set ($extVelocityToolUtil = $utilLocator.findUtil('com.ext.portal.util.ExtVelocityToolUtil'))
#set ($articleId = $reserved-article-id.Data)
#set ($results = $extVelocityToolUtil.getRelatedArticles($companyId,
$groupId, $articleId, 3))
#set ($assetId = $extVelocityToolUtil.getAssetId($groupId,
$articleId))
#set ($currentURL = $request.render-url.substring(0, $request.render-
url.indexOf("?")))

<div style="border: 1px solid #cccccc;padding:4px; text-align:left;">
 <div><h5>Related Content:</h5></div>
 #foreach ($result in $results) #if($result.getAssetId() !=
$assetId)
 <div>
 <a href="$currentURL?p_p_id=extJournalContent&p_p_lifecycle=0
&_extJournalContent_assetId=$result.getAssetId()">
 $result.Title

 </div>
 #end #end </div>
```

As shown in the code above, the `RELATED_CONTENT` template first receives an article ID by `reserved-article-id` and then it consumes the Velocity service `extVelocityToolUtil getRelatedArticles`. It also provides a simple look and feel. Similar to other templates, you can update the look and feel as you desire.

## Building dynamic articles with polls

As mentioned earlier in the use case *related content*, it was nice that we could not only display related content for a given article, but also add poll for this article. Thus, the user can view an article, select related content, rate the article, provide his or her comments on the article, and most importantly, contribute his or her opinions on the article through polls.

Let's consider how to add a poll to an article via a template. As shown in the following screenshot, when building a structure for the use case *Template Node Poll*, we can use the customized template node **Link to Poll**. That is, we can use the default template nodes: **Text**, **Text Box**, **Text Area (HTML)**, **Image**, **Image Gallery**, **Document Library**, **Boolean Flag**, **Selection List**, **Multi-Selection List**, and **Link to Page**. We can also use the customized template node **Link to Poll**.



Moreover, when building an article, content editors can select a poll question for current article as well. As shown in the following screenshot, when creating a poll in the Polls portlet, a list of questions will be displayed.



Content editors can select a poll question from the polls list. At the same time, we need a feature where we could put all metadata together as one view: name, content, author, original author job title, original created date, medium-size image, polls, and related content. Let's implement this feature.

## Adding template node poll

First of all, we need to add a template node poll in the `TemplateNode.java` Java file. To do so, create a `com.liferay.portlet.journal.util` package in the `/ext/ext-impl/src` folder. Now, copy the Java file `TemplateNode.java` from the `com.liferay.portlet.journal.util` package in the `/portal/portal-impl/src` folder to the package `com.liferay.portlet.journal.util`, which is located in the `/ext/ext-impl/src` folder. Then add the following lines before the last } and save the file:

```
public PollsQuestion getPollsQuestion() {
 if (getType().equals("poll")) {
 try {
 return PollsQuestionLocalServiceUtil.getQuestion(
 Long.parseLong(getData()));
 }
 catch (Exception e) {}
 }
 return null;
}
public List<PollsChoice> getPollsChoices() {
 if (getType().equals("poll")) {
 try {
 return PollsChoiceLocalServiceUtil.getChoices(
 Long.parseLong(getData()));
 }
 }
}
```

```
 catch (Exception e) {}
 }
 return null;
}
public boolean hasVoted(String userId) {
 if (getType().equals("poll")) {
 try {
 PollsVoteLocalServiceUtil.getVote(Long.parseLong(getData()),
 Long.parseLong(userId));
 }
 catch (Exception e) { return false; }
 return true;
 }
 return false;
}
public int getTotalVotes() {
 if (getType().equals("poll")) {
 try {
 return PollsVoteLocalServiceUtil.getQuestionVotesCount(
 Long.parseLong(getData()));
 }
 catch (Exception e) {}
 return 0;
 }
 return 0;
}
public int getChoiceVotes(long choiceId) {
 if (getType().equals("poll")) {
 try {
 return PollsVoteLocalServiceUtil.getChoiceVotesCount(
 choiceId);
 }
 catch (Exception e) {}
 return 0;
 }
 return 0;
}
```

As shown in the code above, `TemplateNode` specifies a set of methods for the template node `poll`: `getPollsQuestion`, `getPollsChoices`, `hasVoted`, `getTotalVotes`, and `getChoiceVotes`. In addition, add a message for the link-to-poll node in `Language-ext.properties`. To do so, add the following line of code at the end of the `Language-ext.properties` file:

```
link-to-poll=Link to Poll
```

## Updating the Web Content portlet with template node poll

We need to update the Web Content portlet in order to support the template node poll. You can prepare JSP files as follows. First, create a journal in /ext/ext-web/docroot/html/portlet folder. Now copy the init.jsp, edit\_structure\_xsd\_el.jsp, edit\_article.jsp and edit\_article\_content\_xsd\_el.jsp files from /portal/portal-web/docroot/html/portlet/journal to /ext/ext-web/docroot/html/portlet/journal.

Then add the following lines after the line `<%@ page import="com.liferay.util.RSSUtil" %>` in the init.jsp file and save it:

```
<%@ page import="com.liferay.portlet.polls.model.PollsQuestion" %>
<%@ page import="com.liferay.portlet.polls.service.
 PollsQuestionLocalServiceUtil" %>
```

As shown in the code above, the JSP file init.jsp imports poll-related model and service: PollsQuestion and PollsQuestionLocalServiceUtil. After that, we need to update the edit\_structure\_xsd\_el.jsp file in order to add the template node poll. To do that, add the following lines between the line `<option <%= elType.equals("link_to_layout") ? "selected" : "" %> value="link_to_layout"><liferay-ui:message key="link-to-layout" /></option>` and the line `</select>` in the edit\_structure\_xsd\_el.jsp file:

```
<option <%= elType.equals("poll") ? "selected" : "" %> value="poll">
 <liferay-ui:message key="link-to-poll" />
</option>
```

Further, we need to update the edit\_article.jsp file by adding the following content after the `(elTypeValue == "link_to_layout")` string:

```
|| (elTypeValue == "poll")
```

Finally, we need to update the edit\_article\_content\_xsd\_el.jsp file to show the available poll questions as a list. To do this, add the following lines before the line `<c:if test='<%= elType.equals("link_to_layout") %>'>`:

```
<c:if test='<%= elType.equals("poll") %>'>
 <% List questions = PollsQuestionLocalServiceUtil.getQuestions
 (portletGroupId.longValue()); %>
 <select id=<portlet:namespace />structure_el<%= count.getValue() %>_content"
 onChange=<portlet:namespace />contentChanged();>
 <option value=""></option>
 <% for (int i = 0; i < questions.size(); i++) {
```

```
PollsQuestion question = (PollsQuestion)questions.get(i);
question = question.toEscapedModel();
%>
<option <%= (elContent.equals(String.valueOf(
 question.getQuestionId()))) ? "selected" : "" %>
 value=<%= question.getQuestionId() %>>
 <%= question.getTitle() %>
</option>
<% } %>
</select>
</c:if>
```

That's it. You have updated the Web Content portlet to support polls. By the way, you can add dynamic article touts selection. It is simple; you just need to update the above JSP files.

## Associating journal articles with polls

We have updated the Journal portlet as well. Now let's update the structure and templates. First, we need to add the template node `poll` in the `POLL` structure. To do so, add the following line before the line `<dynamic-element name='text' type='text_area'></dynamic-element>` in the `POLL` structure:

```
<dynamic-element name='poll' type='poll'></dynamic-element>
```

Then create a template named `POLL` with the `POLL` structure and the following content:

```
#if ($poll.getPollsQuestion())
<div style="border: 0px solid #cccccc; padding: 4px;
 text-align: left;">
#set ($question = $poll.getPollsQuestion().toEscapedModel())
#if ($poll.hasVoted($request.attributes.USER_ID) == false)
<form name="cmspollsfm">
 <div>$question.getDescription()</div>

 <table class="lfr-table">
 #foreach ($choice in $poll.getPollsChoices())
 #set ($choice = $choice.toEscapedModel())
 <tr>
 <td>
 <input name="choiceId" type="radio"
 value="$choice.getChoiceId()" />
 </td>
 <td>
 $choice.getName() .
 </td>
```

```
<td>$choice.getDescription()</td>
</tr>
#end
</table>
<input type="submit" value="Submit" onClick="vote()" />
</form>
#else
<!-- ignore details, find it as out in attachment -->
```

As shown in the code above, the POLL template uses a set of methods of the template node poll: getPollsQuestion, getPollsChoices, hasVoted, getTotalVotes, and getChoiceVotes. Finally, we need to update the ARTICLE\_CONTENT template in order to merge the templates POLL and RELATED\_CONTENT. To do so, add the following lines after the line #end in the template ARTICLE\_CONTENT:

```
#parse ("$journalTemplatesPath/POLL")
#parse ("$journalTemplatesPath/RELATED_CONTENT")
```

Cool! You have successfully built dynamic articles with polls and related content. If you created an article with the Article Content type and tags, the polls and related content will be displayed together with a medium-size image and content.

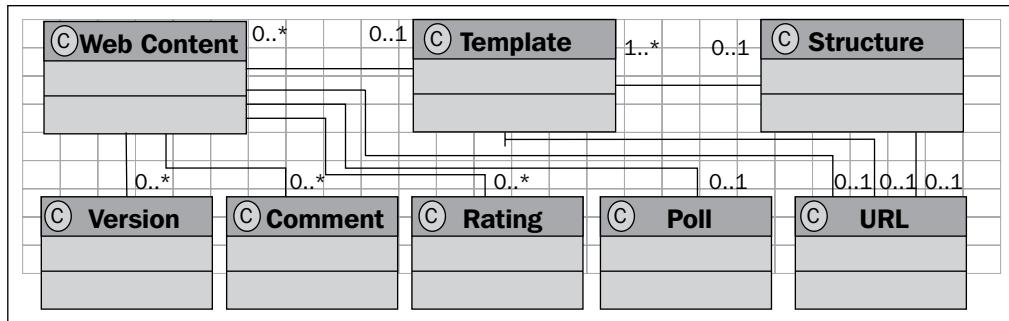
## Extending CMS and WCM

CMS and WCM provide a high ability to manage, integrate, and publish your content. This built-in CMS lets your information take the form of a public web site, a shared workspace, or an enterprise content repository.

## Employing articles, structures, and templates

The following diagram conceptually depicts the relationships among article, template, and structure. Normally, an article may have one associated template. It is possible that an article may not use any template, whereas a template may serve many articles. This is the reason why we can view articles by a given template. Each article may have a poll, a number of comments (that is, posts and replies), and associated ratings. Moreover, each article may have a list of versions. Interestingly, each article has a different status: approved or not approved, expired, or review.

Similarly, a template may have one structure associated, or a template may not employ any structure. Meanwhile, a structure may serve many templates. Thus, on the one hand, we can edit structure by a given template; and on the other, we can view templates and articles by a given structure. Note that each template and structure has a unique URL to be referred to:



## Using journal template—Velocity templates

There are various types of fields available in the structured content, which allow the velocity macros to be used to display structured content. The following is a list of the Velocity macros used for journal articles:

- reserved-article-idreserved-article-version
- reserved-article-title
- reserved-article-create-date
- reserved-article-modified-date
- reserved-article-display-date
- reserved-article-author-id
- reserved-article-author-name
- reserved-article-author-email-address
- reserved-article-author-comments
- reserved-article-author-organization
- reserved-article-author-location
- reserved-article-author-job-title

Moreover, you can use the Velocity macros in `JournalVmUtil` and `VelocityVariables`, `request` (for example, `request.attributes.USER_ID`), `company`, `companyId`, `groupId`, `journalTemplatesPath` (for example `journalTemplatesPath/POLL` and `journalTemplatesPath/RELATED_CONTENT`), `locale`, and `randomNamespace`.

Of course, you can also use customized Velocity macros in your `request.attributes.featuredContentPosition` and `extVelocityToolUtil` article templates as mentioned earlier.

## Enjoying the Web Content search portlet

The Web Content search portlet provides an ability to search journal articles smoothly. Powered by the **Apache Lucene** search engine, search results can be restricted to journal articles.



Apache Lucene is a high-performance, full-featured text search engine library written entirely in Java. Refer to <http://lucene.apache.org>.



To enable the Web Content search portlet, you need to add the following lines in the `portal-ext.properties` file:

```
index.on.startup=true
lucene.analyzer=org.apache.lucene.analysis.SimpleAnalyzer
```

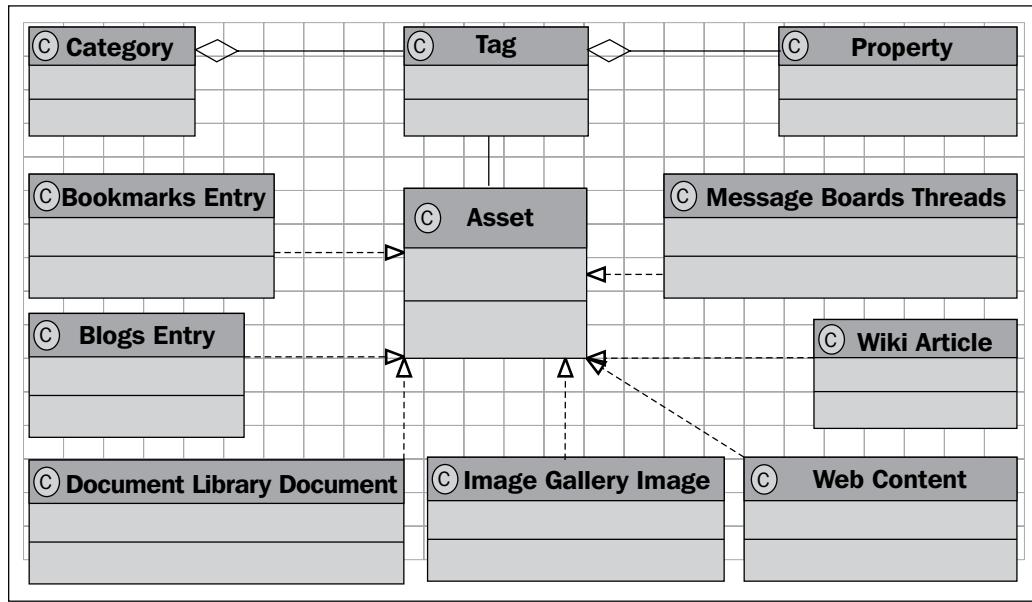
As shown in the code above, the properties file `portal-ext.properties` tells the portal to index the entire library of files on startup. It also sets the default analyzer used for indexing and retrieval.

## Tagging content

Liferay portal tagging system allows you to tag the web content, documents, message board threads, and more. It also allows you to dynamically publish content by tags. Tags provide a way of organizing and aggregating content. Basically, the tag admin determines which tags are available for usage. The users use these tags on their content. Any content (called asset) that is tagged can be grouped or aggregated.

The following figure depicts an overview of tags and contents (that is, assets). A tag may belong to a category. That is, a category may have many tags. When you create a tag, a predefined category or dynamic created category will be assigned to it. In a word, tags are managed and grouped by categories.

A tag may have many properties. Each property is made up of a name and a value. A tag may be associated with content (that is, asset). By using tags, you can tag almost anything: bookmark entries, blog entries, Wiki articles, Document Library documents, Image Gallery images, web content, message board threads, and so on. You can also use these tags to pull content (that is, asset) with the Asset Publisher portlet.



Tags can be used as knowledge for journal articles. That is, you can use tags to classify journal articles with the Article Content type first. You can also use tags to group journal articles. For example, there are two dimensions on journal articles: product family and product category. Product family includes EDI Products, Collaboration Network, and so on; whereas product category involves B2B Simplified, File Transfer Gateway, and so on. When creating journal articles with the Article Content type, you can add tags either to product family or product category. After that, you can show journal articles with tags-based knowledge as mentioned earlier.

## Extending Image Gallery and Document Library

The images dimension is configurable in the Image Gallery portlet. We can set the maximum thumbnail height and width in pixels, and set the dimension of the custom images to 0 to disable creating a scaled image of that size. The following are the sample dimension settings of thumbnail and custom images. You can configure them in the portal-ext.properties file as well:

```
ig.image.thumbnail.max.dimension=150
ig.image.custom1.max.dimension=100
ig.image.custom2.max.dimension=50
```

In addition, we can use the Document Library portlet to save any documents in the JCR repository (for example, Jackrabbit) and use default metadata of Document Library to represent a small set of custom metadata. It would be nice if the dynamic content model could be supported in Document Library. Thus, we could use the Document Library portlet to represent any kind of content types – either zero-content types or normal content types. This could be done via adding two fields – `xmlType` and `xmlMetaData` – in current Document Library. Here, `xmlType` holds current content type, for example `<type>cm:custom-Content-Type</type>`, and `xmlMetaData` holds current values for the current content model.

## Adding Velocity templates in Asset Publisher

It would be good to add Velocity templates for the views of the Asset Publisher portlet. Currently, the views are fixed in JSP files. But you can implement the views through templates. CSS and JavaScript should be available in templates, too. It would be easy for developers to customize Velocity templates (including CSS and JavaScript) directly. The following are the sample implementations: predefine a set of templates (for different views, for example, table, title list, abstracts, and full content) in the /vm folder (or other reasonable folder name) in Asset Publisher portlet, and in the render action, pick up a proper template to generate the view.

## Summary

This chapter first discussed how to manage the terms of use dynamically with a journal article. Then it addressed how to construct featured content by extending the Web Content Display portlet. Accordingly, a customization of Web Content List was introduced as well to display articles in a different look and feel. It especially presented a way to build articles with multiple image icons, rating, comments, polls, related content, recently added content, and so on. Finally, it discussed how to use and extend CMS and WCM. This included journal content search, tags, and relationship among articles, structures, and article templates, CMS extension, and the Asset Publisher portlet extension.

In the next chapter, we're going to discuss how to build a personalized community.

# 8

## Building a Personalized Community

Besides public web sites, it would be nice if we could provide a personal community, that is, My Community, for each registered user. In My Community, users can have a set of public and private pages. For example, in the web site bookpubstreet.com, it would be nice to have ability to build My Community such as My Street, My Activation, and My Book. Personal community is a dynamic feature of Liferay portal. By default, the personal community is a portal-wide setting that will affect all users. Coincidentally, the settings are divided between private and public pages for greater versatility. Moreover, it would be nice to have more features in the personal community, for example, sharing with friends, showing the most popular journal articles, having personalized user comments, managing My Account, and so on.

This chapter will discuss how to build My Community in general and also how to customize and extend this feature. First, it will introduce how to share web sites, pages, and portlets with friends. Then it will address how to set up mostly popular journal articles and view the counter for other assets. It will further discuss how to personalize user comments. It will also talk about how to customize My Account and build My Community with personalized preferences. Finally, it will address best practices to use the personal community.

By the end of this chapter, you will have learned how to:

- Share content with friends
- Set up mostly popular journal articles
- Personalize user comments
- Customize my account
- Build personal community—My Street
- Use personal community efficiently

## Sharing content with friends

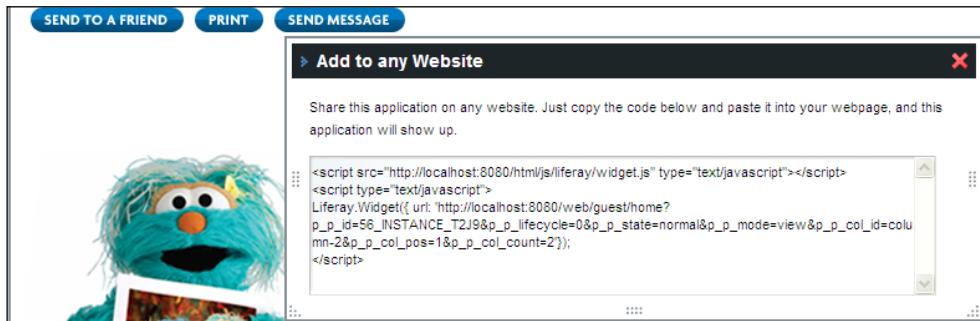
When building a web site, especially a personal community, you may intend to have a feature where users can share content with their friends. The content could be the entire web site (for example, bookpubstreet.com), pages (for example, video), or assets (for example, a video My Own Fairy Tale). The content could be journal articles, Wiki articles, blog entries, and so on. Further, we would also want to share assets with friends as well, for example videos, games, slideshows, video queues, playlists, and so on.

As shown in the following screenshot, we have one page which has a video in playing. Now a user David Berger is playing this video. He is interested in this video and he wants to share this video with his friends as a link. What does he need to do? He just clicks on the **Share** button under this video. A window named **Share** pops up. Then the user enters his/her first name, his/her friend's first name, and his/her friend's email; and then clicks on the **Send** button. His/her friend will receive an email with a link to the web site bookpubstreet, the page video, and the content with the My Own Fairy Tale video. When the friend clicks on the link, he/she will see the same page that David Berger has visited now.



Moreover, we may need to share articles with friends in some other web site (for example, bookpubworkshop.com) as well. For example, the current page has a journal article with three buttons: **SEND TO A FRIEND**, **PRINT**, and **SEND MESSAGE**. It means that this kind of journal article will include three buttons via article templates. When users click on the **SEND TO A FRIEND** button, a window named **Share** will pop up. Users can send an email with the journal article's link to friends. In addition, when users click on the **PRINT** button, a new window with the journal article's preview will appear. Through this feature, users can easily preview the journal articles before printing them.

The ability to send an email to friends with a link is a nice feature. That way, our friends just have to click on the link and they can view the web site, page, or assets. When they are viewing the web site, they may want to go further—sharing the web site, pages, or assets on any web site via the JavaScript code. For doing so, they could just copy the JavaScript code and paste into their web applications and the web site, pages, or assets will show up in their applications.



As shown in screenshot above, the current page has a journal article with a **SEND MESSAGE** button. That is, this kind of journal articles will include the button via article template. When users click on the **SEND MESSAGE** button, a window named **Add to any Website** will pop up with the JavaScript code. The user can share this journal article on any web site. He or she just has to copy the code and paste it into his or her web applications, and this page will show up with the particular journal article. Let's implement these features.

## Building print preview as article template

First of all, we're going to build the print preview feature as article template via the Web Content portlet. To do so, we need to create a structure named **SHARE** with the following content. You can have a different name for the structure; here we are using it just for reference:

```

<root>
 <dynamic-element name='text' type='text_area' repeatable='false'>
 </dynamic-element>
</root>

```

The preceding code shows a structure of journal article. There is only one element `dynamic-element` in the `SHARE` structure, with the name `text` and type `text_area`. We need to create an article template named `SHARE` with the following content. Again, you can have a different name for the template. Here we are using the name `SHARE` just for reference.

```
#set ($articleId = $reserved-article-id.Data)
#set ($articleVersion = $reserved-article-version.Data)
#set ($viewURL = '/c/journal/view_article_content?groupId=' + $groupId
+ '&articleId=' + $articleId + '&version=' + $articleVersion)
<div>$text.data</div>
<div style="padding-top: 10px; padding-bottom: 35px; ">
<a style='text-decoration:none' target='_blank'
 href='$viewURL'>

</div>
```

The code above shows the usage of the VM template variables `articleId` and `articleVersion`. Both of them use journal article reserved elements `reserved-article-id` and `reserved-article-version`, respectively. Moreover, they also use the reserved template variable `groupId`. Based on these variables, this code shows `viewURL` with the `'/c/journal/view_article_content?groupId=' + $groupId + '&articleId=' + $articleId + '&version=' + $articleVersion` value. Especially, the service URL path for viewing article content is `/c/journal/view_article_content?` By the way, the code above shows an image button `btnPrint.gif`. To have it, just place the `/cms_services/images/button/btnPrint.gif` image under the `$CATALINA_HOME/webapps` folder.

If you are creating a journal article via the Web Content portlet with above structure `SHARE` and template `SHARE`, you would see the **PRINT** button at the end of journal article content. If you click on that button, it would bring you to the preview of the current journal article.

## Sharing applications on any web site by widget

We're going to share applications on any web site by widget. As shown in the screenshot previous, it is expected to share applications via JavaScript **jQuery** widget. Let's first experience the `jQuery` method `showWidgetInfo`.

Locate the JavaScript file `portlet_sharing.js` in the `/portal/portal-web/docroot/html/js/liferay` folder and locate the `jQuery` method `showWidgetInfo`. You will see the code like this:

```

showWidgetInfo: function(widgetURL) {
 var portletURL = Liferay.PortletURL.createResourceURL();
 portletURL.setPortletId(133);
 portletURL.setParameter("widgetURL", widgetURL);
 new Expanse.Popup({
 fixedcenter: true,
 header: Liferay.Language.get('add-to-any-website'),
 modal: true, url: portletURL.toString(), width: 550
 });
}

```

The code above shows a JavaScript function `Liferay.PortletSharing.showWidgetInfo` with an input `widgetURL`. The 133 portlet is the portlet name of the `portlet_sharing` portlet. Moreover, it creates a portlet 133, sets the `widgetURL` parameter, and specifies an `Expanse.Popup` pop-up.

Let's then use this JavaScript function `Liferay.PortletSharing.showWidgetInfo` in the article template `SHARE`. To do so, add a line `#set ($widgetURL = $request.render-url)` at the beginning of the `SHARE` template, and add the following lines before the last `</div>`:

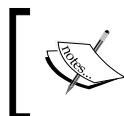
```

<a style='text-decoration: none; '
 href='javascript:Liferay.PortletSharing.showWidgetInfo
 ("$widgetURL")'>


```

The code above shows the usage of the VM template variable `widgetURL`, using reserved element `request.render-url`. Moreover, it also calls JavaScript `javascript:Liferay.PortletSharing.showWidgetInfo` with the `widgetURL` value. Similar to the image of the print preview button, it shows an image button of `btnSendMsg.gif`.

In the pop-up window, you will see the message **Share this application on any website. Just copy the code below and paste it into your web page and this application will show up.** The JavaScript code includes `http://localhost:8080/html/js/liferay/widget.js`, which shows how to insert the link into a page via `iframe`. Of course, you can find the JavaScript `jQuery` file `widget.js` in the `/portal/portal-web/docroot/html/js/liferay/` folder.



`jQuery` is a fast and concise JavaScript Library that simplifies HTML document traversing, event handling, animating, and AJAX interactions for rapid web development. Refer to <http://jquery.com/>.

## Sending emails with sharing links to friends

We have successfully implemented two buttons, **PRINT** and **SEND MESSAGE**, via article templates and JavaScript jQuery. Now let's implement another button via article templates, JavaScript jQuery, and an additional portlet Share. The following are the main steps to implement it:

1. Build a portlet called share
2. Set up view action and email
3. Set up view page with jQuery
4. Prepare jQuery services
5. Build an article template

## Building the Share portlet

Similar to building the Ext Web Content Display portlet, we can build a Share portlet. Configure the Share portlet in both `portlet-ext.xml` and `liferay-portlet-ext.xml` first, and then set title mapping in `Language-ext.properties`. Moreover, add the portlet to the Book category in the `liferay-display.xml` file. Finally, specify Struts actions and forward paths in the `struts-config.xml` and `tiles-defs.xml` files, respectively. In addition, we need to provide the default `email_from` address at the end of the `portal-ext.properties` file as follows:

```
ext.default_email_from.share_with_friends=info@book.com
```

The code above shows that the default `email_from` address is `info@book.com`. Of course, you can configure it in different values.

## Setting up view action and email

We need to set up view action and emails as well. Let's create a Struts view action in the following manner:

1. Create a `com.ext.portlet.sharewithfriend.action` package in the `ext/ext-impl/src` folder.
2. Create a Java file `ViewAction.java` at the `com.ext.portlet.sharewithfriends.action` package and open it.
3. Add the following lines in `ViewAction.java` and save it:

```
public class ViewAction extends PortletAction{
 public void serveResource(ActionMapping mapping, ActionForm
 form, PortletConfig portletConfig, ResourceRequest
 resourceRequest, ResourceResponse resourceResponse)
 throws Exception {
```

```

String path = "/html/portlet/ext/share_with_friends/
view.jsp";
String shareURL = resourceRequest.getParameter("shareURL");
String action = resourceRequest.getParameter("action");
if(action != null && action.equalsIgnoreCase("success")) {
 sendEmail(escape(shareURL));
}
PortletRequestDispatcher portletRequestDispatcher =
portletConfig.getPortletContext().getRequestDispatcher(path);
portletRequestDispatcher.include(resourceRequest,
resourceResponse);
// ignore method sendEmail, see details in attached code
}

```

The code above shows `serveResource` of the Share portlet. If the variable values are entered properly, it will send an email and, moreover, it will bring users to an information page—proving that the action was successful. Otherwise, it will stay on the current page. Note that `import com.ext.*` is available only in the attached code.

## Setting up the view page with jQuery

Now, we need to set up the view page with jQuery JavaScript functions. To do so, use the following steps:

1. Create a folder named `share_with_friends` in the folder `/ext/ext-web/docroot/html/portlet/ext/`.
2. Create a JSP file `view.jsp` in the folder `/ext/ext-web/docroot/html/portlet/ext/share_with_friends` and open it.
3. Add the following lines in `view.jsp` file and save it:

```

<%@ include file="/html/portlet/init.jsp" %>
<%
 String shareURL = ParamUtil.getString(request, "shareURL");
 String action = ParamUtil.getString(request, "action");
%>
<c:choose>
 <c:when test="<% Validator.isNotNull(action) %>">
 <!-- ignore, see details in attached code -->
 </c:when>
 <c:otherwise>
 <div>
 <form method="post" name=<portlet:namespace />fm">
 <input name=<portlet:namespace />sfURL"
 type="hidden" value="<% shareURL %>" />

```

```
<!-- ignore, see details in attached code -->
<table border="0" cellpadding="0" cellspacing="0">
<!-- ignore, see details in attached code -->
<tr>
 <td colspan="2">
 <table border="0" width="100%">
 <tr>
 <td width="45%" style="height:15px;" valign="top" style="padding-top:7px;">
 <div style="padding-top:2px;">
 <a href="#" onClick=<portlet:namespace />
 shareWithFriends(document.
 <portlet:namespace /> fm, this);
 return false;">

 <a href="#" onClick="Expanse.Popup.close(
 this);">

 </div>
 </td>
 </tr>
 </table>
 </td>
</tr>
</table>
</form>
</div>
</c:otherwise>
</c:choose>
<script type="text/javascript">
 function <portlet:namespace />shareWithFriends(form, instance)
 // ignore, see attached code
 var inputs = jQuery("input, textarea, select", form);
 Liferay.PortletSharing.showShareInfo(inputs.serialize(), 'Share
 - Success', 'success');
 Expanse.Popup.close(instance);
 // ignore, see attached code
</script>
```

The preceding code shows how to open and close pop-ups. For the `close` button, just use `jQuery Expanse.Popup.close(this)`. When you click on this button, it will call `jQuery Expanse.Popup.close` to close the current pop-up. Now click on the **SEND TO FRIEND** button. It will sequentially call the jQuery function `shareWithFriends` to verify inputs, open a new pop-up via `Liferay.PortletSharing.showShareInfo` for success information, and close the previous pop-up via `Expanse.Popup.close`. Note that it will get the current URL and call `Liferay.PortletSharing.showShareInfo` with `url` and `title` as `Share - Success`, and `action` as `success`.

## Preparing jQuery service

Accordingly, we need to prepare jQuery service by adding a `showShareInfo` method in `portlet_sharing.js`. As we're going to customize the portlet-sharing feature, it would be better to override `portlet_sharing.js`. To do so, use the following steps:

1. Locate the JavaScript file `portlet_sharing.js` in the folder `/portal/portal-web/docroot/htm/js/liferay`.
2. Copy the `portlet_sharing.js` file to the folder `/ext/ext-web/docroot/htm/js/liferay` and open it.
3. Add the following lines after the line `Liferay.PortletSharing = {` and save it:

```
showShareInfo: function(shareURL, title, action) {
 var portletURL = Liferay.PortletURL.createResourceURL();
 portletURL.setPortletId("shareWithFriends");
 portletURL.setParameter("shareURL", shareURL);
 portletURL.setParameter("action", action);
 new Expanse.Popup({ fixedcenter: true,
 header: title, width: 400,
 height: 350, url: portletURL.toString(),
 modal: true });
},
```

The code above shows a JavaScript function `Liferay.PortletSharing.showShareInfo` with inputs `url`, `title`, and `action`. Moreover, it specifies a pop-up with customized `title`, and calls `Expanse.popup` to show the pop-up with `title`.

Note that the JavaScript function `showShareInfo` specifies a generic pop-up, as it just takes `url`, `title`, and `action` as inputs. If you provide `Share` as an input for `title`, it will show a pop-up with the **Share** title. If you provide `My Street` as an input for `title`, it will show a pop-up with the **My Street** title. We will show this feature later.

## Building the article template

Finally, we need to update the SHARE article template with additional feature—showing sharing info—via the Web Content portlet. You can find the Web Content portlet in the Control Panel. To do so, add the following lines at the beginning of the SHARE template:

```
#set ($shareTitle = 'Share')
#set ($currentURL = $request.render-url.substring(0, $request.render-
url.indexOf("?")))
```

And add the following lines before the last </div>:

```
<a style='text-decoration:none'
 href='javascript:Liferay.PortletSharing.showShareInfo
 ("$currentURL", "$shareTitle")'>


```

The code above shows the VM template variables `shareTitle` and `currentURL`, using the reserved element `request.render-url`. Moreover, it also calls JavaScript `javascript:Liferay.PortletSharing.showShareInfo` with values `currentURL` and `shareTitle`. Similar to the image button of the widget, it shows an image button with a `/cms_services/images/button/btnSend.gif` image.

## Setting up the most popular journal articles

In a web site, we will have a lot of journal articles (that is, web content) for a given article type. For example, for the article type **Article Content**, we will have articles talking about product family. We may want to know how many times the end users read each article. Meanwhile, it would be nice if we could show the most popular articles (for example, **TOP 10 articles**) for this given article type.

As shown in the following screenshot, a journal article **My EDI Product I** is shown via a portlet Ext Web Content Display. Rating and comments on this article are also exhibited. At the same time, the medium-size image, polls, and related content of this article are listed, too. A view counter of this article is especially displayed under the ratings. Moreover, the most popular articles are exhibited with article title and number of views under related content. All these articles belong to the article type **article-content**. That is, the article in the current portlet Ext Web Content Display has the most popular articles only for the article type **article-content**.

Of course, you can customize the portlet Web Content Display directly through changing JSP files. For demo purposes, we will implement the view counter in the portlet Ext Web Content Display. Meanwhile, we will implement the mostly popular articles via VM services and article templates. In addition, we will analyze the view counter for other assets such as Image Gallery images, Document Library documents, Wiki articles, Blog entries, Message Boards threads, and so on.

**My EDI Product I**  
David Berger, Editor, July 24, 2007

Electronic Data Interchange (EDI) refers to the structured transmission of data between organizations by electronic means. It is more than mere E-mail; for instance, organizations might replace bills of lading and even checks with appropriate EDI messages. It also refers specifically to a family of standards, including the X12 series. However, EDI also exhibits its pre-Internet roots, and the standards tend to focus on ASCII(American Standard Code for Information Interchange)-formatted single messages rather than the whole sequence of conditions and exchanges that make up an inter-organization business process.

In 1992, a survey of Canadian businesses found at least 140 that had adopted some form of EDI, but that many (in the sample) "had not benefited from implementing EDI, and that they [had] in fact been disadvantaged by it."

The National Institute of Standards and Technology in a 1996 publication defines Electronic Data Interchange as "the computer-to-computer interchange of strictly formatted messages that represent documents other than monetary instruments. EDI implies a sequence of messages between two parties, either of whom may serve as originator or recipient. The formatted data representing the documents may be transmitted from originator to recipient via telecommunications or physically transported on electronic storage media.". It goes on further to say that "In EDI, the usual processing of received messages is by computer only. Human intervention in the processing of a received message is typically intended only for error conditions, for quality review, and for special situations. For example, the transmission of binary or textual data is not EDI as defined here unless the data are treated as one or more data elements of an EDI message and are not normally intended for human interpretation as part of online data processing."

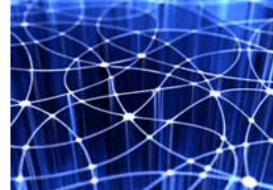
How about the IA product?

a. Good  
 b. OK

**Submit**

**Related Content:**  
[EDI Online](#)  
[EDI Tools](#)  
[My EDI Product II](#)

**Most Popular Content:**  
[My EDI Product II](#) 19 views  
[My EDI Product I](#) 17 views  
[EDI Online](#) 4 views  
[EDI Tools](#) 1 views



Your Rating

18 Views, [Post Reply](#)

Average (1 Vote)

5 stars (1 vote)

## Adding a view counter in the Web Content Display portlet

First of all, let's add a view counter in the Ext Web Content Display portlet. As the function of view counter for assets (including journal articles) is provided in the model `TagsAssetModel` of the `com.liferay.portlet.tags.model` package in the folder `/portal/portal-service/src`, we could use this feature in this portlet directly. To do so, use the following steps:

1. Create a folder `journal_content` in the folder `/ext/ext-web/docroot/html/portlet/`.

2. Copy the JSP file `view.jsp` in the folder `/portal/portal-web/docroot/html/portlet/` to the folder `/ext/ext-web/docroot/html/portlet/journal_content` and open it.
3. Add the line `<%@ page import="com.liferay.portlet.tags.model.TagsAsset" %>` after the line `<%@ include file="/html/portlet/journal_content/init.jsp" %>`, and check the following lines:

```
JournalArticleDisplay articleDisplay = (JournalArticleDisplay)
 request.getAttribute(
 WebKeys.JOURNAL_ARTICLE_DISPLAY);
if (articleDisplay != null) {
 TagsAssetLocalServiceUtil.incrementViewCounter(
 JournalArticle.class.getName(),
 articleDisplay.getResourcePrimKey());
}
```

4. Then add the following lines after the line `<c:if test="<%enableComments %>">` and save it:

```

<% TagsAsset asset = TagsAssetLocalServiceUtil.getAsset
 (JournalArticle.class.getName(),
 articleDisplay.getResourcePrimKey());%>
<c:choose>
<c:when test="<%asset.getViewCount() == 1 %>">
<%asset.getViewCount() %>
<liferay-ui:message key="view" />,
</c:when>
<c:when test="<%asset.getViewCount() > 1 %>">
<%asset.getViewCount() %>
<liferay-ui:message key="views" />,
</c:when>
</c:choose>

```

The code above shows a way to increase the view counter via the `TagsAssetLocalServiceUtil.incrementViewCounter` method. This method takes two parameters `className` and `classPK` as inputs. For the current journal article, the two parameters are `JournalArticle.class.getName()` and `articleDisplay.getResourcePrimKey()`. Then, this code shows a way to display view counted through the `TagsAssetLocalServiceUtil.getAsset` method. Similarly, this method also takes two parameters, `className` and `classPK`, as inputs. This approach would be useful for other assets, as the `className` parameter could be Image Gallery, Document Library, Wiki, Blogs, Message Boards, Bookmark, and so on.

## Setting up VM service

We can set up the VM service to exhibit the most popular articles. We can also add the `getMostPopularArticles` method in the custom velocity tool `ExtVelocityToolUtil`.

To do so, first add the following method in the `ExtVelocityToolService` interface:

```
public List<TagsAsset> getMostPopularArticles(String companyId,
 String groupId, String type, int limit);
```

And then add an implementation of the `getMostPopularArticles` method in the `ExtVelocityToolServiceImpl` class as follows:

```
public List<TagsAsset> getMostPopularArticles(String companyId,
 String groupId, String type, int limit) {
 List<TagsAsset> results = Collections.synchronizedList(new
 ArrayList<TagsAsset>());
 DynamicQuery dq0 = DynamicQueryFactoryUtil.forClass(
 JournalArticle.class, "journalarticle").
 setProjection(ProjectionFactoryUtil.property(
 "resourcePrimKey")).add(PropertyFactoryUtil.
 forName("journalarticle.companyId")).
 eqProperty("tagsasset.companyId").add(
 PropertyFactoryUtil.forName(
 "journalarticle.groupId").eqProperty(
 "tagsasset.groupId")).add(PropertyFactoryUtil.
 forName("journalarticle.type").eq(
 "article-content"));
 DynamicQuery query = DynamicQueryFactoryUtil.forClass(
 TagsAsset.class, "tagsasset")
 .add(PropertyFactoryUtil.forName(
 "tagsasset.classPK").in(dq0))
 .addOrder(OrderFactoryUtil.desc(
 "tagsasset.viewCount"));
 try{ List<Object> assets = TagsAssetLocalServiceUtil.
 dynamicQuery(query);
 int index = 0; for (Object obj: assets) {
 TagsAsset asset = (TagsAsset) obj;
 results.add(asset);
 index++;
 if(index == limit)
 break;
 }
 } catch (Exception e){
 return results;
 }
 return results;
}
```

The preceding code shows a way to get the most popular articles by company ID, group ID, article type, and limited articles to be returned. **DynamicQuery API** allows us to leverage the existing mapping definitions through access to the Hibernate session. For example, DynamicQuery dq0 selects the journal articles by companyID, groupId, and type; DynamicQuery query selects tagsassets by classPK, which exists in DynamicQuery dq0; and tagsassets are ordered by viewCount as well.

Finally, add the following method to register the above method in ExtVelocityToolUtil:

```
public List<TagsAsset> getRelatedArticles(String companyId, String
 groupId, String articleId, int limit){
 return _extVelocityToolService.getRelatedArticles(companyId,
 groupId, articleId, limit);
}
```

The code above shows a generic approach to get **TOP 10** articles for any article types. Of course, you can extend this approach to find TOP 10 assets. This can include Image Gallery images, Document Library documents, Wiki articles, Blog entries, Message Boards threads, Bookmark entries, slideshow, videos, games, video queue, video list, playlist, and so on. You may practice these TOP 10 assets feature.

## Building article template for the most popular journal articles

We have added view counter on journal articles. We have already built VM service for the most popular articles too. Now let's build an article template for them.

### Setting up the default article type

As mentioned earlier, there is a set of types of journal articles, for example, announcements, blogs, general, news, press-release, updates, article-tout, article-content, and so on. In real case, only some of these types will require view counter, for example article-content. Let's configure the default article type for mostly popular articles. We can add the following line at the end of portal-ext.properties.

```
ext.most_popular_articles.article_type=article-content
```

The code above shows that the default article type for `most_popular_articles` is `article-content`.

## Setting up the article template

Let's create a template named MOST\_POPULAR\_ARTICLES via the portlet Web Content with the following content:

```
#set ($extVelocityToolUtil = $utilLocator.findUtil
 ('com.ext.portal.util.ExtVelocityToolUtil'))
#set ($currentURL = $request.render-url.substring(0, $request.render-
url.indexOf("?")))
#set ($results = $extVelocityToolUtil.getMostPopularArticles(
 $companyId, $groupId, $propsUtil.get(
 "ext.most_popular_articles.article_type"), 10))
<div style="border: 1px solid #cccccc; padding: 4px;
 text-align: left; ">
 <div><h5>Most Popular Content </h5></div>
 #foreach ($result in $results)
 <div>
 <a href="$currentURL?p_p_id=extJournalContent&p_p_lifecycle=0
 &_extJournalContent_assetId=$result.getAssetId() ">
 $result.Title

 $result.viewCount views
 </div>
 #end
</div>
```

The code above shows how to use the `extVelocityToolUtil.getMostPopularArticles` method to get the most popular articles via `companyId`, `groupId`, `article_type` and the number of articles (10). Then it shows the most popular articles with title and view count.

## Putting all article templates together

Finally, we need to update the ARTICLE\_CONTENT template via the portlet Web Content in order to merge the POLL, RELATED\_CONTENT, and MOST\_POPULAR\_ARTICLES templates. Thus, we will have one integrated view for polls, related articles, and the most popular articles. To do so, add the following line after the line `#parse ("$journalTemplatesPath/RELATED_Content")` in the ARTICLE\_CONTENT template:

```
#parse ("$journalTemplatesPath/MOST_POPULAR_ARTICLES")
```

Cool! You have built dynamic articles with polls, related content, and the most popular articles successfully. If you created an article with the Article Content type and tags, the polls, related content, and most popular articles (TOP 10) will come out altogether.

## Having a handle on view counter for assets

Assets could be journal articles, Wiki articles, blog entries, Message Boards threads, Image Gallery images, Document Library documents, bookmark entries, and so on. Although the function of view counter for these assets is provided in the `TagsAssetModel` model at the `com.liferay.portlet.tags.model` package in the folder `/portal/portal-service/src`, Message Boards threads and bookmark entries still use their own models. At the same time, Image Gallery images and Document Library documents do not use the `TagsAssetModel` model for view counter yet. Therefore, it would be nice to look at view counter for these assets in detail.

## Using journal article tokens

We have added the view counter feature in JSP file successfully. Here we have one more option: adding the views counter through the journal articles token. You can just add a `@view_counter@` token to either the content of journal article or the output of article template used. This token is automatically translated to the logic of view counter increment. To find out the logic, locate the Java file `ViewCounterTransformerListener.java` at the `com.liferay.portlet.journal.util` package in the folder `/portal/portal-impl/src` and open it. Check out the following lines:

```
protected String replace(String s) {
 Map<String, String> tokens = getTokens();
 String articleResourcePK = tokens.get("article_resource_pk");
 String counterToken = StringPool.AT + "view_counter" +
 StringPool.AT;
 StringBuilder sb = new StringBuilder();
 sb.append("<script type=\"text/javascript\">");
 sb.append("Liferay.Service.Tags.TagsAsset.incrementViewCounter");
 sb.append("{className:'");
 sb.append("com.liferay.portlet.journal.model.JournalArticle', ");
 sb.append("classPK:");
 sb.append(articleResourcePK); sb.append("});");
 sb.append("</script>");
 s = StringUtil.replace(s, counterToken, sb.toString());
 return s;
}
```

The code above shows the same logic as that of the journal article JSP file that was mentioned earlier. Because it is AJAX-based, even if the page is cached somewhere in a proxy, you will get the correct number of views counted.

For example, when editing the text of the article named My EDI Product I, you can simply add @view\_counter@ anywhere in the text and save it. That's it. Besides the @view\_counter@ token, there is a set of tokens you can use at runtime, translating to their applicable runtime value at processing time. The following is a complete list of tokens and their runtime values:

```
@cdn_host@: themeDisplay.getCDNHost()
@company_id@: themeDisplay.getCompanyId()
@group_id@: groupId
@cms_url@: themeDisplay.getPathContext() + "/cms/servlet"
@image_path@: themeDisplay.getPathImage()
@friendly_url_private_group@: themeDisplay.
getPathFriendlyURLPrivateGroup()
@friendly_url_private_user@: themeDisplay.
getPathFriendlyURLPrivateUser()
@friendly_url_public@: themeDisplay.getPathFriendlyURLPublic()
@main_path@: themeDisplay.getPathMain()
@portal_ctx@: themeDisplay.getPathContext()
@portal_url@: Http.removeProtocol(themeDisplay.getURLPortal())
@root_path@: themeDisplay.getPathContext()
@theme_image_path@: themeDisplay.getPathThemeImages()
@language_id@: the language_id of the current request
```

## Get view count on Wiki articles

The view count of Wiki articles is provided by default via the TagsAssetModel model at the com.liferay.portlet.tags.model package in the folder /portal/portal-service/src. You can find it out from view.jsp in the folder /portal/portal-web/docroot/html/portlet/wiki as follows:

```
TagsAssetLocalServiceUtil.incrementViewCounter(WikiPage.class.
getName(), wikiPage.getResourcePrimKey());
<!-- ignore details -->
<% TagsAsset asset = TagsAssetLocalServiceUtil.getAsset(WikiPage.
class.getName(), wikiPage.getResourcePrimKey()); %>
<c:choose>
<c:when test="<% asset.getViewCount() == 1 %>">
<%= asset.getViewCount() %> <liferay-ui:message key="view" />,
</c:when>
<c:when test="<% asset.getViewCount() > 1 %>">
<%= asset.getViewCount() %>
<liferay-ui:message key="views" />,
</c:when>
</c:choose>
```

The code above shows an increasing view counter of the current Wiki article via the TagsAssetLocalServiceUtil service. Then it shows the views counted for the current Wiki article.

## Getting views count on blog entries

Similarly, the view count of blog entries is provided by default via the TagsAssetModel model too. You can find it from the `view_entry.jsp` and `view_entry_content.jsp` file in the folder `/portal/portal-web/docroot/html/portlet/blogs` as follows:

```
TagsAssetLocalServiceUtil.incrementViewCounter(BlogsEntry.class.
 getName(), entry.getEntryId());
<!-- ignore details -->

 <% TagsAsset asset = TagsAssetLocalServiceUtil.getAsset(
 BlogsEntry.class.getName(), entry.getEntryId());%>
 <c:choose>
 <c:when test="<% asset.getViewCount() == 1 %>">
 <%= asset.getViewCount() %> <liferay-ui:message key="view" />
 </c:when>
 <c:when test="<% asset.getViewCount() > 1 %>">
 <%= asset.getViewCount() %>
 <liferay-ui:message key="views" />
 </c:when>
 </c:choose>

```

The code above first shows an increasing view counter of the current blog entries via the `TagsAssetLocalServiceUtil` service in the `view_entry.jsp` file. Then it shows the views counted for the current blog entry in `view_entry_content.jsp`.

## Getting views on Message Boards threads

Message Board threads have their own model to manage the views counted. You can find related methods – locate the Java file `MBThreadModel.java` at the `com.liferay.portlet.messageboards.model` package in the `/portal/portal-service/src` folder and open it. Check the following lines:

```
public int getViewCount();
public void setViewCount(int viewCount);
```

The code above shows get and set methods for view counts.

Then you can check the method which is used to increase views count. Locate the Java file `MBMessageLocalServiceImpl.java` in the `com.liferay.portlet.messageboards.service.impl` package in the `/portal/portal-impl/src` folder and open it. Check the following lines:

```
public MBMessageDisplay getMessageDisplay(MBMessage message)
throws PortalException, SystemException {
 // ignore details
```

```

MBThread thread = mbThreadPersistence.findByPrimaryKey(
 message.getThreadId());
thread.setViewCount(thread.getViewCount() + 1);
mbThreadPersistence.update(thread, false);
// ignore details
}

```

The code above reuses the setter to increase the views counted when displaying Message Boards message.

## Setting up view counter on the Image Gallery images

Although the function of the view count for Image Gallery images is provided in the TagsAssetModel model, the feature of view counter is not provided in the portal by default. Of course, we can set up view counter on the Image Gallery images simply by using the following steps:

1. Create a package com.liferay.portal.servlet in the folder /ext/ext-impl/src.
2. Copy the Java file ImageServlet.java from the package com.liferay.portal.servlet in the folder /portal/portal-impl/src to the package com.liferay.portal.servlet in the folder /ext/ext-impl/src.
3. Locate the Java file ImageServlet.java at the package com.liferay.portal.servlet in the /ext/ext-impl/src folder and open it.
4. Add the following lines after the line `image = ImageLocalServiceUtil.getImage(imageId);` in the `getImage` method :

```

try{
 IGImage igImage = IGImageLocalServiceUtil.getImageByLargeImageId(
 imageId);
 TagsAssetLocalServiceUtil.incrementViewCounter(
 IGImage.class.getName(), igImage.getPrimaryKey());
}
catch (Exception e) {}

```
5. Then add the following lines before the line `image = ImageLocalServiceUtil.getImage(` in the `getImage` method.

```

TagsAssetLocalServiceUtil.incrementViewCounter(IGImage.class.
 getName(), igImage.getPrimaryKey());

```

The code above shows a way to set up a view count for the Image Gallery images. When getting an image via either `imageId` or `uuid` and `groupId`, it uses the `TagsAssetLocalServiceUtil` service to increase view counter for the current image

## Setting up view counter on Document Library documents

Similar to view counter of the Image Gallery images, the feature of view counter for Document Library documents is not provided in the portal by default. But we can set it up easily by using the following steps:

1. Create a package `com.liferay.portlet.documentlibrary.action` in the folder `/ext/ext-impl/src`.
2. Copy the Java file `GetFileAction.java` from the package `com.liferay.portlet.documentlibrary.action` in the `/portal/portal-impl/src` folder to the package `com.liferay.portlet.documentlibrary.action` in the `/ext/ext-impl/src` folder.
3. Locate the Java file `GetFileAction.java` at the package `com.liferay.portlet.documentlibrary.action` in the `/ext/ext-impl/src` folder and open it.
4. Add the following lines after the line `String contentType = MimeTypesUtil.getContentType(fileName);` in the `GetFile` method and save it:  
`TagsAssetLocalServiceUtil.incrementViewCounter(DLFileEntry.class.getName(), fileEntry.getFileEntryId());`

The code above shows a way to increase view counter for Document Library documents. When called, the `getFile` method uses the `TagsAssetLocalServiceUtil` service to increase the view counter for the current document.

## Getting visits on bookmark entries

Similar to Message Boards threads, bookmark entries have their own model to manage views counter. You can find related methods in the Java file `BookmarksEntryModel.java` in the package `com.liferay.portlet.bookmarks.model` in the folder `/portal/portal-service/src`. Open it and then check the following lines:

```
public int getVisits();
public void setVisits(int visits);
```

The code above shows the `get` and `set` methods for visits. Then check the method to increase visits: locate the Java file `BookmarksEntryLocalServiceImpl.java` in the package `com.liferay.portlet.bookmarks.service.impl` in the `/portal/portal-impl/src` folder and open it; check the following lines:

```
public BookmarksEntry openEntry(long entryId)
throws PortalException, SystemException {
BookmarksEntry entry =
bookmarksEntryPersistence.findByPrimaryKey(entryId);
entry.setVisits(entry.getVisits() + 1);
bookmarksEntryPersistence.update(entry, false);
return entry;}
```

As shown in the code above, it reuses the `set` method to increase visits (as explained earlier) when a Bookmark entry is opened.

## Personalizing user comments

Liferay portal provides Message Boards which can be used to represent comments by categories, threads, and messages. As it uses the `discussionId-classNameId-classPK` model, these comments can be used on pages, journal articles, Wiki articles, blog entries, and so on. For example, if `classNameId` is a journal article, comments are used for journal articles; whereas if `classNameId` is a Wiki, comments are used for Wiki articles.

You can post comments on pages, journal articles, Wiki articles, blog entries, and so on. Once comments are added, they are visible to anyone. Other users can view existing comments, reply to comments, or post new comments. It is cool for most of the requirements. But no workflow for approval or deletion is available in this model. For example, a normal user **Lotti Stein** created comments in a journal article. These comments will not be published immediately. A CMS admin user David Berger will review these comments first. He may approve or delete them. Once these comments are approved, normal users and guests will be able to see it. Otherwise, these comments will be deleted, that is, no one can see them anymore. This feature is not available in Message Boards yet.

But in some use cases, we need a workflow for approval or deletion of comments. As shown in the following screenshot, the **Approve** or **Delete** workflow was added in comments. For instance, **Lotti Stein** logged in and posted comments on a journal article. She could not see her comments immediately. After logging in, David Berger will see and review these comments. He would either approve it or delete it. At the same time, an email must be sent to CMS admin and/or other persons when comments are created. Moreover, the *TO*, *CC*, and *BCC* email addresses should be configurable and multiple email addresses for all these three fields should be supported as well. The format of email body and subject must be configurable too.

In this section, we're going to implement this feature—personalizing user comments. Meanwhile, we will replace the default discussion UI tag with this custom user comments.

This comment is awaiting your review and approval!

**Lotti Stein wrote:**  
12/12/08 8:44 PM  
This is a good idea!

Approve   Delete

Add your Comment.\*

All comments have a 1000 character limit (1000 left).

Submit

\*Your comment will be posted within 24 hours

## Creating user comments model

First of all, let's create user comments model in `service.xml` using the following steps:

1. Create a package `com.ext.portlet.comment` in the folder `/ext/ext-impl/src`.
2. Create an XML file `service.xml` in the package `com.ext.portlet.comment` and open it.
3. Add the following lines in this file and save it:

```
<?xml version="1.0"?>
<!DOCTYPE service-builder PUBLIC "-//Liferay//DTD Service Builder
5.2.0//EN" "http://www.liferay.com/dtd/liferay-service-builder_5_
2_0.dtd">
<service-builder package-path="com.ext.portlet.comment">
 <namespace>ExtComment</namespace>
 <entity name="ExtComment" uuid="false"
 local-service="true" remote-service="true"
 persistence-class="com.ext.portlet.comment.service.
 persistence.ExtCommentPersistenceImpl">
 <column name="commentId" type="long" primary="true" />
 <column name="classNameId" type="long" />
 <column name="classPK" type="long" />
 <column name="createDate" type="Date" />
```

---

```

<column name="approvalDate" type="Date" />
<column name="userId" type="long" />
<column name="approverId" type="long" />
<column name="text_" type="String" />
<column name="approved" type="boolean" />
</entity>
<exceptions>
 <exception>ExtComment</exception>
</exceptions>
</service-builder>

```

The code above shows user comments model, which will be used by any asset as it is using `classNameId` and `classPK`. It not only specifies the creation date and the user who created it, but also the approval date and the user who approved it.

Now, just create the following table into the database:

```

create table ExtComment (
 commentId bigint not null primary key,
 classNameId bigint, classPK bigint,
 createDate datetime, approvalDate datetime,
 userId bigint, approverId bigint,
 text_ longtext, approved boolean
);

```

The code above shows database SQL script for the user comments model. Similarly, the XML model shows table fields such as `commentId`, `classNameId`, and `classPK`.

Afterwards, we need to build a service with ServiceBuilder. After preparing the `service.xml` file, you can build services using the following steps:

1. Locate the XML file `/ext/ext-impl/build-parent.xml` and open it.
2. Add the following lines between `</target>` and `<target name="build-service-portlet-reports">` and save it:

```

<target name="build-service-portlet-extComment">
 <antcall target="build-service">
 <param name="service.file"
 value="src/com/ext/portlet/comment/service.xml" />
 </antcall>
</target>

```

When you are ready, double-click on the Ant target `build-service-portlet-extComment`. ServiceBuilder will build related models and service for the user comments.

## Building the Ext Comment portlet

Similar to building the portlet Share, we can build the portlet Ext Comment as well by using the following steps:

1. Configure the portlet Ext Comment in both the `portlet-ext.xml` and `liferay-portlet-ext.xml` files.
2. Set title mapping in the `Language-ext.properties` file.
3. Add the portlet Ext Comment to the Book category in `liferay-display.xml`.
4. Finally, specify Struts actions and forward paths in the `struts-config.xml` and `tiles-defs.xml` files, respectively.

Next, we need to create Struts actions in the following manner:

1. Create a package `com.ext.portlet.comment.action` in the `/ext/ext-impl/src` folder.
2. Add Java files `ViewAction.java` and `CommetsPropsValues.java`, `AddCommentLocalServiceUtil.java` in the package `com.ext.portlet.comment.action`.
3. Open the Java file `AddCommentLocalServiceUtil.java`, add the following lines, and save it:

```
public class AddCommentLocalServiceUtil {
 public static ExtComment getComment(long commentId)
 {/* ignore details */}
 public static List<ExtComment> getComments(long classNameId,
 long classPK) throws Exception {/* ignore details */}
 public static void deleteComment(ActionRequest actionRequest)
 {/* ignore details */}
 public static ExtComment addComment(ActionRequest actionRequest)
 {/* ignore details */}
 public static void updateComment(ActionRequest actionRequest)
 {/* ignore details */}
}
```

The code above shows the methods to get, delete, add, and update comments. We have used the `AddCommentLocalServiceUtil` name only for demo purpose; you can use a different name. Moreover, ServiceBuilder generated services include CRUD operations and handling transaction. We should not modify these services directly. Instead, we simply add a customized wrapper `AddCommentLocalServiceUtil` on top of ServiceBuilder-generated service. Thus, when upgrading ServiceBuilder, we only need to upgrade this wrapper if any changes are involved.

## Adding permissions based on user groups

Before building Struts view pages, we need to set up user groups: Admin and CMS Admin. To do so, locate the properties file `portal-ext.properties` in `/ext/ext-impl/src/` and open it. Add the following lines at the end of `portal-ext.properties` and save it:

```
ext.ondportal.cmsadmin=ONDPortal_CMSAdmin
ext.ondportal.admin=ONDPortal_Admin
```

The code above shows the default user group names `ONDPortal_Admin` and `ONDPortal_CMSAdmin` for Admin and CMS Admin, respectively. For test purpose, you can assign a member (David Berger) to the user group `ONDPortal_CMSAdmin`.

Last but not the least, we need to build Struts view pages `init.jsp` and `view.jsp` using the following steps:

1. Create a folder `comments` in `/ext/ext-web/docroot/html/portlet/ext/ext-impl/src/`
2. Create JSP files `init.jsp` and `view.jsp` in `/ext/ext-web/docroot/html/portlet/ext/comments/`
3. Locate JSP file `view.jsp` and open it
4. Add the following lines in `view.jsp` and save it:

```
/* ignore details */
UserGroup userGroupAdmin = UserGroupLocalServiceUtil.getUserGroup(
 company.getCompanyId(),
 PropsUtil.get("ext.ondportal.admin"));
UserGroup userGroupCMSAdmin = UserGroupLocalServiceUtil.
 getUserGroup(company.getCompanyId(),
 PropsUtil.get("ext.ondportal.
cmsadmin"));
List<UserGroup> userGroups = UserGroupLocalServiceUtil.
 getUserUserGroups(themeDisplay.
 getUserId());
 boolean hasAdminGroup = false;
boolean hasCMSAdminGroup = false;
for(UserGroup obj: userGroups){
 if(obj.getUserId() == userGroupAdmin.getUserId())
 hasAdminGroup = true;
 else if(obj.getUserId() == userGroupCMSAdmin.
 getUserId())
 hasCMSAdminGroup = true;
}
// ignore details, see attached code
if (comments != null){
 for (ExtComment comment : comments)
 {
 User poster = UserLocalServiceUtil.getUserById(
 comment.getUserId());
 poster.setComments(comment);
 UserLocalServiceUtil.updateUser(poster);
 }
}
```

```
comment.getUserId());
if (comment.isApproved()) {
 /* ignore details */
}
else if (isAdmin || hasAdminGroup || hasCMSAdminGroup) {
 /* ignore details */
}
/* ignore details */
if (themeDisplay.isSignedIn()) {
 /* ignore details */
}
}
```

The code above shows logic to add comments. It first finds user groups Admin and CMSAdmin. Then it checks whether or not the current user belongs to the user groups Admin and CMSAdmin. Finally, it specifies the conditions to add new comments, display approved comments, or to approve/delete the submitted comments.

Similar to the approach stated in the previous chapter, you can define a permission model on the Approve and Delete buttons instead of checking permissions of user groups in a JSP file. Thus, the end users can assign permissions of using these buttons to User, Organization, User Group, and Role by referring to the book *Liferay Portal Enterprise Intranets*.

## Updating the UI tag

Liferay portal provides UI tag for comments (that is, discussion). You can find discussion UI tag in /portal/portal-web/docroot/html/taglib/ui/discussion/page.jsp. Here is an example code for adding comments to a journal article:

```
<liferay-ui:discussion formAction="<%=" discussionURL %>">
 className="<%=" JournalArticle.class.getName() %>"
 classPK="<%=" articleDisplay.getResourcePrimKey() %>"
 userId="<%=" articleDisplay.getUserId() %>"
 subject="<%=" articleDisplay.getTitle() %>"
 redirect="<%=" currentURL %>"
 ratingsEnabled="<%=" enableCommentRatings %>"/>
```

The code above shows a UI tag discussion and its inputs. You can find the specification of UI tag liferay-portlet:discussion in /portal/util-taglib/src/META-INF/liferay-ui.tld, where a tag com.liferay.taglib.ui.DiscussionTag is defined. Moreover, you may be interested in the definition of UI tag com.liferay.taglib.ui.DiscussionTag. You may check definitions in the folder /portal/util-taglib/src.

Now we're going to use the portlet Ext Comments as liferay-ui:discussion UI tag by using the following steps:

1. Create a folder `/taglib/ui/discussion` in the folder `/ext/ext-web/docroot/html`.
2. Create a JSP file `page.jsp` in `/ext/ext-web/docroot/html/taglib/ui/discussion` and open it.
3. Add the following lines in this file and save it:

```
<%@ include file="/html/taglib/init.jsp" %>
<div style="margin-top: 10px;">
 <liferay-portlet:runtime portletName="extComments" />
</div>
<%!private static Log _log = LogFactoryUtil.getLog("portal-
 web.docroot.html.taglib.ui.discussion.page.jsp") ;
%>
```

This code shows a new UI tag discussion. It will load the portlet `extComments` for the tag lib UI discussion. It uses portlet tag `liferay-portlet:runtime` to load the portlet `extComments` in runtime. You can find specification of portlet tag `liferay-portlet:runtime` in `/portal/portal-web/docroot/WEB-INF/tld/liferay-portlet-ext.tld`. From now on, the `extComments` portlet will appear whenever the UI tag discussion was attached to `<liferay-ui:discussion>` for any assets.

## Setting up email notification

As mentioned earlier, an email must be sent to CMS Admin and/or other persons when comments are created. Moreover, the *TO*, *CC*, and *BCC* email addresses should be configurable, multiple email addresses for them should be supported, and format of email body and subject should be configurable too. Let's implement these features. First, we need to set up email addresses for *to*, *cc* and *bcc*. To do so, locate the `portal-ext.properties` file in `/ext/ext-impl/src` and open it. Add the following lines at the end of `portal-ext.properties` and save it:

```
ext.comment.ratings.enabled=false
ext.comment.character.limit=1000
ext.comment.email.enabled=true
ext.comment.from.name=Palm Tree
ext.comment.from.address=admin@book.com
ext.comment.to.names=David Berger,Lotti Stein
ext.comment.to.emails=david@book.com,lotti@book.com
ext.comment.cc.names=David Berger,Lotti Stein
ext.comment.cc.emails=david@book.com,lotti@book.com
ext.comment.bcc.names=David Berger,Lotti Stein
ext.comment.bcc.emails=david@book.com,lotti@book.com
```

The code above shows the character limit and enabled rating for user comments portlets first. Then it configures the email notification settings. Of course, you can configure the values of the above properties according to your requirements.

Secondly, we need to consume the above settings. To do so, create a Java file ExtMailUtil.java in the package com.ext.portlet.comment.action under the folder /ext/ext-impl/src. Open it and add the sendEmail method in ExtMailUtil.java and save it.

```
public static void sendEmail(String text, ActionRequest req) throws
Exception {
 /* ignore details */
 String subjectTemplate = ContentUtil.get(
 "com/ext/portlet/comment/dependencies/comments_subject.vm");
 String bodyTemplate = ContentUtil.get(
 "com/ext/portlet/comment/dependencies/comments_body.vm");
 /* ignore details */
 if(toNames != null && toEmails != null)
 mailMessage.setTo(getEmailAddresses(toNames, toEmails));
 /* ignore details */
 if(ccNames != null && ccEmails != null)
 mailMessage.setCC(getEmailAddresses(ccNames, ccEmails));
 /* ignore details */
 if(bccNames != null && bccEmails != null)
 mailMessage.setBCC(getEmailAddresses(bccNames, bccEmails));
 /* ignore details */
}
```

The code shows a way to consume body template and subject template. It also shows a way to set up the to, cc, and bcc email address.

Finally, we need to create a package com.ext.portlet.comment.dependencies in /ext/ext-impl/src, and create two template files comments\_subject.vm and comments\_body.vm in this package.

In short, you can configure emails notification by updating the portal-ext.properties file. You can also configure email body and subject by updating the comments\_subject.vm and comments\_body.vm templates.

## Customizing My Account

For different kinds of requirements, we may need different kinds of customization and extension on top of the portal. Of course, the portal provides a flexible framework where out of the box portlets could be customized and extended easily. For instance, it is useful to quickly customize My Account on top of the portal in order to provide a different look and feel. As shown in the following screenshot, we are going to change the login's look and feel for the use case named *Customizing login view*. After signing in, we are expected to display a greeting message, **Welcome {userName}!**, with your profile and logout links at **Edit your profile | logout**. Meanwhile, the look and feel with a specific background color can be changed easily.



In addition, let's consider another requirement to add fields in the **Create Account** page. It is very helpful that new users are able to create an account on the fly. After creating an account, new users will receive an email with newly created account information.

As shown in above screenshot, for the use case *Creating a customized account on the fly*, we want to add the **Your Password**, **Verify Password**, **Reminder Query Question**, and **Reminder Query Answer** fields in the **Create Account** page. That is, when creating an account, you enter your basic information such as **First Name**, **Middle Name**, **Last Name**, **Screen Name**, **Email Address**, **Gender**, and **Text Verification**; and you also give advanced information such as **Your Password**, **Verify Password**, **Reminder Query Question**, and **Reminder Query Answer**.

## Customizing login view

In this section, we're going to implement the use case *Customizing login view*. First, let's take a deep look at how to create an account and how the login view works. Then we will see how to customize the login view.

## Locating the portlet My Account

Let's first start the portal, and then click on the **Sign In** link. Further, let's click on the **Create Account** tab and we would get a link like:

```
http://localhost/web/guest/test?p_p_id=2&p_p_lifecycle=1&p_p_state= maximized&p_p_mode=view&saveLastPath=0&_2_struts_action=%2Fmy_ account%2Fcreate_account
```

The code above shows `p_p_id=2` (portlet name for creating account page), `p_p_lifecycle=1` (an action URL that is a Struts action), and `_2_struts_action=%2Fmy_account%2Fcreate_account` (Struts mapping `/my_account/create_account` defined in the `struts-config.xml` file).

First, let's check the portlet 2 in `/portal/portal-web/docroot/WEB-INF/portlet-custom.xml`. You will find the portlet 2 as follows:

```
<portlet>
 <portlet-name>2</portlet-name>
 <display-name>My Account</display-name>
 <portlet-class>com.liferay.portlet.StrutsPortlet</portlet-class>
 <init-param>
 <name>view-action</name>
 <value>/my_account/view</value>
 </init-param>
 <!-- ignore details -->
</portlet>
```

The code above shows the portlet 2 with a name My Account. It is a Struts portlet `com.liferay.portlet.StrutsPortlet`. At the same time, the `init-param` element contains a name-value pair `view-action - /my_account/view` as an initialization parameter of the portlet, that is, the Struts action path.

Now, let's find the Struts action in `portal/portal-web/docroot/WEB-INF/struts-config.xml`. You will find the Struts action paths `/login/create_account` and `/login/login` as follows:

```
<action path="/login/create_account"
 type="com.liferay.portlet.login.action.CreateAccountAction">
 <forward name="portlet.login.create_account"
 path="portlet.login.create_account" />
</action>
<action path="/login/login"
 type="com.liferay.portlet.login.action.LoginAction">
 <forward name="portlet.login.login" path="portlet.login.login" />
</action>
```

The preceding code shows the Struts action path /login/login and /login/create\_account. Moreover, under the Struts action /login/create\_account, you will see the com.liferay.portlet.login.action.CreateAccountAction type and the portlet.login.create\_account forward path. Similarly, under the Struts action /login/login, you will see the com.liferay.portlet.login.action.LoginAction type and the portlet.login.login forward path.

Last but not the least, let's check the forward path and the related JSP files in /portal/portal-web/docroot/WEB-INF/tiles-defs.xml. Search the forward paths portlet.login.create\_account and portlet.login.login in the tiles-defs.xml file. You will find the forward paths and JSP files as follows:

```
<definition name="portlet.login" extends="portlet" />
<definition name="portlet.login.create_account"
 extends="portlet.login">
 <put name="portlet_content"
 value="/portlet/login/create_account.jsp" />
</definition>
<definition name="portlet.login.login" extends="portlet.login">
 <put name="portlet_content" value="/portlet/login/login.jsp" />
</definition>
```

The code above shows the forward paths portlet.login.create\_account and portlet.login.login, and related JSP files portlet/login/create\_account.jsp and /portlet/login/login.jsp. Obviously, you can modify these pages using Ext. Without overwriting forward paths in the original tiles-def.xml file, create the same directory structure under Ext environment with the same filename, and then put modifications there.

Note that labels of Struts action, Struts path, and forward path might be changed in different portal versions. But it would be helpful to customize and extend the look and feel of My Account.

## Overriding login view

First of all, we need to provide message specification for the login view in the Language-ext.properties file. To do so, locate the JSP file Language-ext.properties in /ext/ext-impl/src/content and open it. Add the following line at the end of Language-ext.properties and save it:

```
x={0}
```

The code above shows a mapping between x and a {0} parameter.

Then, we need to override the JSP file `login.jsp` using the following steps:

1. Create a folder `login` in `/ext/ext-web/docroot/html/portlet`.
2. Copy the JSP file `login.jsp` from `/portal/portal-web/docroot/html/portlet/login` to `/ext/ext-web/docroot/html/portlet/login`.
3. Locate the JSP file `login.jsp` in the `/ext/ext-web/docroot/html/portlet/login` folder and open it.
4. Replace the lines between `<c:when test="<%=" themeDisplay.isSignedIn() %>">` (not included) and `</c:when>` (not included) with the following lines and save it:

```
<%String signedInAs = "Edit your profile";
 String logout="logout";
 String signout= "<a style=\"text-decoration:none\" href=\""
 + themeDisplay.getURLSignOut().toString() + "\">"
 + logout + "";
if (themeDisplay.isShowMyAccountIcon()) {
 signedInAs = "<a style=\"text-decoration:none\" href=\""
 + themeDisplay.getURLMyAccount().toString() +
 "\">" + signedInAs + "";
}
%>
<div style="background-color: #EEEEEE; padding: 5px; " >

<font face="Verdana, Arial, Helvetica, sans-serif"
 font color="#000000" font size="2em">
<%= user.getGreeting()%>

<font face="Verdana, Arial, Helvetica, sans-serif"
 font color="#ABABAC" font size="1.6em">
<%= LanguageUtil.format(pageContext, "x", signedInAs) %> |
<%= LanguageUtil.format(pageContext, "x", signout) %>

</div>
```

The code above shows the current user's greeting, profile, and sign out. Meanwhile, the profile has a link to edit profile and sign out has a link associated to the action of signing out. The background color is provided too. Of course, you can change all this according to your requirements.

## Creating a customized account on the fly

We have successfully implemented the use case *Customizing login view*. Now let's implement the use case *Creating a customized account on the fly*. To do so, we will add a custom code to the action and JSP view. That is, we're going to show how to override the action in Ext.

First, we're going to customize the action in the following manner:

1. Create a package `com.liferay.portlet.login.action` in the `/ext/ext-impl/src` folder.
2. Copy the Java file `CreateAccountAction.java` from the package `com.liferay.portlet.login.action` in the `/portal/portal-impl/src` folder to the package `com.liferay.portlet.login.action` in the `/ext/ext-impl/src` folder.
3. Locate `CreateAccountAction.java` in the package `com.liferay.portlet.login.action` under the `/ext/ext-impl/src` folder.
4. Update the line `boolean autoPassword = true;` as  
`boolean autoPassword = false;`
5. Now add the following lines before the line `if (openIdPending) {` in the `addUser` method and save it:

```
String question = ParamUtil.getString(actionRequest,
 "reminderQueryQuestion");
String answer = ParamUtil.getString(actionRequest,
 "reminderQueryAnswer");
UserServiceUtil.updateReminderQuery(user.getUserId(),
 question, answer);
```

The code above disables `autoPassword` first, that is, the password entered by the users on the fly will be saved. Then it uses the `UserServiceUtil.updateReminderQuery` method to update reminder query question and reminder query answer.

Then, we need to override the JSP file `create_account.jsp` as follows:

1. Copy the JSP file `create_account.jsp` from the `/portal/portal-web/docroot/html/portlet/login` folder to `/ext/ext-web/docroot/html/portlet/login` folder.
2. Locate the JSP file `create_account.jsp` in the `/ext/ext-web/docroot/html/portlet/login` file and open it.

3. Add the following lines before the line `<c:if test="<%=_PropsValues.LOGIN_CREATE_ACCOUNT_ALLOW_CUSTOM_PASSWORD %>">` in this file and save it:

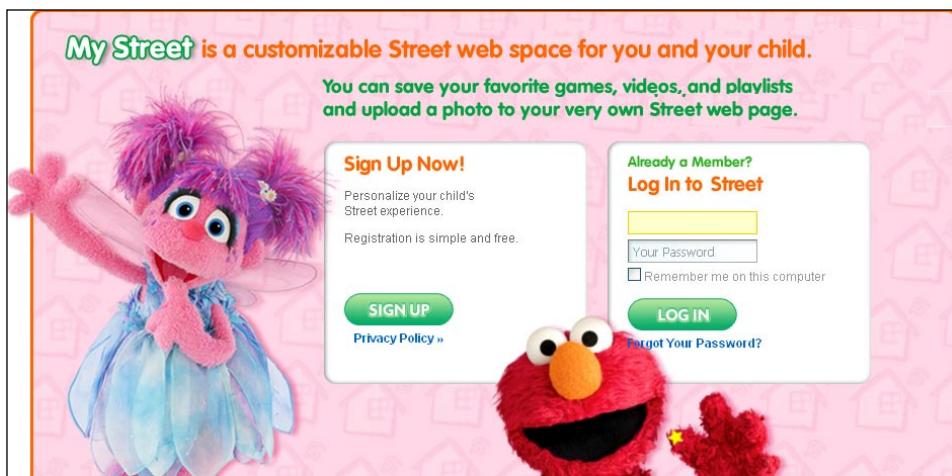
```
<div class="ctrl-holder">
 <label class="prompt">Your Password</label>
 <input type="password" id="password1"
 name="password1"
 value="" class="buffer"
 onKeyPress="Liferay.Util.checkMaxLength(this, 36);"
 onChange="Liferay.Util.checkMaxLength(this, 36);"/>
</div>
<div class="ctrl-holder">
 <label>Verify Password</label>
 <input type="password" id="password2"
 name="password2"
 value="" class="buffer"
 onKeyPress="Liferay.Util.checkMaxLength(this, 36);"
 onChange="Liferay.Util.checkMaxLength(this, 36);"/>
</div>
<div class="ctrl-holder">
 <label>Reminder Query Question</label>
 <liferay-ui:input-field model="<%=_User.class %>"
 bean="<%=_user2 %>" field="reminderQueryQuestion" />
</div>
<div class="ctrl-holder">
 <label>Reminder Query Answer</label>
 <liferay-ui:input-field model="<%=_User.class %>"
 bean="<%=_user2 %>" field="reminderQueryAnswer" />
</div>
```

The code above shows inputs of Your Password, Verify Password, Reminder Query Question, and Reminder Query Answer.

Cool! You have successfully implemented the use case *Creating a customized account on the fly*. If you deploy the above updates, you would see an update in the look and feel of use cases *Creating a customized account on the fly* and *Customizing login view*.

## Building personal community—My Street

My Street—an example of a personal community, is a customizable Book Street web space for parents and children. In My Street, you can save your favorite games, videos and playlists, and the My Street theme—your background color. As shown in the following screenshot, there is a page `my_street` with a portlet `myStreet`, where an end user can sign up, log in, or handle an issue such as **Forgot Your Password?**.



When clicking on the **SIGN UP** button, this **My Street** portlet will allow the end users to set up his/her account such as creating a nickname, password hint question, password hint answer, your password, and verify password. Further, as the end user, you can set up favorite games, videos and playlists, and background color.

You can have your own favorites: number of favorites displayed in My Street (for example, 4, 20, and 35) and My Street theme, (for example, default, Abby Cadabby, Bert, Big Bird, Cookie Monster, and so on). A set of default My Street themes is predefined in `/cms_services/images/my_street/` under the folder `$CATALINA_HOME/webapps/` and you can choose any one of them at any time. At the same time, you can upload a photo to your own Book Street web page.

When logged in, the My Street theme will be applied on Home page, Games landing page, Videos landing page, and Playlist landing page. For example, current user's My Street theme could be applied on Playlist landing page.

You may play videos, games, and playlists when you visit the web site [www.bookpubstreet.com](http://www.bookpubstreet.com). When you find your favorite videos, games, and playlists, you can add them into My Street. As shown in the following screenshot, you could be playing a playlist **Tickle Time**. If you are interested in this playlist, just click on the **Add to My Street** button. The playlist **Tickle Time** will be added into My Street as your favorite playlist. In this section, we will show how to implement these features.



## Customizing user model

First, let's customize user model in `service.xml` in order to support extended user and user preferences. To do so, use the following steps:

1. Create a package `com.ext.portlet.user` in the `/ext/ext-impl/src` folder.
2. Create an XML file `service.xml` in the package `com.ext.portlet.comment` and open it.
3. Add the following lines in this file and save it:

```
<?xml version="1.0"?><!DOCTYPE service-builder PUBLIC "-//
Liferay//DTD Service Builder 5.2.0//EN" "http://www.liferay.com/
dtd/liferay-service-builder_5_2_0.dtd">
<service-builder package-path="com.ext.portlet.user">
 <namespace>ExtUser</namespace>
 <entity name="ExtUser" uuid="false" local-service="true"
 remote-service="true"
 persistence-class="com.ext.portlet.user.service.
 persistence. ExtUserPersistenceImpl">
 <column name="userId" type="long" primary="true" />
```

```

<column name="favorites" type="int" />
<column name="theme" type="String" />
<column name="printable" type="boolean" />
<column name="plainPassword" type="String" />
<column name="creator" type="String" />
<column name="modifier" type="String" />
<column name="created" type="Date" />
<column name="modified" type="Date" />
</entity>
<entity name="ExtUserPreference" uuid="false"
 local-service="true" remote-service="true"
 persistence-class="com.ext.portlet.user.service.
 persistence.ExtUserPreferencePersistenceImpl">
 <column name="userPreferenceId" type="long" primary="true" />
 <column name="userId" type="long"/>
 <column name="favorite" type="String" />
 <column name="favoriteType" type="String" />
 <column name="date_" type="Date" />
</entity>
<exceptions>
 <exception>ExtUser</exception>
 <exception>ExtUserPreference</exception>
</exceptions>
</service-builder>

```

The code above shows the customized user model, including `userId` associated with `USER_` table, `favourites`, `theme`, `printable`, `plainPassword`, and so on. This code also shows user preferences model, including `userPreferenceId`, `userId`, `favorite` (for example, game/video/playlist UID), `favoriteType` (for example, game/video/playlist), and the updated date `date_`. Of course, these models are extensible. You can extend these models for your unique current needs or future requirements.

Enter the following tables into database through command prompt:

```

create table ExtUser (
 userId bigint not null primary key,
 creator varchar(125), modifier varchar(125),
 created datetime null, modified datetime null,
 favorites smallint, theme varchar(125),
 plainPassword varchar(125), printable boolean);
create table ExtUserPreference (
 usePreferenceId bigint not null primary key,
 userId bigint not null, favorite varchar(125),
 favoriteType varchar(125), date_ datetime null
) ;

```

The preceding code shows the database SQL script for customized user model and user preference model. Similar to XML model ExtUser, this code shows the userId, favorites, theme, plainPassword, and printable table fields. Again, for the XML model ExtUserPreference, this code shows the userId, favorite, favoriteType, date\_, and userPreferenceId table fields.

Afterwards, we need to build a service with ServiceBuilder. After preparing service.xml, you can build services. To do so, locate the XML file /ext/ext-impl/build-parent.xml, open it, add the following lines between </target> and <target name="build-service-portlet-reports">, and save it:

```
<target name="build-service-portlet-extUser">
 <antcall target="build-service">
 <param name="service.file"
 value="src/com/ext/portlet/user/service.xml" />
 </antcall>
</target>
```

When you are ready, just double-click on the Ant target build-service-portlet-extUser. ServiceBuilder will build related models and services for extUser and extUserPreference.

## Building the portlet My Street

Similar to how we had built portlet Ext Comment, we can build the portlet My Street as follows:

1. Configure the portlet My Street in both portlet-ext.xml and liferay-portlet-ext.xml files.
2. Set title mapping in the Language-ext.properties file.
3. Add the My Street portlet to the Book category in the liferay-display.xml file.
4. Finally, specify Struts actions and forward paths in the struts-config.xml and tiles-defs.xml files, respectively.

Then, we need to create Struts actions as follows:

1. Create a package com.ext.portlet.my\_street.action in the folder /ext/ext-impl/src.
2. Add the Java files ViewAction.java, EditUserAction.java, and CreateAccountAction.java in this package.
3. Create a Java file AddUserLocalServiceUtil.java in this package and open it.

4. Add the following methods in `AddUserLocalServiceUtil.java` and save it:

```

public static ExtUser getUser(long userId) {
 ExtUser user = null;
 try{
 user = ExtUserLocalServiceUtil.getExtUser(userId);
 }
 catch (Exception e){}
 if(user == null){
 user = ExtUserLocalServiceUtil.createExtUser(userId);
 try{
 ExtUserLocalServiceUtil.updateExtUser(user);
 }
 catch (Exception e) {}
 }
 return user;
}
public static void deleteUser(long userId) {
 try{
 ExtUserLocalServiceUtil.deleteExtUser(userId);
 }
 catch (Exception e){}
}
public static void updateUser(ActionRequest actionRequest,
 long userId) {
 /* ignore details */
}
public static List<ExtUserPreference> getUserPreferences(
 long userId, int limit){
 /* ignore details */
}
public static ExtUserPreference addUserPreference(
 long userId, String favorite, String favoriteType){
 /* ignore details */
}

```

As shown in the code above, it shows methods to get `ExtUser` and `ExtUserPreference`, to delete `ExtUser`, and to add and update `ExtUser` and `ExtUserPreference`.

In addition, we need to provide default values for private page and friendly URL in `portal-ext.properties` as follows:

```

ext.default_page.my_street.private_page=false
ext.default_page.my_street.friend_url=/my_street

```

The code above shows the default private page of `my_street` as `false`, default friendly URL of `my_street` as `/my_street`. Therefore, you can use VM service to generate a URL in order to add videos, games, and playlists into My Street.

## Adding Struts view page

Now we need to build Struts views pages `view.jsp`, `forget_password.jsp`, `create_account.jsp`, `congratulation.jsp`, and `congrates_uid.jsp`. The following are main steps to do so:

1. Create a folder `my_street` in `/ext/ext-web/docroot/html/portlet/ext/`.
2. Create JSP files pages `view.jsp`, `view_password.jsp`, `userQuestions.jsp`, `edit_account.jsp`, `create_account.jsp`, `congratulation.jsp`, and `congrates_uid.jsp` in `/ext/ext-web/docroot/html/portlet/ext/my_street/`.

Note that `congrates_uid.jsp` is used for the pop-up congratulation of **Add to My Street**. When you click on the **Add to My Street** button, a window with `congrates_uid.jsp` will pop up. `userQuestions.jsp` is used when the user has forgotten the password of My Street, `view_password.jsp` is for general view of My Street, `congratulation.jsp` is used to represent successful information after creating user account, and `edit_account.jsp` is used to create/update user account.

## Sharing the My Street theme

The My Street theme can be shared by other portlets on top of Liferay portal. The `com.ext.portlet.user.service.ExtUserLocalServiceUtil` service resides in `ext-service.jar`, which is a part of external services as well as that of `portal-service.jar`. So the other portlets, whether they are developed in Ext or in Plugins SDK, can use this service and directly get the My Street theme of the current user. The following code shows this ability:

```
ThemeDisplay themeDisplay = (ThemeDisplay) renderRequest.getAttribute(WebKeys.THEME_DISPLAY);
ExtUser extUser = ExtUserLocalServiceUtil.getExtUser(themeDisplay.getUserId());
String theme = extUser.getTheme();
```

For instance, the portlet Playlist landing can use the code above to get the My Street theme first. Then it can display the My Street theme as background color in its view JSP file.

## Adding videos, games, and playlists into My Street

We can reuse jQuery service as mentioned earlier in order to add videos, games, and playlists into My Street. First of all, we can generate a URL follows:

```
String shareTitle = "Add to My Street";
String url = "/c/portal/render_portlet?p_p_id=myStreet&p_l_id=" + p_l_
id + "&uid=23&assetType=PLAYLIST";
```

Then we can get the value of p\_l\_id using the following code:

```
Layout shareLayout = ExtJournalUtil.getFriendlyURLLayout(
 themeDisplay.getGroupId(), PropsUtil.get(
 ext.default_page.my_street.private_page),
 PropsUtil.get("ext.default_page.my_street.friend_url"));
long p_l_id = shareLayout.getPlid();
```

At the same time, we need to add portlet render in ViewAction.java under the package com.ext.portlet.my\_street.action.

```
public ActionForward render(ActionMapping mapping, ActionForm form,
 PortletConfig portletConfig, RenderRequest renderRequest,
 RenderResponse renderResponse)
 throws Exception {
 String fwd = "portlet.ext.my_street.view";
 String uid = renderRequest.getParameter(myStreetUrl_uid);
 String assetType = renderRequest.getParameter
 (myStreetUrl_assetType);
 if(uid != null && assetType != null){
 ThemeDisplay themeDisplay = (ThemeDisplay)renderRequest.
 getAttribute(WebKeys.THEME_DISPLAY);
 AddUserLocalServiceUtil.addUserPreference(
 themeDisplay.getUserId(), uid, assetType);
 fwd = "portlet.ext.my_street.congrates_uid";
 }
 return mapping.findForward(fwd);
}
```

The code above shows two rendering scenarios. If uid or assetType is null, then render a normal view. Otherwise, add user preference and, moreover, render a success view for Add to My Street. It is simple for other portlets to share this feature in order to add videos, games, or playlists into My Street. For the **Add to My Street** button, it generates a URL as stated above. Then it calls `Liferay.PortletSharing.showShareInfo` as follows:

```
javascript: Liferay.PortletSharing.showShareInfo("<%= url %>",
 "<%= shareTitle %>").
```

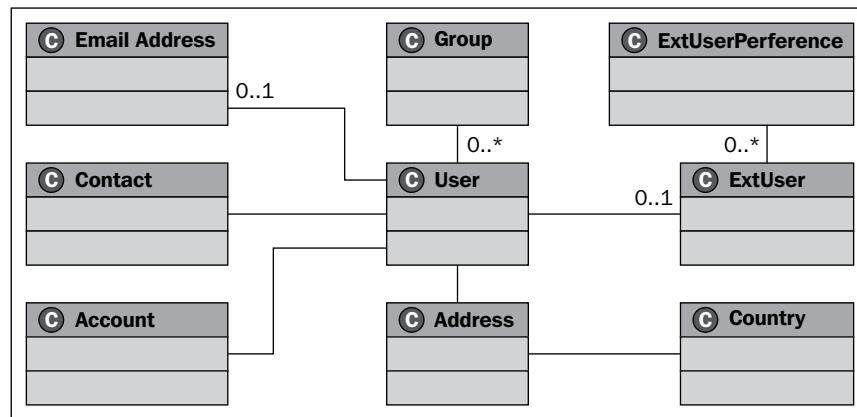
## Using personal community efficiently

On top of the portal, all of the users get personal space that can be made public or kept private. These public and private pages will be published as a web site with a unique friendly URL. You can customize My Community (or called personal community) for example, how the space looks via the layout templates, what tools and applications are included, what goes into Document Library and Image Gallery, and who can see and access it. This would be sufficient for most of the requirements. But in some requirements, users want to have preference data as well. For example, users may want to have their own favorite games and videos. In this case, we have to extend the user table and user preferences.

## Extending user account and user preferences

It would be pretty helpful to recover, and moreover extend, the user data if we know the user model very well. As shown in the following figure, a *User* may belong to *Group*, and *Group* may have many users as its members. A user may have *Email Address*, *Address*, *Account*, and an address may belong to a *Country*. More importantly, each user must have a *Contact*. Thus, when you recover users, keep in mind to recover contacts at the same time.

When adding user preferences, do not overwrite the user table `USER_` directly. If you do so, you would have issues while upgrading your custom code to a higher version. But you can create a new table `ExtUser` to hold single preference data, for example `20` – the number of your favorites (videos, games, or playlists), `elmo` – your favorite theme, and so on. Further, you can create a new table `ExtUserPreference` to save multiple preferences, for example 5 favorite videos, 10 favorite games and 15 favorite playlists as mentioned earlier.



## Setting My Community

Liferay portal provides the ability to configure default user public and private pages (or called layouts), and default guest public pages. The maximum number of members a personal community can have is one. A user with a personal zone cannot make anyone else a member of his/her community. If you own a personal community, you can have administrative rights over the pages within the community. You can find default settings for default user public pages in the `portal.properties` file as follows:

```
default.user.public.layout.name=Welcome
default.user.public.layout.template.id=2_columns_ii
default.user.public.layout.column-1=82,23
default.user.public.layout.friendly.url=/home
default.user.public.layout.regular.theme.id=classic
default.user.public.layouts.lar=${resource.repositories.root}/deploy/
default_user_public.lar
layout.types=portlet,panel,embedded,article,url,link_to_layout
layout.user.public.layouts.enabled=true
layout.user.public.layouts.modifiable=true
layout.user.public.layouts.auto.create=true
```

The code above shows how to specify a LAR file that can be used to create the user private layouts. If this property was set, the previous layout properties will be ignored. It also shows default theme, default layout template, default friendly URL, and so on. The properties `layout.user.public.layouts.enabled` and `layout.user.public.layouts.auto.create` are both set to true. It means that the users will have public layouts that will be automatically created. For even more complex behavior, you can override the `addDefaultUserPublicLayouts` method in `com.liferay.portal.events.ServicePreAction` under the `/portal/portal-impl/src/` folder.

## Using Control Panel to manage My Account

You can also use Control Panel to manage My Account and My Pages in a uniform view. For look and feel of Control Panel, you can check theme specifications in `/portal/portal-web/docroot/html/themes/control_panel`. You can also find the default settings of Control Panel in the `portal.properties` file as follows:

```
control.panel.layout.name=Control Panel
control.panel.layout.friendly.url=/manage
control.panel.layout.regular.theme.id=controlpanel
```

The code above shows name, friendly URL, and theme of the layout for Control Panel.

By default, the portal allows the users who forgot their passwords to obtain a new one through email. This functionality will allow a second security mechanism based on reminder queries. This mechanism can be configured through several properties in the `portal.properties` file as follows:

```
users.reminder.queries.enabled=true
users.reminder.queries.custom.question.enabled=true
users.reminder.queries.questions=what-is-your-primary-frequent-flyer-
number,what-is-your-library-card-number,what-was-your-first-phone-
number,what-was-your-first-teacher's-name,what-is-your-father's-
middle-name
```

As shown in the code above, the first property enables the mechanism of reminder queries and makes them essential to obtain a new password. The second property allows the user to write his/her own question so that he/she can choose his/her own question in addition to the ones offered to him/her by default. The third property allows us to write preset reminder queries. By the way, any organization can define its own reminder queries instead of the default ones in the Control Panel.

## Using dynamic query API

The Dynamic Query API provides an elegant way to define complex queries without a complex setup, or a stiff and abstract learning curve. This API allows us to leverage the existing mapping definitions through access to the Hibernate Session. The interface of the API is specified in the `com.liferay.portal.kernel.dao.orm.DynamicQuery` class under the folder `/portal/portal-kernel/src`. The following is the interface:

```
public DynamicQuery add(Criterion criterion);
public DynamicQuery addOrder(Order order);
public void compile(Session session); public List list();
public List list(boolean unmodifiable);
public void setLimit(int start, int end);
public DynamicQuery setProjection(Projection projection);
```

Subqueries, associations, projections, and aliases are the features available in Dynamic Query API. For example, let us suppose that you want to find Ext comments by `classPK` and `classNameId`. You can specify it with the Dynamic Query API as follows:

```
public static List<ExtComment> getComments(long classNameId, long
classPK) throws Exception {
List<ExtComment> comments = Collections.synchronizedList(new
ArrayList<ExtComment>());
DynamicQuery query = DynamicQueryFactoryUtil.forClass(ExtComment
.class).add(PropertyFactoryUtil.forName(

```

```

 "classPK") .eq(new Long(classPK)) .add(
 PropertyFactoryUtil.forName("classNameId") .
 eq(new Long(classNameId)));
 try{
 List<Object> assets = ExtCalEventLocalServiceUtil.
 dynamicQuery(query);
 for (Object obj: assets) {
 ExtComment asset = (ExtComment)obj;
 comments.add(asset);
 }
 } catch (Exception e){
 return comments;
 }
}

```

The code above shows a simple dynamic query on the `ExtComment` table. As mentioned earlier, we could use subquery, order, associations, projections, and aliases to find the most popular articles on the `TagsAsset` and `JournalArticle` tables.

## Using pop ups

There are two kinds of pop ups—Floating Div pop up and Window pop up. Let's look at these pop ups in detail.

### Applying Floating DIV pop up

Liferay portal provides a class called `Expanse.Popup` to implement the Floating DIV pop up. The following JavaScript code will make an asynchronous call to a URL, which will place content in a page as follows:

```

showPopupDialog: function(url, title) {
 new Expanse.Popup({ fixedcenter: true,
 header: title, width: 400, height: 350,
 url: url, modal: true });
},

```

The code above shows a JavaScript function `showPopupDialog`. It defines a `Expanse.Popup` pop up with parameters: `title`, `position`, `modal`, `width`, and `height`. If the URL is a portlet URL, it must have the `windowState` parameter set to `LiferayWindowState.EXCLUSIVE`. To implement a button or a link to close the pop up, you can use the code: `Expanse.Popup.close(this)`. Moreover, you can find more information about pop ups in the JavaScript file `/portal/portal-web/docroot/html/js/liferay/popup.js`.

## Employing window pop up

Window pop up is loaded in a new window. In order to do that, we must set the `windowState` parameter in the portlet URL to `LiferayWindowState.POP_UP`. The following code shows a slideshow as a window pop-up in Image Gallery:

```
function <portlet:namespace />viewSlideShow() {
 var slideShowWindow = window.open('
 <portlet:renderURL windowState="<%=_LiferayWindowState.
 POP_UP.toString()%>">
 <portlet:param name="struts_action"
 value="/image_gallery/view_slide_show" />
 <portlet:param name="folderId"
 value="<%=_String.valueOf(folderId)%>" />
 </portlet:renderURL>',
 'slideShow','directories=no,location=no,menubar=no, resizable=yes,
 scrollbars=yes,status=no, toolbar=no');
 slideShowWindow.focus();
}
```

If you want to close the pop up, you can use the `window.close()` code. Further, if you want to close the parent page, you can use the `opener.close()` code.

## Summary

This chapter discussed how to build My Community in general, and how to customize and extend this feature as well. First, it introduced how to share web site, pages, or portlets with friends. Then it addressed how to set up the most popular journal articles and view counter for any assets. This chapter further discussed how to personalize user comments. It also introduced how to customize My Account and how to build My Street with personalized preferences. Finally, it addressed the best practices to use My Community efficiently, including dynamic query API, pop-up JavaScript, My Community settings, My Account Control Panel, user account extension, and user preferences.

In the next chapter, we're going to develop layout templates and themes.

# 9

## Developing Layout Templates and Themes

Our web sites [www.bookpubstreet.com](http://www.bookpubstreet.com) and [www.bookpubworkshop.com](http://www.bookpubworkshop.com) are made up of a great amount of pages. Each page consists of a set of portlets with a specific look and feel specified by themes. Moreover, all of the portlets in a page are arranged via layout templates. Normally, themes specify the overall look and feel of these web sites; whereas, layout templates specify the arrangement of portlets in pages. Generally speaking, a theme is a user interface design that makes the portal more user friendly and visually pleasing.

Liferay portal provides layout templates in order to describe how various columns and rows are arranged to display portlets. It also provides themes that can be used to customize the overall look and feel of web sites and pages. A theme controls the whole look and feel of the pages generated by Liferay portal using CSS, images, JavaScript, and Velocity templates.

This chapter will discuss how to develop layout templates in both Ext and Plugins SDK, and show how to build themes in Plugins SDK, in general. This chapter will also discuss how to build layout templates in Ext and Plugins SDK. Meanwhile, it will introduce how to add Velocity services in themes. Finally, it will introduce how to use Plugins SDK more efficiently.

By the end of this chapter, you will have learned how to:

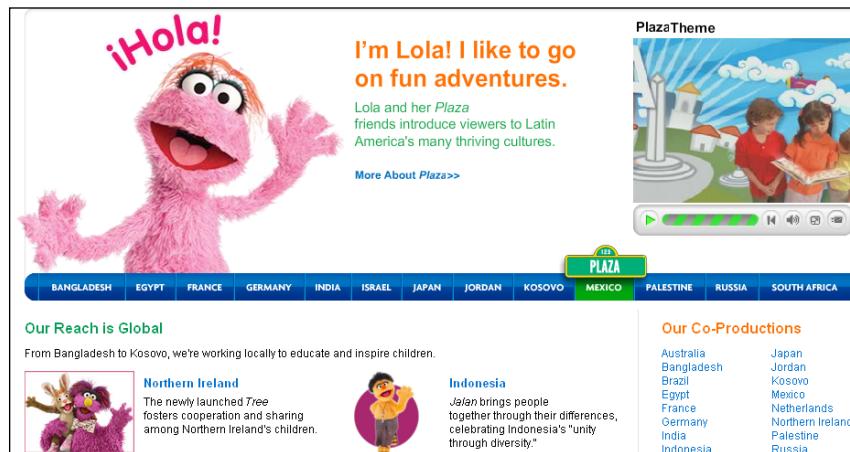
- Build layout templates in Ext
- Develop layout templates in Plugins SDK
- Build themes in Plugins SDK
- Add Velocity services in themes
- Use Plugins SDK more efficiently

## Building layout templates in Ext

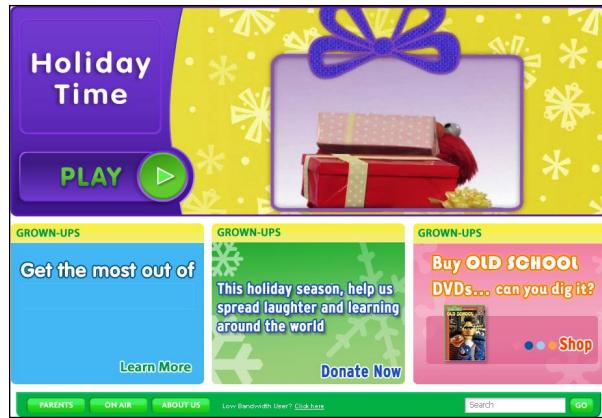
Layout Templates are ways of choosing how portlets will be arranged on a page.

Layout templates are usually a grid-like structure and are mostly created with HTML tables. They form the body of a page the large area, where you can drag and drop the portlets to create pages. In short, a layout template describes how various columns and rows are arranged to display the portlets.

As shown in the following screenshot for the use case *Book Workshop Home*, the web site [www.bookpubworkshop.com](http://www.bookpubworkshop.com) is made up of many workshop pages. Each page consists of a set of portlets. Portlets are arranged in a specific way in pages. For instance, the **AROUND THE WORLD** page has three portlets: **I'm Lola**, **Our Research is Global**, and **Our Co-Productions**. The whole page is aligned in the centre, while **I'm Lola**, **Our Research is Global**, and **Our Co-Productions** are represented by layout template 1-2 columns. As you can see, Lola's hand is around 25 pixels out of the box. This means that **I'm Lola** portlet should have padding-left 0 pixel; the **Our Research is Global** portlet should have padding-left 25 pixels; and the **Our Co-Productions** portlet should have padding-right 25 pixels with a border on the left side.

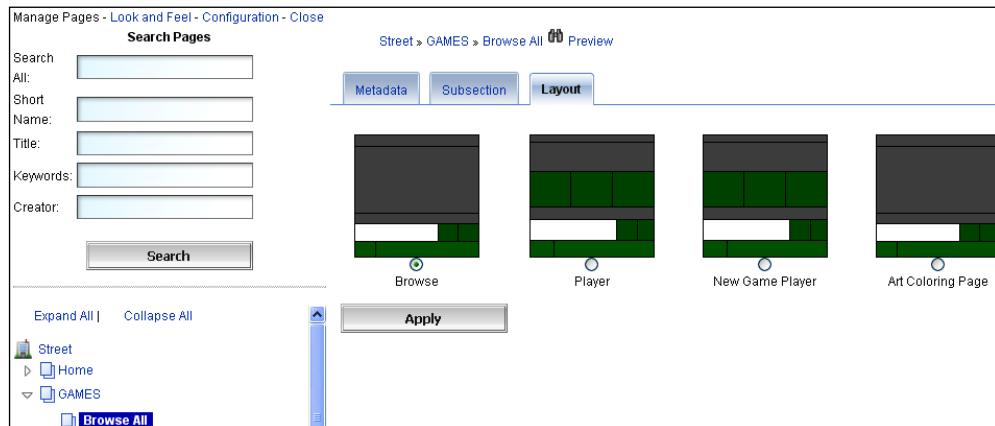


A similar situation happens in the web site [bookpubstreet.com](http://bookpubstreet.com). As shown in the following screenshot for the use case *Book Street Home*, the web site [bookpubstreet.com](http://bookpubstreet.com) is made up of many street pages. Each page consists of a set of portlets arranged in a specific way. For example, the **Home** page has five portlets: **Holiday Time**, **Get the most out of...**, **This holiday season...**, **Buy OLD SCHOOL DVDs**, and **Parents**. The whole page is aligned in center, while these five portlets are represented as 1-3-1 columns. As you can see, the portlets **Holiday Time**, **Get the most out of...**, **This holiday season...**, and **Buy OLD SCHOOL DVDs...** should have padding-left 0 pixel; whereas the portlet **Parents** should have padding-left 25 pixels and padding-right 25 pixels.



The feature of dragging and dropping portlets to build pages is helpful and flexible for experienced users who are familiar with the portal. But some end users, such as content producers, prefer to use predefined layout templates in building pages. For example, the content producers in Palm Tree Publications do not expect to drag-and-drop portlets to create pages. They expect Admin to predefine the layout templates first, and then they just choose one of the layout templates and apply it on a specific page. All of the portlets related to this page as well as the selected layout template should be populated automatically. Then, they just update the content of portlets if applicable. Therefore, the layout templates should be applied in different ways: drag-and-drop, bundle with pages, and so on.

As shown in the following screenshot, a set of predefined layout templates is available for a content producer to build pages. When a content producer chooses a layout template and applies the selected layout template on a page, all portlets associated with the selected layout templates will be populated to this page. After that, a content producer just populates the content of portlets in this page.



As mentioned in previous chapters, we have discussed how to dynamically apply layout templates to pages. How do we build these layout templates? In this section, we're going to build the layout templates in Ext. By the way, there is also a feature to build layout templates in Plugins SDK. This will be introduced in the coming section.

## Constructing custom layout templates

In brief, a layout template describes how various columns and rows are arranged in pages to display the portlets on top of the portal. By default, the portal comes with several built-in layout templates. But if you have a layout of complex pages as mentioned above, especially for your customized pages, you need to create custom layout templates.

## Experiencing default layout templates

Layout templates are located in the /portal/portal-web/docroot/html/layouttpl/standard and /portal/portal-web/html/custom directories as a series of .tpl files and .png files. By default, there are two groups of layout templates: **standard** and **custom**. You can find these two groups of layout templates in the following manner:

1. Locate the XML file `liferay-layout-templates.xml` in the /portal/portal-web/docroot/WEB-INF folder and open it.
2. Check the `<standard>` and `</standard>` XML tags:

```
<?xml version="1.0"?>
<!DOCTYPE layout-templates PUBLIC
"-//Liferay//DTD Layout Templates 5.2.0//EN"
"http://www.liferay.com/dtd/liferay-layout-
templates_5_2_0.dtd">
<layout-templates>
 <standard>
 <layout-template id="exclusive">
 <template-path>
 /layouttpl/standard/exclusive.tpl
 </template-path>
 <wap-template-path>
 /layouttpl/standard/exclusive.wap.tpl
 </wap-template-path>
 <thumbnail-path>
 /layouttpl/standard/exclusive.png
 </thumbnail-path>
 </layout-template>
 <layout-template id="max">
 <template-path>
 /layouttpl/standard/max.tpl
 </template-path>
 </layout-template>
 </standard>
</layout-templates>
```

```
</template-path>
<wap-template-path>
 /layouttpl/standard/max.wap.tpl
</wap-template-path>
<thumbnail-path>
 /layouttpl/standard/max.png
</thumbnail-path>
</layout-template> <!--ignore details -->
</standard>
</layout-templates>
```

In the same way, you can find the `<custom>` and `</custom>` XML tags as follows:

```
<custom>
 <layout-template id="freeform" name="Freeform">
 <template-path>
 /layouttpl/custom/freeform.tpl
 </template-path>
 <wap-template-path>
 /layouttpl/custom/freeform.wap.tpl
 </wap-template-path>
 <thumbnail-path>
 /layouttpl/custom/freeform.png
 </thumbnail-path>
 <roles><role-name>User</role-name></roles>
 </layout-template>
 <layout-template id="1_column" name="1 Column">
 <template-path>
 /layouttpl/custom/1_column.tpl
 </template-path>
 <wap-template-path>
 /layouttpl/custom/1_column.wap.tpl
 </wap-template-path>
 <thumbnail-path>
 /layouttpl/custom/1_column.png
 </thumbnail-path>
 </layout-template> <!--ignore details -->
</custom>
```

The code above shows that the standard layout templates include `exclusive`, `max`, and `pop_up`; whereas the custom layout templates include by default `freeform`, `1_column`, and so on. Each layout template is made up of three files: `.png` for thumbnail icon, `.tpl` for regular web browsers, and `.wap.tpl` for the **WAP** version (**Wireless Application Protocol**). For instance, the `1_column` layout template includes the `1 Column` display name, the `/layouttpl/custom/1_column.tpl` template path, the WAP template path `/layouttpl/custom/1_column.wap.tpl`, and the `/layouttpl/custom/1_column.png` thumbnail path.

## Adding customized layout templates

As stated above, the default layout templates are located in the /portal/portal-web/docroot/html/layouttpl/standard and /portal/portal-web/html/custom folders. Each layout template is made up of a .png thumbnail icon, .tpl for regular web browsers, and .wap.tpl for **WAP** version. The layout templates are registered in /portal/portal-web/docroot/WEB-INF/liferay-layout-templates.xml.

Now let's create customized layout templates in Ext as well. Considering the above use cases *Book Workshop Home* and *Book Street Home*, we're going to create two layout templates: book\_workshop\_home and book\_street\_home.

The following are the main steps to build the book\_street\_home layout template:

1. Create a folder named layouttpl under the /ext/ext-web/docroot folder.
2. Create a folder named custom under the /ext/ext-web/docroot/layouttpl folder.
3. Create a book\_street\_home.png thumbnail icon under the /ext/ext-web/docroot/layouttpl/custom folder.
4. Create a book\_street\_home.tpl file in the /ext/ext-web/docroot/layouttpl/custom folder and open it.
5. Add the following lines in book\_street\_home.tpl and save it.

```
<div id="content-wrapper">
 <table class="lfr-grid" id="layout-grid">
 <tr id="row-1">
 <td class="lfr-column" id="column-1"
 valign="top" colspan="3"
 style="border: 0px solid green;
 padding-left: 0px; width: 917px; ">
 $processor.processColumn("column-1")
 </td>
 </tr>
 <tr id="row-2">
 <td class="lfr-column thirty" id="column-2"
 valign="top" style="border: 0px solid green;
 padding-left: 0px; ">
 $processor.processColumn("column-2")
 </td>
 <td class="lfr-column thirty" id="column-3"
 valign="top" style="border: 0px solid green;
 padding-left: 0px; ">
 $processor.processColumn("column-3")
 </td>
 <td class="lfr-column thirty" id="column-4"
 valign="top" style="border: 0px solid green;
 padding-right: 0px; ">
```

```
$processor.processColumn("column-4")
</td>
</tr>
<tr id="row-3">
 <td class="lfr-column" id="column-5" valign="top"
 colspan="3" style="border: 0px solid green;
 padding-left: 25px; padding-right: 25px;
 width: 917px; ">
 $processor.processColumn("column-5")
 </td>
</tr>
</table>
</div>
```

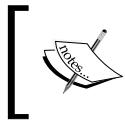
The code above shows a way to specify five evenly spaced columns with a table. Each table cell `<td>` has an associated CSS class (for example, `lfr-column`), a style, as well as an ID (for example, `column-1`). Here it uses `style` of `<td>`. You can modify `style` with different values in order to get a different look and feel. Especially, columns (for example, `column-1` and `column-5`) are arranged by `padding-left` and `padding-right`. In addition, the CSS class may be customized by modifying the theme you are using to display the layout.

Then, you can create a `book_street_home.wap.tpl` file in the folder `/ext/ext-web/docroot/layouttpl/custom` and add the following lines:

```
<table>
<tr><td colspan="3">
 $processor.processColumn("column-1")
</td></tr>
<tr>
 <td>$processor.processColumn("column-2")</td>
 <td>$processor.processColumn("column-3")</td>
 <td>$processor.processColumn("column-4")</td>
</tr>
<tr>
 <td colspan="3">
 $processor.processColumn("column-5")
 </td>
</tr>
</table>
```

As shown in the code above, WAP version doesn't have the benefit of CSS and so it settles for five evenly spaced columns. In runtime, the portal will automatically detect the client being used to connect to `bookpubstreet.com` and serve an appropriate layout template. If the client is a phone, it will serve the `book_street_home.wap.tpl` file. Otherwise, it will serve the `book_street_home.tpl` file.

The thumbnail icon specifies what the layout looks like. You can customize thumbnail icons `book_workshop_home.png` and `book_street_home.png` in an image manipulation program such as **GIMP**, Adobe Photoshop, and so on.



**GIMP** is the GNU image manipulation program, which is a distributed piece of software for tasks such as photo retouching, image composition, and image authoring. Refer to <http://www.gimp.org>.

Similarly, you can create the `book_workshop_home` layout template. Just create a `book_workshop_home.png` thumbnail icon, a `book_workshop_home.tpl` file, and `book_workshop_home.wap.tpl`.

## Registering layout templates

We have created customized layout templates successfully. Now let's register the above layout templates `book_street_home` and `book_workshop_home`:

1. Locate the XML file `liferay-layout-templates.xml` in the `/portal/portal-web/docroot/WEB-INF` folder.
2. Copy the XML file `liferay-layout-templates.xml` from the `/portal/portal-web/docroot/WEB-INF` to folder `/ext/ext-web/docroot/WEB-INF` folder.
3. Locate the XML file `liferay-layout-templates.xml` in the `/ext/ext-web/docroot/WEB-INF` folder and open it.
4. Add the following lines before the lines `</custom> </layout-templates>` in `liferay-layout-templates.xml` and save it:

```
<layout-template id="book_street_home" name="Street Home">
 <template-path>
 /layouttpl/custom/book_street_home.tpl
 </template-path>
 <wap-template-path>
 /layouttpl/custom/book_street_home.wap.tpl
 </wap-template-path>
 <thumbnail-path>
 /layouttpl/custom/book_street_home.png
 </thumbnail-path>
</layout-template>
<layout-template id="book_workshop_home" name="Workshop Home">
 <template-path>
 /layouttpl/custom/book_workshop_home.tpl
 </template-path>
 <wap-template-path>
 /layouttpl/custom/book_workshop_home.wap.tpl
 </wap-template-path>
</layout-template>
```

---

```
</wap-template-path>
<thumbnail-path>
 /layouttpl/custom/book_street_home.png</thumbnail-path>
</layout-template>
```

The code above shows registration of the layout templates `book_street_home` and `book_workshop_home`. Each layout template is specified as:

- ID – such as `1_Column`
- Name – such as `1 Column`
- Template path – for example, `/layouttpl/custom/book_street_home.tpl`
- WAP template path – such as `/layouttpl/custom/book_street_home.wap.tpl`
- Thumbnail path – for example `/layouttpl/custom/book_street_home.png`

You can specify WAP layout templates there as well.

By the way, you may be interested in layout template **DTD (Document Type Definition)**. Definitely, you can find details of layout templates DTD in `/portal/definitions/liferay-layout-templates_5_2_0.dtd`.

As you can see, we have specified the Velocity template, for example `$processor`, in layout templates. Basically, each Velocity template is a plain text file with simple special characters. These special characters are translated just prior to the server sending the requested web page to the browser. The following are the available (but not limited) variables of the Velocity template, which we can use in layout templates:

Variable	Template type
<code>processor</code>	<code>com.liferay.portlet.layoutconfiguration.util.velocity.TemplateProcessor</code>
<code>request</code>	<code>javax.servlet.http.HttpServletRequest</code>
<code>pageContext</code>	<code>javax.servlet.jsp.PageContext</code>
<code>portletConfig</code>	<code>com.liferay.portlet.PortletConfigImpl</code>
<code>renderRequest</code>	<code>javax.portlet.RenderRequest</code>
<code>renderResponse</code>	<code>javax.portlet.RenderResponse</code>
<code>themeDisplay</code>	<code>com.liferay.portal.theme.ThemeDisplay</code>
<code>company</code>	<code>com.liferay.portal.model.Company</code>
<code>user/realUser</code>	<code>com.liferay.portal.model.User</code>
<code>layout</code>	<code>com.liferay.portal.model.Layout</code>
<code>layouts</code>	<code>java.util.List&lt;com.liferay.portal.model.Layout&gt;</code>

Variable	Template type
plid	java.lang.String
layoutTypePortlet	com.liferay.portal.model.LayoutTypePortlet
portletGroupId	java.lang.String
locale	java.util.Locale
timeZone	java.util.TimeZone
theme	com.liferay.portal.model.Theme
colorScheme	com.liferay.portal.model.ColorScheme
portletDisplay	com.liferay.portal.theme.PortletDisplay

For example, the `$processor.processColumn("column-1")` code is handling the `column-1` column. You can find the above Velocity template variables in detail in the `insertVariables` method of `com.liferay.portal.velocity.VelocityVariables` under the folder `/portal/portal-impl/src`.

Great! You have created and registered customized layout templates `book_street_home` and `book_workshop_home` successfully. When you deploy the above layout templates and restart the application server, you will see that thumbnail icons for both `book_street_home` and `book_workshop_home` layout templates are displayed when you click on **Layout Template** from the **Dock** menu.

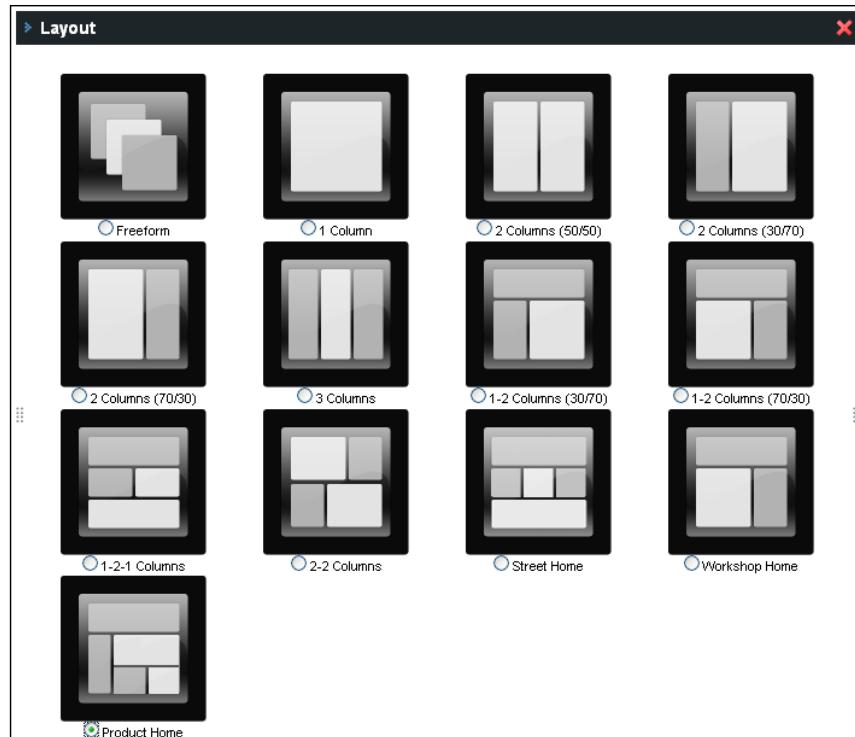
## Developing layout templates in Plugins SDK

We have discussed how to build layout templates in Ext. We also introduced Plugins SDK in the previous chapters. Now let's focus on how to develop hot-deployable layout templates in Plugins SDK.

The screenshot shows the Customer Center Home page with a navigation bar at the top: Support Center, Tools and Applications, Product Communities, Learning Center, and Resource Center. Below the navigation bar, there is a welcome message for 'Customer Center Home' and a user profile link for 'Lotti Stein'. On the left side, there are links for 'My support requests', 'My Cases', 'Submit a Case', 'Online tools and applications' (Inflight Data Tracking, Document Tracking), 'Community forums', and 'User forums'. In the center, there is a 'Product Family Centers' section with 'Integration Products' and 'Business Applications' categories, each containing links for 'Collaboration Network', 'Managed File Transfer', 'Integration Products', 'Multi-Channel Fulfillment', 'Multi-Channel Selling', and 'Total Payments'. Below this is a 'Featured Customer Case Study' for 'Sterling Collaboration Network consolidates AT&T's Business Processes'. To the right is a 'Customer Connection' box for 'Grand Hyatt San Antonio April 27 - 29, 2009 San Antonio, Texas'. At the bottom, there are sections for 'Latest Content' (Starting EDI with Asian Partners) and 'News Forum Topics' (Sterling Customer Center 6 posts). A 'Featured Webinar' box is also present.

Look at the preceding screenshot. For the **Product Home** use case, we're going to build a page with a row on the top that says **Welcome!**, one column in the bottom left with a logged-in message and left navigation bar, a column in the middle right having **Product Family Centers** and **Images (Featured Customer Case Study, Customer Connection, Featured Webinar)**, and two columns in the bottom right with contents **Latest Content** and **News Forum Topics**, respectively. As a content editor, you can flexibly drag-and-drop different portlets to the **Product Home** page. That is, there is only a small set of layout templates that will be in use for the entire web site **Customer Centre**. Thus, this layout page could be specified by a hot-deployable layout template.

Moreover, this layout template is reusable for other pages. As shown in the following screenshot, a content editor can view it by thumbnail icon, by clicking on **Layout Template** from the **Dock** menu. The thumbnail image **Product Home** shows what the layout looks like.



Obviously, this kind of layout template could be developed in both Ext and Plugins SDK. But in order to make this layout template hot and deployable, it should be developed in Plugins SDK. In this section, we're going to discuss how to build a hot and deployable layout template in Plugins SDK.

## Building layout templates

First of all, we could create a folder named `product-home-layouttpl` in the `$PLUGINS_SDK_HOME/layouttpl` folder. Everything for the use case *Product Home* will go in the `$PLUGINS_SDK_HOME/layouttpl/product-home-layouttpl` folder.

In the `product-home-layouttpl` folder, create another `docroot` folder and an XML file `build.xml`. Open it and add the following lines at the beginning of this file:

```
<?xml version="1.0"?>
<project name="layouttpl" basedir=". " default="deploy">
 <property name="plugin.version" value="1" />
 <import file="../build-common-layouttpl.xml" />
</project>
```

The code above shows a `layouttpl` project with default target `deploy`. It takes the `plugin.version` property with value 1. This property should be imported from the `build.${user.name}.properties` file as mentioned in Chapter 3, *ServiceBuilder and Development Environments*. Meanwhile, it imports a `build-common-layouttpl.xml` file from a parent folder.

Then in the `/product-home-layouttpl/docroot/` folder, create a `WEB-INF` folder. Further, create an XML file `liferay-plugin-package.xml` in the `/product-home-layouttpl/docroot/WEB-INF/` folder and add the following lines at the beginning of this file:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plugin-package PUBLIC
 "-//Liferay//DTD Plugin Package 5.2.0//EN"
 "http://www.liferay.com/dtd/liferay-plugin-
 package_5_2_0.dtd">
<plugin-package>
 <name>Product Home Templates</name>
 <module-id>
 liferay/product-home-layouttpl/5.2.3.1/war
 </module-id>
 <types>
 <type>layout-template</type>
 </types>
 <short-description>
 This plugin is Product Home layout template.
 </short-description>
 <change-log></change-log>
 <page-url>http://www.book.com</page-url>
 <author>Book, Inc.</author>
 <licenses>
 <license osi-approved="true">MIT</license>
 </licenses>

```

---

```
<liferay-versions>
 <liferay-version>5.2.0+</liferay-version>
</liferay-versions>
</plugin-package>
```

The code above shows the plugin package name as Product Home Templates, and module id as liferay/product-home-layouttpl/5.2.3.1/war, type as layout-template, and so on. The `liferay-plugin-package.xml` file is used to keep track of versions and compatibility. If the portal was upgraded—for example, from 5.2 to 5.3 (or 5.4 or 5.5), you need to replace 5.2 and 5\_2 with 5.3 (or 5.4 or 5.5) and 5\_3 (or 5\_4 or 5\_5) respectively. That is, only the major version (for example, 5.2, 5.3, 5.4, 5.5, and so on.) needs to be considered.

By the way, you may be interested in the plugin package DTD. Fortunately, you can find details of the plugin package DTD in the folder `/portal/definitions/liferay-plugin-package_5_2_0.dtd`.

Finally, we need to register the product-home layout template. Create an XML file `liferay-layout-templates.xml` in the `/product-home-layouttpl/docroot/WEB-INF/` folder and add the following lines at the beginning of this file:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE layout-templates PUBLIC
"-//Liferay//DTD Layout Templates 5.2.0//EN"
"http://www.liferay.com/dtd/liferay-layout-
templates_5_2_0.dtd">
<layout-templates>
<custom>
 <layout-template id="product_home" name="Product Home">
 <template-path>/product_home.tpl</template-path>
 <wap-template-path>
 /product_home.wap.tpl
 </wap-template-path>
 <thumbnail-path>/product_home.png</thumbnail-path>
 </layout-template>
</custom>
</layout-templates>
```

The code above shows registration of the product-home layout template under the custom XML tag with id as `product_home`, name as `Product Home`, template-path as `/product_home.tpl`, wap-template-path as `/product_home.wap.tpl`, and thumbnail-path as `/product_home.png`. You can specify WAP layout templates there as well.

Optionally, you could run a script to create a blank layout template project. For example, for the above project, we have a project named `product_home` and layout template display name `Product Home`. On Linux or Mac, you would change the directory to `$PLUGINS_SDK_HOME/layouttpl` and then type the following command:

```
./create.sh product-home "Product Home"
```

On Windows, you would change the directory to `$PLUGINS_SDK_HOME/layouttpl` and then type the following command:

```
create.bat product-home "Product Home"
```

The above command will create a blank layout template in the folder `$PLUGINS_SDK_HOME/layouttpl`. In fact, the script uses a predefined ZIP file `layouttpl.zip` to create a blank layout template with the following Ant command:

```
ant -Dlayouttpl.name=$1 -Dlayouttpl.display.name=\"$2\" create
```

## Creating layout templates

We have built a project for the `product_home` layout template. We have also registered the layout template. As mentioned earlier, this layout template is registered with `id` as `product_home`, `name` as `Product Home`, `template-path` as `/product_home.tpl`, `WAP-template-path` as `/product_home.wap.tpl`, and `thumbnail-path` as `/product_home.png`. But the files `product_home.tpl`, `product_home.wap.tpl`, and `/product_home.png` do not exist yet. Now, let's create these files for this layout template.

First, create a thumbnail icon file `product_home.png` in the `/product-home-layouttpl/docroot` folder. This thumbnail icon specifies what the layout looks like. You can customize the thumbnail icon `product_home.png` in an image manipulation program such as GIMP, Adobe Photoshop, and so on.

Then create a `.wap.tpl` file `product_home.wap.tpl` in the `/product-home-layouttpl/docroot` folder. The WAP version specifies the arrangement of portlets in a page in WAP devices. You need to add the following lines at the beginning of `product_home.wap.tpl`:

```
<table>
 <tr>
 <td colspan="2">
 $processor.processColumn("column-1")
 </td>
 </tr>
 <tr>
 <td>
```

```
$processor.processColumn("column-2")
</td>
<td>
 <table>
 <tr>
 <td colspan="2">
 $processor.processColumn("column-3")
 </td>
 </tr>
 <tr>
 <td>
 $processor.processColumn("column-4")
 </td>
 <td>
 $processor.processColumn("column-5")
 </td>
 </tr>
 </table>
<td>
</tr>
</table>
```

The code above shows five columns in a table and a sub-table. As you can see, the WAP version doesn't have the benefit of CSS.

Last but not the least, create a .tpl file `product_home.tpl` in the folder `/product-home-layouttpl/docroot`. This file specifies the arrangement of portlets of a page in a regular web browser. You need to add the following lines at the beginning of `product_home.tpl`:

```
<div id="content-wrapper">
 <table class="lfr-grid" id="layout-grid" border="0">
 <tr>
 <td class="lfr-column" valign="top" colspan="2"
 style="border: 0px solid green;
 padding: 5px; width: 970px; ">
 $processor.processColumn("column-1")
 </td>
 </tr>
 <tr>
 <td class="lfr-column" style="width: 240px;
 padding-right: 15px; " valign="top">
 $processor.processColumn("column-2")
 </td>
 <td class="lfr-column" style="width: 715px;
 padding: 0px; " valign="top">
```

```
<table width="100%" cellspacing="0"
 cellpadding="0" border="0">
<tr>
 <td class="lfr-column" colspan="2"
 style="width: 715px; padding: 0px;
 border: 0px solid green; " valign="top">
 $processor.processColumn("column-3")
 </td>
</tr>
<tr>
 <td class="lfr-column" style="width: 345px;
 padding-right: 12px; border: 0px solid green;
 " valign="top">
 $processor.processColumn("column-4")
 </td>
 <td class="lfr-column" style="width: 345px;
 padding-left: 13px;
 border: 0px solid green; " valign="top">
 $processor.processColumn("column-5")
 </td>
</tr>
</table>
</td>
</tr>
</table>
</div>
```

The code above shows five evenly spaced columns with a table. Each table cell `<td>` has a CSS class (such as `lfr-column`) associated with it, a style, as well as an ID (such as `column-5`). Here this code uses `style` of HTML tag `<td>`. You may want to modify `style` with different values in order to get a different look and feel. Especially, columns such as `column-5` and `column-3` are arranged by the `width`, `border`, `padding-left`, and `padding-right` styles. Again, the CSS class may be customized by modifying the theme you are using to display the layout.

You are set! You can first drop the XML file `build.xml` in the folder `$PLUGINS_SDK_HOME/layouttpl/product-home-layouttpl` to the Ant view. Then you can click on the Ant target `deploy` under `layouttpl` of the Ant view to deploy this layout template. Before deploying, you may need to clean the previous deployment of the layout template by clicking on the Ant target `clean` in the Ant view.

## Building themes in Plugins SDK

The web sites [bookpubstreet.com](http://bookpubstreet.com) and [bookpubworkshop.com](http://bookpubworkshop.com) have their own look and feel, which is specified by the themes on top of the portal. As mentioned earlier, themes customize the overall look and feel of pages generated by the portal.

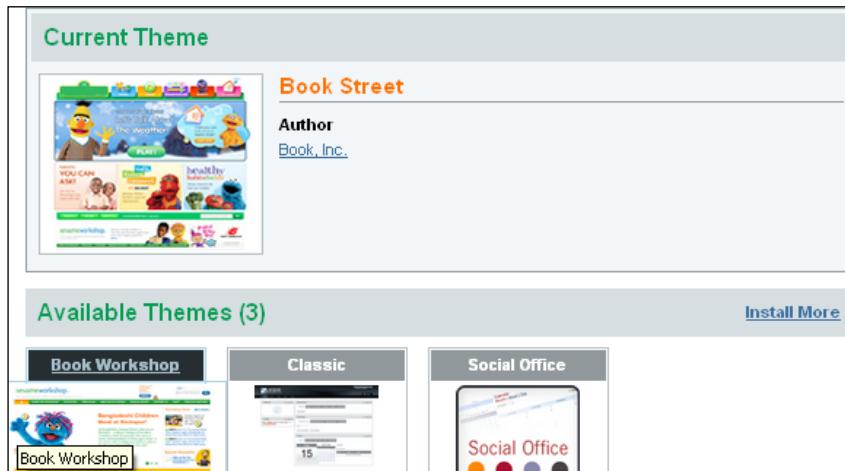
As shown in the following screenshot, for the use case *Book Street Theme*, the [bookpubstreet.com](http://bookpubstreet.com) web site has its own look and feel. For example, all the content will stay in the center of pages. Moreover, the width must be fixed as 917 pixels. All the pages in the web site [bookpubstreet.com](http://bookpubstreet.com) will share the same overall look and feel. Especially, a default MUPPET image will be displayed on the **MUPPETS** button if current user has not signed in yet. When the user signs in, your portrait image will appear on the **MUPPET** button.



Similarly, as shown in the following screenshot, the web site [bookpubworkshop.com](http://bookpubworkshop.com) has its own look and feel the *Book Workshop Theme* use case. For instance, all the content will stay in the centre of page and the width must be fixed as 917 pixels. All the pages in the web site [bookpubworkshop.com](http://bookpubworkshop.com) will share the same overall look and feel. Note that the **Donate** icon will become bigger in size if the mouse comes over the **SUPPORT US** button, and it will shrink back to its normal size if the mouse moves away from the **SUPPORT US** button. At the same time, the **Donate** icon is presented via JavaScript and it will change its look and feel at random.



Themes can be shared by any pages of any web site. In other words, the other pages from different web sites can share this overall look and feel if applicable. As shown in the following screenshot, you can choose one of the four themes (Book Street, Book Workshop, Classic, and Social Office) for any page you want. Moreover, these themes should be hot-deployable. Using this feature as a developer, you can make your changes, hot deploy them to the portal, and test it quickly.



In this section we're going to address how to develop and deploy themes in the Plugins SDK environment. We will specially address the general processes to develop and deploy themes, and the above use cases will be used only as examples. In real cases, you can use the same processes to build your own themes for your own web sites.

## Creating a customized theme

A theme uses CSS, images, JavaScript, and Velocity templates to control the whole look and feel of the pages generated by the portal. Thus, when creating customized themes, we need to consider these four groups as well. The `book-street-theme` theme is made up of a folder `_diffs` with four subfolders `css`, `images`, `javascript`, and `templates`; and a folder `WEB-INF` with the properties file `liferay-plugin-package.properties` and XML file `liferay-look-and-feel.xml`. Let's build this customized theme as follows.

## Setting up the theme project

First of all, we need to create a folder named `book-street-theme` under the folder `$PLUGINS_SDK_HOME/themes`. Everything for the *Book Street Theme* use case will go in the folder `$PLUGINS_SDK_HOME/themes/book-street-theme`.

Under the folder `book-street-theme`, create another folder `docroot` and an XML file `build.xml`. Open this XML file and add the following lines at its beginning:

```
<?xml version="1.0"?>
<project name="theme" basedir=". " default="deploy">
 <import file="../build-common-theme.xml" />
 <property name="theme.parent" value="_styled" />
</project>
```

The code above shows a project theme with the default target `deploy`. It takes one `theme.parent` property with the `_styled` value. You can find `_styled` in the folder `/portal/portal-web/docroot/html/themes/`. Meanwhile, it imports a `build-common-theme.xml` file from parent folder.

Then, under the folder `/book-street-theme/docroot`, create a folder `/WEB-INF`. Now create an XML file `liferay-plugin-package.properties` in the folder `/docroot/WEB-INF` and add the following lines at the beginning of this file:

```
name=Book Street
module-group-id=book
module-incremental-version=1
tags=
short-description=
change-log=
page-url=http://www.book.com
author=Book, Inc.
licenses=AGPL
```

The code above shows the plugin package name as Book Street, the module group ID as book, the module incremental version as 1, the author as Book, Inc., and so on. Of course, you can have your own settings for above items.

Accordingly, we need to register the `book-street-theme` theme. Create an XML file `liferay-look-and-feel.xml` in the folder `/docroot/WEB-INF` and add the following lines in this file:

```
<?xml version="1.0"?><!DOCTYPE look-and-feel PUBLIC
"-//Liferay//DTD Look and Feel 5.2.0//EN"
"http://www.liferay.com/dtd/liferay-look-and-
feel_5_2_0.dtd">
<look-and-feel>
 <compatibility>
 <version>5.2.0+</version>
 </compatibility>
 <theme id="book_street" name="Book Street">
 <settings>
```

```
<setting key="portlet-setup-show-borders-default"
 value="false" />
</settings>
</theme>
</look-and-feel>
```

The code above shows the registration of the theme `book-street-theme`, with the ID as `book_street` and name as Book Street. At the same time, a `portlet-setup-show-borders-default` property key is specified with the value `false`. This means that Liferay portal will turn off borders by default for all the portlets. Each theme can define a set of settings in order to make it configurable. The `liferay-look-and-feel.xml` file is also used to keep a track of versions and compatibility. As shown in the code above, if the portal was upgraded from 5.2 to 5.3 (or 5.4 or 5.5), for example, you need to replace `5.2` and `5_2` with `5.3` (or `5.4` or `5.5`) and `5_3` (or `5_4` or `5_5`) respectively.

By the way, you may be interested in the Liferay look and feel DTD. You can find its details in `/portal/definitions/liferay-look-and-feel_5_2_0.dtd`.

Finally, we need to create a folder `_diffs` in the folder `/book-street-theme/docroot`. Under the folder `_diffs`, create four subfolders: `css`, `images`, `javascript`, and `templates`.

Optionally, you can run a script to create a blank theme project. For example, for the above project, we have a project named `book-street-theme` and theme display named Book Street. On Linux or Mac, you would change the directory to `$PLUGINS_SDK_HOME/themes` and then type the following command:

```
./create.sh book-street "Book Street"
```

On Windows, you would change directory to `$PLUGINS_SDK_HOME/themes` and then type the following command:

```
create.bat book-street "Book Street"
```

The above command will create a blank theme in the `$PLUGINS_SDK_HOME/themes` folder. In fact, similar to the project of layout template, the script uses predefined ZIP file `theme.zip` to create a blank theme with the following Ant command:

```
ant -Dtheme.name=$1 -Dtheme.display.name=\"$2\" create
```

Similarly, you can build a theme project named `book-workshop-theme` just like you had built the `book-street-theme` project.

## Building differences of themes

As much as we can say, the best practice of building a customized theme is to put only the differences of customized theme into the folder \${theme-name}/docroot/\_diffs. Here \${theme-name} refers to any theme project name, for example, book-street-theme. Using the best practice, we need to put customized CSS, images, JavaScript, and templates in the folder /\_diffs only.

In the folder /\_diffs/css, create a CSS file custom.css. We should place all of the CSS that is different from the other files. By placing custom CSS in this file, and not touching the other files, we can be assured that the upgrading of their theme later on will be much smoother. In the folder /\_diffs/images, put all customized images with subfolders. For example, create two images: screenshot.png and thumbnail.png to show how a page with the current theme looks like. Further, create a subfolder searchbar and put all search-related images in this folder /searchbar.

Create a JavaScript file javascript.js in the folder /\_diffs/javascript. Liferay portal includes the jQuery Javascript library. Thus, we can include any plugins that jQuery supports in the theme. In the folder /\_diffs/templates, create customized template files such as dock.vm, init\_custom.vm, navigation.vm, portal\_normal.vm, portal\_pop\_up.vm, and portlet.vm. Note that you can use JSP files in template files under the folder templates. However, you won't have access to the Velocity variables if JSP files were in use.

Cool! When you are ready, you can drop the XML file build.xml in the folder \$PLUGINS\_SDK\_HOME/themes/\${theme-name} to the Ant view first. Then just double-click on the Ant target deploy under theme of the Ant view.

## What's happening after deploying themes?

In general, when you double-click on the Ant target deploy under theme of the Ant view, it will copy all of the files from the folder \${app.server.portal.dir}/html/themes/\_unstyled/ to the folder \$PLUGINS\_SDK\_HOME/themes/\${theme-name}/docroot/ first. Then it will copy all of the files from the folder \${app.server.portal.dir}/html/themes/\_styled/ to the folder \$PLUGINS\_SDK\_HOME/themes/\${theme-name}/docroot/\_diffs/ too. Afterwards, it will copy all of the files from the folder \$PLUGINS\_SDK\_HOME/themes/\${theme-name}/docroot/\_diffs/ to the folder \$PLUGINS\_SDK\_HOME/themes/\${theme-name}/docroot/. It means that you will place all of your new and changed files into the folder \$PLUGINS\_SDK\_HOME/themes/\${theme-name}/docroot. Here \${theme-name} refers to a real theme project name, for example, book-street-theme.

Afterwards, you will see four folders: `css`, `images`, `javascript`, and `templates` under the folder `$PLUGINS_SDK_HOME/themes/${theme-name}/docroot`. Each of these folders will contain all merged files and subfolders from `/_unstyled`, `/_styled`, and `/_diffs`. As mentioned earlier, the `theme.parent` property is specified with the `_styled` value in `$PLUGINS_SDK_HOME/themes/${theme-name}/build.xml`. Of course, you can configure this property with the `_unstyled` value. Fortunately, you can find details from the XML file `$PLUGINS_SDK_HOME/themes/build-common-theme.xml` as follows:

```
<if>
 <equals arg1="${theme.parent}" arg2="_unstyled" />
 <then>
 <copy todir="docroot" overwrite="yes">
 <fileset dir="${app.server.portal.dir}/
 html/themes/_unstyled"
 excludes="templates/init.vm"/>
 </copy>
 </then>
 <elseif>
 <equals arg1="${theme.parent}" arg2="_styled" />
 <then>
 <copy todir="docroot" overwrite="yes">
 <filesetn dir="${app.server.portal.dir}/
 html/themes/_unstyled"
 excludes="templates/init.vm" />
 </copy>
 <copy todir="docroot" overwrite="yes">
 <fileset dir="${app.server.portal.dir}/
 html/themes/_styled"/>
 </copy>
 </then>
 </elseif>
```

The code above shows the process to deploy themes. For `/_styled`, it just copies all files from `/_unstyled` to `/docroot`. For `/_styled`, it first copies all of the files from `/_unstyled` to `/docroot`, and then it copies all of the files from `/_styled` to `/docroot` and overwrites all the changes on the folder `/docroot` from the folder `/_styled`.

In short, the main technologies used in theme development involve CSS, Velocity, and HTML. CSS provides a great degree of control over the look and feel of the page; Velocity (or optionally JSP) provides a series of templates offering control over the HTML produced; and HTML provides the most common output of a theme template. The portal includes two template themes, `_unstyled` and `_styled`, which are the basis for other themes. The `_unstyled` template contains the default templates and images. Its CSS files only contain placeholders to add formatting rules. `_styled` adds CSS files with formatting rules. For instance, the `classic` theme is based on `/_unstyled` and `/_styled` and, moreover, it adds different color schemes.

## Putting HTML to use

Themes are organized as a logical file structure. The `/templates` folder contains the Velocity (or JSP) templates that control the HTML generated by the theme. By default, many of the templates in the `/templates` directory have been consolidated. The `portal_normal.vm` file contains the overall site structure, from the opening HTML tag to the closing. It includes the header and footer, two templates (`dock.vm` and `navigation.vm`), and also the system files. This file is the main index file that contains the base HTML. The `dock.vm` file contains the entire HTML for the dock, the file `navigation.vm` contains the entire HTML for the navigation, and the file `portal_pop_up.vm` contains the entire HTML structure for pop-up windows. It is similar to the file `portal_normal.vm`, except it is shown in pop-up windows.

By the way, the file `portlet.vm` contains the HTML that wraps every portlet, including the portlet title and portlet icons. If you do not want show portlet icons, just comment on the following lines in `portlet.vm`:

```
<div class="portlet" id="portlet-wrapper-$portlet_id">
 <div class="portlet-topper">

 $theme.iconPortlet() $portlet_title

 <div class="portlet-icons"
 id="portlet-small-icon-bar_$portlet_id">
 #if ($portlet_display.isShowBackIcon())
 <a href="$portlet_back_url"
 class="portlet-icon-back">
 #language ("return-to-full-page")

 #else
 //{$theme.iconOptions()} //{$theme.iconMinimize()}
 //{$theme.iconMaximize()} //{$theme.iconClose()}
 #end
 </div>
 </div>
 <div class="portlet-content">$portlet_content</div>
</div>
```

The code above shows the portlet content as `<div class="portlet-content">$portlet_content</div>`. Portlet icons are commented already. In the same way, if in need, you can comment the portlet title and the portlet back URL as well. The HTML is set as the standard mode by default. For example, the following is a sample code is specified in the file `portal_normal.vm`:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
```

In addition, it would be better to have Velocity as the default format for templates using Velocity for cleaner code and better maintainability. Here we strongly recommend Velocity for formatting templates. But it is also possible for you to have JSP as default format for templates. If that's the case, just change the .vm extension in the files above to .jsp.

## Experiencing CSS and images

As stated earlier, themes are broken into a logical file structure. The /css folder contains the CSS files of the theme organized in several files, the /images folder contains the images of the theme organized in subfolders by purpose, the /javascript folder contains optional JavaScript code to control behavior aspects, whereas the /WEB-INF folder contains configuration files. As mentioned above, the /templates folder contains the Velocity (or JSP) templates that control the HTML generated by themes.

The following CSS file main.css in the /css folder is the main CSS that includes other CSS files:

```
@import url(base.css);
@import url(application.css);
@import url(layout.css);
@import url(navigation.css);
@import url(portlet.css);
@import url(forms.css);
@import url(custom.css);
```

As shown in the code above, the CSS file main.css includes other CSS files; for example, base.css, application.css, layout.css, navigation.css, navigation.css, forms.css, and custom.css. The CSS file base.css contains all of the base generic styling. This styling is used for all elements that are not directly related to another aspect of the site, for example, forms, navigation, and dock. The CSS file application.css has the capacity for all of the styling that is used for application elements, for example, dialogs, inline pop-up windows, tabs, tags, and other elements. The CSS file layout.css accommodates all of the styling related to the layouts. It is fairly low level and should most likely not be edited, unless there is something specific we need. The CSS file navigation.css contains all of the styling related to the navigation, as well as the dock. Whereas, the CSS file portlet.css accommodates all of the styling related to the portlets, including the JSR-168 class-names and the JSR-286 class-names. Moreover, the CSS file forms.css contains all of the CSS styling related to form elements on the page. Last but not least, the CSS file custom.css should contain all the customized CSS that is different from the other files. By placing the customized CSS in this file and not touching the other files, we can be assured that the upgrading of the theme later on will be much smoother.

Further, you can find all of the images of the theme in the /images folder organized in the subfolders by purpose. The /common subfolder normally contains all of the general images; the /dock subfolder contains images for styling the dock; the /messages subfolder contains the images for the different portal messages; the /navigation subfolder contains the images for the navigation styling. While the /portlet subfolder contains images related to the portlet decoration, the /form subfolder contains images related to form elements on the page and the other portlet-specific image subfolders that are used for other purposes.

Now, you can easily reference your themes images directory from CSS without the use of obscure variables. You can also easily feed different CSS styles into different browsers or operating systems, and even different browser versions as shown in the following code (abstracted from /css/form.css):

```
.ie6 input.text, input.password, .ie6 input.submit, .ie6 input.file,
.ie6 input.button {
 background-image: url(..../images/forms/input_shadow.png);
 background-repeat: no-repeat; border: 1px solid;
 border-color: #BFBFBF #DEDEDE #DEDEDE #BFBFBF;
 font: 1em Arial,Helvetica,Verdana,sans-serif;
 padding: 5px 1px;
}
```

As shown in the code above, to reference images directory, you would just put the relative path in this manner: background-image: url(..../images/forms/input\_shadow.png). The way to select different browsers would be to use .ie, .ie6, .gecko, .safari, and so on. To feed rules to different operating systems, you would provide input of the extension as: .win, .linux, .mac, and so on.

## Using jQuery JavaScript library

Liferay portal includes the jQuery JavaScript library. Thus, you can include any plugins that jQuery supports. Note that the VM template \$variable is not supported for better compliance with different portlets.

Inside the javascript.js file in the /docroot/\_diffs/javascript/ folder, you will find three different function calls as follows:

```
jQuery(document).ready(function() {});
Liferay.Portlet.ready(
 function(portletId, jqueryObj) {});
jQuery(document).last(function() {}
);
```

The preceding code shows the three function calls: `jQuery(document).ready`, `Liferay.Portlet.ready`, and `jQuery(document).last`. The function `jQuery(document).ready` gets loaded when all the HTML, not including the portlets, is loaded; the `Liferay.Portlet.ready` function gets loaded after every portlet on the page is loaded; the `jQuery(document).last` function gets loaded when everything, including the portlets, is on the page.

Besides theme-wide JavaScript, the VM template also supports page-specific JavaScript. The page settings form provides three separate JavaScript pieces that you can insert anywhere in your theme. You can find the following code for these settings in `/portal/portal-web/docroot/html/themes/_unstyled/templates/init.vm`.

```
#set ($typeSettingsProperties =
 $layout.getTypeSettingsProperties())
#set ($page_javascript_1 =
 $typeSettingsProperties.getProperty("javascript-1"))
#set ($page_javascript_2 =
 $typeSettingsProperties.getProperty("javascript-2"))
#set ($page_javascript_3 =
 $typeSettingsProperties.getProperty("javascript-3"))
```

In brief, JavaScript has a **Document Object Model (DOM)**. It allows us to grab and interact with different elements on the page. JavaScript also has an **Event Model (EM)**. It allows us to perform actions when a user interacts with the web sites or when changes are made. Browsers handle both DOM and EM in different, and sometimes buggy, ways. Different JavaScript libraries have been developed to compensate for the different ways the browsers behave, and to make it simpler to interact with the page.

jQuery is a JavaScript library that allows us to use CSS selectors (for example `#banner`, `.logo`, `body`, and so on) to select the elements on the page in the same way with CSS. jQuery supports up to **CSS 3 (Cascading Style Sheets 3**, refer to <http://www.w3.org/TR/css3-roadmap>).

In the previous chapter, we have briefly introduced the jQuery pop up. Here, let's take an in-depth look on general usage of jQuery. Firstly, with jQuery we could use CSS selectors to select the elements on the page. For instance, to style all links on a page with CSS, we would use the following code:

```
a {color: #FAFAFA; text-decoration: none; display: block; }
```

Similarly, to grab all of the links on a page in jQuery, we would provide input as:

```
jQuery('a');
```

Secondly, to execute JavaScript as soon as the page is ready, we would simply pass a function into jQuery directly. In other words, JavaScript will execute as soon as the page has loaded. Suppose we have a link with a `muppet` class on it as follows:

```

 Submit Muppet

```

We would like to have an alert box to pop up when users click on the above link, and later remove the link so that they cannot see it anymore. To do so, we would use the following JavaScript:

```
jQuery(function(){
 jQuery('.muppet'){
 alert('You've clicked on the link MUPPET!');
 jQuery(this).remove();
 }
});
```

Thirdly, jQuery also comes with many built-in effects, for example fading in and out, and sliding elements open and closed. Assume that we have the following HTML:

```
<a class="show-street-info"
 href="http://www.bookstreet.com/info">
 Show Book Street

<div class="streetinfo" style="display: none; ">
 Welcome to Book Street
</div>
```

Using the following JavaScript, when a user clicks on the `show-street-info` link, the information will slide into view. When they click on it again, it will slide away from view:

```
jQuery(function() {
 jQuery('.show-street-info').click(
 function (event){
 jQuery('.streetinfo').slideToggle();
 }
);
});
```

Moreover, as mentioned earlier, the portal has a couple of custom events that make working with the page easier:

- `Liferay.Portlet.ready()`: This event is called whenever a portlet is loaded onto the page
- `jQuery(document).last()`: This event is called after all the portlets have been rendered onto the page
- `jQuery(document).ready()`: This event is called when all the HTML, not including the portlets, is loaded onto the page

For more details of jQuery, please check the JavaScript file `jquery.js` in the folder `/portal/portal-web/docroot/html/js/jquery`.

## Employ theme settings

The portal defines settings that allow the theme to determine certain behaviors. So far, there are only two predefined settings at the time of writing: `portlet-setup-show-borders-default` and `bullet-style-options`. Certainly, this number would grow in the future. The `portlet-setup-show-borders-default` key shows whether the portal will turn off the borders by default for all the portlets or not. The default value for this property is `true`. For example, in the `book-street-theme` theme, we added the following code in  `${theme-name}/docroot/WEB-INF/liferay-look-and-feel.xml`.

```
<theme id="book_street" name="Book Street">
 <settings>
 <setting key="portlet-setup-show-borders-default"
 value="false" />
 </settings>
</theme>
```

As shown in the code above, the `portlet-setup-show-borders-default` key has a `false` value—it means that the portal will turn off the borders by default for all the portlets. The default value is `true`. This default behavior can be overridden for individual portlets using the `liferay-portlet.xml` file and Portlet CSS pop-up settings.

The `bullet-style-options` key shows the bullet style options for theme. You can find the `bullet-style-options` key in `liferay-look-and-feel.xml` under the `/portal/portal-web/docroot/WEB-INF/` folder.

```
<theme id="classic" name="Classic">
 <!-- ignore details -->
 <settings>
 <setting key="bullet-style-options" value="1,2" />
```

---

```

<setting key="hello" value="world" />
<setting key="hi" value="mom" />
</settings> <!-- ignore details -->
</theme>
```

As shown in the code above, the value must be a comma-separated list of valid bullet styles (for example, 1, 2) to be used by the navigation portlet. You can certainly add customized settings to make the theme configurable. As shown in the code above, two customized settings keys hello and hi are defined in the classic theme.

By the way, these settings (either predefined or customized) can be accessed in the theme templates using the following code:

```
$theme.getSetting("hello");
```

## Adding color schemes

Color schemes are specified as CSS. With CSS in color schemes, we cannot only change colors, but also choose different background images, different border colors, and so on. You can find color schemes from the liferay-look-and-feel.xml file in the /portal/portal-web/docroot/WEB-INF/ folder as follows:

```

<look-and-feel> <!-- ignore details -->
<theme id="classic" name="Classic">
 <root-path>/html/themes/classic</root-path>
 <!-- ignore details -->
 <color-scheme id="01" name="Blue">
 <css-class>blue</css-class>
 <color-scheme-images-path>
 ${images-path}/color_schemes/${css-class}
 </color-scheme-images-path>
 </color-scheme>
 <color-scheme id="02" name="Green">
 <css-class>green</css-class>
 </color-scheme>
 <color-scheme id="03" name="Orange">
 <css-class>orange</css-class>
 </color-scheme>
</theme>
<theme id="control-panel" name="Control Panel">
 <root-path>/html/themes/control_panel</root-path>
</theme> <!-- ignore details -->
</look-and-feel>
```

The preceding code shows a theme with ID as `classic` and name as `Classic`. The root path of the theme `classic` is `/html/themes/classic`. Three color schemes are specified in the theme `classic`. The `01` color scheme has the name `Blue`, with CSS class `blue`, the  `${images-path}/color_schemes/${css-class}` image path. Similarly, the color schemes `02` and `03` are specified as well. At the same time, another theme with ID as `control-panel` and name as `Control Panel` is also specified. The root path of the theme `control-panel` is `/html/themes/control_panel`.

Moreover, you can find CSS of color scheme of the theme `classic` in `/portal/portal-web/docroot/html/themes/classic/_diffs/css/color-schemes/`. You can also find images of color scheme of the theme `classic` in `/portal/portal-web/docroot/html/themes/classic/_diffs/images/color-schemes/`.

Of course, you can add color schemes in themes, for example `book-street-theme` and `book-workshop-theme`. As stated above, you can specify color schemes in `liferay-look-and-feel.xml` under the folder  `${theme-name}/docroot/WEB-INF/`, and then add images and CSS for color schemes in the  `${theme-name}/docroot/_diffs/css/color-schemes/` and the  `${theme-name}/docroot/_diffs/images/color-schemes/` folders, respectively.

## **Adhering to WAP standard**

Now, let's have a look in detail at the WAP themes. Locate the XML file `liferay-look-and-feel.xml` in the `/portal/portal-web/docroot/WEB-INF/` folder. You will see the following code:

```
<look-and-feel> <!-- ignore details -->
 <theme id="mobile" name="Mobile">
 <root-path>/wap/themes/${theme-id}</root-path>
 <wap-theme>true</wap-theme>
 </theme>
</look-and-feel>
```

The code above shows a theme with ID as `mobile` and name as `Mobile`. It also specifies the `/wap/themes/${theme-id}` root path and the WAP theme `true`. This means that the theme was developed for a mobile device.

You will also notice the addition of a `wap` folder in `/portal/portal-web/docroot/`. Locate the `mobile` folder in the `/portal/portal-web/docroot/wap/themes/` folder. To make a theme show up correctly in a mobile device, the following doc type is required in `/portal/portal-web/docroot/wap/themes/mobile/templates/portal_normal.vm`.

```
<?xml version="1.0"?>
<!DOCTYPE html PUBLIC "-//WAPFORUM//DTD XHTML Mobile 1.1//EN"
"http://www.wapforum.org/DTD/xhtml-mobile11.dtd">
```

## Adding runtime portlets to a theme

You can especially add runtime portlets to a theme. Suppose that we have portlet ID \${PORTLET\_ID} with a value, for example, extComments. Now we're going to add this portlet as a runtime portlet to a theme, for example book-street-theme. To add a runtime portlet to a theme, simply add the following line to the Velocity template:

```
$theme.runtime("${PORTLET_ID}")
```

## Using theme, CSS, and JavaScript

As mentioned earlier, themes use CSS, JavaScript, and Velocity templates to control the entire look and feel of the pages that are generated by the portal. Once theme, CSS, and JavaScript are engaged, we should use them in a proper way. Here we list a set of suggestions, as follows.

### Making use of themes

When developing themes, we should take account of the following comments. Note that these suggestions from the best practices would make theme development successful and smooth. First, try to use semantic HTML whenever possible and avoid using `<div>` when the `<a>` and `<p>` tags will do.

Secondly, use the `<div>` and `<span>` tags appropriately. The `<div>` tag can contain other block level elements such as `<p>` and other `<div>`, but `<span>` can only contain inline elements such as images, anchors, and so on.

Then, make sure that an `id` can only appear once on a page. If you require multiple identifying elements, use a `class`.

Last but not the least, avoid both **classitis** and **divitis**. Divitis is the unnecessary use of the `<div>` tags in markups. As a substitute, remove unnecessary div elements. Likewise, classitis is the overuse of the class attribute, for example placing a class on every child of an element. But instead, use elements and class names sparingly.

## Applying CSS

When developing CSS, we should take the following advices into account. Note that these advices belong to the nice-to-have category.

1. First, avoid using CSS hacks to target different browsers. Liferay portal provides browser selectors that allow us to target a specific browser, as well as a specific version of that browser, using a simple selector namespace such as `.ie`, `.ie6`, `.firefox`, `.safari`, and so on.
2. Second, strive to use dashes in CSS class names instead of underscores or **CamelCasing**. (CamelCasing is the practice of writing compound words or phrases in which the words are joined without spaces and are capitalized within the compound.) Then, make an effort to use CamelCasing for `id`.

## Employing JavaScript

Here we have some recommendations on developing JavaScript. First, we should avoid using global variables in JavaScript files. Global variables can break other scripts inside the portal. They cause endless amounts of conflicts, and also make troubleshooting difficult. To avoid using the global variables, make sure you define variables only inside a function, and always place `var` in front of that variable when defining it.

Secondly, for the theme, we should use the **JSON** style to contain all of the functions, properties, values, and so on. For example, if a theme is named `book-street-theme` in JavaScript, we would place all of the methods and properties inside one object—`bookstreet`—as follows:

```
var bookstreet={ setUserImagePortrait: function() {
 var so = new SWFObject(
 escape("/cms_services/street/PolaroidSmall.swf"),
 "PolaroidSmall", "48", "48", "9", "#FFFFFF");
 bookStreet.addParam("menu", "false"); // ignore details
 so.write("user_portait");
},
getTimeDate: function() {
// ignore details
 var ts_url = "&t=" + ts; return ts_url;
},
};
```



**JSON (JavaScript Object Notation)** is a lightweight data-interchange format. Refer to <http://www.json.org>.

## Experiencing the developing and debugging tools

There is a list of tools that you can use to develop and debug the theme, CSS, and JavaScript. First, **Firebug** is a good tool to view the CSS, which is being applied and inherited to edit the page's CSS live, run JavaScript from a command line, and debug JavaScript with breakpoints, as well as profile tools and track AJAX requests. Refer to <http://getfirebug.com/>.

Then, you may use **IE Developer Toolbar**. IE Developer Toolbar is the closest equivalent to Firebug for Internet Explorer. While it is not anywhere near as powerful, it is still a great help to view what CSS is being applied to an object and edit that CSS live. Refer to <http://www.microsoft.com>.

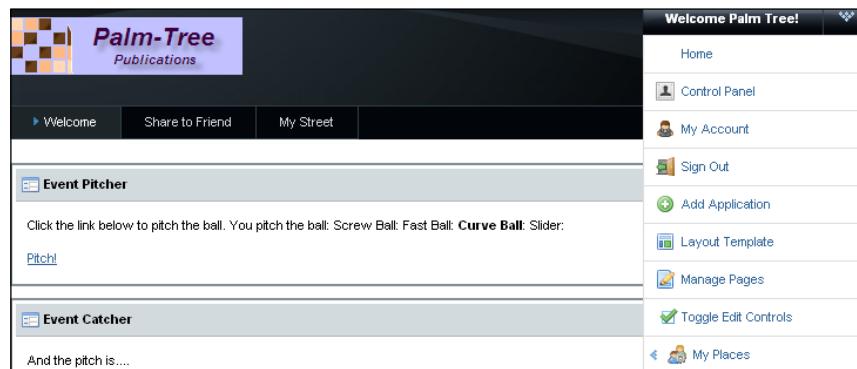
In addition, you may like **DebugBar**. DebugBar is a bit like IE Developer Toolbar and Firebug for Internet Explorer. It is quite powerful as it allows us to run JavaScript from a command line and also edit CSS live. Refer to <http://www.debugbar.com>.

## Customizing Velocity templates in themes

The hot-deployable themes use Velocity for cleaner code and better maintainability by default. In this section, we will first discuss which default Velocity templates are being used in themes. Then we will address how to add customized Velocity templates in both the drop-down menu and the navigation bar. In addition, we will introduce how to set up default customized themes and layout templates in the portal.

## Using default Velocity templates

Themes with the Velocity templates customize the overall look and feel of the portal. As you can see, the dock **Welcome Test Test!** with a list of drop-down links is specified via Velocity templates. Before logging in, you only see two links in the dock by default: **Home** and **Sign In**. After you log in as an admin, you will see a lot of links in the dock. Among many, these links include **Home**, **Control Panel**, **My Account**, **Sign out**, **Add Application**, **Layout Templates**, **Manage Pages**, **Toggle Edit Control**, **My Places**, and so on. Thus, several questions pop up – What is the dock with these kinds of links? How does it work? How do we customize Velocity templates? As mentioned earlier, the file `dock.vm` contains the entire HTML for the dock. Let's have a look at the details of dock.



## Experiencing default Velocity variables

First of all, locate the theme  `${theme-name}`, for example `book-street-theme`, and locate the VM file `dock.vm` in the  `${theme-name}/docroot/_diffs/templates` folder. When you open it, you will see links with the Velocity templates in the dock as follows:

```
<div class="lfr-dock interactive-mode">
 <h2 class="user-greeting">$user_greeting</h2>
 <ul class="lfr-dock-list">
 #if ($show_home)
 <li class="home">
 $home_text

 #end #if ($show_control_panel)
 <li class="control-panel">

 $control_panel_text


```

---

```

#end #if ($show_sign_in)
 <li class="sign-in">
 $sign_in_text

#end /* ignore details */
#if ($show_my_places)
 <li class="my-places">
 <a>$my_places_text
 $theme.myPlaces()

#end

</div>

```

The code above shows the dock with a set of HTML tags: `<div>`, `<h2>`, `<ul>`, `<li>`. It also shows the dock with a list of Velocity templates such as `#if` and `#end`. For the user greeting, it uses the `$user_greeting` Velocity template. For Home, it uses the `$show_home`, `$show_url`, `$show_text` templates. Similarly, it shows templates for Control Panel, My Account, Sign out, Add Application, Layout Templates, Manage Pages, Toggle Edit Control, My Places, and so on.

Then, we would ask, where are these templates specified? Here is the answer: in the file `init.vm`. Locate the file `init.vm` in the `/portal/portal-web/docroot/html/themes/_unstyled/templates` folder and you will see predefined Velocity templates for themes as follows:

```

#set ($theme_display = $themeDisplay)
#set ($portlet_display = $portletDisplay)
#set ($theme_timestamp = $themeDisplay.getTheme() .
 getTimestamp())
#set ($theme_settings = $themeDisplay.getTheme() .
 getSettings())

/* ignore details */
#set ($user_greeting = $htmlUtil.escape($user.getGreeting()))
/* ignore details */
#set ($full_templates_path = $fullTemplatesPath)
/* ignore details */
#if ($show_home)
 #set ($home_text = $languageUtil.get($company_id,
 $locale, "home"))
 #set ($home_url = $theme_display.getURLHome())
 #if (!$request.isRequestedSessionIdFromCookie())
 #set ($home_url = $portalUtil.getURLWithSessionId($home_url,
 $request.getSession().getId()))
 #end #end /* ignore details */
 #parse ("$full_templates_path/init_custom.vm")

```

The preceding code shows the definition of template variables: \$user\_greeting, \$show\_home, \$show\_url, \$show\_text, and so on. More interestingly, it parses the file `init_custom.vm` from current theme. Thus, we could override the existing template variables in the file `init.vm` and define new Velocity variables in the file `init_custom.vm` of our customized themes, for example `book-street-theme` and `book-workshop-theme`. We will discuss this usage in the coming section.

You have noticed that a set of template variables are used in the file `init.vm` as default, which include `$themeDisplay`, `$htmlUtil`, `$user`, `$fullTemplatesPath`, and so on. Where are these templates variables coming from? They are coming from the Java file `VelocityVariables.java`. Let's have a deep look at these template variables in the Java file `VelocityVariables.java`. Locate the Java file `VelocityVariables.java` in the package `com.liferay.portal.velocity` under the `/portal/portal-impl/src` folder and open it. You will see the following code:

```
// ignore details
velocityContext.put("htmlUtil", HtmlUtil.getHtml());
// ignore details
public static void insertVariables(VelocityContext velocityContext,
HttpServletRequest request) {
 ThemeDisplay themeDisplay = (ThemeDisplay)request.
 getAttribute(WebKeys.THEME_DISPLAY);
 if (themeDisplay != null) {
 Theme theme = themeDisplay.getTheme();
 Layout layout = themeDisplay.getLayout();
 List<Layout> layouts = themeDisplay.getLayouts();
 velocityContext.put("themeDisplay", themeDisplay);
 velocityContext.put("company", themeDisplay.getCompany());
 velocityContext.put("user", themeDisplay.getUser());
 // ignore details
 velocityContext.put("fullTemplatesPath", servletContextName
 + theme.getVelocityResourceListener()
 + theme.getTemplatesPath());
 // ignore details
 }
}
```

The code above shows a definition of template variables, including `$themeDisplay`, `$htmlUtil`, `$user`, `$fullTemplatesPath`, and so on.

## Customizing Velocity variables

We have answered the first two questions: What is the dock with these kinds of links, and how does it work. Now let's focus on the third question: How do we customize Velocity templates?

As mentioned in the previous chapters, we have added the journal article template variable `ExtVelocityToolUtil` via `<beans>` in `/ext/ext-impl/src/META-INF/ext-spring.xml`. It would be nice to add a template variable `extJournalUtil` in themes via service events. Afterwards, we could use the template variable `extJournalUtil` in any themes, including `book-street-theme` and `book-workshop-theme`.

Let's customize the template variable `extJournalUtil` as a pre-service event. First, create a Java file `ExtJournalUtil.java` in the package `com.ext.portal.util` under the `/ext/ext-impl/src` folder. Add the following lines at its beginning:

```
public class ExtJournalUtil {
 public static ExtJournalUtil getInstance() {
 return _instance;
 }
 private ExtJournalUtil() {}
 private static ExtJournalUtil _instance = new
 ExtJournalUtil();
}
```

As shown in the code above, `ExtJournalUtil` is specified as a singleton. Then create a package `com.ext.portal.events` in the folder `/ext/ext-impl/src`, create a pre-service event `ExtServicePreAction.java` in the package `com.ext.portal.events`, and add the following lines:

```
public class ExtServicePreAction extends Action {
 public void run(HttpServletRequest request,
 HttpServletResponse response) throws ActionException {
 Map<String, Object> vmVariables = Collections.
 synchronizedMap(new HashMap<String, Object>());
 vmVariables.put("extJournalUtil",
 ExtJournalUtil.getInstance());
 request.setAttribute(WebKeys.VM_VARIABLES, vmVariables);
 }
}
```

The code above shows that the instance of `ExtJournalUtil` is added as a Velocity variable `extJournalUtil`. Then, we need to register this pre-service in `portal-ext.properties`, including `ServicePreAction` at the end of `portal-ext.properties` file as follows:

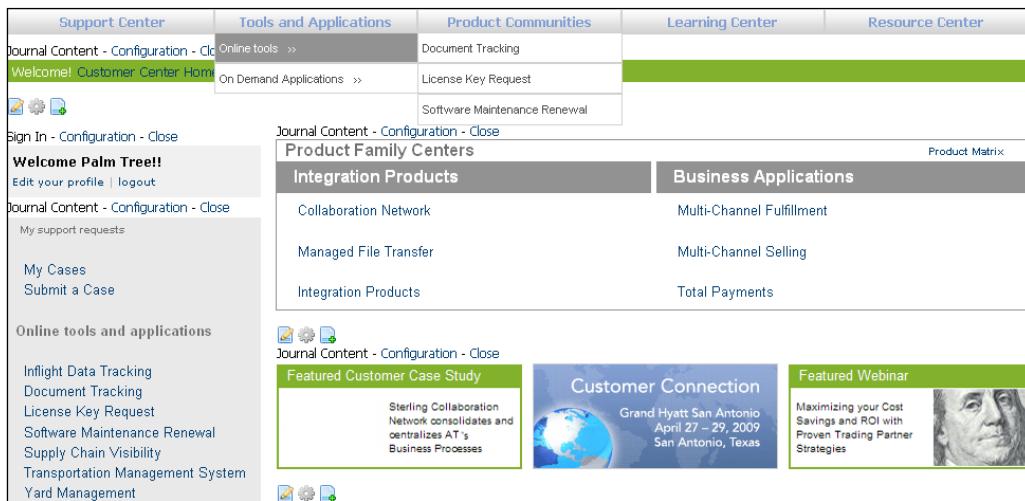
```
servlet.service.events.pre=com.liferay.portal.events.ServicePreAction
, com.ext.portal.events.ExtServicePreAction
```

That's it. The Velocity theme templates now include the `extJournalUtil` variable. Similarly, you can use post-service event `ExtServicePostAction.java` as well. In short, the pre-service events process before Struts processes the request, while the post-service events process after Struts processes the request. Further, you can customize the other pre/post-service events, for example `LoginPreAction.java`, `LoginPostAction.java`, `LogoutPreAction.java`, `LogoutPostAction.java`, and so on in at the package `com.liferay.portal.events` under the `/portal/portal-impl/src/` folder.

## Adding customized Velocity templates

We have introduced what the default Velocity templates are using in themes. Now let's consider the second topic: how to add customized Velocity templates in both drop-down menu and navigation bar.

As shown in the following screenshot for the use case, *Drop-down Menu* the drop-down menu **Tools and Applications | Online tools >>** has three items: **Document Tracking**, **License Key Request**, and **Software Maintenance Renewal**. The **Document Tracking (DT)** item is visible to any logged-in users. The **License Key Request (LKR)** item is only visible to the users coming from Admin user group and LKR user group. The **Software Maintenance Renewal (SMR)** item is only visible to the users coming from Admin user group and SMR user group.



A similar feature is required on the left navigation bar under **Online tools and applications**. As shown in the above screenshot for the use case *Navigation Bar*, the left navigation bar has many items: **Inflight Data Tracking**, **Document Tracking**, **License Key Request**, and **Software Maintenance Renewal**, and so on. The DT and **Inflight Data Tracking (IDT)** are visible to any logged-in users; the LKR is only visible to the users coming from Admin user group and LKR user group; and the SMR is only visible to the users coming from the Admin user group and SMR user group. In the following section, we're going to show how to implement the two use cases, *Drop-down Menu* and *Navigation Bar*, via Velocity templates.

## Using Velocity templates in drop-down menu

To implement the use case *Drop-down Menu*, we need to add the following user groups after the line `ext.ondportal.admin=ONDPortal_Admin` in the `portal-ext.properties` file:

```
ext.ondapp.lkr=ONDApp_LKR
ext.ondapp.smr=ONDApp_SMR
```

The code above shows the mappings between user group keys and user group names. The `ext.ondportal.admin` key has the value `ONDPortal_Admin`, and the `ext.ondapp.lkr` and `ext.ondapp.smr` keys have the values `ONDApp_LKR` and `ONDApp_SMR`, respectively.

Next, we need to create three user groups with names `ONDPortal_Admin`, `ONDPortal_LKR`, and `ONDPortal_SMR` in the User Groups of Control Panel. At the same time, assign expected users into the above user groups, for example assign David Berger to `ONDPortal_LKR`, and assign Lotti Stein to `ONDPortal_SMR`.

Now, create a VM file `init_custom.vm` in `$(theme-name)/docroot/_diffs/templates/` and add the following lines at its beginning:

```
#set ($userGroupService = $serviceLocator.findService(
 com.liferay.portal.service.UserGroupService"))
#set ($userGroups = $userGroupService.
 getUserUserGroups($user.getUserId()))
#set ($show_add_content_admin = false)
#foreach($userGroup in $userGroups)
 #if($userGroup.getName().equalsIgnoreCase($propsUtil.get(
 "ext.ondportal.admin")))
 #set ($show_add_content_admin = true)
 #end #end
 #set ($show_add_content_lkr = false)
 #foreach($userGroup in $userGroups)
 #if($userGroup.getName().equalsIgnoreCase($propsUtil.get(
 "ext.ondapp.lkr")))
```

```
#set ($show_add_content_lkr = true)
#end #end
#set ($show_add_content_smr = false)
#foreach($userGroup in $userGroups)
#if($userGroup.getName().equalsIgnoreCase($propsUtil.get(
 "ext.onapp.smr")))
#set ($show_add_content_smr = true)
#end #end
```

This code shows how to use the template variable `$serviceLocator` to find the service `$userGroupService` for user group. Afterwards, it specifies the template variables `show_add_content_admin`, `show_add_content_lkr`, and `show_add_content_smr`.

Finally, consume the customized Velocity templates `$show_add_content_admin`, `$show_add_content_lkr`, and `$show_add_content_smr` in the VM file navigation.vim under the folder  `${theme-name}/docroot/_diffs/templates/` as follows:

```
/* ignore details */
#set ($subsubmenu = 1)
#foreach ($nav_child_child in $nav_child.getChildren())
#if ($show_add_content_admin ||
$show_add_content_lkr
 && $nav_child_child.getName().toLowerCase() !=
$propsUtil.get("sterling.commerce.item.smr").
toLowerCase()) ||
($show_add_content_smr &&
$nav_child_child.getName().toLowerCase() !=
$propsUtil.get("sterling.commerce.item.lkr").
toLowerCase())
#if ($nav_child_child.getChildren())

<a onMouseOver="clrTimer();
showLayer('dd$root_menu','dds$root_menu$submenu',
'ddss$root_menu$submenu$subsubmenu')"
onMouseOut="setTimer()" href="#">
$nav_child_child.getName() &nbsp&nbsp&nbsp
››

#else
<a onMouseOver="clrTimer();
showLayer('dd$root_menu',
'dds$root_menu$submenu',
'ddss$root_menu$submenu$subsubmenu')"
onMouseOut="setTimer()"
href="$nav_child_child.getURL()"
```

---

```

 $nav_child_child.getTarget() >
 $nav_child_child.getName()

#end
/* ignore details */

```

In short, we have invoked the portal services through Velocity templates in themes. Liferay portal provides a utility to obtain references to services by \$serviceLocator. For example, we can invoke the layoutLocalService, userLocalService, roleService, and userGroupService services when we need them via the following code:

```

#set ($layoutLocalService = $serviceLocator.findService(
 "com.liferay.portal.service.LayoutLocalService"))
#set ($roleService = $serviceLocator.findService(
 "com.liferay.portal.service.RoleService"))
#set ($userLocalService = $serviceLocator.findService(
 "com.liferay.portal.service.UserLocalService"))
#set ($userGroupService = $serviceLocator.findService(
 "com.liferay.portal.service.UserGroupService"))

```

The code above shows how to use service locator to invoke specific services. You can find the Java file ServiceLocator.java in the package com.liferay.portal.velocity under the /portal/portal-impl/src folder. You can also find \$serviceLocator in the insertHelperUtilities method in the Java file com.liferay.portal.velocity.VelocityVariables.java under the /portal/portal-impl/src folder.

```

ServiceLocator serviceLocator = ServiceLocator.
 getInstance();
// ignore details
_insertHelperUtility(velocityContext,
 restrictedVariables,
 "serviceLocator", serviceLocator);

```

Similarly, we could invoke the other services existing in the package com.liferay.portal.service under the /portal/portal-service/src folder. For example, these services include the following, but are not limited to: accountService, contactService, groupService, imageLocalServices, organizationService, permissionService, and so on.

## Using Velocity templates in journal article-based navigation

To implement the use case *Navigation Bar*, we need to add a VM service (that is, a method called `getUserGroups`) after the first { in `ExtJournalUtil.java` as follows:

```
public List<UserGroup> getUserGroups(String userId) {
 List<UserGroup> results = Collections.
 synchronizedList(new ArrayList<UserGroup>());
 try{
 results = UserGroupLocalServiceUtil.
 getUserUserGroups(new Long(userId));
 }
 catch (Exception e) { return results; }
 return results;
}
```

The code above shows the `getUserGroups` method which gets a list of user groups by user ID. Then, we need to create a structure called `Navigation_Home` in the Web Content of Control Panel as follows. It will provide a structure for the `Navigation_Home` journal article.

```
<root>
 <dynamic-element name='LKR' type='text'>
 </dynamic-element>
 <dynamic-element name='SMR' type='text'>
 </dynamic-element>
</root>
```

The code above shows two elements, `LKR` and `SMR`, in the `Navigation_Home` structure. These elements have the type as `text`. That is, the display names of these elements are open to the content editors. They can provide any display names as input for both `LKR` and `SMR` when they are building journal articles based on the `Navigation_Home` structure. Note that you can have different names; we are using the name `Navigation_Home` for demo purpose only.

Now, we need to create an article template called `Navigation_Home` in the Web Content of Control Panel in order to consume the above VM service and structure as follows:

```
#set ($userGroups = $extJournalUtil.getUserGroups($request.
 attributes.USER_ID))
#set ($show_add_content_admin = false)
#foreach($userGroup in $userGroups)
 #if($userGroup.getName().equalsIgnoreCase($propsUtil.
 get("ext.ondportal.admin")))
```

```
#set ($show_add_content_admin = true)
#end #end #set ($show_add_content_lkr = false)
#foreach($userGroup in $userGroups)
#if($userGroup.getName().equalsIgnoreCase($propsUtil.
 get("ext.ondapp.lkr")))
#set ($show_add_content_lkr = true)
#end #end #set ($show_add_content_smr = false)
#foreach($userGroup in $userGroups)
#if($userGroup.getName().equalsIgnoreCase($propsUtil.get(
 "ext.ondapp.smr")))
#set ($show_add_content_smr = true)
#end #end #* ignore details *#
#if ($show_add_content_admin || $show_add_content_lkr)

 $LKR.data

#end
#if ($show_add_content_admin || $show_add_content_smr)

 $SMR.data

#end #* ignore details *#
```

The code above shows a way to consume VM service and structure. It first gets user groups by the \$extJournalUtil.getUserGroups VM service. Then it specifies the template variables \$show\_add\_content\_admin, \$show\_add\_content\_lkr, and \$show\_add\_content\_smr. Finally, it uses the above template variables as a precondition to display data, for example \$LKR.data, \$SMR.data.

From now on, if you created an article (for example Navigation\_Home) associated with above template and structure, you would see similar content as that of the use case *Navigation Bar*. Note that when creating a template, for example Navigation\_Home, you need to uncheck the **Cacheable** button in the edit mode.

## Setting up customized themes and layout templates as default

Whenever a community (for example, Book Street) or a user (for example, David Berger) is created, a default theme is applied on the default page of the community Book Street or the user David Berger's personal community. The portal comes up with the theme, `classic`, and layout template, `2_columns_ii`, as default. But we can change this behavior and set the default theme to one of our own themes.

First, let's have a look at the default settings of look and feel. To do so, locate the properties file `portal.properties` in the `/portal/portal-impl/src` folder. Now locate the `look.and.feel` string in this file and you will see the following code:

```
look.and.feel.modifiable=true
default.layout.template.id=2_columns_ii
default.regular.theme.id=classic
default.regular.color.scheme.id=01
default.wap.theme.id=mobile
default.wap.color.scheme.id=01
theme.sync.on.group=false
```

The code above shows default value of `look.and.feel.modifiable` as `true`. Set this to `false` if you do not want to allow users to modify the look and feel. It sets the default layout template ID `2_columns_ii` and default theme ID for regular themes `classic`. It also sets the default color scheme ID for both regular themes and WAP themes `01`, and the default theme ID for WAP themes `mobile`. Moreover, it sets `theme.sync.on.group` as `false`. Set this to `true` if you want a change in the theme selection of the public or private group which will be applied automatically to the other.

Then, let's consider our requirement. Suppose we need to set up our default theme as *Book Street* and default layout template as *Product Home*. Meanwhile, we do not want to allow the users to modify the look and feel; and moreover, we want a change in the theme selection of the public or private group to automatically be applied to the other. To implement these requirements, locate the properties file `portal-ext.properties` in the `/ext/ext-impl/src` folder and open it. Add the following lines at the end of this file and save it:

```
look.and.feel.modifiable=false
default.layout.template.id=product_home
default.regular.theme.id=book_street
theme.sync.on.group=true
```

Congratulations! You've just changed the portal default theme and layout template successfully.

## Using Plugins SDK more efficiently

Plugins SDK is useful to develop layout templates and make them hot-deployable. In short, you can update, deploy, and test layout templates in Plugins SDK without restarting your application server. Plugins SDK is also helpful to develop themes. It manages the differences between existing themes, creates a plugin WAR file with the resulting theme, and it is able to install the theme in a local installation. More importantly, everything in themes is customizable.

Moreover, Plugins SDK is helpful in developing portlets with a hooking feature. Portlets developed in Plugins SDK only import classes from `portal-kernel.jar`, `portal-service.jar`, and other JARs contained in the portlet folder  `${portlet-name} /docroot/WEB-INF/lib`. This forces the portlets to rely completely on the portal API, and not to depend on the implementation classes defined in `portal-impl.jar`. Why not? As `portal-impl.jar` is an implementation of Liferay portal, it is invisible and inaccessible to any plugins, including portlets.

In Plugins SDK, portlets can make use of any application framework (especially, MVC) that the portal supports, for example Struts, Spring, Tapestry, JSF, and so on. Hooks allow **hooking** into the portal. They specifically allow us to hook into events system, model listeners, JSP files, and portal properties of Liferay portal. For more details about developing portlets with a hooking feature in Plugins SDK, refer to the next chapter.

## How does it work?

Plugins SDK requires an application server bundle in order to compile against JAR files included within the portal. In runtime, the portal detects any WAR files in the current hot-deploy folder and, meanwhile, explodes the WAR files into the application server's deployment folder automatically.

## When to use Plugins SDK?

As mentioned above, portlets developed in Plugins SDK only import classes from `portal-kernel.jar` and `portal-service.jar`. That is, if customized portlets only depend on these two JARs and not on `portal-impl.jar`, we should develop them in Plugins SDK. More specifically, if the customized portlets do not involve the following areas, then we should use only Plugins SDK or a combination of Plugins SDK and Ext—otherwise, we have to use Ext only.

1. Does not require changing default user interface that is not theme related.
2. Does not require integration with other products that are not yet provided.
3. Does not require an algorithm for a particular functionality.
4. Does not require changing the behavior of built-in portlets in a way that is not configurable via UI and/or properties files.

## Summary

This chapter discussed how to develop layout templates in both Ext and Plugins SDK, and how to build themes in Plugins SDK. It first introduced how to build layout templates in Ext. Then it discussed how to build layout templates and themes in Plugins SDK and how to add Velocity services in themes. Finally, it addressed how to use Plugins SDK in an efficient way.

In the next chapter, we're going to discuss how to build a portal instance—Social Office.

# 10

## Building My Social Office

It would be nice to be able to build social office and social networking in our web site bookpub.com. Thus, an end user could be a part of the team without sacrificing his/her own preferences; a manager could raise productivity and improve team dynamics while watching the bottom line; and a team could keep everyone up-to-date on each other's activity, no matter where they are. Meanwhile, the users could modify portal settings and controls as well.

**Social Office** is a social collaboration on top of the portal – a full virtual workspace that streamlines communication and builds up group cohesion. All components are tied together seamlessly, which gets everyone on the same page by sharing the same look and feel. Especially, the dynamic activity tracking gives us a birds-eye view of who has been doing what and when within each individual site. Social Office is not another separate download bundle, but a specific instance of Liferay portal. **Control Panel** is a feature of Liferay portal that allows the users to modify portal settings and controls. In my Social Office, the Control Panel is called as **My Profile**.

This chapter will first introduce the Control Panel – how it works and how to customize it. Then we will address **Inter-Portlet Communication (IPC)**. Later, we will discuss how to build **My Social Office** with theme and portlets, and how to hook the properties and JSP files into it. Finally, we will discuss an efficient way to use hooks – a feature to catch hold of the properties and JSP files into an instance of Liferay portal (such as My Social Office) as if catching them with a hook.

By the end of this chapter, you will have learned how to:

- Experience the Control Panel
- Build inter-portlet communication
- Develop the Social Office theme
- Add mail and chat portlets
- Build Social Office with portlets
- Hook portal properties and custom JSP files into Social Office
- Use hooks more efficiently

## Experiencing the Control Panel

The Control Panel provides a centralized administration for all the content, users, organizations, communities, roles, server resources, and more. Additionally, it also provides full customizability with the ability to hide different parts of the form as desired, or add custom parts with the portlets.

## What's Control Panel?

Generally speaking, the Control Panel is a feature of Liferay portal that allows us to modify portal settings and controls. As shown in the following screenshot, in **My Account** every user will have access to the Control Panel and will at least have access to edit his/her account details. These account details include user information (details, password, reminder question and answer, organizations, communities, user groups, roles, and so on), user identification (addresses, phone number, additional email addresses, web sites, instant messenger, social network, SMS, OpenID, and so on), and miscellaneous (announcements, display settings, comments and custom attributes, and so on), among many others. Similarly, in **My Pages**, every user has the ability to manage personal pages, that is, My Community.



As shown in the following screenshot, you can edit any type of content that might be published through community or organization pages from the Control Panel. For example, through web content (that is, journal) you can manage articles, structures, and templates. Further, you can also manage feeds, permissions, recent articles, recent structures, and recent templates. All of the content must belong to a community or an organization on top of the Liferay portal. Thus, whenever the administration selects a tool to manage content, the title (for example, **Content for Guest**) shows for which community or organization you administer the content by default. It can either be the one you came from or another one that you are allowed to select. If there is only one community or administrator on the UI, things will be very simple. But if you have many communities or organizations, for example **Book Street**, **Book Workshop**, **Guest**, and **Palm Tree Publications** (an organization), you still have a fast way to move around them.



Of course, you can manage the users of the portal in the Control Panel. The users who an administrator can manage are filtered by his/her permissions. As shown in the following screenshot, you can manage users (for example **Edit**, **Permissions**, **Impersonate Users**, **Deactivate/Activate**) with your own permissions. For example, as an admin, you have full permissions for user management. But as an editor David Berger, you may have limited permissions for management of some users.

Likewise, you can manage organizations, communities, user groups, roles, and so on in the portal category of the Control Panel. Note that the **Plugins Configuration** portlet is under the **Portal** category with the render weight as **9.0**. In short, the Control Panel is a UI that allows administrating the complete portal from one community to thousands of communities, and from users and organizations to documents, images, articles, polls, tags, blogs, Wiki, forums, and so on. It puts all portlets together in one single place.

The screenshot shows the Alfresco Control Panel interface. On the left, there is a vertical navigation bar with categories: 'Palm Tree' (My Account, My Pages), 'Content' (Content, Portal, Plugins Configuration), 'Portal' (Users, Organizations, Communities, User Groups, Roles, Password Policies, Settings, Monitoring), and 'Server' (Server). The 'Portal' category is currently selected, and the 'Users' sub-category is highlighted. The main content area is titled 'Users' and contains a table with two rows of data:

	First Name	Last Name ▾	Screen Name	Job Title	Organization	Actions
<input type="checkbox"/>	David	Berger	david	Editor		
<input type="checkbox"/>	Lotti	Stein	lotti	Editor		

The Control Panel groups all the portlets into four categories: **My**, **Content**, **Portal**, and **Server**. The **My** category provides access sign to edit account details and manage personal pages, for example, **My Account** and **My Pages**. The **Content** category furnishes access to manage all types of contents: Web Content, Document Library, Image Gallery, Bookmarks, Calendar, Message Boards, Blogs, Wiki, Polls, Software Catalog, Tags and Categories, and so on. It can even include content from custom portlets. For instance, if we had a portlet named **Manage Ads**, we could put it under the **Content** category smoothly.

The **Portal** category gives access to manage users, organizations, communities, user groups, roles password polices, settings, monitoring, plugins configuration, and all portal wide elements; whereas the **Server** category renders access to server-related administration tasks such as checking the memory usage, installing plugins, managing portal instances, setting up WSRP producer and consumer, and so on.

## How does it work?

As you can see, the Control Panel provides a preconfigured UI with all the administration tools provided by the portal, and it allows the automatic delegation of administration. For example, if a user is assigned the *Organization Admin* role, he/she would be able to administer that organization automatically. The Control Panel also adds support for disabling these parts of the administration when they are not being used in a given installation. For example, if the users don't use user groups, they would be able to hide that option. By the way, the Control Panel supports adding custom administration tools in a consistent and integrated way. Moreover, it provides an easy-to-use UI for portal installations no matter how many web sites are managed.

## Using the Control Panel theme

All of the users will have access to the Control Panel by following a link to **Control Panel** (or **My Profile** in Social Office) in the top right menu under the dock. Once accessed, the Control Panel will automatically show the sections (that is, categories) based on the permissions of the current user. Each user will have the ability to atleast edit his/her account details. How does it work? Let's have a deep look at it.

The Control Panel theme is specified in the `/portal/portal-web/docroot/html/themes/control_panel` folder where you would see a customized theme in the `/_diffs` folder. Customized CSS, images, JavaScript, and templates are specified only in the `/_diffs` folder. In the `/_diffs/css` folder, a CSS file `custom.css` is created and customized. In the `/_diffs/images` folder, all of the customized images with subfolders are added. For example, the two images `screenshot.png` and `thumbnail.png` show how a page with the current theme looks. Further, a common subfolder is created and all general images are customized in this subfolder. In the `/_diffs/javascript` folder, a JavaScript file `javascript.js` is created as well as customized. Last but not the least, in the `/_diffs/templates` folder, a Velocity template file `portal_normal.vm` is created and customized.

Likewise, you can find the definition of Control Panel theme as well. Locate the XML file `liferay-look-and-feel.xml` in the `/portal/portal-web/docroot/WEB-INF/` folder. You will see the following code:

```
<look-and-feel>
 <!-- ignore details -->
 <theme id="control-panel" name="Control Panel">
 <root-path>/html/themes/control_panel</root-path>
 </theme>
 <!-- ignore details -->
</look-and-feel>
```

The code above shows a theme with ID `control-panel` and name `Control Panel`.

## Employing Control Panel settings

We have discussed the Control Panel theme. Now let's look at the Control Panel settings in detail. First, locate the properties file `portal.properties` in the `/portal/portal-impl/src` folder and you will see the following code:

```
layout.edit.page[control_panel] = /portal/layout/edit/control_panel.jsp
layout.view.page[control_panel] = /portal/layout/view/control_panel.jsp
layout.url[control_panel] = ${liferay:mainPath}/portal/layout?p_1_
id=${liferay:plid}
layout.url.friendliable[control_panel] = true
layout.parentable[control_panel] = true
layout.first.pageable[control_panel] = true
```

The code above shows the settings for the Control Panel layouts. The layout has an edit page `/portal/layout/edit/control_panel.jsp`, and a view page `/portal/layout/view/control_panel.jsp`. Both of them are specified in the `/portal/portal-web/docroot/html/` folder.

Then, let's see the default categories definition of control panel. Locate Java file `PortletCategoryKeys.java` at the `com.liferay.portal.util` package in the `/portal/portal-service/src` folder. You will see the following code:

```
public class PortletCategoryKeys {
 public static final String CONTENT = "content";
 public static final String MY = "my";
 public static final String PORTAL = "portal";
 public static final String SERVER = "server";
 public static final String[] ALL = {MY, CONTENT, PORTAL, SERVER}; }
```

The code above shows category keys for Control Panel. By default, four categories are specified: `content`, `my`, `portal`, and `server`. As stated above, both the edit page `/portal/layout/edit/control_panel.jsp` and the view page `/portal/layout/view/control_panel.jsp` use these categories.

Now, let's study the Velocity templates for Control Panel. The **Control Panel** link of the dock menu is specified in `/portal/portal-web/docroot/html/themes/_unstyled/templates/dock.vm` as follows:

```
#if ($show_control_panel)
<li class="control-panel">
 $control_panel_text
#end
```

The preceding code shows template variables \$show\_control\_panel, \$control\_panel\_url, and \$control\_panel\_text for Control Panel. These template variables are specified in /portal/portal-web/docroot/html/themes/\_unstyled/templates/init.vm as follows:

```
#set ($theme_display = $themeDisplay) ## ignore details *#
#set ($show_control_panel = $theme_display.isShowControlPanelIcon())
#if ($show_control_panel)
 #set ($control_panel_text = $languageUtil.get($company_id,
 $locale, "control-panel"))
 #set ($control_panel_url = $theme_display.getURLControlPanel())
#endif
```

Further, the Control Panel URL (set by themeDisplay.setURLControlPanel) is specified in com.liferay.portal.events.ServicePreAction.java as follows:

```
/* ignore details */
String urlControlPanel = friendlyURLPrivateGroupPath +
 "/control_panel";
if (Validator.isNotNull(doAsUserId)) {
 urlControlPanel = HttpUtil.addParameter(urlControlPanel,
 "doAsUserId", doAsUserId);
}
if (scopeGroupId > 0) {
 urlControlPanel = HttpUtil.addParameter(urlControlPanel,
 "doAsGroupId", scopeGroupId);
}
if (refererPlid > 0) {
 urlControlPanel = HttpUtil.addParameter(urlControlPanel,
 "refererPlid", refererPlid);
}
else if (plid > 0) {
 urlControlPanel = HttpUtil.addParameter(urlControlPanel,
 "refererPlid", plid);
}
themeDisplay.setURLControlPanel(urlControlPanel);
// ignore details
```

The code above shows a way to build the control panel URL. The URL would look like this: /control\_panel?doAsUserId=value&doAsGroupId= value&refererPlid=value.

## Configuring portlets for Control Panel

As a developer, you can actually decide which items in the categories will be displayed. In fact, each of these items is a portlet, so you can disable the portlet if you don't want it to be shown either through the UI or through the `liferay-portlet-ext.xml` file.

Suppose you want to disable the Communities portlet, you can make it invisible through the UI. To do so, just click on the **Plugins Configuration** portlet (portlet ID 132) first, and then search the **Communities** portlet. Finally, click on the **Communities** portlet, uncheck the **Active** checkbox, and click on the **Save** button.

You can also add any custom portlet to the desired place in that menu and make it part of the Control Panel. In short, you just need to add a few elements in the `liferay-portlet.xml` file under the folder `/portal/portal-web/docroot/WEB-INF`. You can find Control Panel specification of the portlet ID 132 (that is, Plugins Configuration) as follows:

```
<control-panel-entry-category>portal</control-panel-entry-category>
<control-panel-entry-weight>9.0</control-panel-entry-weight>
```

This code shows the Control Panel specification of the 132 portlet. The first element `control-panel-entry-category` determines the category of the menu where the portlet ID 132 will be added. It is `portal` for the portlet ID 132. The second element `control-panel-entry-weight` determines the relative ordering for the portlet within a given category. The higher the number, the lower in the list the portlet will appear within that category. It is `9.0` for the portlet ID 132.

Similarly, the following code shows the Control Panel specification of the portlet ID 2 (that is, My Account):

```
<control-panel-entry-category>my</control-panel-entry-category>
<control-panel-entry-weight>1.0</control-panel-entry-weight>
<control-panel-entry-class>
 com.liferay.portlet.myaccount.MyAccountControlPanelEntry
</control-panel-entry-class>
```

The code above shows Control Panel specification of the portlet ID 2. The first element `control-panel-entry-category` determines the category of the menu where the portlet ID 2 will be added; it is `My` for the portlet ID 2. The second element `control-panel-entry-weight` determines the render weight. It is `1.0` for the portlet ID 2. The last one `control-panel-entry-class` is optional, and it allows deciding under which conditions the item will be shown or not shown. The `control-panel-entry-class` element has the value `com.liferay.portlet.myaccount.MyAccountControlPanelEntry`.

Further, you can find similar definition of the Control Panel on Users and My Pages. The control-panel-entry-class of Users (portlet ID 125) has the value com.liferay.portlet.enterpriseadmin.UsersControlPanelEntry, whereas the control-panel-entry-class of My Pages (portlet ID 140) has the value com.liferay.portlet.mypages.MyPagesControlPanelEntry. Together MyAccountControlPanelEntry, MyPagesControlPanelEntry, and UsersControlPanelEntry in the folder /portal/portal-impl/src/ extend the BaseControlPanelEntry class. Whereas the BaseControlPanelEntry class implements the interface ControlPanelEntry. Both BaseControlPanelEntry and ControlPanelEntry exist in the package com.liferay.portlet under the /portal/portal-service/src/ folder.

The ControlPanelEntry interface specifies if the condition isVisible(PermissionChecker permissionChecker, Portlet portlet) for an item in Control Panel will be shown or not. For example, the MyAccountControlPanelEntry class extends BaseControlPanelEntry and implements ControlPanelEntry as follows:

```
public class MyAccountControlPanelEntry extends BaseControlPanelEntry
{
 public boolean isVisible(PermissionChecker
 permissionChecker, Portlet portlet)
 throws Exception { return true; }
}
```

The code above shows that MyAccountControlPanelEntry extends ControlPanelEntry. It always returns true. It means that the My Account portlet is visible for any user.

## How to customize it?

We have introduced the mechanism that explains how the Control Panel works. Now, let's use the mechanism to customize it as well. In general, there are three levels to customize the Control Panel: changing theme, updating edit and view pages, and configuring custom portlets in categories (for example, My, Content, Portal, and Server).

### Changing theme

You can customize the look and feel of the Control Panel by following these steps:

1. Create a folder named /themes in the /ext/ext-impl/docroot/html/ folder.
2. Copy the entire /control\_panel folder from the /portal/portal-impl/docroot/html/themes/ folder to the /ext/ext-impl/docroot/html/themes/ folder.

3. Change `/_diffs/css/custom.css` if you want to update the CSS of the Control Panel.
4. Change `/_diffs/images/*` if you want to update images of the Control Panel.
5. Change `/_diffs/javascript/javascript.js` if you want to add or update JavaScript functions of the Control Panel.
6. Change `/_diffs/templates/*` if you want to add or update Velocity template files of Control Panel, including `dock.vm`, `init_custom.vm`, `navigation.vm`, `portal_normal.vm`, `portlet_pop_uo.vm`, and `portlet.vm`.

In short, through updating CSS, images, JavaScript, and Velocity template files, you can update the entire look and feel of the Control Panel.

## **Updating both edit page and view page**

Besides updating the theme to change look and feel, you can also modify the edit page and view directly. By doing this, you can change the view of the Control Panel as well. You can do it as follows:

1. Create a folder named `/layout` in the `/ext/ext-impl/docroot/html/portal` folder.
2. Create the `/edit` and `/view` folders in the `/ext/ext-impl/docroot/html/portal/layout` folder.
3. Copy the `control_panel.jsp` file from the `/portal/portal-impl/docroot/html/portal/layout/edit` folder to the `/ext/ext-impl/docroot/html/portal/layout/edit` folder.
4. Copy the `control_panel.jsp` file from the `/portal/portal-impl/docroot/html/portal/layout/view` folder to the `/ext/ext-impl/docroot/html/portal/layout/view` folder.
5. Change the `control_panel.jsp` file in the `/ext/ext-impl/docroot/html/portal/layout/edit` folder if you want to update the edit page of the Control Panel.
6. Change the `control_panel.jsp` file in the `/ext/ext-impl/docroot/html/portal/layout/view` folder if you want to update the view page of Control Panel.

In short, by updating the edit and view page, you can change the user experience of the Control Panel as well.

## Configuring customized portlets

Moreover, we can also configure customized portlets in categories of the Control Panel. Suppose we plan to put the Ext Communities portlet in the portal category and the render weight is 10.0. At the same time, control-panel-entry-class is com.ext.portlet.communities.ExtUserControlPanelEntry, which allows us to decide the conditions under which the Ext Communities item will or will not be shown in the portal category. Let's configure the Ext Communities portlet in the portal category with the render weight 10.0.

First of all, we need to create a Control Panel entry for the Ext Communities portlet as follows:

1. Create a Java file ExtUserControlPanelEntry.java in the com.ext.portlet.communities package under the /ext/ext-impl/src folder and open it.
2. Add the following lines at the beginning of this file and save it:

```
public class ExtUserControlPanelEntry extends
BaseControlPanelEntry {
 public boolean isVisible(PermissionChecker permissionChecker,
 Portlet portlet) throws Exception {
 List<Organization> organizations =
 OrganizationLocalServiceUtil.getManageableOrganizations(
 permissionChecker.getUserId());
 for (Organization organization : organizations) {
 if (permissionChecker.isCommunityAdmin(
 organization.getGroup().getGroupId())) {
 return true;
 }
 if (OrganizationPermissionUtil.contains(
 permissionChecker, organization.getOrganizationId(),
 ActionKeys.MANAGE_USERS)) {
 return true;
 }
 if (OrganizationPermissionUtil.contains(
 permissionChecker, organization.getOrganizationId(),
 ActionKeys.MANAGE_SUBORGANIZATIONS)) {
 return true;
 }
 }
 return false;
 }
}
```

The preceding code shows that `ExtUserControlPanelEntry` extends the `BaseControlPanelEntry` class and implements the `isVisible` method of the `ControlPanelEntry` interface indirectly. As mentioned earlier, `BaseControlPanelEntry` implements the `ControlPanelEntry` interface. Inside the `isVisible` method, it checks whether it is community admin or not, and also whether it has the `MANAGE_USERS` and `MANAGE_SUBORGANIZATIONS` permissions or not.

Then, we need to configure the Ext Communities portlet in the Control Panel by using the following steps:

1. Locate the XML file `liferay-portlet-ext.xml` in the `/ext/ext-web/docroot/WEB-INF/` folder and open it.
2. Locate the portlet ID `extCommunities` and add the following lines after the line `<struts-path>ext/communities</struts-path>` in `liferay-portlet-ext.xml` and save it:

```
<control-panel-entry-category>
 portal
</control-panel-entry-category>
<control-panel-entry-weight>
 10.0
</control-panel-entry-weight>
<control-panel-entry-class>
 com.ext.portlet.communities.ExtUserControlPanelEntry
</control-panel-entry-class>
```

The code above shows that the Ext Communities portlet will be displayed in the `portal` category with the render weight `10.0`. At the same time, it specifies `control-panel-entry-class` as `com.ext.portlet.communities.ExtUserControlPanelEntry`, which allows us to decide the conditions under which the Ext Communities item will be shown or not in the `portal` category.

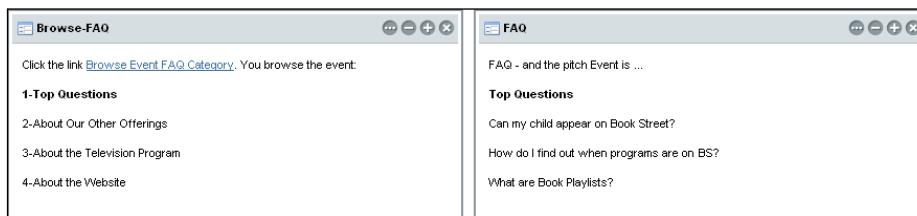
Note that the Control Panel render weight for `control-panel-entry-weight` is the only thing that matters for determining the uniqueness of a portlet within a control panel category. So you need to keep your portlet (for example, Ext Communities) render weight (for example, `10.0`) different from the portlets (for example, from `1.0` to `9.0` in the category `portal`) that exist in Liferay portal, and from other portlets that you may want to add to the portal.

That's it! If you deployed these updates, log in as an admin and click on the **Control Panel** link. You would see the Ext Communities portlet displayed in the `Portal` category with the render weight `10.0`. In short, the Control Panel allows us to administrate the complete portal with capability of inter-portlet communication from users and organizations to documents, images, articles, polls, tags, blogs, Wiki, and so on.

## Building Inter-Portlet Communication

Inter-portlet communication (or IPC) becomes important with portlets applications, for example, the web sites `bookpubstreet.com` and `bookpubworkshop.com` with the capability of Control Panel and Social Office. These applications are composed of more than one portlet for their functionality. As stated in Chapter 2, *Working with JSR-286 Portlets*, JSR-286 portlets address IPC – an event model that allows portlets to send and receive events. We will go over the simple example FAQ of IPC that consists of two portlets: **FAQ** and **Browse-FAQ**. The **Browse-FAQ** portlet contains categories: **Top Questions**, **About Our Other Offerings**, **About the Television Program**, and **About the Website**. The portlet **FAQ** also contains the categories **Top Questions**, **About Our Other Offerings**, **About the Television Program**, and **About the Website**. Each category may contain a set of questions with an answer. For example, the **Top Questions** category contains three questions: **Can my child appear on Book Street?**, **How do I find out when programs are on BS?**, and **What are Book Playlists?**.

When a category (for example, **Top Questions**) is selected in the portlet **Browse-FAQ**, the category should be highlighted. At the same time, the portlet **FAQ** should display questions with answers if applicable for the selected category **Top Questions** – that is, IPC is applicable on this scenario. In this section, we're going to implement IPC with the example **FAQ**. Two portlets **Browse-FAQ** and **FAQ** will be developed in Plugins SDK:



## Creating IPC portlet project

First of all, we need to create a folder named `ipc-faq-portlet` in the `$PLUGINS_SDK_HOME/portlets` folder in Plugins SDK. Everything for the portlets FAQ and Browse-FAQ will go in this folder `ipc-faq-portlet`. In the folder `ipc-faq-portlet`, create another folder `docroot` and an XML file `build.xml`. Open it and add the following lines at its beginning:

```
<?xml version="1.0"?>
<project name="portlet" basedir=". " default="deploy">
 <import file="..../build-common-portlet.xml" />
</project>
```

The preceding code shows a portlet project with the default target deploy. It takes one property plugin.version with value 1. Similar to layouttpl and theme, this property should be imported from custom properties file build.\${user.name}.properties. Meanwhile, it imports a build-common-portlet.xml file from the parent folder. Now in the /docroot folder, create the folders /WEB-INF, /css, /images, /javascripts, and /jsp. The /WEB-INF folder contains portlet application specification files, subfolders /lib, /tld, and /src. The /css folder accommodates all CSS files, if applicable, using subfolders; whereas all JavaScript files goes to the /javascripts folder; all JSP files go to the /jsp folder, if applicable, using subfolders; and all image files go to the /images folder.

Optionally, you could run a script to create a blank portlet project. For example, for the above project, we have a project named ipc-faq and a portlet display name IPC FAQ. On Linux or Mac, you would change the directory to \$PLUGINS\_SDK\_HOME/portlets and then type the following command:

```
./create.sh ipc-faq "IPC FAQ"
```

On Windows, you would change the directory to \$PLUGINS\_SDK\_HOME/portlets and then type the following command:

```
create.bat ipc-faq "IPC FAQ"
```

This command will create a blank portlet in the folder portlets. In fact, the above script uses predefined ZIP file portlet.zip to create a blank portlet with the following Ant command:

```
ant -Dportlet.name=$1 -Dportlet.display.name=\"$2\" create
```

## Constructing IPC portlets

As stated above, we have successfully created a portlet project ipc-faq-portlet. Now, let's build IPC portlets FAQ and Browse-FAQ as well. First, we need to set up these two portlets in the portlet.xml file. To do so, first create an XML file portlet.xml in \$PLUGINS\_SDK\_HOME/portlets/ipc-faq-portlet/docroot/WEB-INF and open it. Add the following lines at its beginning and save it. Note that the portlets would be in the same project (one WAR file), so there's only one portlet.xml.

```
<?xml version="1.0"?>
<portlet-app xmlns="http://java.sun.com/xml/ns/portlet/portlet-
 app_2_0.xsd"
 version="2.0"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://java.sun.com/xml/ns/
 portlet/portlet-app_2_0.xsd
 http://java.sun.com/xml/ns/portlet/
 portlet-app_2_0.xsd">
```

```
<portlet>
 <portlet-name>pitcher-portlet</portlet-name>
 <display-name>Event Pitcher</display-name>
 <portlet-class>com.book.ipc.PitcherPortlet</portlet-class>
 <init-param>
 <name>view-jsp</name>
 <value>/jsp/pitcher/view.jsp</value>
 </init-param>
 <!-- ignore details -->
 <portlet-info>
 <title>Browse-FAQ</title>
 <short-title>Browse-FAQ</short-title>
 <keywords>IPC Event Pitcher</keywords>
 </portlet-info>
 <!-- ignore details -->
 <supported-publishing-event>
 <qname xmlns:x="http://book.com/events">x:ipc.pitch</qname>
 </supported-publishing-event>
</portlet>
<portlet>
 <portlet-name>catcher-portlet</portlet-name>
 <display-name>Event Catcher</display-name>
 <portlet-class>com.book.ipc.CatcherPortlet</portlet-class>
 <init-param>
 <name>view-jsp</name>
 <value>/jsp/caught/view.jsp</value>
 </init-param>
 <!-- ignore details -->
 <portlet-info>
 <title>FAQ</title>
 <short-title>FAQ</short-title>
 <keywords>IPC Event Catcher</keywords>
 </portlet-info>
 <!-- ignore details -->
 <supported-processing-event>
 <qname xmlns:x="http://book.com/events">x:ipc.pitch</qname>
 </supported-processing-event>
</portlet>
 <!-- ignore details -->
 <event-definition>
 <qname xmlns:x="http://book.com/events">x:ipc.pitch</qname>
 <value-type>java.lang.String</value-type>
 </event-definition>
</portlet-app>
```

The code above shows the definition of two portlets (for example Browse-FAQ and FAQ), and an event (for example, x:ipc.pitch). Incidentally, you could find the portlet app **XSD (XML Schema Definition)** in /portal/definitions/portlet-app\_2\_0.xsd.

## Defining portlets

The preceding code specifies the portlet `pitcher-portlet` with a name `pitcher-portlet`, display name `Event Pitcher`, and portlet class `com.book.ipc`.

`PitcherPortlet`. The `init-param` points to a JSP file `/jsp/pitcher/view.jsp` in `$PLUGINS_SDK_HOME/portlets/${portlet.name}/docroot`. The `${portlet.name}` is the actual portlet project name, for example `ipc-faq-portlet`. In addition, the `portlet-info` tag is specified with title, short title (for example, `Browse-FAQ`), and keywords (for example, `IPC Event Pitcher`). Then, it enumerates the portlet `catcher-portlet` with a name `catcher-portlet`, display name `Event Catcher`, and portlet class `com.book.ipc.CatcherPortlet` in the same pattern. `init-param` points to a JSP `/jsp/caught/view.jsp`. The `portlet-info` tag is specified with title, short title (for example, `FAQ`), and keywords (for example, `IPC Event Catcher`).

## Defining events

Below the last portlet `FAQ`, an event has been defined with the `event-definition` tag. `qname` is a qualified name that allows events to be specified with namespace properly so that no two events are identical. For example, the code above defines an event called `ipc.pitch` within the namespace `http://book.com/events`. The payload for this event is a `String` containing the type of pitch event that the portlet `pitcher-portlet` generated. It also tells the portlet container which portlet sends the event. For instance, the portlet `pitcher-portlet` will publish an event. The publishing event is specified with the `supported-publishing-event` tag and it has the value `ipc.pitch` within a namespace `http://book.com/events`.

In general, one or many portlets can be configured to receive events. For example, the portlet `catcher-portlet` will receive and process the event. The processing event is specified with the `supported-processing-event` tag and it has the value `ipc.pitch` within the namespace `http://book.com/events`.

## Registering portlets

Later, we need to register portlets. To do so, create an XML file `liferay-portlet.xml` in `$PLUGINS_SDK_HOME/portlets/${portlet.name}/docroot/WEB-INF` and open it. Add the following lines at its beginning and save it:

```
<?xml version="1.0"?>
<!DOCTYPE liferay-portlet-app PUBLIC "-//Liferay//DTD Portlet
Application 5.2.0//EN" "http://www.liferay.com/dtd/
liferay-portlet-app_5_2_0.dtd">
<liferay-portlet-app>
 <portlet>
 <portlet-name>pitcher-portlet</portlet-name>
```

```
<icon>/images/icon.png</icon>
<instanceable>true</instanceable>
<header-portlet-css>/css/common.css</header-portlet-css>
<header-portlet-javascript>
 /javascripts/common.js
</header-portlet-javascript>
</portlet>
<portlet>
 <portlet-name>catcher-portlet</portlet-name>
 <icon>/images/icon.png</icon>
 <instanceable>true</instanceable>
 <header-portlet-css>/css/common.css</header-portlet-css>
 <header-portlet-javascript>
 /javascripts/common.js
 </header-portlet-javascript>
</portlet>
 <!-- ignore details -->
</liferay-portlet-app>
```

The code above shows registration of portlets `pitcher-portlet` and `catcher-portlet`. It specifies portlet names `pitcher-portlet` and `catcher-portlet` by the `portlet-name` tag. It also specifies `icon` with a value `/images/icon.png`, `instanceable` with a value `true`, `header-portlet-CSS` with a value `/css/common.css`, and `header-portlet-javascript` with a value `/javascripts/common.js`. By the way, you could find details of portlet DTD in `/portal/definitions/liferay-portlet_5_2_0.dtd`. In addition, we expect to put portlets in the `IPC` category. To do so, create an XML file `liferay-display.xml` in `$PLUGINS_SDK_HOME/portlets/${portlet.name}/docroot/WEB-INF`. Open it, add the following lines at its beginning, and save it:

```
<?xml version="1.0"?>
<!DOCTYPE display PUBLIC "-//Liferay//DTD Display 5.2.0//EN" "http://
www.liferay.com/dtd/liferay-display_5_2_0.dtd">
<display>
 <category name="IPC">
 <portlet id="pitcher-portlet" />
 <portlet id="catcher-portlet"/>
 </category>
</display>
```

Parenthetically, you can find details of Liferay display DTD in `/portal/definitions/liferay-display_5_2_0.dtd`.

## Specifying portlet process actions

Now we will implement the event. One of the nice things about JSR-286 portlets is that `GenericPortlet` is reliant on Java SE 5 annotations. Thus, we can create methods to process specific actions without having to override the `processAction` method. We now create two Java files `PitcherPortlet.java` and `CatcherPortlet.java` in the `com.book.ipc` package for portlets `pitcher-portlet` and `catcher-portlet`, respectively. `com.book.ipc.PitcherPortlet` in the `$PLUGINS_SDK_HOME/portlets/${portlet.name}/docroot/WEB-INF/src` folder is used to generate an event and then publish it. The JSP file displays a URL to the event. When a user clicks on the URL, the event is called. We can implement this in `com.book.ipc.PitcherPortlet` as follows:

```
public class PitcherPortlet extends GenericPortlet {
 // Ignore details
 @ProcessAction(name="pitchEvent")
 public void pitchEvent(ActionRequest request, ActionResponse
 response) {
 String pitchType = null;
 Random random = new Random(System.currentTimeMillis());
 int pitch = random.nextInt(5);
 switch (pitch) {
 case 1: pitchType = "Top Questions"; break;
 case 2: pitchType = "About Our Other Offerings"; break;
 case 3: pitchType = "About the Television Program"; break;
 case 4: pitchType = "About the Website"; break;
 default: pitchType = "Top Questions";
 }
 response.setRenderParameter("pitch", pitchType);
 QName qName = new QName("http://book.com/events", "ipc.pitch");
 response.setEvent(qName, pitchType);
 }
 /* ignore details */
}
```

The code above shows that `PitcherPortlet` extends `GenericPortlet`. This calls any method with the same name as the process action name, for example `pitchEvent`. The `pitchEvent` method starts with an annotation `@ProcessAction(name="pitchEvent")`. It generates a string representing the kind of pitch, that is, FAQ category. It then sends an event with the name defined in the `portlet.xml` file containing the string payload of the type of event—FAQ category. Similarly, `com.book.ipc.CatcherPortlet` in the `$PLUGINS_SDK_HOME/portlets/${portlet.name}/docroot/WEB-INF/src` folder is used to take the value of the event payload and set it as a render parameter. Thus, it will be displayed during the render phase of the portlet. We can implement `com.book.ipc.CatcherPortlet` as follows:

```
public class CatcherPortlet extends GenericPortlet {
 // ignore details
 @ProcessEvent(qname={"http://book.com/events}ipc.pitch")
 public void catchEvent(EventRequest request, EventResponse
 response) {
 Event event = request.getEvent();
 String pitch = (String)event.getValue();
 response.setRenderParameter("pitch", pitch);
 }
 /* ignore details */
}
```

As shown in the code above, it uses the `processEvent` method of `GenericPortlet` to direct the processing to the `catchEvent` method starting with an annotation `@ProcessEvent`. In this case, `qname` of the event is the parameter.

## Specifying portlet views

As shown in the `portlet.xml` file, the `pitcher-portlet` portlet has `init-param` which points to a JSP `/jsp/pitcher/view.jsp`. The `catcher-portlet` portlet has `init-param` which points to a JSP `/jsp/catcher/view.jsp`. That is, the view of this application consists of two JSP files (for example, `/jsp/pitcher/view.jsp` and `/jsp/catcher/view.jsp`) that implement the view mode for each portlet. Let's implement these JSP files. First, let's create the JSP `/jsp/pitcher/view.jsp` by using the following steps:

1. Create a folder named `pitcher` in the `$PLUGINS_SDK_HOME/portlets/${portlet.name}/docroot/jsp/` folder.
2. Create JSP file `view.jsp` in the folder `/docroot/jsp/pitcher/` and open it.
3. Add the following lines at the beginning of this file and save it:

```
<!-- ignore details -->
<portlet:defineObjects />
<% List<String> list = Collections.synchronizedList(new
 ArrayList<String>());
 list.add("Top Questions");
 list.add("About Our Other Offerings");
 list.add("About the Television Program");
 list.add("About the Website");
 String pitch = (String)renderRequest.getParameter("pitch"); %>
<p>
 Click the link <a href=" </portlet:actionURL>">Browse Event FAQ
 Category.
 You browse the event:
```

```
</p>
<% int index = 1;
for(String key: list){
 if(key.equals(pitch)) {
 %><p><%= index %>-<%= key %></p> <%
 }
 else {
 %><p><%= index %>-<%= key %></p> <%
 }
 index++;
}
%>
```

The code above shows a portlet action URL with the action name as `pitchEvent`, which maps to the method name annotation `@ProcessAction(name="pitchEvent")` as stated previously.

Then, let's create the JSP `/jsp/catcher/view.jsp`. Similarly, create a folder `catcher` in the `$PLUGINS_SDK_HOME/portlets/${portlet.name}/docroot/jsp/` folder. Then create the JSP file `view.jsp` in the `/docroot/jsp/catcher/` folder and open it. Add the following lines at its beginning and save it:

```
<!-- ignore details -->
<portlet:defineObjects />
<% String pitch = (String)renderRequest.getParameter("pitch");
String subc1 = "Can my child appear on Book Street?";
String subc2 = "How do I find out when programs are on PBS?";
String subc3 = "What are Book Playlists?"; %>
<p>FAQ - and the pitch Event is ... </p>
<p><% if (pitch != null) { %> <p><%= pitch %></p>
<% if(pitch.equals("Top Questions")){
 %><p><%= subc1 %></p><p><%= subc3 %></p><p><%= subc3 %></p><%
}
}
else { %> ... waiting for pitch Event. <% } %>
```

The code above shows the render parameter. If the render parameter has the value `Top Questions`, it would show this value and its questions.

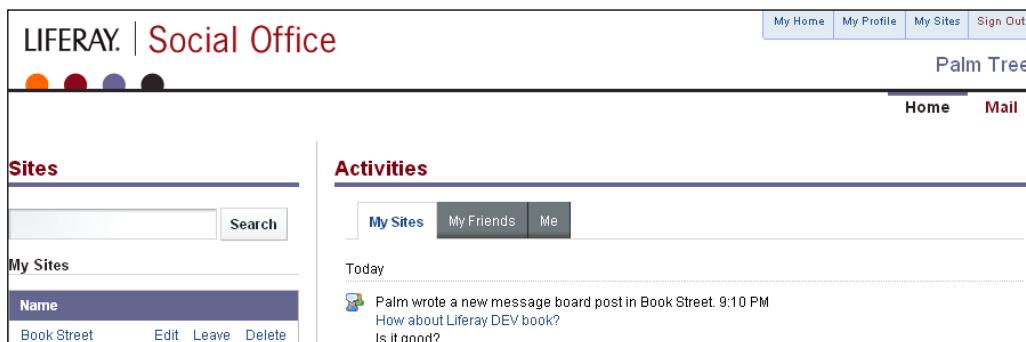
Cool! You are ready to deploy these IPC portlets. You could drop the file `build.xml` in the `$PLUGINS_SDK_HOME/portlets/${portlet.name}` folder into the Ant view. Then the only thing you need to do is to double-click on the `deploy` target under `portlet` of the Ant view. The portlet will be compiled and deployed as well. Note that we only provided a framework to show how to build IPC. You can definitely reuse the framework and customize these portlets (or process actions or views) anytime in Plugins SDK for your own requirements.

## Developing Social Office theme

Generally speaking, Social Office is a social collaboration solution for the enterprise. It allows people to collaborate effectively and efficiently. One of the cool features of Social Office is its use of Microsoft Office integration. It is not another separate portal package, but a specific instance of the portal. Here we're going to build My Social Office.

All of the features of Social Office are available in the portal as well. In fact, Liferay portal is the framework and the Social Office is a customization of this framework (for example, custom portlets, themes, properties, and so on). The core functionality is identical between the portal and the Social Office. What differs is the user experience—Social Office is for a specific use case, whereas the portal is for a more generic use case.

How to build My Social Office according to my requirements? The simple answer is that My Social Office is a customization of the framework of the portal, involving custom themes, custom portlets, and custom properties. This section will discuss how to build the theme for My Social Office.



As shown in the above screenshot, **My Social Office** has its own overall look and feel, that is, Social Office theme. This theme uses CSS, images, JavaScript and Velocity templates and it controls the whole look and feel of pages generated by My Social Office. Let's have an in-depth look at how to build this Social Office theme.

## Setting up the theme project

The Social Office theme (**so-theme** in short), as well as a normal theme, consists of CSS, images, JavaScript, and Velocity templates. It controls the whole look and feel of pages generated by Social Office. Therefore, when creating the so-theme, we need to consider these four groups as well.

First of all, we need to create a folder named **so-theme** in the **/themes** folder. Everything for so-theme will go in this folder. Under the **so-theme** folder, create another folder **/docroot** and an XML file **build.xml**. The **build.xml** file has same content as that of **book-street-theme**.

Then in the **/docroot** folder, create folders **/\_diffs** and **/WEB-INF**. In the **/\_diffs** folder, create four subfolders: **css**, **images**, **javascript**, and **templates**. In the **/WEB-INF** folder create a properties file **liferay-plugin-package.properties** and an XML file **liferay-look-and-feel.xml**. Add the following lines at its beginning:

```
<?xml version="1.0"?>
<!DOCTYPE look-and-feel PUBLIC "-//Liferay//DTD Look and Feel
5.2.0//EN" "http://www.liferay.com/dtd/liferay-look-and-
feel_5_2_0.dtd">
<look-and-feel>
 <compatibility>
 <version>5.2.0+</version>
 </compatibility>
 <theme id="so" name="Social Office" />
</look-and-feel>
```

The code above shows registration of the **book-street-theme** theme with ID as **so** and name as **Social Office**.

## Constructing differences of the so-theme

We have built the theme project so-theme successfully. Now let's construct differences of the so-theme. We need to put custom CSS, images, JavaScript, and templates in the **/docroot/\_diffs** folder only.

First, in the **\_diffs/css** folder create a CSS file **custom.css**. We should place all of CSS that is different from the other files. Then in the **\_diffs/images** folder, put all the customized images with subfolders. In the **\_diffs/javascript** folder, create a JavaScript file **javascript.js**. Add the following lines in **javascript.js**:

```

var LiferayInc = function() {
 var $ = jQuery; return { init: function() {
 var instance = this; instance.handleMySitesDropDown();
 handleMySitesDropDown: function() {
 $('#navigation-top .my-sites').hoverIntent(
 { interval: 0, timeout: 500, over: function() {
 $(this).addClass('open'); $('.child-menu', $(this)).show();
 },
 out: function() {
 $(this).removeClass('open');
 $('.child-menu', $(this)).hide();
 }
 });
 jQuery(document).ready(function() {LiferayInc.init();});
 Liferay.Layout = null;
 }
}

```

The code above shows JavaScript functions: `init`, `jQuery(document).ready`, `handleMySitesDropDown`, and so on. Finally, in the `/_diffs/templates` folder create customized template files, for example `dock.vm`, `init_custom.vm`, `navigation.vm`, `portal_normal.vm`, `portal_pop_up.vm`, and `portlet.vm`.

That's it! You are ready to deploy this theme now. You could drop the `build.xml` file in the `$PLUGINS_SDK_HOME/themes/${theme.name}` folder into the Ant view. `${theme.name}` is an actual theme project name, for example, `so-theme`. Then just double-click on the `deploy` target under the theme of the Ant view. By the way, you can also check out the sample theme `so-theme` from `svn://svn.liferay.com/repos/public/plugins/trunk/themes/so-theme`.

## Adding mail and chat portlets

There are two basic requirements for My Social Office. First, we should be able to manage emails and send messages to any friends via email engine. Before using the mail engine, we should be able to configure the mail page with a specific mail server. Meanwhile, we should be able to know the number of online friends and chat with them at any time.

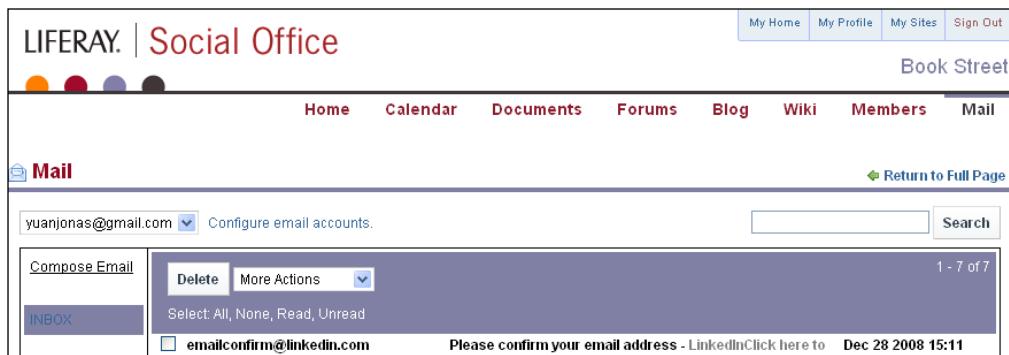
### Setting up the mail portlet

Liferay portal has unified the configuration of **JavaMail** so that it's the same for all application servers. It involves an SMTP server and an IMAP server. As shown in the following screenshot, a user in Book Street of My Social Office could manage his/her emails.



JavaMail API provides a platform-independent and protocol-independent framework to build mail and messaging applications. Refer to <http://java.sun.com/products/javamail/> for more details.

Mail portlet is also helpful and useful for other purposes. For instance, the Message Boards portlet has a feature that allows the users to receive and send posts as mails. Users can subscribe to the categories in the message board to receive by mail all the messages of that category and its subcategories. The mails corresponding to the same thread would be nested by the mail reader if it supports that functionality. Upon receiving an email, the user can answer directly from the mail reader to post a response in the same thread. It is also possible to create a new thread by saving the email address associated with a category and sending an email directly to it.



In order to manage emails in My Social Office, we're going to set up a mail portlet in Plugins SDK. As mentioned earlier, all portlets stay in the folder `$PLUGINS_SDK_HOME/portlets`, while `$PLUGINS_SDK_HOME` is the current directory of Plugins SDK home. The mail portlet is available as a plugin in Subversion. Simply check out the mail portlet from `svn://svn.liferay.com/repos/public/plugins/trunk/portlets/mail-portlet` to `$PLUGINS_SDK_HOME/portlets/mail-portlet`.

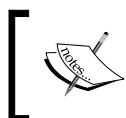
Then you just refresh the `$PLUGINS_SDK_HOME` project in Eclipse IDE, and you will see the `mail-portlet` folder in the `/portlets` folder. Under the `mail-portlet` folder, you will see the `/docroot` folder and the `build.xml` file. Particularly, you can find the mail portlet definition and registration in the `/docroot/WEB-INF` folder.

From now on, you can get the latest version of mail portlet by the `Update` command of Subversion. At the same time, you can also customize the mail portlet according to your own requirements.

You are ready to deploy this theme now. You could drop the `build.xml` file in the `$PLUGINS_SDK_HOME/portlets/${portlet.name}` folder into the Ant view. `${portlet.name}` is an actual portlet project name, for example `mail-portlet`. Then just double-click on the `deploy` target under `portlet` of the Ant view.

## Setting up the chat portlet

As stated above, we have added a mail portlet in My Social Office. Now, let's see how to add a chat portlet in it. As shown in the following screenshot, the chat portlet should have **Facebook** style. That is, users can see which of their friends are online and chat with them.



Facebook is a social utility that connects people with their friends and others who work, study, and live around them. Refer to <http://www.facebook.com> for more information.

In order to keep the chat feature in My Social Office, we need to set up a chat portlet in Plugins SDK.

## Deploying the chat portlet

The chat portlet is also available as a plugin in Subversion. Let's build the chat portlet. Similar to the mail portlet, just check out the chat portlet from `svn://svn.liferay.com/repos/public/plugins/trunk/portlets/chat-portlet` to `$PLUGINS_SDK_HOME/portlets/chat-portlet`.

Then, you can refresh the `$PLUGINS_SDK_HOME` project in Eclipse IDE. Further, you will see the `chat-portlet` folder (represented as  `${portlet.name}` ) in the `$PLUGINS_SDK_HOME/portlets/${portlet.name}/portlets` folder. Going further, under the `chat-portlet` folder, you will see the `/docroot` folder and the `build.xml` file. Especially, you can find the chat portlet definition and registration in the `/docroot/WEB-INF` folder.

From now on, you can get the latest version of the chat portlet by clicking on the `Update` command of Subversion. That's all. When you are ready, you can drop the `build.xml` file in the `$PLUGINS_SDK_HOME/portlets/${portlet.name}` folder into the Ant view. Then, just double-click on the `deploy` target under `portlet` of the Ant view.

## What's happening behind?

After deployment, you can see which friends are online and you can chat with them. For example, David Berger and Lotti Stein are friends. Both of them are logged in to My Social Office. David Berger sees Lotti Stein online and he wants to chat with her, or vice versa. Lotti Stein says **Hi David, How are you?** and David Berger answers **I am fine. And you?**.



You may ask: What's happening behind? In fact, two models are provided: `chat entry` and `chat status`. You can find the details of these models in the `com.liferay.chat.model` package under the `$PLUGINS_SDK_HOME/portlets/${portlet.name}/docroot/WEB-INF/src` folder.

The `chat status` model specifies the user's current status in My Social Office, for example user ID, online, awake, message, and so on. When a user is logged in, his/her friends will see his/her status via this model. The `chat entry` model represents an entry of chat in My Social Office, for example, entry ID, from user ID, to user ID, content, and so on. When a user sends a message to his/her friend, an entry will be created via this model.

## Building Social Office with portlets

We have discussed the Control Panel and IPC, which would be helpful to build My Social Office. We have also introduced the theme of My Social Office, which controls the overall look and feel of pages in My Social Office. Moreover, two portlets (for example, chat and mail) are also added into My Social Office. Now, let's build Social Office portlets in the Plugins SDK.

The following screenshot shows the user's home page—**My Home**. The left side of the user's home page lists the communities (called **sites** in this case) and a button to add sites if the user has an admin role. The right side of the page has three view activity options—**My sites**, **My Friends**, and **Me**. The **My sites** option lists all of the recent activities for the sites that the user belongs to, the **My friends** option lists the recent activities for user's friends, and the **Me** option lists all of the activities of the logged-in user. For the management purpose of sites, we can reuse the portlet communities. But for users' activities and their friends' activities, we're going to develop a portlet called Activities in my Social Office.

LIFERAY | Social Office

My Home | My Profile | My Sites | Sign Out

Palm Tree

Home Mail

Sites

Activities

My Sites | My Friends | Me

Today

Palm Tree wrote a new message board post in Book Street. 9:10 PM  
How about Liferay DEV book?  
Is it good?

As shown in the following screenshot, the **Members** page lists all the members of that particular community on the right, for example **Book Street**. You can click on a specific person to see more information, that is, **Profile**. Moreover, My Social Office should have the ability to let users send a request to join or to leave a site. For example, David Berger has joined the web site [bookpubstreet.com](http://bookpubstreet.com). He will have a choice to leave the web site. In other words, we're going to develop another three portlets in My Social Office: **members**, **profiles**, and **invite-members**.

LIFERAY | Social Office

My Home | My Profile | My Sites | Sign Out

Book Street

Home Calendar Documents Forums Blog Wiki Members Mail

Members

Palm Tree : Profile

Palm Tree CMS Admin admin@book.com

Contact Information

Email Address admin@book.com

Facebook yuanjonas@gmail.com

In short, we're going to provide general process—developing four portlets for My Social Office: **activities**, **members**, **profiles**, and **invite-members**. Of course, your requirements may be different. Fortunately, you can use the same process to develop your own portlets for your own Social Office.

## Rearing the Social Office portlets project

First of all, we're going to create a folder named `so-portlet` in the `$PLUGINS_SDK_HOME/portlets` folder in Plugins SDK. Everything for the use case *Social Office portlets* will go in this folder. Under it, create a `/docroot` subfolder and an XML file `build.xml`. You can copy the XML file `build.xml` from the `/ipc-faq-portlet` folder to the `/so-portlet` folder.

Then in the folder `/docroot`, create folders `/WEB-INF`, `/css`, `/images`, `/javascripts`, `/jsp`. The folder `/WEB-INF` contains portlet application specification files, subfolders `/lib`, `/tld`, and `/src`. Also the folder `/css` accommodates all CSS files, if applicable, using subfolders. Whereas all JavaScript files goes to the folder `/javascripts`, all JSP files go to the folder `/jsp`, if applicable, using subfolders; and all image files go to the folder `/images`.

In addition, when updating the source code in the `/docroot/WEB-INF/src` folder, you would see that related dependencies are included as well. To do this, first right-click on the Java project `$PLUGINS_SDK_HOME`, use **Build Path | Configure Build Path | Libraries** and add **User Library** `liferay-portal`. At the same time, add `portlet.jar`, `portlet-service.jar`, and `portlet-kernel.jar` from `$CATALINA_HOME/lib/ext` to `liferay-portal`. Sure, you will have different value in your own machine. Just use the real value for `$CATALINA_HOME`. Secondly, in the Navigator view, open the `.classpath` file and add a line: `<classpathentry kind="src" path="portlets/so-portlet/docroot/WEB-INF/src"/>` in the `.classpath` file and save it. From now on, you can compile the code with related dependencies immediately.

Optionally, you could run a script to create a blank portlet project. Similar to that of the portlet `ipc-faq-portlet`, you can create a blank portlet under the `portlets` folder.

## Assembling social portlets

We have created the portlet project `so-portlet` successfully. Now, let's build Social Office portlets Activities, Members, Profiles, and Invite-Members as well. First, we need to set up these four portlets in `portlet.xml`. Create an XML file `portlet.xml` in the `$PLUGINS_SDK_HOME/portlets/so-portlet/docroot/WEB-INF` folder. Open it and add the following lines at its beginning, and save it:

```
<!-- ignore details -->
<portlet-app>
 <portlet>
 <portlet-name>1</portlet-name>
 <display-name>Activities</display-name>
 <portlet-class>
 com.liferay.util.bridges.jsp.JSPPortlet
 </portlet-class>
 </portlet>
</portlet-app>
```

```
</portlet-class>
<init-param>
 <name>view-jsp</name>
 <value>/jsp/activities/view.jsp</value>
</init-param>
<!-- ignore details -->
</portlet>
<portlet>
 <portlet-name>2</portlet-name>
 <display-name>Invite Members</display-name>
 <portlet-class>
 com.liferay.util.bridges.jsp.JSPPortlet
 </portlet-class>
 <init-param>
 <name>view-jsp</name>
 <value>/jsp/invite_members/view.jsp</value>
 </init-param>
 <!-- ignore details -->
</portlet>
<portlet>
 <portlet-name>3</portlet-name>
 <display-name>Members</display-name>
 <portlet-class>
 com.liferay.util.bridges.jsp.JSPPortlet
 </portlet-class>
 <init-param>
 <name>view-jsp</name>
 <value>/jsp/members/view.jsp</value>
 </init-param>
 <!-- ignore details -->
</portlet>
<portlet>
 <portlet-name>4</portlet-name>
 <display-name>Profiles</display-name>
 <portlet-class>
 com.liferay.util.bridges.jsp.JSPPortlet
 </portlet-class>
 <init-param>
 <name>view-jsp</name>
 <value>/jsp/profiles/view.jsp</value>
 </init-param>
 <!-- ignore details -->
</portlet>
</portlet-app>
```

The preceding code shows the definition of four portlets: Activities, Invite Members, Members, and Profiles. These portlets are all JSP portlets, as the portlet-class is com.liferay.util.bridges.jsp.JSPPortlet. Moreover, init-param has a value for name view-jsp; /jsp/profiles/view.jsp for the Profiles portlet; /jsp/invite\_members/view.jsp for the Invite Members portlet; /jsp/members/view.jsp for the Members portlet; and /jsp/activities/view.jsp for the Activities portlet.

Afterwards, we need to register these portlets and create an XML file liferay-portlet.xml in the \$PLUGINS\_SDK\_HOME/portlets/so-portlet/docroot/WEB-INF folder. Open it and add the following lines at its beginning, and then save it:

```
<!-- ignore details -->
<liferay-portlet-app>
 <portlet>
 <portlet-name>1</portlet-name>
 <css-class-wrapper>so-portlet-activities</css-class-wrapper>
 </portlet>
 <portlet>
 <portlet-name>2</portlet-name>
 <css-class-wrapper>
 so-portlet-invite-members
 </css-class-wrapper>
 </portlet>
 <portlet>
 <portlet-name>3</portlet-name>
 <css-class-wrapper>so-portlet-members</css-class-wrapper>
 </portlet>
 <portlet>
 <portlet-name>4</portlet-name>
 <friendly-url-mapper-class>
 com.book.so.profiles.portlet.ProfilesFriendlyURLMapper
 </friendly-url-mapper-class>
 <header-portlet-javascript>
 /javascripts/javascript.js
 </header-portlet-javascript>
 <css-class-wrapper>so-portlet-profiles</css-class-wrapper>
 </portlet>
 <!-- ignore details -->
</liferay-portlet-app>
```

The code above shows the registration of portlets, for example 1 (activities), 2 (invite-member), 3 (members), and 4 (profiles) in the portal. It specifies the header portlet JavaScript for the portlet 4 (profiles) with the value /javascripts/javascript.js. Moreover, it specifies friendly-url-mapper-class for portlet 4 (profiles) with the value com.book.so.profiles.portlet.ProfilesFriendlyURLMapper.

In addition, we're going to put these portlets in the Social Office category. Create an XML file `liferay-display.xml` in the `$PLUGINS_SDK_HOME/portlets/so-portlet/docroot/WEB-INF` folder and open it. Add the following lines at its beginning and save it:

```
<!-- ignore details -->
<display>
 <category name="category.social.office"></category>
</display>
```

The code above shows a display category of Social Office, for example `category.social.office`.

## Raising JavaScript functions and friendly URL

As stated above, the portlet 4 (profiles) has header portlet JavaScript with a value `/javascripts/javascript.js` and friendly-url-mapper-class with a value `com.book.so.profiles.portlet.ProfilesFriendlyURLMapper`. In this section, we will raise these JavaScript function and friendly URL for profiles. First, we need to create JavaScript file `javascript.js` in the `/docroot/javascripts/` folder. Open it and add the following lines in it, and save it:

```
Liferay.SO = Liferay.SO || {};
Liferay.SO.Profiles = { init: function(params) {
 var instance = this; instance._assignEvents(); },
 displayUserProfile : function (userId) {
 jQuery.ajax({ url: themeDisplay.getLayoutURL() + '/-/profiles/
 user_profile', data: {userId: userId}, success: function(result) {
 jQuery('.profile-wrapper').html(result); },
 type: 'POST' }), _assignEvents: function() {
 var instance = this; jQuery('.so-portlet-members .user').click(
 function() { var userId = jQuery(this).attr('data-userId');
 instance.displayUserProfile(userId); });
 }
 }
 }
```

The code above shows the JavaScript functions: `init`, `displayUserProfile`, and `_assignEvents`. It uses jQuery JavaScript library, for example `jQuery.ajax` and `jQuery().attr`.

Then, we need to create a `com.book.so.profiles.portlet` package in the `/docroot/WEB-INF/src` folder. Moreover, create Java file `ProfilesFriendlyURLMapper.java` in the `com.book.so.profiles.portlet` package in the `/docroot/WEB-INF/src` folder. Open it, add the following lines in it, and save it:

```
the /docroot/WEB-INF/src folder. Open it, add the following lines in
it, and save it:
public class ProfilesFriendlyURLMapper extends BaseFriendlyURLMapper {
```

```
/* ignore details */
public String getMapping() { return _MAPPING; }
public String getPortletId() { return _PORTLET_ID; }
public void populateParams(String friendlyURLPath, Map<String,
 String[]> params) {
 int x = friendlyURLPath.indexOf("/", 1);
 int y = friendlyURLPath.indexOf("/", x + 1);
 if (y == -1) { y = friendlyURLPath.length(); }
 String jspPage = friendlyURLPath.substring(x + 1, y);
 if (Validator.isNull(jspPage)) { return; }
 addParam(params, "p_p_id", _PORTLET_ID);
 addParam(params, "p_p_lifecycle", "2");
 addParam(params, "p_p_state", WindowState.NORMAL);
 addParam(params, "p_p_mode", PortletMode.VIEW);
 addParam(params, "jspPage", "/jsp/profiles/" + jspPage + ".jsp");
}
private static final String _MAPPING = "profiles";
private static final String _PORTLET_ID = "4_WAR_soportlet";
}
```

The code above shows that `ProfilesFriendlyURLMapper` extends `BaseFriendlyURLMapper` with a `populateParams` method. You can find `BaseFriendlyURLMapper` in the `com.liferay.portal.kernel.portlet` package in the `/portal/portal-kernel/src` folder.

## Erecting social views

As mentioned in the `portlet.xml` file, `init-param` has a value for the name `view-jsp-/jsp/profiles/view.jsp` for the `Profiles` portlet, `/jsp/invite_members/view.jsp` for the `Invite Members` portlet, `/jsp/members/view.jsp` for the `Members` portlet, and `/jsp/activities/view.jsp` for the `Activities` portlet. In this section, we're going to build these views.

First, create a subfolder named `/activities` in the `/docroot/jsp` folder. All JSP files of the `activities` portlet go to this folder. Similarly, create subfolders named `/members`, `/invite_members`, and `/profiles` for portlets `Members`, `Invite Members`, and `Profiles`, respectively. Meanwhile, create JSP file `init.jsp` in `/docroot/jsp` and import models and services in JSP file `init.jsp` for the above four portlets. Then, create a JSP file `view.jsp` in the `/docroot/jsp/profiles/` folder and open it. Add the following lines at its beginning and save it:

```
<%@ include file="/jsp/init.jsp" %><div class="profile-wrapper"> <%@
include file="/jsp/profiles/user_profile.jsp" %>
</div>
```

The preceding code shows view of the Profiles portlet. It includes /jsp/init.jsp and /jsp/profiles/user\_profile.jsp. Accordingly, create a JSP file user\_profile.jsp in the /docroot/jsp/profiles/ folder and open it. Add the following lines at its beginning and save it:

```
<%@ include file="/jsp/init.jsp" %>
<% long userId = ParamUtil.getLong(request, "userId");
User curUser = null; /* ignore details */
%>
<c:choose>
<c:when test="<%= curUser != null %>">
/* ignore details */
</c:when>
<c:otherwise>
<h1>
<liferay-ui:message key="profile" />
</h1>
<div class="portlet-msg-error">
<c:choose>
<c:when test="<%= !themeDisplay.isSignedIn() %>">
<liferay-ui:message key="please-login-to-view-
user-profiles" />
</c:when>
<c:otherwise>
<liferay-ui:message key="this-site-has-
no-members" />
</c:otherwise>
</c:choose>
</div>
</c:otherwise>
</c:choose>
```

The code above shows the view of the Profiles folder. If the current user is not null then the portal will display this user's profile; otherwise, the portal will show the message `please-login-to-view-user-profiles` in case the current user is not signed in yet, or the portal will show the message `this-site-has-no-members` in case the current user is signed in already. By the way, we should add /jsp/invite\_members/view.jsp for the Invite Members portlet, /jsp/members/view.jsp for the Members portlet, and /jsp/activities/view.jsp for the Activities portlet. Note that /jsp/members/view.jsp uses jQuery JavaScript function Liferay.Scripts.init in the /docroot/javascripts/javascript.js folder.

Cool! You could deploy these portlets as well. Simply drop the file build.xml in the folder \$PLUGINS\_SDK\_HOME/portlets/so-portlet into the Ant view. Then just double-click on the deploy target under portlet of the Ant view. The social portlets activities, invite-member, members, and profiles will be compiled and deployed. Further, you could manage activities, members, and profiles, and invite members to join web sites. By the way, you can also check out the sample so-portlet portlet at svn://svn.liferay.com/repos/public/plugins/trunk/portlets/so-portlet.

## What's happening?

We have developed the social portlets: activities, invite-member, members, and profiles. Using these portlets, we could manage activities, members, and profiles, and invite members to join web sites. Let's see what's happening on these Social Office portlets.

## Experiencing social models

As shown in the following screenshot, three models are provided for My Social Office. They are socialrequest, socialactivity, and socialrelation. The socialrequest model represents users' request—whether they want to join a web site or leave it. It is made up of uuid\_, requestId, groupId, companyId, userId, classNameId, classPK, type\_, and so on.

The socialactivity model represents the users' activities. It consists of activityId, groupId, companyId, userId, classNameId, classPK, type\_, and so on. Moreover, the socialrelation model represents relationship among users. It is made up of uuid\_, relationId, companyId, userId1, userId2, type\_, and so on.

socialrequest		socialactivity		socialrelation	
uuid_	varchar(75)	activityId	bigint(20)	uuid_	varchar(75)
requestId	bigint(20)	groupId	bigint(20)	relationId	bigint(20)
groupId	bigint(20)	companyId	bigint(20)	companyId	bigint(20)
companyId	bigint(20)	userId	bigint(20)	createDate	datetime(19)
userId	bigint(20)	createDate	datetime(19)	userId1	bigint(20)
createDate	datetime(19)	mirrorActivityId	bigint(20)	userId2	bigint(20)
modifiedDate	datetime(19)	classNameId	bigint(20)	type_	int(11)
classNameId	bigint(20)	classPK	bigint(20)	extraData	longtext(2147483647)
classPK	bigint(20)	type_	int(11)	receiverUserId	bigint(20)
type_	int(11)	extraData	longtext(2147483647)		
receiverUserId	bigint(20)	receiverUserId	bigint(20)		
status	int(11)				

## Experiencing social services

Based on the above social models, the portal provides a set of social services, for example `SocialActivityLocalServiceUtil`, `SocialActivityInterpreterLocalServiceUtil`, `SocialRelationLocalServiceUtil`, and `SocialRequestLocalServiceUtil`. All these services are packaged as `portal-service.jar`. You can find detailed information at the `com.liferay.portlet.social.service` package in the `/portal/portal-service/src` folder.

The `SocialActivityLocalServiceUtil` service specifies a list of methods, for example `addActivity`, `deleteActivity`, `updateSocialActivity`, `getSocialActivities`, `getUserGroupsActivities`, `getRelationActivities`, `getUserActivities`, and so on. For example, to add a social activity, you have the following interface:

```
public static com.liferay.portlet.social.model.SocialActivity addActivity(long userId, long groupId,
 java.lang.String className, long classPK, int type,
 java.lang.String extraData, long receiverUserId)
```

The code above shows the parameters for the `addActivity` method. The `type` parameter identifies the type of activity; the `extraData` parameter is a string that can contain any additional info; `receiverUserId` is the user who has the activity. To delete social activities, you have the following interface:

```
public static void deleteActivities(long classNameId, long classPK)
```

The code above shows the parameters for the `deleteActivities` method. The `classNameId` parameter identifies class name ID, for example Wiki, blogs, message boards, calendar, and so on. The `classPK` parameter identifies the class primary key. Further, in `$PLUGINS_SDK_HOME/portlets/so-portlet/docroot/jsp/activities/view.jsp`, we have used these methods as follows:

```
have used these methods as follows:
/* ignore details */
List<SocialActivity> activities = null;
if (tabs1.equals("my-sites")) {
 activities = SocialActivityLocalServiceUtil.
 getUserGroupsActivities(user.getUserId(), 0,
 SearchContainer.DEFAULT_DELTA);
}
else if (tabs1.equals("my-friends")) {
 activities = SocialActivityLocalServiceUtil.
 getRelationActivities(user.getUserId(),
 SocialRelationConstants.TYPE_BI_FRIEND, 0,
 SearchContainer.DEFAULT_DELTA);
}
```

```
else {
 activities = SocialActivityLocalServiceUtil.
 getUserActivities(user.getUserId(), 0,
 SearchContainer.DEFAULT_DELTA);
<liferay-ui:social-activities activities="<% activities %>">
 feedEnabled="<% true %>"
 feedTitle='<%= LanguageUtil.format(pageContext, "subscribe-to-these-
 activities", user.getFirstName()) %>'
 feedLink="<% rssURL.toString() %>"
 feedLinkMessage='<%= LanguageUtil.format(pageContext, "subscribe-to-
 these-activities", user.getFirstName()) %>' />
/* ignore details */
feedEnabled="<% true %>" feedTitle='<%= LanguageUtil.
format(pageContext, "subscribe-to-these-activities", user.
getFirstName()) %>' feedLink="<% rssURL.toString() %>"
feedLinkMessage='<%= LanguageUtil.format(pageContext, "subscribe-to-
 these-activities", user.getFirstName()) %>' />
/* ignore details */
```

The code above shows the usage of the `SocialActivityLocalServiceUtil` service. Three methods of this service are in use: `getUserGroupsActivities`, `getRelationActivities`, and `getUserActivities`. Meanwhile, it shows activities via UI tag `liferay-ui:social-activities`. Of course, you can find this UI tag lib definition in `/portal/portal-web/docroot/WEB-INF/tld/liferay-ui.tld` and the page specification in `/portal/portal-web/docroot/html/taglib/ui/social_activities/page.jsp`.

The following is a sample code to get and display activities in the Activities portlet from `/portal/portal-web/docroot/html/taglib/ui/social_activities/page.jsp`:

```
/* ignore details */
List<SocialActivity> activities = (List<SocialActivity>)
 request.getAttribute("liferay-
 ui:social-activities:activities");
/* ignore details */
if (activities == null) {
 activities = SocialActivityLocalServiceUtil.getActivities(
 0, className, classPK, QueryUtil.ALL_POS,
 QueryUtil.ALL_POS);
}
/* ignore details */
for (SocialActivity activity : activities) {
 SocialActivityFeedEntry activityFeedEntry = SocialActivity
 InterpreterLocalServiceUtil.interpret(
 activity, themeDisplay);
/* ignore details */
```

The preceding code first shows a way to get activities via the `SocialActivityLocalServiceUtil` service. Then it uses the `SocialActivityInterpreterLocalServiceUtil` service to find the social activity feed entry.

## Adding social activity tracking

As you can see, activities include Wiki articles, blog entries, message board entries, calendar events, and so on. That is, the `Activities` portlet will keep tracking social activities of Wiki articles, blog entries, message board entries, calendar events, and so on. Undoubtedly, recorded social activities will appear on the `Activities` portlet. How does it work? How can you add social activity tracking? Let's have a deep look at these activities. Here we use the `Calendar` portlet as an example in order to show how to add social activity tracking.

First, we need to create an activity interpreter that extends `com.liferay.portlet.social.model.BaseSocialActivityInterpreter` and implements `com.liferay.portlet.social.model.SocialActivityInterpreter` in the `/portal/portal-service/src` folder. The activity interpreter class needs a `getClassName` method that returns an array of class names. This should include `className` used to call the `addActivity` method. We will also need a `doInterpret` method that returns `SocialActivityFeedEntry`. The code should parse the `SocialActivity` argument to create `SocialActivityFeedEntry`. In particular, we need a link, a title, and a body. For instance, `CalendarActivityInterpreter` in the `com.liferay.portlet.calendar.social` package in the `/portal/portal-impl/src` folder has the following implementation:

```
public class CalendarActivityInterpreter extends
BaseSocialActivityInterpreter {
 public String[] getClassNames() {
 return _CLASS_NAMES;
 }
 protected SocialActivityFeedEntry doInterpret(
 SocialActivity activity, ThemeDisplay themeDisplay)
 throws Exception {
 /* ignore details */
 CalEvent event = CalEventLocalServiceUtil.getEvent(
 activity.getClassPK());
 String link =
 themeDisplay.getURLPortal() +
 themeDisplay.getPathMain() +
 "/calendar/find_event?eventId=" +
 activity.getClassPK();
 /* ignore details */
 }
}
```

```
String title = themeDisplay.translate(titlePattern,
 titleArguments);
StringBuilder sb = new StringBuilder();
sb.append("<a href=\"");
sb.append(link);
sb.append("\">");
sb.append(cleanContent(event.getTitle()));
sb.append("
");
sb.append(cleanContent(event.getDescription()));
String body = sb.toString();
return new SocialActivityFeedEntry(link, title, body);
}
private static final String[] _CLASS_NAMES = new String[] {
 CalEvent.class.getName(),
};
}
```

The code above shows how to extend `BaseSocialActivityInterpreter` and how to build the `getclassNames` and `doInterpret` methods. Similarly, you can find implementation of Wiki articles, blogs entries and message board entries, as `com.liferay.portlet.wiki.social.WikiActivityInterpreter`, `com.liferay.portlet.blogs.social.BlogsActivityInterpreter`, and `com.liferay.portlet.messageboards.social.MBActivityInterpreter`, respectively. Finally, we can check the following lines in `/portal/portal-web/docroot/WEB-INF/liferay-portlet.xml` for the Calendar portlet:

```
<social-activity-interpreter-class>
 com.liferay.portlet.calendar.social.CalendarActivityInterpreter
</social-activity-interpreter-class>
```

The code above shows that the `social-activity-interpreter-class` tag has the value `com.liferay.portlet.calendar.social.CalendarActivityInterpreter` as an activity interpreter class. Similarly, you can specify the specific activity interpreter class for other portlets such as Wiki, blogs, and message boards. Further, you can follow the same processes to add social activity tracking on your custom portlets as well.

## Hooking properties and JSP files into Social Office

As mentioned above, we have developed the theme for Social Office, added a chat portlet and a mail portlet in Social Office, and built social portlets as well. In this section, we're going to customize the portal in order to build My Social Office.

First of all, a default user admin has to be predefined for My Social Office. That means that when My Social Office is starting, a default user account admin has been created smoothly. With this account, we could create other user accounts easily.

Besides a default user account, we need to predefine a list of pages, for example, Home, Calendar, Documents, Forums, Blogs, Wiki, and Members. Further, each page will be specified with predefined layout templates and portlets. For instance, the Documents page will have layout template `2_columns_iii` and the Document Library and Asset Publisher portlets. In `column_1` we add the Document Library portlet; and in `column_2` we add the Asset Publisher portlet. In short, all pages should be configurable.

One of the biggest aspects of implementing the portal is customization of the user experience by modifying portal JSP files generally. For example, we're going to customize the user experience of portlets, for example, Asset Publisher, Communities, Calendar, Document Library, Message Boards, Blogs, and Wiki. Of course, you can customize JSP in Ext. But the problem will arise while migrating from a lower version to a higher version. Hooks—a feature to seize properties and JSP files into an instance of the portal—would be the best solution for the above use cases. Here we will discuss a generic solution on how to hook the properties and custom JSP files into My Social Office. This solution will be generalized, and it should be helpful to build any other instances of the portal, for example, your own Social Office, your own special web sites, and so on.

## Building hooks

First of all, let's consider hooks configuration. Hooks allow us to hook into the event system, model listeners, JSP files and portal properties. We'll begin by building hooks in the following manner:

1. Create an XML file `liferay-hook.xml` in the `$PLUGINS_SDK_HOME/portlets/so-portlet/docroot/WEB-INF` folder and open it.
2. Add the following lines at its beginning and save it:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hook PUBLIC "-//Liferay//DTD Hook 5.2.0//EN" "http://
www.liferay.com/dtd/liferay-hook_5_2_0.dtd">
<hook>
 <portal-properties>portal.properties</portal-properties>
 <custom-jsp-dir>/META-INF/custom_jsp</custom-jsp-dir>
</hook>
```

The preceding code shows a hook configuration. It specifies portal-properties with a value portal.properties, and custom-jsp-dir with a value /META-INF/custom\_jsps. You can find the liferay-hook DTD in /portal/definitions/liferay-hook\_5\_2\_0.dtd. How does the hooks deployment work? You can check the details of hook hot-deploy com.liferay.portal.deploy.hot. HookHotDeployListener in the /portal/portal-impl/src folder. Here is a piece of code:

```
public class HookHotDeployListener extends BaseHotDeployListener
implements PropsKeys {
 public void invokeDeploy(HotDeployEvent event){ /* ignore details
 */
 protected void doInvokeDeploy(HotDeployEvent event) {/* ignore
 details */
 public void invokeUndeploy(HotDeployEvent event) {/* ignore details
 */
 protected void doInvokeUndeploy(HotDeployEvent event){ /* ignore
 details */
 protected void getCustomJspss(ServletContext servletContext, String
 webDir, String resourcePath,List<String> customJspss){/* ignore
 details */
 // ignore details
```

The code above shows hot-deploy of hooks. The HookHotDeployListener extends BaseHotDeployListener, and moreover, it implements PropsKeys. It also specifies the methods: invokeDeploy, doInvokeDeploy, invokeUndeploy, and doInvokeUndeploy.

## Applying portal event handlers

Then, let's consider how to create default user account. It means when an application is starting up, use portal event handler to create default user account. Create a package com.book.so.hook.events in the \$PLUGINS\_SDK\_HOME/portlets/so-portlet/docroot/WEB-INF/src folder. Then, you could create a Java file StartupAction.java in the com.book.so.hook.events package and add the following lines in it:

```
public class StartupAction extends SimpleAction {
 public void run(String[] ids) throws ActionException {
 try {
 doRun(GetterUtil.getLong(ids[0]));
 }
 catch (Exception e) { throw new ActionException(e); }
 }
 protected void doRun(long companyId) throws Exception {
 Group guestGroup = GroupLocalServiceUtil.getGroup(
```

```

 companyId, GroupConstants.GUEST) ;
Layout layout = LayoutLocalServiceUtil.getLayout(
 guestGroup.getGroupId(), false, 1);
LayoutTypePortlet layoutTypePortlet =
(LayoutTypePortlet)
 layout.getLayoutType();
if (!layoutTypePortlet.hasPortletId("47")) {return;}
 setupCompany(companyId); setupPermissions(companyId);
 setupLayouts(layout); setupUsers(companyId);
}
<!-- ignore details -->
}

```

The code above shows a customized start-action, `StartupAction`, which extends `com.liferay.portal.kernel.events.SimpleAction`. It overwrites the `run` method and sets up company, permissions, layouts, and default user as well. You also see portlet ID 47. This is the `Hello World` portlet that is used as a flag—detecting whether the start action has been taken into account or not.

## Putting model listeners to use

Finally, we need to set up the default layout pages. It means when application starts up, it uses model listener to set up default pages. Model listeners are similar in behavior to portal event handlers, except that they handle events with respect to models. To do so, create a `com.book.so.hook.listeners` package in the `$PLUGINS_SDK_HOME/portlets/so-portlet/docroot/WEB-INF/src` folder. Then create a Java file `LayoutSetListener.java` in the `com.book.so.hook.listeners` package. Add the following lines in it:

```

public class LayoutSetListener extends BaseModelListener {
 public void onAfterCreate(BaseModel model) throws
 ModelListenerException {
 try {
 LayoutSet layoutSet = (LayoutSet)model;
 if (layoutSet.isPrivateLayout()) {
 return;
 }
 Group group = GroupLocalServiceUtil.getGroup (
 layoutSet.getGroupId());
 if (group.isCommunity()) {
 addCommunityLayouts(group);
 updateFriendlyURL(group);
 }
 else if (group.isUser()) {
 addUserLayouts(group);
 }
 }
 }
}

```

```
 }
 catch (Exception e) {
 throw new ModelListenerException(e);
 }
 }
protected void addCommunityLayouts(Group group) throws Exception {
 updateLookAndFeel(group);
 Layout layout = addLayout(group, "Home", "/home", "1_column");
 removePortletBorder(layout, "1_WAR_soportlet");
 layout = addLayout(group, "Calendar", "/calendar", "1_column");
 removePortletBorder(layout, "8");
 layout = addLayout(group, "Documents", "/documents",
 "2_columns_iii");
 removePortletBorder(layout, "20");
 /* ignore details */
}
}
```

The code above shows that a model listener `LayoutSetListener` extends `com.liferay.portal.model.BaseModelListener`, which implements the `com.liferay.portal.model.ModelListener` interface. It overwrites the `onAfterCreate` method with adding community layouts, updating group friendly URL, and adding user layouts. Further, in the `addCommunityLayouts` method, it creates the `Home` page with a friendly URL `/home` and a layout template `1_column`. Similarly, it creates the `Calendar` page with a friendly URL `/calendar` and a layout template `1_column`. Especially, the `Documents` page is built with friendly URL `/documents` and `2_column_iii` layout template. Note that `1_WAR_soportlet` is the ID of the Activities portlet; `8` points to the ID of the Calendar portlet; while `20` refers to the ID of the Document Library portlet.

## Erecting portal properties

As mentioned above, each page was specified with a predefined layout template and friendly URL. Now we're going to use these features in portal properties and also add static portlets in the page. The portal properties file (for example, `portal.properties`) has been specified in `/docroot/WEB-INF/liferay-hook.xml`. That is, this portal properties file will override the existing portal properties in My Social Office. To do so, you just create a properties file `portal.properties` in the `$PLUGINS_SDK_HOME/portlets/so-portlet/docroot/WEB-INF/src` folder. Now add the following lines in it and save it:

```
ignore details
layout.static.portlets.start.column-1[user] [/home]=29
layout.static.portlets.start.column-2[user] [/home]=1_WAR_soportlet
layout.static.portlets.start.column-1[user] [/mail]=1_WAR_mailportlet
layout.static.portlets.start.column-1[community] [/home]=116
layout.static.portlets.start.column-1[community] [/calendar]=8
layout.static.portlets.start.column-1[community] [/documents]=20
layout.static.portlets.start.column-2[community] [/documents]=101_INSTANCE_abcd
layout.static.portlets.start.column-1[community] [/forums]=19
ignore details
application.startup.events=com.book.so.hook.events.StartupAction
com.liferay.portal.model.LayoutSet=com.book.so.hook.listeners.
LayoutSetListener
```

The code above shows how to set up layouts with static portlets, layout templates, and friendly URLs. It specifies the user's home page /home with the 29 My Communities portlet at the start of the iteration of the first column `column_1`, and the 1\_WAR\_soportlet Activities portlet at the start of the iteration of the second column `column_2`. More interestingly, it specifies the community's /documents page with the portlet 20 Document Library at the start of the iteration of the first column `column_1`, and the 101 Asset Publisher portlet at the start of the iteration of the second column `column_2`. As the 101 Asset Publisher portlet is instance-able, it adds the suffix `_INSTANCE_abcd` to the portlet ID, where `abcd` is any random alphanumeric string. Similarly, it specifies static portlets for layouts /forums, /blog, /wiki, and /members. Last but not the least, it configures the start-up event `application.startup.events` with a value `com.book.so.hook.events.StartupAction`, and model listener `com.liferay.portal.model.LayoutSet` with a value `com.book.so.hook.listeners.LayoutSetListener` as defined previously.

Suppose you want the portlet 28 Bookmarks to always appear at the end of the second column for user layouts, you can set the `layout.static.portlets.end.column-2[user]` property to 28. Especially, you can input a list of comma-delimited portlet IDs to specify more than one portlet.

The static portlets are fetched based on the properties controlled by custom filters. By default, the available filters are user, community, and organization. Note that static portlets cannot be moved and will always appear on every layout. Static portlets will take precedence over the portlets that may have been dynamically configured for the layout. On the whole, we can alter the portal configuration properties by specifying an override file. The properties will immediately take effect in My Social Office.

## Employing JSP hooks

We have already erected portal properties via hooks in My Social Office. Now let's use JSP hooks to customize the look and feel of existing portlets such as Asset Publisher, Communities, Calendar, Document Library, Message Boards, Blogs, and Wiki. As mentioned earlier, the custom JSP directory, for example /META-INF/custom\_jsp, has been specified in /docroot/WEB-INF/liferay-hook.xml. Let's see how to employ JSP hooks on the user experience of existing Calendar portlet as an example. There are two kinds of user experience which could be customized: portlet JSP file and portlet tag-lib UI.

First, create a folder named /custom\_jsp in the /docroot/META-INF/ folder. As specified in /docroot/WEB-INF/liferay-hook.xml, JSP hooks will start from this custom\_jsp folder. Then, create a folder named /html under the /docroot/META-INF/custom\_jsp folder. This folder will map to the folder /html in the /portal/portal-web/docroot folder.

For the portlet JSP file, create a folder /portlet under the /html folder. Further, create a subfolder /calendar under the /html/portlet folder. Note that, the /html/portlet/calendar folder must map to the /html/portlet/calendar folder under the /portal/portal-web/docroot folder exactly. In the /html/portlet/calendar folder, we customize the JSP files: edit\_event.jsp, init-ext.jsp, mini\_calendar.jsp, sidebar.jsp, tabs1.jsp, view\_event.jsp, and view.jsp. For instance, we can customize the edit\_event.jsp file as follows:

```
<%@ include file="/html/portlet/calendar/init.jsp" %>
<% String redirect = ParamUtil.getString(request, "redirect");
CalEvent event = (CalEvent)request.getAttribute(
 WebKeysCALENDAR_EVENT); %>
<liferay-util:include page="/html/portlet/calendar/sidebar.jsp" />
<liferay-util:include page="/html/portlet/calendar/
 view_event.portal.jsp" />
<c:if test="<%= CalEventPermission.contains(permissionChecker,
 event, ActionKeys.UPDATE) %>">

<portlet:renderURL windowState="<%=WindowState.MAXIMIZED.
 toString() %>">
 var="editURL">
<portlet:param name="struts_action"
 value="/calendar/edit_event" />
<portlet:param name="redirect" value="<%= redirect %>" />
<portlet:param name="eventId"
 value="<%= String.valueOf(event.getEventId()) %>" />
</portlet:renderURL>
```

---

```
<input type="button"
 value=<liferay-ui:message key="edit" />
 onclick="location.href = '<%= HtmlUtil.escape(editURL) %>';" />
</c:if>
```

The code above shows an option to customize the JSP file `edit_event.jsp`. For the use case portlet tag-lib UI, we need to create a folder named `/taglib` under the `/html` folder. Furthermore, create a subfolder named `/ui/calendar` under the `/html/taglib` folder. Note that the `/html/taglib/ui/calendar` folder must map to the `/html/taglib/ui/calendar` folder under the `/portal/portal-web/docroot` folder precisely.

The calendar portlet uses the `liferay-ui:calendar` tag to present summary view and year-based view. The default summary view has code as follows (abstracted from `/portal/portal-web/docroot/html/portlet/calendar/summary.jsp`):

```
<liferay-ui:calendar month=<%= selMonth %>
 day=<%= selDay %>
 year=<%= selYear %>
 headerFormat=<%= DateFormat.getDateInstance(
 DateFormat.LONG, locale) %>
 data=<%= data %> />
```

The code above shows usage of the `liferay-ui:calendar` tag. To have a custom user experience on the summary view and year-based view, we could update the JSP file `page.jsp` of the `liferay-ui:calendar` tag. That is, the only thing you need to do is to hook the JSP file `page.jsp` of the `liferay-ui:calendar` tag into My Social Office. Simply create a JSP file `page.jsp` in the `/html/taglib/ui/calendar` folder. Just add the following lines at the beginning of this file and save it:

```
<%@ include file="/html/taglib/init.jsp" %>
<!-- ignore details -->
<% int x = html.indexOf("<tr class=\"calendar-header\""); %>
 int y = html.indexOf("</tr>", x); %>
<%= html.substring(0, x) %>
<tr>
 <th valign="center">
 <a href="javascript: <%= namespace %>
 updateCalendar(<%= selMonth - 1 %>, <%= selDay %>,
 <%= selYear %>);">

 </th>
 <th valign="center" colspan="5">
```

```
<%= dateFormat.format(selCal.getTime()) %>
</th>
<th valign="center">
 <a href="javascript: <%= namespace %>
 updateCalendar(<%= selMonth + 1 %>,
 <%= selDay %>, <%= selYear %>);">

</th>
</tr>
<%= html.substring(y + 5) %>
```

The code above shows a way to customize the view of the `liferay-ui:calendar` tag. In the same way, you can customize the view of the Asset Publisher, Communities, Document Library, Message Boards, Blogs and Wiki portlets as well. As you can see, JSP hooks provide a way to easily modify JSP files without having to alter the core part of the portal. You could simply specify a folder in the hook plugin from which to obtain JSP files. The portal will automatically use these files in place of the existing ones in the portal. This works for any JSP files in the `/portal`, `/portlet`, and `/taglib` folders under the `/html` folder.

## Using hooks more efficiently

Hook plugins are more powerful plugins that come to complement portlets, themes, layout templates, and web modules. A hook plugin is always combined with a portlet plugin. For instance, the `so-portlet` project is a portlet plugin for Social Office with hooks.

## General usage

In general, there are three kinds of hook parameters: `portal-properties` (called **portal properties hooks**), `language-properties` (called **language properties hooks**), and `custom-jsp-dir` (called **JSP hooks**) as specified in `/portal/definitions/liferay-hook_5_2_0.dtd`.

```
<!ELEMENT hook (portal-properties?, language-properties*, custom-jsp-
dir?)>
```

As shown in the preceding code, the ordering of elements is significant in the DTD – you need to have your portal properties, language properties, and custom-JSPs declared in the same order. Language properties hooks allow us to install new translations or override few words in existing translations. JSP hooks provide a way to easily modify JSP files without having to alter the core of the portal, whereas portal properties hooks allow runtime re-configuration of the portal. The portal configuration properties can be altered by specifying an override file, where the properties will immediately take effect when deployed. Through portal properties hooks, we could change certain configuration properties dynamically and inject behavior into the hooks defined in the `portal.properties` file. All of the hooks that we have discussed above will revert, and their targeted functionality will be disabled immediately as soon as they are undeployed from the portal. Also, each type of hook can easily be disabled via the `portal.properties` file.

In any case, hooks can be built, packaged, and deployed, combined with portlets and using Plugins SDK. Not all of the portal properties can be overridden via hooks. The supported properties are `auth.forward.by.last.path`, `javascript.fast.load`, `layout.template.cache.enabled`, `layout.user.private.layouts.auto.create`, `layout.user.private.layouts.enabled`, `layout.user.private.layouts.modifiable`, `layout.user.public.layouts.enabled`, `layout.user.public.layouts.modifiable`, `my.places.show.community.private.sites.with.no.layouts`, `my.places.show.community.public.sites.with.no.layouts`, `my.places.show.organization.private.sites.with.no.layouts`, `my.places.show.organization.public.sites.with.no.layouts`, `my.places.show.user.private.sites.with.no.layouts`, `my.places.show.user.public.sites.with.no.layouts`, `terms.of.use.required`, `theme.css.fast.load`, `passwords.passwordpolicytoolkit.generator`, `passwords.passwordpolicytoolkit.static`, `layout.static.portlets.all`, `auto.login.hooks`, `application.startup.events`, `login.events.post`, `login.events.pre`, `logout.events.post`, `logout.events.pre`, `servlet.service.events.post`, `servlet.service.events.pre`.

Let's consider one more example for a hook plugin combined with a portlet plugin – **World of Liferay (WOL)**.

## **WOL—World of Liferay**

WOL provides social network in a single plugin. It showcases how to build a social network by leveraging social networking services. The following is a list of the portlets included in WOL:

- **Profile Summary** exhibits the picture and presentation of the user along with a summary of the participation of the user in the blogs and forums.
- **Activity Tracker** displays the activity of a user in any of the tools: Blogs, Message Boards, Wiki, Wall, JIRA, SVN (Code repository). In addition, the Activity tracker makes it easy to integrate third-party portlets or applications with minimal effort.
- **Friend List** provides the ability to invite friends and respond to friend invitations.
- **Facebook-style Wall** provides the ability for leaving messages to users
- **User Location Map** geo-locates the user based on his/her IP address and shows his/her location and that of his/her friends automatically.

WOL uses a set of models: `WOL_MeetupEntry`, `WOL_MeetupsRegistration`, `WOL_SVNRepository`, `WOL SVNRevision`, and `WOL_WallEntry`. How can you, as a developer, use WOL? Simply check out the WOL portlet from `svn://svn.liferay.com/repos/public/plugins/trunk/portlets/wol-portlet` to the `$PLUGINS_SDK_HOME/portlets/wol-portlet` folder; while `$PLUGINS_SDK_HOME` is current Plugins SDK home. That's it. When you refresh the `$PLUGINS_SDK_HOME` project in the Eclipse IDE, you will see the `wol-portlet` folder in the `/portlets` folder.

## **Special usage**

Besides the above stated general usage of hooks, there is a set of special usage, involving, document library hooks, auto-login hooks, and mail hooks.

## **Document library hooks**

Document library hooks allow us to connect to different repository servers. The available hooks are classes `FileSystemHook`, `JCRHook` and `S3Hook`:

```
com.liferay.documentlibrary.util.FileSystemHook,
com.liferay.documentlibrary.util.JCRHook,
com.liferay.documentlibrary.util.S3Hook.
```

The preceding code shows a set of document library hooks (that is, classes with package names). You can set the name of a class that implements `com.liferay.documentlibrary.util.Hook`, for example `dl.hook.impl=com.liferay.documentlibrary.util.FileSystemHook` in `/portal/portal-impl/src/portal.properties`. The document library server will use the file system to persist documents. `JCRHook` implements `com.liferay.documentlibrary.util.Hook` using an implementation of Content Repository API for Java (**JCR**), that is, Apache **Jackrabbit**. `JCR` is a specification for a Java platform API for accessing content repositories in a uniform manner; while Apache Jackrabbit is a fully-conforming implementation of `JCR` (both `JCR-170` and `JCR-283`; refer <http://jackrabbit.apache.org>). Similarly, `S3Hook` implements `com.liferay.documentlibrary.util.Hook` using Amazon **S3**. Amazon S3 is storage for the Internet (<http://aws.amazon.com/s3>).

## Auto-login hooks

Auto-login hooks allow us to easily configure the portal working with the other SSO servers. You can find the following mapping of `auto.login.hooks` in `/portal/portal-impl/src/portal.properties`:

```
auto.login.hooks=com.liferay.portal.security.auth.CASAutoLogin,
com.liferay.portal.security.auth.NtlmAutoLogin,
com.liferay.portal.security.auth.OpenIdAutoLogin,
com.liferay.portal.security.auth.OpenSSOAutoLogin,
com.liferay.portal.security.auth.RememberMeAutoLogin,
com.liferay.portal.security.auth.SiteMinderAutoLogin
```

The code above shows a list of comma-delimited class names that implement `com.liferay.portal.security.auth.AutoLogin`. These classes will run in consecutive order for all unauthenticated users until one of them returns a valid user ID and password combination. These classes include `CASAutoLogin`, `NTLMAutoLogin`, `OpenIDAutoLogin`, `OpenSSOAutoLogin`, `RememberMeAutoLogin`, and `SiteMinderAutoLogin` among many others. `RememberMeAutoLogin` reads from a cookie to automatically log in a user who previously logged in, while checking on the **Remember Me** box. Computer Associate's (CA) **SiteMinder** is a centralized web access management. Refer to <http://www.ca.com> for more details.

## Mail hooks

As mentioned earlier, Liferay portal has unified the configuration of JavaMail so that it's the same for all application servers. Mail hooks allow us to connect to the different mail servers. The available hooks are: `com.liferay.mail.util.CyrusHook`, `com.liferay.mail.util.DummyHook`, `com.liferay.mail.util.FuseMailHook`, `com.liferay.mail.util.SendmailHook`, and `com.liferay.mail.util.ShellHook`.

You can set the name of a class that implements `com.liferay.mail.util.Hook` for the `mail.hook.impl` key, for example, `mail.hook.impl=com.liferay.mail.util.DummyHook` in `/portal/portal-impl/src/portal.properties`. The mail server will use this class (for example, `DummyHook`), to ensure that the mail and portal servers are synchronized on user information. By the way, the portal will not know how to add, update, or delete users from the mail server except through this hook.

## Summary

This chapter focused on how to build My Social Office in general. First, it introduced Control Panel—how it works and how to customize it. Then it addressed IPC—how to build IPC portlets. Later it discussed how set up Social Office theme and portlets, including IPC capabilities. It also discussed how to hook language properties and portal properties—including model listener and event handlers, and custom JSP files into Social Office. Finally, it discussed an efficient way to use hooking features.

In the next chapter we're going to discuss staging and publishing features.

# 11

## Staging and Publishing

It would be nice if we could stage, schedule, and publish the web content in our web site `bookpub.com` remotely. As a content creator, you may update what you've created and publish it in a staging. Then other users could review and modify it. Moreover, content editors could make a decision whether to publish web content from staging to live. In short, it would be nice if users could easily create and manage everything from a simple article of text and images to fully functional web sites in staging and then publish them live.

Before going live, you may schedule web content as well. For instance, you may publish web content immediately or schedule it for publishing on a specific date. Further, you may schedule a specific portlet for specific web content. For publishing features, you might choose either a local publishing or a remote publishing; you may publish either the entire web site or just a subset of all pages. Liferay portal provides the capability to not only stage web content of the web site and to publish web content either locally or remotely, but also provides a flexible framework to make customization and extension of staging and publishing easy to use.

This chapter will discuss a simple extension—how to build dynamic navigation and how to construct a customized site map. Then, it will address how to handle events and model listeners. Based on these features, it will introduce local staging and publishing, and staging workflow. A way to schedule pages and assets will be discussed, too. Finally, it will address how to publish web content remotely.

By the end of this chapter, you will have learned how to:

- Build dynamic navigation and site map
- Customize event handlers and model listeners
- Experience local staging and publishing
- Employ staging workflow and other workflows
- Schedule pages and assets for publishing
- Experience remote staging and publishing

## Building dynamic navigation and site map

Liferay portal provides two portlets related to dynamic navigation: the Breadcrumb portlet and the Navigation portlet. The Breadcrumb portlet displays a trail of parent pages for the current page. It can be placed on the public portal pages as a navigational aid to publish web sites, which will help the user to visualize the structure of the site and quickly move from a page to a broader grouping of information. Meanwhile, the Navigation portlet provides a directory of links to reflect the portal's page structure with a drill down into the current page. The style and appearance are adjustable. The navigation portlet displays links for other pages outside of the current page's trail of parent pages, helping the user to visualize the structure of the site and provide links to quickly move from page to page. Moreover, it displays more information about the current page.

These two portlets are pretty useful for most generic cases. But for specific requirements, we have to customize them. As shown in the following screenshot, the first level navigation menu is made up of kid-friendly icons on the top, for example **Home**, **Games**, **Videos**, **Play List**, **Muppets**, **My Street** and normal image buttons close to the bottom, for example **PARENTS**, **ON AIR**, and **ABOUT US** plus global search button as well. The kid-friendly buttons will be built in the theme. Here we're going to build these normal image buttons for navigations.



Moreover, the second-level navigation menu also has a different view. For example, the first-level navigation layout (such as **GAMES**) has sublevel pages such as **Browse All**, **By Subject**, **By Theme**, **By Character**, **ART** and the searching games button **GO**. All second-level navigation plus their parent page (that is, **GAMES**) share the same color, for example dodger blue.

But for the **VIDEOS** menu bar, the look and feel is different although it shares the same mechanism. As shown in the following screenshot, the first-level navigation layout, for example **VIDEOS** has sublevel pages, for example **Browse All**, **By Subject**, **By Theme**, **By Character**, **Songs**, **Classic Clips** and the searching videos button **GO**. All the second-level navigation plus their parent page, that is **VIDEOS**, share the same color, for example dark orange.



A similar thing will happen on the **PLAYLIST** menu bar and its sub-navigations. Moreover, the menu bars **ON AIR** and **PARENTS** will share the same framework with different look and feel, as shown in the following screenshot. All main menu bars and submenu bars in this group share the same color, for example forest blue.



As you can see, there is only one portlet with a different look and feel (on the top), which depends on where the portlet resides. So how can we implement Ext Navigation and Street Navigation (normal image buttons close to the bottom)? Let's customize the breadcrumb portlet and the navigation portlet.

## Constructing custom navigation and street navigation

First of all, let's build a portlet called Ext Navigation. To do so, you need to specify and register a portlet in the `/ext/ext-web/docroot/WEB-INF` folder. To do so, add the following lines before the line `</portlet-app>` in the `portlet-ext.xml` file:

```
<portlet>
 <portlet-name>extNavigation</portlet-name>
 <display-name>Ext Naviagtion</display-name>
 <!-- ignore details -->
</portlet>
```

To register the `extNavigation` portlet, add the following lines after the line `<!-- Custom Portlets -->` in the `liferay-portlet-ext.xml` file:

```
<portlet>
 <portlet-name>extNavigation</portlet-name>
 <struts-path>ext/navigation</struts-path>
 <use-default-template>false</use-default-template>
 <restore-current-view>false</restore-current-view>
</portlet>
```

To put the `extNavigation` portlet in the Book category, add the following line after the line `<portlet id="myStreet" />` in the `liferay-display.xml` file:

```
<portlet id="extNavigation" />
```

Then add the following line for page flow after the line `<action-mappings>` in the `struts-config.xml` file:

```
<action path="/ext/navigation/view"
 forward="portlet.ext.navigation.view" />
```

Add the following lines for page layout after the line `<tiles-definitions>` in the `tiles-defs.xml` file:

```
<definition name="portlet.ext.navigation.view" extends="portlet">
 <put name="portlet_content"
 value="/portlet/ext/ navigation /view.jsp" />
</definition>
```

Finally, add the following line for title mapping before the line `javax.portlet.title.EXT_1=Reports` in the `Language-ext.properties` file:

```
javax.portlet.title.extNavigation=Ext Navigation.
```

You can use the above process to build the Street Navigation portlet. We especially need to provide entries mapping at the end of the `portal-ext.properties` file with the following lines:

```
theme.css.name.parent=parents
theme.css.name.air=onair
theme.css.name.about=aboutus
theme.css.name.bandwidth=lowbandwidhttxt
street.theme.css.names=home,game,video,playlist,muppet,street
street.nav.css.names=parent,air,about,bandwidth
theme.gamessubmenu.css=browse-all,by-subject,by-theme,by-
character,art
theme.videossubmenu.css=browse-all,by-subject,by-theme,by-
character,songs,classic-clips
theme.playlistssubmenu.css=browse-all,by-subject,by-theme,by-
character
theme.parentssubmenu.css=topics,newslatters,how-to
theme.on_airsubmenu.css=episode-guide,celebrity-guests,cast,characte
rs,history
```

The code above shows the entries of navigation, for example name mappings and CSS mappings. Note that you may have different menu bars—just reconfigure them.

## Build portlets' views

The views of the Ext Navigation portlet are made up of two JSP files: `init.jsp` and `view.jsp`. Thus, we can customize the look and feel of the portlet via these JSP files. First of all, let's create a JSP file `init.jsp` for the Ext Navigation portlet in the following manner:

1. Create a folder named `navigation` page in the `/ext/ext-web/docroot/html/portlet/ext` folder.
2. Create a JSP file `init.jsp` in the `/ext/ext-web/docroot/html/portlet/ext/navigation` folder and open it.
3. Add the following lines in the `init.jsp` file and save it:

```
<!-- ignore details -->
<% String[] cssNames = null;
 String cNames = PropsUtil.get("street.nav.css.names");
 if(cNames != null || !cNames.equals(""))
 cssNames = cNames.split(",");
%>
```

The code above shows a way to get properties entries. Similarly, you will create a folder called `streetNavigation` and a file JSP called `init.jsp` for the Street Navigation portlet in the `/ext/ext-web/docroot/html/portlet/ext/streetNavigation` folder.

Let's create a JSP file `view.jsp` for the Ext Navigation portlet in the following manner:

1. Create a JSP file `view.jsp` in the `/ext/ext-web/docroot/html/portlet/ext/navigation` folder and open it.
2. Add the following lines in this file and save it:

```
<%@ include file="/html/portlet/ext/navigation/init.jsp" %>
<div id="searchbar" class="searchbar">

 <% if (layout != null) { /* ignore details */ }
 for (int i = 0; i < navItems.size(); i++) {
 NavItem navItem = (NavItem) navItems.get(i);
 <!-- ignore details -->
 for(int j = 0; j < cssNames.length; j++){
 if((pageName.equals(cssNames[j])) ||
 (pageName.indexOf(cssNames[j]) != -1)) {
 if(cssNames[j].equals("bandwidth") &&
 ((layoutName.indexOf("game") != -1) ||
 (layoutName.indexOf("home") != -1))){
 %> <li class="<%="searchbar-"
 +
```

```
PropsUtil.get("theme.css.name." +cssNames[j]) %>">
 Low Bandwidth User?
 Click here

<%
}
else /* ignore details */
}
if(cssNames[j].indexOf("bandwidth") == -1) {
%>

<a href=<%= navItem.getURL() %>" id=<%="searchbar-" + PropsUtil.get(
 "theme.css.name." +cssNames[j]) +
 navItemClass %>" class="search-button">
bookpub
<%
}
<!-- ignore details -->
```

The code above first shows a way to retrieve the navigation menu bars. Then it picks up a different kind of CSS to represent navigations. Similarly, you can create a `view.jsp` file for the Street Navigation portlet in the `/ext/ext-web/docroot/html/portlet/ext/streetNavigation` folder. Just add the following lines in it:

```
/* ignore details */
if(layoutName.equalsIgnoreCase("games") || layoutName.toLowerCase().indexOf("games") !=-1) {
 sessionName = "games"; searchName = "Games";
}
/* ignore details */
<div id="submenu" class="<%= sessionName %>">
<!-- ignore details -->
```

The code above shows layout names, for example, Games, Videos, Play-List, Parents and On Air pages. Then it uses a different kind of CSS to represent a different group navigations.

## Establishing custom site map

The site map portlet provides an ability to display a structured directory of links to all the pages in the portal. It is used to navigate directly to any page on the site. Further, it can be configured to display the entire site or a subsection of pages.

As shown in the following screenshot, we're going to customize the **Site Map** portlet with a different view. More specifically, first, set up a box around the title and different page levels with the green color, then show all pages in three columns, and finally, display the different page levels in different colors.

The screenshot shows a Site Map portlet with a header titled "Site Map". Below the header, there are three main sections arranged horizontally:

- GAMES** (left column):
  - [Browse All Games](#)
  - [By Subject](#)
  - [By Theme](#)
  - [By Character](#)
  - [Art](#)
- On Air** (middle column):
  - [Episode Guide](#)
  - [Celebrity landing page](#)
  - [Cast landing page](#)
  - [Characters](#)
  - [History of Sesame Street](#)
- Parents** (right column):
  - [Parents Topics Page](#)
  - [Coping with Stressful Events](#)
  - [Comforting Children in a Disaster](#)
  - [Easing the Transition to Daycare or P...](#)
  - [Letting Go Is A Life Skill](#)
  - [Understanding How Preschoolers Handle...](#)
  - [Understanding How Six- to Eight-Year-...](#)
  - [CreativityImagination](#)

We keep the original Site Map portlet as it is while building another site map portlet called Street Site Map with a customized look and feel.

## Constructing the street site map portlet

Just like the Ext Navigation portlet, you can construct a portlet in Ext in the following manner:

1. Specify the Street Site Map portlet in the `portlet-ext.xml` file.
2. Register this portlet in the `liferay-portlet-ext.xml` file.
3. Put it in the Book group in the `liferay-display.xml` file.
4. Add the page flow in the `struts-config.xml` file.
5. Add the page layout in the `tiles-defs.xml` file.
6. Finally, add title mapping in the `Language-ext.properties` file.

## Building up portlet view

Later we need to customize the look and feel for site map. First let's create the JSP file `init.jsp` in the following manner:

1. Create a folder named `streetSitemap` in the `/ext/ext-web/docroot/html/portlet/ext` folder.
2. Create a JSP file `init.jsp` in the `/ext/ext-web/docroot/html/portlet/ext/streetSitemap` folder and open it.

3. Add the following lines in this file and save it:

```
<!-- ignore details -->
<% PortletPreferences prefs = renderRequest.getPreferences();
 String portletResource = ParamUtil.getString(request,
 "portletResource");
 // ignore details
 long rootPlid = GetterUtil.getLong(prefs.getValue("root-plid",
 StringPool.BLANK));
 int displayDepth = 0;
%>
```

As shown in the code above, `init.jsp` gets the portlet preference and resource. It also resets the root page and display depth. Now, let's create the JSP file `view.jsp` in the following manner:

1. Create a JSP file `view.jsp` under the `ext/ext-web/docroot/html/portlet/ext/streetSitemap` folder and open it.
2. Add the following lines in `view.jsp` and save it:

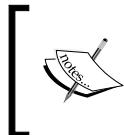
```
<%@ include file="/html/portlet/ext/streetSitemap/init.jsp" %>
<% List rootLayouts = null;
 if (rootPlid > 0) {
 Layout rootLayout = LayoutLocalServiceUtil.getLayout(
 rootPlid);
 rootLayouts = rootLayout.getChildren();
 }
 else {
 rootLayouts = LayoutLocalServiceUtil.getLayouts(
 layout.getGroupId(), layout.isPrivateLayout(),
 LayoutConstants.DEFAULT_PARENT_LAYOUT_ID);
 }
 StringBuilder sm = new StringBuilder();
 sm.append(
 "<table style='width:100%'>
 <tr>
 <td style='width:25%' valign=\"top\">
 <table >");
 _buildSiteMap(rootLayouts, displayDepth, 1,
 themeDisplay, sm);
%> <div class="box outline-green">
 <div class="bi">
 <div class="bt">
 <div></div>
 </div>
 <div class="grid-row">
 <div class="grid-col4 margin-top-20">
 <h1>Site Map</h1>
```

```

 <%= sm.toString() %>
 </div>
 </div>
</table>
</td>
</tr>
</table>
<div class="bb">
 <div></div>
</div></div>
</div>
<!-- ignore -->

```

The code above shows the root, that is, the community name Book Street first. Then it tracks the first-level layout pages, for example Home, Games, Videos, Playlists, and so on. For each layout page, it tracks its children pages. In addition, a set of CSS is applied as well.



Note that you can find the original Site Map portlet as reference at the /portal/portal-web/docroot/WEB-INF folder for portlet definition, and in the /portal/portal-web/docroot/html/portlet/site\_map folder for JSP pages.

## Customizing event handlers and model listeners

Liferay portal is highly configurable and extensible. Its default configuration settings are defined in the /portal/portal-impl/src/portal.properties source tree. It is read-only and must not be edited. Fortunately, Ext provides an ability to override the values in the portal.properties file. Custom values are normally placed in /ext/ext-impl/src/portal-ext.properties. In this section, we're going to look in detail at the customization of event handlers and model listeners via the portal-ext.properties file.

### Handling events

Generally speaking, the portal supports event listeners to hook into the system. Event listening is configured in the portal.properties file and it should be re-configured in the portal-ext.properties file by configuring event handlers and/or model listeners.

## Configuring global startup and shutdown actions

The portal supports a mechanism for code to hook into the portal during startup and/or shutdown. The default global startup and/or shutdown actions are originally configured in the `portal.properties` file and these events could be overridden via the `portal-ext.properties` file later. The `portal.properties` file defines key-value pairs with `global.startup.events` and `global.shutdown.events`. The value for each key is a comma-delimited list of class names as follows:

```
global.startup.events=com.liferay.portal.events.GlobalStartupAction
global.shutdown.events=com.liferay.portal.events.GlobalShutdownAction
```

The code above shows that the global startup event runs when the portal initializes, whereas the global shutdown event runs when the portal shuts down. Let's configure the global startup action in the following manner:

1. Create a package named `com.ext.portal.action` in the `/ext/ext-impl/src` folder.
2. Create a file named `GlobalStartupAction.java`, which extends `SimpleAction` in this package and open it.
3. Add the following lines at the beginning of this file and save it:

```
public class GlobalStartupAction extends SimpleAction {
 public void run(String[] ids) throws ActionException {
 _log.debug("Inside global startup action!");
 }
 private static Log _log = LogFactory.getLog(
 GlobalStartupAction.class);
}
```

Similarly, you can override the global shutdown action in the following manner:

1. Create a file named `GlobalShutdownAction.java` which extends `SimpleAction` inside the `com.ext.portal.action` package in the `/ext/ext-impl/src` folder and open it.
2. Add the following lines in `GlobalShutdownAction.java` and save it:

```
public class GlobalShutdownAction extends SimpleAction {
 public void run(String[] ids) throws ActionException {
 if (_log.isDebugEnabled()) {
 _log.debug("Inside global shutdown action!");
 }
 }
 private static Log _log = LogFactory.getLog(
 GlobalShutdownAction.class);
}
```

The preceding code shows a global startup and shutdown action. We need to update the custom values by adding the following lines at the end of the `portal-ext.properties` file. Normally, you just add a comma at the end of the property and put classes.

```
global.startup.events=com.liferay.portal.events.GlobalStartupAction,
 com.ext.portal.action.GlobalStartupAction
global.shutdown.events=com.liferay.portal.events.GlobalShutdownAction
 ,com.ext.portal.action.GlobalShutdownAction
```

As shown in the code above, the classes will be called in order. Finally, let's test these configurations using the following steps:

1. Run Ant targets `clean` and `deploy` in the Ant view `ext`.
2. Start the portal and add the Admin portlet to a page.
3. Click on **Server | Log Levels**.
4. Change the log level for `com.ext` to `DEBUG` and click on **Save**.
5. Stop the portal and restart it.
6. After stopping and then restarting, you would see a debug message in the console similar to the following:

```
22:11:10,901 DEBUG [GlobalStartupAction:17] Inside global startup
action!
22:31:13,911 DEBUG [GlobalShutdownAction:17] Inside global
shutdown action!
```

In the same way, you can override the events `servlet.session.create`, `events, servlet.session.destroy.events, application.startup.events, application.shutdown.events, servlet.service.events.pre, servlet.service.events.post`, and so on.

## Creating a custom cookie on login

Liferay portal provides a pluggable mechanism whereby you can register classes that get fired on pre-login and post-login. The default pre-login and post-login actions are configured in `portal.properties` and can be overridden via `portal-ext.properties`. The `portal.properties` file defines a key-value pair with the keys `login.events.pre` and `login.events.post`. The value for each key is a comma-delimited list of class names as follows:

```
login.events.pre=com.liferay.portal.events.LoginPreAction
login.events.post=com.liferay.portal.events.LoginPostAction,
 com.liferay.portal.events.DefaultLandingPageAction
```

Let's create a cookie with login info using the following steps:

1. Create a file named `CreateCookieAction.java` which extends `com.liferay.portal.kernel.events.Action` in the `com.ext.portal.action` package of the `/ext/ext-impl/src` folder, and open it.
2. Add the following lines at the beginning of this file and save it:

```
public class CreateCookieAction extends Action {
 public void run(HttpServletRequest req, HttpServletResponse res)
 throws ActionException {
 try {
 long companyId = PortalUtil.getCompanyId(req);
 long userId = PortalUtil.getUserId(req);
 String domain = PropsUtil.get(
 PropsKeys.SESSION_COOKIE_DOMAIN);
 String cookieValue = "companyId=" + companyId +
 ",userId=" + userId; Cookie cookie = new Cookie(
 "MY_COOKIE", cookieValue);
 if (Validator.isNotNull(domain)) {
 cookie.setDomain(domain);
 }
 cookie.setPath(StringPool.SLASH);
 CookieKeys.addCookie(res, cookie);
 if (_log.isDebugEnabled()) {
 _log.debug("Added MY_COOKIE value to response:" +
 cookieValue);
 }
 }
 catch (Exception e) {
 throw new ActionException(e);
 }
 }
 private static Log _log = LogFactory.getLog(
 CreateCookieAction.class);
}
```

Now we need to override the custom values in the `portal-ext.properties` file, add a comma at the end of the property, and put the class as follows:

```
login.events.post=com.liferay.portal.events.LoginPostAction,com.
liferay.portal.events.DefaultLandingPageAction,com.ext.portal.action.
CreateCookieAction
```

After restarting, you will see a debug message in the console similar to the following:

```
22:12:56,250 DEBUG [CreateCookieAction:42] Added MY_COOKIE value to
response: companyId=10110,userId=11301
```

In the same way, you can set it as a pre-login action, too. Moreover, you may set it as pre-logout and post-logout. That is, you can register classes that get fired on pre-logout and post-logout. In short, various events can have one or more handler classes listed. If a class is listed, it is called by the event management system. Some of the key players are as follows:

- `com.liferay.portal.events.EventsProcessor`—the dispatcher
- `com.liferay.portal.servlet.MainServlet`—calling `EventsProcessor` for most events
- `com.liferay.portal.servlet.PortalSessionListener`—calling `EventsProcessor` for session creation and destruction events
- `com.liferay.portal.action.LogoutAction / LoginAction`—calling `EventsProcessor` when user logs out/logs in
- `com.liferay.portal.action.LayoutAction`—calling `EventsProcessor` when layout creation and destruction
- `com.liferay.portal.util.InitUtil`—calling `EventsProcessor` during initialization

## Building custom model listeners

We have discussed how to handle events. Now let's consider model listeners, which are similar to those of My Social Office. Using model listeners, the default user admin has to be predefined. That is, when My Social Office is starting, a default user account admin has been created smoothly. With this account, we could create other user accounts easily. Besides the default user account, we need to predefined a list of pages, for example Home, Calendar, Documents, Forums, Blogs, Wiki, and Members. It means when the portal starts up, it uses model listener to set up default pages.

What is a model listener? What is the motivation behind it? Model listener is an event listener that has been defined in a model class. The event being listened to can be `onCreate`, `onDelete`, and so on. This model listener could be used, for example, for audit trail implementation. In this section, we will show how to use model listeners in order to add community layouts, to update group friendly URL, and to add user layouts, and so on.

## Creating custom model listener

To create a custom model listener, just create a package named `com.ext.portal.model` in the `/ext/ext-impl/src` folder. Then create a Java file named `LayoutSetListener.java` in the `com.ext.portal.model` package and add the following lines in it:

```
public class LayoutSetListener extends BaseModelListener {
 public void onAfterCreate(BaseModel model)
 throws ModelListenerException {
 try {
 LayoutSet layoutSet = (LayoutSet)model;
 if (layoutSet.isPrivateLayout()) { return; }
 Group group = GroupLocalServiceUtil.getGroup(
 layoutSet.getGroupId());
 if (group.isCommunity()) {
 addCommunityLayouts(group);
 updateFriendlyURL(group);
 }
 else if (group.isUser()) {
 addUserLayouts(group);
 }
 }
 catch (Exception e) {
 throw new ModelListenerException(e);
 }
 }
}
```

The code above shows that the `LayoutSetListener` model listener extends `com.liferay.portal.model.BaseModelListener`, which implements the `com.liferay.portal.model.ModelListener` interface. It overwrites the `onAfterCreate` method by adding community layouts, updating group friendly URL, and adding user layouts.

Then, register this model listener in `portal-ext.properties` as follows;

```
value.object.listener.com.liferay.portal.model.LayoutSet=com.liferay.
portal.model.LayoutSetListener,com.ext.portal.model.LayoutSetListener
```

When the portal is started, it will create default user admin. Besides the default user account, it will create a list of pages, for example Home, Calendar, Documents, Forums, Blogs, Wiki, and Members. Each page will be specified with predefined layout templates and portlets.

## What's happening?

By default, the portal has set up value objects in `portal.properties` as follows:

```
value.object.listener.com.liferay.portal.model.Layout=com.liferay.
 portal.model.LayoutListener
value.object.listener.com.liferay.portal.model.LayoutSet=com.liferay.
 portal.model.LayoutSetListener
```

The code above shows a list of models mapped to model listeners. You can add a listener for a specific class by setting the `value.object.listener` property with a list of comma-delimited class names that extend the `com.liferay.portal.model.BaseModelListener` class, which implements the `com.liferay.portal.model.ModelListener` interface. For instance, the `com.liferay.portal.model.LayoutSet` model is mapped to the `com.liferay.portal.model.LayoutSetListener` model listener.

The `LayoutSetListener` model listener is specified in the `/portal/portal-impl/src` folder as follows:

```
public class LayoutSetListener extends BaseModelListener {
 public void onAfterRemove(BaseModel model) {
 clearCache(model);
 }
 public void onAfterUpdate(BaseModel model) {
 clearCache(model);
 }
 protected void clearCache(BaseModel model) {
 LayoutSet layoutSet = (LayoutSet) model;
 if (!layoutSet.isPrivateLayout()) {
 CacheUtil.clearCache(layoutSet.getCompanyId());
 }
 }
}
```

The code above shows that `LayoutSetListener` extends `BaseModelListener`. It overrides the `onAfterRemove` and `onAfterUpdate` methods in order to clear the cache as well.

The `BaseModelListener` base class implements the `ModelListener` interface. Both of them are specified in the `/portal/portal-service/src` folder. The operations specified in `ModelListener` include `onAfterAddAssociation`, `onAfterCreate`, `onAfterRemove`, `onAfterRemoveAssociation`, `onAfterUpdate`, `onBeforeAddAssociation`, `onBeforeCreate`, `onBeforeRemove`, `onBeforeRemoveAssociation`, and `onBeforeUpdate`. That is, the model listeners (for example, `UserListener`, `ContactListener`, `LayoutListener`, `LayoutSetListener`, `PortletPreferenceListener`, `JournalArticleListener`, and `JournalTemplateListener`) extend the base class `com.liferay.portal.model.BaseModelListener` in the `/portal/portal-impl/src` folder.

In short, events can be notified when most valued objects (that is, database entities) are created, read, updated, deleted, and so on. You can define model listeners for the CRUD operations on users, layouts, organizations, locations, web sites, and so on. You can first define a class that extends the `BaseModelListener` base class, which implements the `ModelListener` interface. Then you can make an entry in the `portal-ext.properties` file in order to register.

## **Undergoing local staging and publishing**

In the web sites of the enterprise "Palm Tree Publications", it will be nice if users can stage their work – the ability to work on a working copy of the web site. For example, as a content creator, you can manipulate this working copy and preview it as if it were the web site. You should be able to preview a working copy at any time without disrupting the live pages. The purpose of the staging feature is to deploy a new version of the web site in a fully functional form, which can be tested and reviewed by content producers or content editors. Content producers or content editors, who are evaluating the web content changes, are able to navigate the site without having to choose which version to see.

Similarly, it would be nice if the users could publish web content smoothly – to push one or more assets from a staging to a live (or called production) environment. Generally speaking, publishing should include the capability to publish to both the local portal instances and the remote portal instances. From the functional point of view, publishing should be as simple as a push of a button or it should be included as a step in a workflow. Most importantly, publishing should not disrupt the production environment except the published change. Liferay portal provides the ability to stage and publish web content. In this section we're going to discuss local staging and publishing features.

## Activating staging

First of all, let's activate staging on a community, for example, Book Street. Note that the staging is activated in an organization/community level only. Before taking a deep look at staging, we use the Book Street community as an example. Here are steps to create a community named Book Street:

1. Log in as an admin, for example, Palm Tree.
2. Go to **My Community** and **Private Pages** where you can find the **Communities** portlet.
3. Click on the **Add Community** button, enter the values for the **Name** as **Book Street**, and the **Description** as **Book Street Website**. Select **Type** as **Open** and select the checkbox **Active**.
4. Click on the **Save** button.

You will see that the **Book Street** community has been created. Click on **Actions** which is next to the **Book Street** community and click on the **Manage Pages** icon. As shown in the following screenshot, you will see **Edit Pages for Community: Book Street**. Here you can manage **Public Pages** and **Private Pages**, and set **Staging**, **Virtual Host**, **Sitemap**, **Monitoring**, **Logo**, and **Merge Pages** as well.



To activate staging, just select the **Activate Staging** checkbox. You will receive a message when the staging environment for the Book Street community has been created properly. To deactivate staging, just deselect the **Activate Staging** checkbox. You will receive a message when the staging environment for the Book Street community has been removed properly.

## What's happening?

What happens when you select the **Activate Staging** checkbox? What happens when you deselect the checkbox **Activate Staging**? Let's have a deep look at this function. First, the entry point to activate or deactivate staging is specified in the Java file `com.liferay.portlet.communities.action.EditPagesAction.java` in the `/portal/portal-impl/src` folder. The following is a piece of code:

```
public void processAction(ActionMapping mapping, ActionForm form,
 PortletConfig portletConfig, ActionRequest actionRequest,
 ActionResponse actionResponse)
 throws Exception {
 String cmd = ParamUtil.getString(actionRequest, Constants.CMD);
 else if (cmd.equals("staging")) {
 StagingUtil.updateStaging(actionRequest);
 }
}
```

The code above shows an implementation of the `processAction` method. When the `cmd` command is equal to the `staging` value, it updates the staging environment by either creating a new one or deleting an old one. You can find more details of updating the staging environment in the Java file `com.liferay.portlet.communities.util.StagingUtil.java`. The following is a piece of code:

```
public static void updateStaging(ActionRequest actionRequest)
 throws Exception {
// ignore details
 if ((stagingGroupId > 0) && !stagingEnabled) {
 GroupServiceUtil.deleteGroup(stagingGroupId);
 GroupServiceUtil.updateWorkflow(liveGroupId, false, 0, null);
 }
 else if ((stagingGroupId == 0) && stagingEnabled) {
 Group liveGroup = GroupServiceUtil.getGroup(liveGroupId);
 Group stagingGroup = GroupServiceUtil.addGroup(
 liveGroup.getGroupId(), liveGroup.getName(),
 +" (Staging)", liveGroup.getDescription(),
 GroupConstants.TYPE_COMMUNITY_PRIVATE,
 null, liveGroup.isActive());
 if (liveGroup.hasPrivateLayouts()) {
 Map<String, String[]> parameterMap = getStagingParameters();
 publishLayouts(liveGroup.getGroupId(),
 stagingGroup.getGroupId(), true, parameterMap, null, null);
 }
 if (liveGroup.hasPublicLayouts()) {
 Map<String, String[]> parameterMap = getStagingParameters();
 publishLayouts(liveGroup.getGroupId(),
 stagingGroup.getGroupId(), false, parameterMap, null, null);
 }
 }
}
```

The preceding code shows a way to activate or deactivate the staging environment. It first gets a staging group id `stagingGroupId` and a flag `stagingEnabled`. If the current group is a staging group and has a value `false`, it removes the current staging group. Otherwise, it creates a group for staging environment. As you can see, the name of staging group is the name of the current community name and `(Staging)`. The live group ID of staging group is the current community group ID. For instance, the current community group has the name `Book Street` with a group ID as `10401`. Thus, the staging group has the name `Book Street (Staging)` with a live group ID as `10401` and a group ID as `10501`. Note that the group ID value is showed as an example. During runtime, these values will be different. At the same time, all of the layouts (either private pages or public pages) have been populated from the current `Book Street` community to the `Book Street (Staging)` staging group.

## How does it work?

There are three models associated with the staging environment. These models are `Group`, `Layout`, and `LayoutSet`. Let's see how it works. We could benefit a lot from customization and extension of the staging feature if we know the models and their mechanisms very well.

First, we're going to have a deep insight into the `Group` model. The `Group` model (`com.liferay.portal.model.Group`) represents group records, including normal community, private user community (that is, my community), organizations, and user groups. For instance, you will see the `className` and `classPK` columns in the `Group` model:

- If `className` and `classPK` are blank, then the record is a Community
- If `className` is `com.liferay.portal.model.User`, then the record represents a private user community
- If `className` is `com.liferay.portal.model.Organization`, then the record represents an organization
- If `className` is `com.liferay.portal.model.UserGroup`, then the record represents a user group

Why do we have `Group` to represent organizations and user groups? The reason is that we can have one entry in this table for each entity in the system that represents a set of users. This model simplifies the relationships between the other entities (for example, permissions or roles) with these sets of users.

The community (a special group where `className` and `classPK` are blank) is assigned one of these types: `open`, `restricted`, or `private`. You can find details of these types in the Java file `com.liferay.portal.model.GroupConstants.java` in the `/portal/portal-service/src` folder.

Considering the preceding scenario, when the Book Street community was created, a group with a name Book Street, a type 1 (this is type open), and a group ID 10401 was created in the Group model. Note that the live group ID is 0 for the Book Street community. It means that the Book Street group is a live group, and not a staging group. Once the staging group is activated, a group with a name Book Street (String), a type 3 (this is the private type) and a group ID 10501 is created in the Group model. Note that, the live group ID is 10401. It means that the Book Street (Staging) group is a staging group, and not a live group. In a word, each group whose live group ID is not 0 represents a staging group.

Especially, the `typeSettings` column of the Group model is used for type settings, for example workflow settings. We will discuss this feature in detail in the coming section.

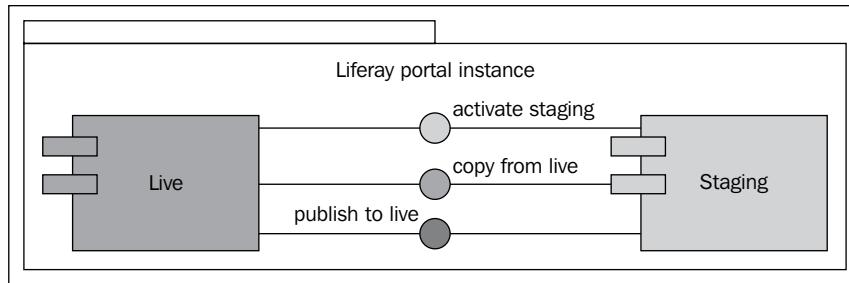
Then, the Layout model represents an instance of a single page composed of one or more portlets arranged inside various columns. When all layouts (either private pages or public pages) have been populated from the current Book Street community to the Book Street (Staging) staging group, the current group ID 10501 of the Book Street (Staging) staging group is added in the layouts. Once the Book Street (Staging) staging group has been removed, all layouts related to the group ID 10501 will be removed. A layout is assigned one of types: `portlet`, `panel`, `control_panel`, `embedded`, `article`, `url`, and `link_to_layout`. You can find details of types in the Java file `com.liferay.portal.model.LayoutConstants.java` in the `/portal/portal-service/src` folder.

In addition, the LayoutSet model layout set represents a group of layout pages. Each layout (`com.liferay.portal.model.Layout`) is a member of layout set (`com.liferay.portal.model.LayoutSet`). The layout set has group ID associated. If the group was a live group, for example Book Street, then the layout set represented a collection of layout pages of live group. If the group was a staging group, for example Book Street (Staging), then the layout set represented a collection of layout pages of the staging group. Similarly, the private pages and public pages are differentiated as well. Further, the additional information is also stored in the LayoutSet model, for example CSS, logo, theme, count of page, virtual host, and so on.

## **Staging and publishing locally**

We have discussed how to activate staging in a portal instance, that is, the live group and the staging group exist in the same portal instance. In other words, we have activated the staging environment for local web content staging and publishing. For example, the Book Street live group and the Book Street (Staging) staging group exist in the same portal instance. From now on, we can manage the staging group as well as the live group. As shown in the following figure, we can do the following tasks:

- *copy from live*—copy pages from the live group to the staging group
- *publish to live*—publish selected pages or an entire web site from the staging group to the live group



## Copying from live

Once the staging group is created, you can update the staging group any time. This feature is called *copy from live*. That is, copy all pages from the live group to the staging. For instance, the Book Street (Staging) staging group was created based on the Book Street live group. Then we just work on the Book Street (Staging) staging group. In the middle of updating the staging group, we may need to roll back to the live group. In such a case, we can use the *copy from live* feature—copying all pages from the live group to the staging group. Thus, we can make the Book Street (Staging) staging group synchronized with the Book Street live group.

How does it work? Let's take an in-depth look at this *copy from live* feature. The entry point to copy from live is specified in the Java file `EditPagesAction.java` in the `com.liferay.portlet.communities.action` package. The following is a piece of code:

```
public void processAction(ActionMapping mapping, ActionForm form,
 PortletConfig portletConfig, ActionRequest actionRequest,
 ActionResponse actionResponse)
 throws Exception {
 String cmd = ParamUtil.getString(actionRequest, Constants.CMD);
 else if (cmd.equals("copy_from_live")) {
 StagingUtil.copyFromLive(actionRequest);
 }
}
```

The preceding code shows an implementation of the `processAction` method. When the `cmd` command is equal to the `copy_from_live` value, it will copy all pages from the live group to the staging group. You can find more details of copying from live in the Java file `StagingUtil.java`. The following is a piece of code:

```
public static void copyFromLive(ActionRequest actionRequest)
throws Exception {
 long stagingGroupId = ParamUtil.getLong(actionRequest,
 "stagingGroupId");
 Group stagingGroup = GroupLocalServiceUtil.getGroup(
 stagingGroupId);
 long liveGroupId = stagingGroup.getLiveGroupId();
 Map<String, String[]> parameterMap = getStagingParameters();
 _publishLayouts(actionRequest, liveGroupId, stagingGroupId,
 parameterMap, false);
}
```

The code above shows a way to copy from live. It first gets the staging group ID `stagingGroupId` and the live group ID `liveGroupId`. Then it copies all pages from the live group to the staging group. Note that whatever we have in the staging group will be overridden. The pages of the staging group will be synchronized with those of the live group once the `copy from live` feature has been applied.

## Publishing to live

Once the staging group updates are ready, you can publish all the pages of the staging to the live group. This feature is called `publish to live`. That is, copy all pages from the staging group to the live group. For instance, the Book Street (Staging) staging group is ready and we want to apply all changes of the staging group to the Book Street live group. In this case, we can use the `publish to live` feature—copying all pages from the staging group to the live group. Thus, we can make the Book Street live group synchronized with the Book Street (Staging) staging group. How to implement the `publish to live` feature? The entry point to `publish to live` is specified in the Java file `EditPagesAction.java`. The following is a piece of code:

```
public void processAction(ActionMapping mapping, ActionForm form,
 PortletConfig portletConfig, ActionRequest actionRequest,
 ActionResponse actionResponse)
throws Exception {
 String cmd = ParamUtil.getString(actionRequest, Constants.CMD);
 else if (cmd.equals("publish_to_live")) {
 StagingUtil.publishToLive(actionRequest);
 }
}
```

The preceding code shows an implementation of the `processAction` method. When the `cmd` command has the `publish_to_live` value, it will publish all pages from the staging group to the live group. You may be interested in more details of publishing to live. Of course, you can find details in the Java file `StagingUtil.java` in the `com.liferay.portlet.communities.util` package. The following is a piece of code:

```
public static void publishToLive(ActionRequest actionRequest)
throws Exception {
 long stagingGroupId = ParamUtil.getLong(actionRequest,
 "stagingGroupId");
 Group stagingGroup = GroupLocalServiceUtil.getGroup(
 stagingGroupId);
 long liveGroupId = stagingGroup.getLiveGroupId();
 Map<String, String[]> parameterMap = getStagingParameters(
 actionRequest);
 _publishLayouts(actionRequest, stagingGroupId, liveGroupId,
 parameterMap, false);
}
```

The code above shows a way to publish from a staging to a live. It first gets the staging group ID `stagingGroupId` and the live group ID `liveGroupId`. Then it publishes selected pages from the staging group to the live group. Note that whatever we have in the live group will be overridden. The selected pages of the live group will be synchronized with those of the staging group once the `publish to live` feature has been applied.

In short, the approach of staging and publishing web content locally would be a good idea when the web site is small and the loading traffic is a minor concern—the processes of content management and publishing can share the same portal instance. This feature, called **local staging and publishing**, would be useful for intranets. As a content creator, you can manipulate a working copy and preview it as if it was the web site to work on a working copy at any time without disrupting the live pages.

But when the web site is huge and the loading traffic is a major concern, the processes of content management and publishing should be separated. Thus, we have to use the remote staging and publishing feature. That means, a staging box and, at least, a live box exist in different portal instances although in different domains or locations. This feature is called **remote staging and publishing**, which is discussed in the following section.

## Employing staging workflow and other workflows

In a staging environment, users can work on the working copy by a staging workflow. For instance, a **proposal** applies to any web content (for example pages) that enters the workflow; whereas a **review** exists when a **proposal** is assigned to a reviewer. In the staging environment, we discussed that an actor, **Staging Manager**, has the power to create or break a staging environment. This actor is either the community owner, super user, or someone who has been granted the **MANAGE\_STAGING** permission.

In the staging workflow, we can specify other actors: **content creator**, **content producer**, **content reviewer**, and **content editor**. Note that these four roles are not mandatory – one can only have two or three roles instead of four. A content creator represents anyone having the **MANAGE\_LAYOUTS** permission. A content creator can create content and edit pages in which he/she can then propose a change into the workflow. He/she can edit the web content of a proposal and add comments on a proposal.

Content producer can reject a proposal, assign reviewers to the proposal, and delete a proposal outright. Moreover, just like the content creator, he/she can edit the content of a proposal, and add comments to a proposal. A content reviewer is a knowledge expert and represents the QA element in the workflow. Any number of different reviewers can be assigned to a given proposal. The reviewers cannot edit the content of the proposal but they can reject or approve a proposal, or add comments to a proposal. A content editor participates in the final act of a proposal. He/she can publish a proposal from staging to live, reject a proposal, edit the content of a proposal, or add comment on a proposal.

This is called a **role-based staging workflow**. In the following section, we will discuss how to use staging workflow and how to customize it for remote staging and publishing.

## Activating staging workflow

As mentioned earlier, to activate staging, just select the checkbox **Activate Staging**. A function to activate workflow becomes available later. That is, the default workflow is only applied for the staging environment. By default, the roles for the workflow are community roles, and not organization or regular roles.

For demonstration purposes, we're going to create four community roles: content creator, content producer, content reviewer, and content editor. Suppose that we have four user accounts: Lotti Stein, Rolf Hans, Julia Maurer, and David Berger. Create a role content creator by **Roles** in the Control Panel and add Lotti Stein in this role. Similarly create a role content producer and add Rolf Hans in this role; create a role content reviewer and add Julia Maurer in this role; and finally, create a role content editor and add David Berger in this role.

To activate staging workflow, just select the checkbox **Activate Workflow**. After activating the workflow, you have the ability to set up the workflow. Of course, to disable the staging workflow, you just have to deselect the checkbox **Activate Workflow**.

As an admin, you can choose **Number of Stages** and the stage role for each stage in order to set up a workflow. Considering the above scenario, we use **Number of Stages** as 4. The default value of **Stage 1 Role** is **content creator**. Choose the value of **Stage 2 Role** as **content producer**, the value of **Stage 3 Role** as **content reviewer**, and the value of **Stage 4 Role** as **content editor**.

## Creating a proposal

We have activated staging workflow. Now let's experience it. As a content creator, log in as Lotti Stein first. Go to the Book Street community, and then select **Staging | View Staged Page**. You will be in the **Home** page. Go further to update the **Home** page by adding a **Reports** portlet in the **Home** page, for instance. Then select **Staging | Proposals Publication** and enter a name for proposal, for example **How about this page and web content**. Now click on the **Save** button.

You have successfully submitted a proposal in workflow as a content creator. As shown in the following screenshot, you can go further to view the proposal, including name, type, ID, user, due date, status, and the actions with a set of icons. To do so, just select **Staging | View Proposals**:

Name	Type	ID	User	Due Date	Status	Edit	Delete	Actions
Home How about this page and web content	Page	10504	Lotti Stein	Never	Stage 2 Pending Review			

Now you will see that the **Status** of the workflow is **Stage 2 Pending Review**. That is, it is waiting for a content producer to review, edit, and preview the web content, and moreover, add comments, and reject or approve the proposal. If you log in as Rolf Hans, you can edit the web content, add comments on the proposal, and approve the proposal. Suppose you approved the proposal. Thus, you will see that the **Status** of the workflow is **Stage 3 Pending Review**.

Similarly, a content reviewer such as Julia Maurer can preview the page, add comments, and reject or approve the proposal. Suppose Julia Maurer approved the proposal. Finally, you will see that the **Status** of the workflow is **Stage 4 Pending Review**. That is, the proposal is waiting for a content producer to review, edit, and preview the web content, and moreover, add comments, reject or approve it, and publish the web content from staging environment to the live environment. If you log in as David Berger, you can edit the web content, add comments on the proposal, and approve the proposal. Suppose you approved the proposal, and further, you published the updates to live. So, you can see that the **Home** page in the live environment has been updated with the Reports portlet.

## What's happening?

Let's analyze the staging workflow mechanism. First, we will focus on the mechanism on workflow activation and deactivation. The entry point to activate/deactivate a staging workflow is specified in the Java file `EditPagesAction.java`. The following is a piece of code:

```
public void processAction(ActionMapping mapping, ActionForm form,
 PortletConfig portletConfig, ActionRequest actionRequest,
 ActionResponse actionResponse)
 throws Exception {
 String cmd = ParamUtil.getString(actionRequest, Constants.CMD);
 else if (cmd.equals("workflow")) {
 updateWorkflow(actionRequest);
 }
}
```

The code above shows an implementation of the `processAction` method. When the `cmd` command is equal to the `workflow` value, it will activate or deactivate a staging workflow for the current staging environment. Of course, you can find details in the Java file `EditPagesAction.java`. The following is a piece of code:

```
protected void updateWorkflow(ActionRequest actionRequest)
 throws Exception {
 // ignore details
 GroupServiceUtil.updateWorkflow(liveGroupId, workflowEnabled,
 workflowStages, workflowRoleNames);}
```

The preceding code shows how to update the workflow settings. It first gets a live group ID `liveGroupId`, a flag `workflowEnabled`, and a number of workflow stages `workflowStages`. Then it forms the workflow role names `workflowRoleNames`. Note that the above information is stored in the `typeSettings` column of the Group model. For instance, considering the above scenario, the value for `typeSettings` of the Group model would be `workflowEnabled=true workflowStages=3 workflowRoleNames=Content Producer, Content Reviewer, Content Editor`.

Of course, you can disable the staging workflow. Once a staging workflow is disabled, the `workflowEnabled` flag is set to the `false` value. Considering the above scenario again, the value for `typeSettings` of the Group model would be `workflowEnabled=false workflowStages=3 workflowRoleNames=Content Producer, Content Reviewer, Content Editor`.

Then, we're going to focus on the staging workflow itself—how to approve/reject a proposal, how to edit the content, and how to add comments on a proposal. The entry point of staging workflow was specified in `/portal-web/docroot/WEB-INF/struts-config.xml`. The following is a piece of code:

```
<action path="/enterprise_admin_communities/edit_proposal"
 type="com.liferay.portlet.communities.action.
 EditProposalAction">
 <forward name="portlet.communities.edit_proposal"
 path="portlet.communities.edit_proposal" />
 <forward name="portlet.communities.error"
 path="portlet.communities.error" />
</action>
<action path="/layout_management/edit_proposal"
 type="com.liferay.portlet.communities.action.
 EditProposalAction">
 <forward name="portlet.communities.edit_proposal"
 path="portlet.communities.edit_proposal" />
 <forward name="portlet.communities.error"
 path="portlet.communities.error" />
</action>
<action path="/communities/edit_proposal"
 type="com.liferay.portlet.communities.action.
 EditProposalAction">
 <forward name="portlet.communities.edit_proposal"
 path="portlet.communities.edit_proposal" />
 <forward name="portlet.communities.error"
 path="portlet.communities.error" />
</action>
```

The preceding code shows an entry point of staging workflow. Whenever editing a proposal, the `com.liferay.portlet.communities.action.EditProposalAction` portlet action is involved. You can find `EditProposalAction.java` in the `/portal/portal-impl/src` folder for adding or updating a proposal, approving review, deleting a proposal, rejecting a review, and publishing a proposal. The following is piece of code:

```
public void processAction(ActionMapping mapping, ActionForm form,
 PortletConfig portletConfig, ActionRequest actionRequest,
 ActionResponse actionResponse)
throws Exception { String cmd = ParamUtil.getString(actionRequest,
 Constants.CMD);
try {
 if (cmd.equals(Constants.ADD) || cmd.equals(Constants.UPDATE)) {
 updateProposal(actionRequest, actionResponse);
 }
 else if (cmd.equals(Constants.APPROVE)) {
 approveReview(actionRequest);
 }
 else if (cmd.equals(Constants.DELETE)) {
 deleteProposal(actionRequest);
 }
 else if (cmd.equals(Constants.PUBLISH)) {
 publishProposal(actionRequest);
 }
 else if (cmd.equals(Constants.REJECT)) {
 rejectReview(actionRequest);
 }
}
}
```

The code above shows a way to add or update a proposal, to approve a proposal, to delete a proposal, to reject a review, and publish a proposal. It specifies an implementation of the `processAction` method. When the `cmd` command is equal to the `UPDATE` or `ADD` value, it will update a proposal; if the `cmd` command is equal to the `APPROVE` or `REJECT` value, it will approve or reject a review; if the `cmd` command is equal to the `DELETE` value, it will delete a proposal; if the `cmd` command is equal to the `PUBLISH` value, it will publish a proposal—copy the selected page or all of the pages that the proposal is associated with from the staging environment to the live environment.

Now, let's take a deep look at models. You will see that two models are specified: TasksProposal and TasksReview. The TasksProposal model represents a proposal with proposalId, groupId, companyId, userId, and userName, creation date and modification date, class name ID and class primary key, name description, publish date and due date. Note that the group ID is the current live group ID. For example, the group ID of the proposal **How about this page and web content** is the live group ID, for example 10401 of Book Street. But the proposal is only being activated and is available under the Book Street (Staging) staging group in front controls. Thus, it is available for us to apply for proposals based on the current live group. That is, when one box has one portal instance, we can use this box as a staging box without activating an additional staging group. For a content creator and a content producer, it is a live box where he/she can use proposals on the live group, for example Book Street. But for the production box, what the end users or public guests are visiting is their live box. Thus, what the content creator and the content producer are using can be called a staging box.

More interestingly, the code above is using classNameID and classPK to represent target objects. If the class name has a value com.liferay.portal.model.Layout, it means that the proposal is based on the layout pages and if the class name has a value com.liferay.portlet.journal.model.JournalArticle, it means that the proposal is based on the web content (that is, journal articles). Following the same pattern, this TasksProposal model can be used to represent a workflow based on any kind of models such as com.liferay.portlet.bookmarks.model.BookmarksEntry, com.liferay.portlet.polls.model.PollsQuestion, com.liferay.portlet.imagegallery.model.IGImage, com.liferay.portlet.documentlibrary.model.DLFileEntry, and so on. That is, you can apply for a workflow based on bookmarks, polls, images and documents, and so on. This is a pretty cool model for staging workflow. Thus, you can customize it according to your own requirements—either current or future.

The TasksReview model represents any information related to the reviewing processes. It has a set of attributes associated: reviewed, groupId, companyId, userId, and userName, created date, modified date, proposalId, assigned user, stage, flags complete, or rejected. Same as that of the TasksProposal model, the group ID is the current live group ID. For example, the group ID of the TasksReview model is the live group ID, for example, 10401 of Book Street.

## Customizing staging workflow

As you can see, the **How about this page and web content** proposal has been removed at the end of the staging workflow. That is, there is no history data of layout pages and staging workflow proposals. At the same time, there is only simple comment to players, for example, content creator. That is, there is only one version of comments. Multiple comments with specific date information are not supported yet. Further, the workflow is used for the staging environment by default. The workflow is applied for layout pages only. Here we're going to extend staging workflow to keep a history data of layout pages in order to maintain multiple comments to directly apply the workflow on the live group.

## Extending model

Considering the above requirements, we need to maintain the layout pages history. When a layout page (for example **Home**) is published, the previous page must be saved with its version. Besides this, we need to keep the complete workflow as history data. That is, when a proposal has been published at the end of workflow, save the current proposal as history data. Furthermore, we need to maintain multiple comments with a date on task review.

We can implement the above via three models. Of course, here we just show an example in order to keep history data for the staging workflow. You can share this pattern to define different models for your own requirements.

The extended models include `LayoutHistory`, `TasksProposalHistory`, and `TaskCommentsHistory`. The `LayoutHistory` model keeps all information from the `Layout` model, while it adds version, creator, modifier, creation date, and modification date. The `TasksProposalHistory` model keeps all the information from the `TasksProposal` model, while it adds the page state. Similarly, the `TaskCommentsHistory` model supports multiple comments input. Each comment is specified with creator, modifier, creation date, and modification date. The following are the contents of `service.xml` in the `com.ext.portlet.tasks` package:

```
<!-- ignore details -->
<service-builder package-path="com.ext.portlet.tasks">
 <namespace>Tasks</namespace>
 <entity name="LayoutHistory" local-service="true"
 remote-service="true"
 persistence-class="com.ext.portlet.tasks.service.
 persistence.LayoutHistoryPersistenceImpl">
 <!-- PK fields -->
 <column name="id" type="long" primary="true" />
 <column name="plid" type="long"/>
 <column name="version" type="double" />
```

```
<column name="published" type="Date" />
<column name="creator" type="String" />
<column name="modifier" type="String" />
<column name="created" type="Date" />
<column name="modified" type="Date" />
<column name="groupId" type="long" />
<!-- ignore details --> </entity>
<entity name="TasksProposalHistory" local-service="true"
 remote-service="true"
 persistence-class="com.ext.portlet.tasks.service.
 persistence.TasksProposalHistoryPersistenceImpl">
 <!-- ignore details -->
</entity>
<entity name="TasksCommentsHistory" local-service="true"
 remote-service="true"
 persistence-class="com.ext.portlet.tasks.service.
 persistence.TasksCommentsHistoryPersistenceImpl">
 <!-- ignore details -->
</entity>
```

The code above shows an XML file for ServiceBuilder. After preparing `service.xml`, you can build services in the following manner:

1. Locate the XML file `/ext/ext-impl/build-parent.xml` and open it.
2. Add the following lines between `</target>` and `<target name="build-service-portlet-reports">` and save it:

```
<target name="build-service-portlet-extTasks">
 <antcall target="build-service">
 <param name="service.file"
 value="src/com/ext/portlet/tasks/service.xml" />
 </antcall>
</target>
```

When you are ready, double-click on the Ant target `build-service-portlet-extTasks`. ServiceBuilder will build related models and services.

## Building a standalone workflow portlet

We have built models and services successfully. Now let's build a standalone workflow portlet. The workflow portlet should work in the live group directly. It should have all of the features that the staging workflow has and should support the extending model as mentioned above.

## *Staging and Publishing*

---

As shown in the following screenshot, content creator, content producer, content reviewer, and content editor can have **My Tasks**, **Pending Tasks**, **Submitted Tasks**, and **Approval Status**. That is, workflow users can easily find what his/her tasks are, what pending tasks are, what submitted tasks were, and a list approval status of completed tasks.

Title	Creator	Producer	Reviewer	Editor	Status	Options
Privacy Policy	Content Creator				Awaiting Producer	
Oops	Content Creator				Awaiting Producer	
Leaving Workshop	Content Creator				Awaiting Producer	

How to implement it? Just like the Ext Navigation portlet, you can construct a portlet in Ext in the following manner:

1. Specify the Ext tasks portlet in `portlet-ext.xml`.
2. Register the Ext Tasks portlet in `liferay-portlet-ext.xml`.
3. Put it in the Book group in `liferay-display.xml`.
4. Add the page flow in `struts-config.xml`.
5. Add the page layout in `tiles-defs.xml`.
6. Finally, add the title mapping in `Language-ext.properties`.

Then we need to set up a portlet action for the Ext Tasks portlet. To do so, use the following steps:

1. Create a package named `com.ext.portlet.tasks.action` in the `/ext/ext-impl/src` folder.
2. Create a Java file named `AddTasksHistoryLocalServiceUtil.java` (you can use a different name for this Java file) in the `com.ext.portlet.tasks.action` package.
3. Add the following functions in order to create, update, delete, read data from the extended models, for example `LayoutHistory`, `TasksProposalHistory`, and `TasksCommentsHistory`:

```
public class AddTasksHistoryLocalServiceUtil {
 public static TasksProposalHistory getTasksProposalHistory(
 long proposalId) {
 TasksProposalHistory comment = null;
 try{
 comment = TasksProposalHistoryLocalServiceUtil.
 getTasksProposalHistory(proposalId);
 }
 }
}
```

```

 catch (Exception e) {}
 return comment;
 }
 public static TasksCommentsHistory getComment(long commentId) {
 TasksCommentsHistory comment = null;
 try{ comment = TasksCommentsHistoryLocalServiceUtil.
 getTasksCommentsHistory(commentId);
 }
 catch (Exception e) {}
 return comment;
 }
 // ignore details
}

```

The code above shows methods in order to create, update, delete, and read data from the extended models. Then, create a Java file named `ViewWorkflowAction.java` in the `com.ext.portlet.tasks.action` package and add the following functions in order to process action and render view:

```

public class ViewWorkflowAction extends PortletAction {
 public void processAction(ActionMapping mapping, ActionForm form,
 PortletConfig config, ActionRequest req, ActionResponse res)
 throws Exception {
 // ignore details
 public ActionForward render(ActionMapping mapping, ActionForm
 form, PortletConfig config, RenderRequest req,
 RenderResponse res) throws Exception {
 // ignore details
 }
}

```

The code above shows the `processAction` and `render` methods for the Ext Tasks portlet. Finally, we should create views for the Ext Tasks portlet. To do so, create a folder named `/tasks` under the `/ext/ext-web/docroot/html/portlet/ext` folder, and then create the required JSP files in the `/ext/ext-web/docroot/html/portlet/ext/tasks` folder—`init.jsp`, `view.jsp`, `MyTasks.jsp`, `pendingtasks.jsp`, and so on.

The Ext Tasks portlet provides workflow capability for the live group where the portlet is added. The entire staging workflow functions plus history data management are available. You can surely customize and extend this portlet with other features and functions.

## Enjoying other workflows

We have discussed the default staging workflow. We also showed how to customize the staging workflow, how to keep history data, and moreover, how to build a workflow portlet. In this section, we're going to briefly discuss other workflows such as journal article workflow, jBPM workflow, and Intalio | BPMS.

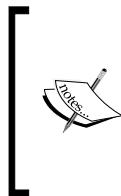
## Employing the journal article workflow

There is a simple fixed workflow on a journal article – save, save and approve, and expire. In other words, there are two states for a given journal article: unapproved or approved, and active or expired. Actually, these kinds of states are specified in the `com.liferay.portlet.journal.model.JournalArticle` model. For instance, the fields `approved`, `approvedById`, `approvedByName`, and `approvedDate` are used for the state of unapproved or approved; the fields `expired` and `expirationDate` are used for the state of active or expired. A similar mechanism is used for display and review by the `displayDate` and `reviewDate` fields.

The above information would be helpful when you would like to customize, or even extend, the journal article workflow. You can make the maximum use of the existing model to implement your requirements smoothly.

## Play with the jBPM workflow

The **jBPM** workflow should be working with an **ESB**. Instead of having the portal directly access the workflow service, it will access the ESB, and the ESB will then determine which workflow component to use. No matter how many times or in what ways the workflow service changes, the portal is never impacted directly. That is, ESBs such as **ServiceMix** and **Mule** provide a powerful mechanism to plug in new services and modify them with little or no impact to the portal configuration.



Service-Mix is truer to the traditional definition of an ESB (it adheres to the **Java Business Integration JBI JSR-208** specification); whereas Mule is based on ESB (it doesn't adhere to the specification), but is more flexible. However, Service-Mix has a smaller footprint than Mule. Refer to <http://mule.mulesource.org> and <http://servicemix.apache.org> for more information.

Let's gain an insight on how to get the jBPM workflow portlet running using plugins webs (Web Applications), `mule-web` and `jbpm-web`. To do so, first check out webs `mule-web` from `svn://svn.liferay.com/repos/public/plugins/trunk/webs/mule-web` to `$PLUGINS_HOME/webs/mule-web`, where `$PLUGINS_HOME` is the current Plugins SDK home. Then drop the `build.xml` file from `$PLUGINS_HOME/webs/mule-web` to the Ant view and run the `deploy` target under `web` in the Ant view. The `deploy` target will copy the WAR file `mule-web.war` from the `$PLUGINS_HOME/dist/` folder to the `$LIFERAY_PORTAL/deploy` folder. Make sure that the value for the `jbi.workflow.url` key in `portal.properties` is `http://localhost:8080/mule-web/workflow`. This ensures that we use Mule as the ESB.

Next, check out webs jbpm-web from `svn://svn.liferay.com/repos/public/plugins/trunk/webs/jbpm-web` to `$PLUGINS_HOME/webs/jbpm-web`. Modify `/jbpm-web/docroot/WEB-INF/src/hibernate.cfg.xml` to make it point to the appropriate database, for example MySQL. Afterwards, drop the `build.xml` file from `$PLUGINS_HOME/webs/jbpm-web` to the Ant view, and run the `deploy` target under the `web` in the Ant view. Similar to `web mule-web`, the Ant target `deploy` will copy the WAR file `jbpm-web.war` from the `$PLUGINS_HOME/dist/` folder to the `$LIFERAY_PORTAL/deploy/` folder.

These two examples, `mule-web` and `jbpm-web`, are typical web applications (that is, plugins webs) in Plugins SDK. Now, we can add the workflow portlet—jBPM workflow. Check out portlet `workflow-portlet` from `svn://svn.liferay.com/repos/public/plugins/trunk/portlets/workflow-portlet` to `$PLUGINS_HOME/portlets/workflow-portlet`. Afterwards, drop the `build.xml` file from `$PLUGINS_HOME/portlets/workflow-portlet` to the Ant view, and run the `deploy` target under `portlet` in the Ant view. From now on, you can add the workflow portlet to any page.

## Using Intalio | BPMS

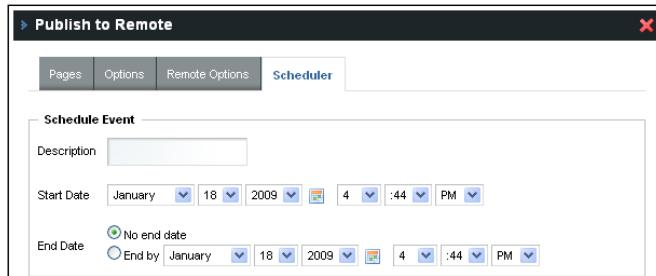
By using Intalio | BPMS, we can integrate the BPMN and BPEL inside the portal. By this, we can smoothly get a BPEL-based workflow in the portal. Intalio | BPMS is the BPMS to natively support the BPMN and BPEL industry standards. Refer to <http://www.intalio.com> for more information

## Scheduling web content

We have introduced staging workflow and workflow customization in the previous section. At the end of staging workflow, we may need to schedule jobs in order to transfer pages from the staging to the live. That is, the scheduling capability is very useful. We can select data and subsets of pages, or all of the pages of a given community, and transfer them to the live site, which can be a separate, remote portal instance—that is, remote staging and publishing.

## Scheduling pages

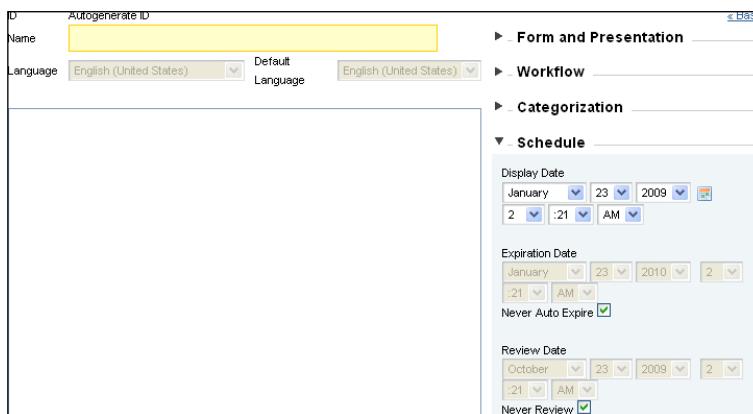
First of all, we're going to invest how to schedule pages for the publishing purpose. As shown in the following screenshot, scheduling capability to publishing pages has been provided by default. In order to publish pages remotely, we should select the scope of publishing—all of the pages, or the selected pages of a given community (for example, Book Street)—in the **Pages** tab first, and then use the **Scheduler** tab to add an event (that is, a job for publishing). You can provide **Description**, **Start Date**, **End Date**, and a repeatable feature.



In addition, we can set up the scheduling feature in order to publish pages immediately.

## Scheduling the web content

Further, we can schedule the web content as well. As shown in the following screenshot, we can schedule **Display Date**, **Expiration Date**, and **Review Date** in the Web Content portlet. For instance, you can specify the **Display Date** (that is, the web content will be available on display date), the **Expiration Date** (that is, the web content will be unavailable on the expiration date), and the **Review Date** (that is, the web content should be reviewed on the review date).



## What's happening?

In general, a scheduling engine with an SOA service is used in the portal. Here the scheduler engine is the **Quartz** scheduler. Quartz is a full-featured job scheduling system that can be integrated with, or used alongside virtually any J2EE or J2SE application from the smallest standalone application to the largest e-commerce system. It can be used to create simple or complex schedules for executing tens, hundreds, or even tens of thousands of jobs. Refer to <http://www.opensymphony.com/quartz> for more details.

## Setting a scheduler engine

Quartz is also set in the `portal.properties` properties file as follows:

```
org.quartz.dataSource.ds.connectionProvider.class=com.liferay.portal.
scheduler.quartz.QuartzConnectionProviderImpl
org.quartz.jobStore.class=org.quartz.impl.jdbcjobstore.JobStoreTX
org.quartz.jobStore.dataSource=ds
org.quartz.jobStore.driverDelegateClass=com.liferay.portal.scheduler.
quartz.DynamicDriverDelegate
ignore details
org.quartz.threadPool.threadPriority=5
```

The code above shows a connection provider, for example, `com.liferay.portal.scheduler.quartz.QuartzConnectionProviderImpl`. It also specifies job store, scheduler, thread pool, and so on.

## Scheduling layouts

Here let's take a look at the layout scheduling. The entry point of layout scheduling is specified in `com.liferay.portlet.communities.util.StagingUtil` of the / `portal/portal-impl/src` folder as follows:

```
if (schedule) {
 String groupName = getSchedulerGroupName(
 DestinationNames.LAYOUTS_REMOTE_PUBLISHER, groupId);
 // ignore details
 LayoutServiceUtil.schedulePublishToRemote(
 groupId, privateLayout, layoutIdMap,
 getStagingParameters(actionRequest), remoteAddress,
 remotePort, secureConnection, remoteGroupId,
 remotePrivateLayout, startDate,endDate, groupName,
 cronText, startCal.getTime(),schedulerEndDate, description);
}
```

The preceding code shows layout scheduling. It first checks the `schedule` flag. Then it collects scheduling information and uses the `LayoutServiceUtil`. `schedulePublishToRemote` layout service to schedule jobs. The `LayoutServiceUtil.schedulePublishToRemote` method service has been implemented in `com.liferay.portal.service.impl.LayoutServiceImpl` as follows:

```
public void schedulePublishToRemote(long sourceGroupId, boolean
 privateLayout, Map<Long, Boolean> layoutIdMap,
 Map<String, String[]> parameterMap, String remoteAddress,
 int remotePort, boolean secureConnection, long remoteGroupId,
 boolean remotePrivateLayout, Date startDate, Date endDate,
 String groupName, String cronText, Date schedulerStartDate,
 Date schedulerEndDate, String description)
throws PortalException, SystemException {
 // ignore details
 SchedulerEngineUtil.schedule(
 groupName, cronText, schedulerStartDate, schedulerEndDate,
 description, DestinationNames.LAYOUTS_REMOTE_PUBLISHER,
 JSONFactoryUtil.serialize(publisherRequest));
}
```

The code above shows that the `SchedulerEngineUtil.schedule` method is used to set up the scheduler. You can find `SchedulerEngineUtil` in the `com.liferay.portal.kernel.scheduler` package of the `/portal/portal-impl/src` folder where a set of scheduling methods have been specified, for example `start`, `shutdown`, `schedule`, `un-schedule`, and so on.

You may be interested in the details of the Java beans definition for scheduler. You can find the XML file `scheduler-spring.xml` in the `/portal/portal-impl/src/META-INF` folder where two beans are specified as well: `com.liferay.portal.kernel.job.JobSchedulerUtil` and `com.liferay.portal.kernel.scheduler.SchedulerEngine`.

As stated above, we have shown the process on how to schedule layouts where the code above is using `SchedulerEngineUtil`. Definitely you can follow the same process to customize/extend the code above in order to satisfy your own current or future requirement. In short, the process to schedule layouts is extendable.

## Configuring scheduler class

We have discussed how to apply scheduler on the layouts. Let's move forward to see how to apply scheduler on the portlets, for example, journal, announcements, blogs, calendar, message boards, and so on. There is a job scheduler interface named `Scheduler` available, and a set of classes that implement `com.liferay.portal.kernel.job.Scheduler`. Some of these classes include `AdminScheduler`, `AnnouncementsScheduler`, `BlogsScheduler`, `JournalsScheduler`, `CalendarScheduler`, `MBScheduler` which allow jobs to be scheduled.

Of course, you can provide a customized class which implements the `com.liferay.portal.kernel.job.Scheduler` interface. In order to have a scheduling capability, you can just put the above class in your customized portlet definition. For instance, the Journal portlet uses the `com.liferay.portlet.journal.job.JournalsScheduler` class in `liferay-portlet.xml` to allow jobs on journal articles to be scheduled as follows:

```
<portlet>
 <portlet-name>15</portlet-name>
 <icon>/html/icons/default.png</icon>
 <struts-path>journal</struts-path>
 <!-- ignore details -->
 <scheduler-class>
 com.liferay.portlet.journal.job.JournalsScheduler
 </scheduler-class>
 <!-- ignore details -->
</portlet>
```

The code above shows the definition of portlet 15 (that is, Journal portlet). It defines portlet name, icon, struts path, configuration action class, and so on. It especially specifies the scheduler class with a value `com.liferay.portlet.journal.job.JournalsScheduler`.

By the way, you can configure the scheduler to be enabled or disabled. This kind of feature is defined in `portal.properties` as follows:

```
scheduler.enabled=true
scheduler.classes=
```

The code above shows features—how to enable or disable scheduling capability on the portlets. Set the `scheduler.enabled` property to `false` to disable all of the scheduler classes defined in the `scheduler.classes` property of `liferay-portlet.xml`. Moreover, you can input a list of comma-delimited class names that implement `com.liferay.portal.kernel.job.Scheduler` for the `scheduler.classes` property. These classes allow jobs to be scheduled on startup. Note that, these classes are not associated to any portlet.

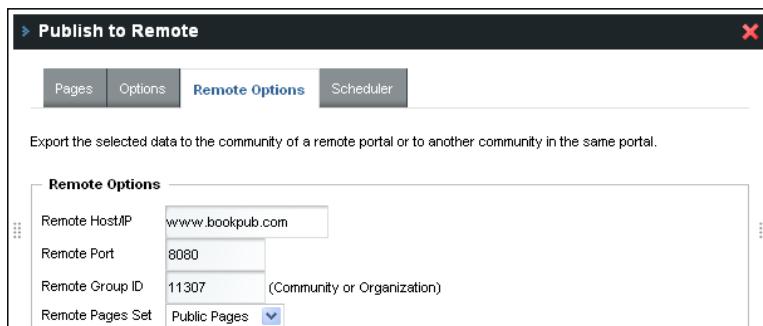
## Experiencing remote staging and publishing

Liferay portal provides remote staging and publishing capability through which the users can select subsets of pages and data, and transfer them to the live site—that is, remote portal instance. Using this, we can export the selected data to the community of a remote portal instance, or to another community in the same portal instance.

First, let's consider the scenario: export the selected pages and data to another community in the same portal instance. In the web site [www.bookpub.com](http://www.bookpub.com), we already have a default community named Guest with a group ID as 10148. In the public pages of the Guest community, create a page named Remote Publishing and add the Hello World and Asset Publisher portlets in this page. Next, we will create a community named Book Workshop with a group ID as 11307. We're going to publish the Remote Publishing page from the public pages of the Guest community to the public pages of the Book Workshop community.

How to publish this page? We can use the publishing feature as follows:

1. Log in as an admin.
2. Go to **Control Panel** and click on **Communities** under the **Portal** category.
3. Locate the **Guest** community and click on the **Manage Pages** action.
4. Click on the **Publish to Remote** button and you will see the following screen:



Now let's set up the publishing feature as follows:

1. Under the **Pages** tab, choose **Scope** as **Selected Pages**
2. Select the **Remote Publishing** page.
3. Under the **Remote Option** tab, enter the values for remote options as shown in the above screenshot – **Remote Host/IP** as **www.bookpub.com** (for example, external IP: **64.71.191.145**), **Remote Port** as **8080**, **Remote Group ID** as **11307**, and **Remote Pages Set** as **Public Pages**.

When you are ready, click on the **Pages** tab, and then click on the **Publish** button. The **Remote Publishing** page will be published from the public pages of the Guest community to the public pages of the Book Workshop community.

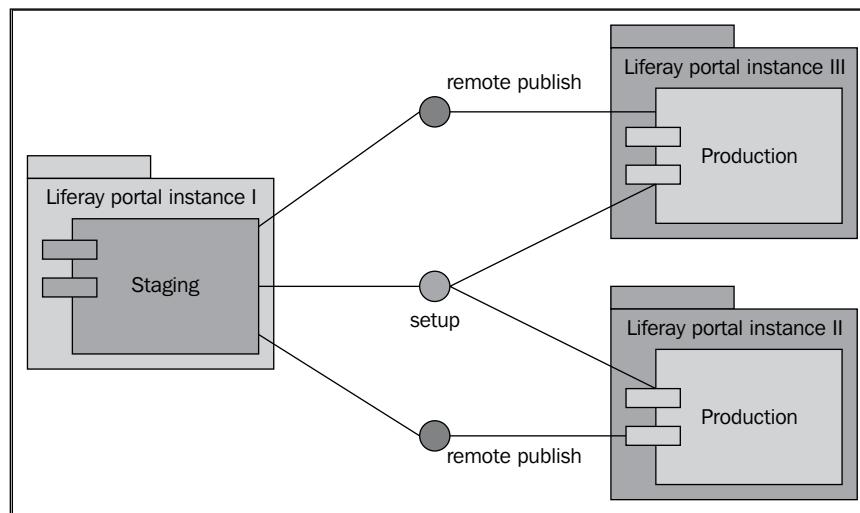
Now we're going to consider another scenario: Export the selected data to the community of a remote portal instance. We're going to publish the **Remote Publishing** page from the public pages of the Guest community to the public pages of the Guest community of a remote portal instance. Suppose the remote portal has external IP **69.198.171.104** (domain name **liferay.cignex.com**), port number **8080**, and group ID of the Guest 1 community.

How to publish this page to a remote portal instance? Under the **Pages** tab, choose **Scope** as **Selected Pages**, and select the **Remote Publishing** page. Under the **Remote Option** tab, input values for remote options, for example **Remote Host/IP** as **liferay.cignex.com** (or external IP **69.198.171.104**, for example. Of course, you can have different remote portal instances), **Remote Port** as **8080**, **Remote Group ID** as **1**, and **Remote Pages Set** as **Public Pages**.

When you are ready, click on the **Pages** tab, and then click on the **Publish** button. The **Remote Publishing** page will be published from the public pages of the Guest community of current portal instance, for example **www.bookpub.com**, to the public pages of the Guest community of the remote portal instance, for example **liferay.cignex.com**.

## What's remote staging and publishing?

We have introduced local staging and publishing in the previous section. Local staging and publishing means that we have only one box and only one Liferay portal instance. For a given group, for example Book Street, we created a staging group Book Street (Staging) – a working copy of the Book Street group. Now users can work on the staging group only. When they are ready, they can publish pages of the Book Street (Staging) staging group to the pages of the Book Street group. As the staging and publishing happen in one box, and the Book Street (Staging) staging group and the Book Street live group belongs to the same Liferay portal instance, it is called local staging and publishing. This would be useful when the web site is small with less traffic and a small group of end users.



But when the web site is huge – high traffic, big groups of end users – we have to consider the remote staging and publishing feature. As shown in the above figure, there is one staging box and two production boxes. All of the boxes have live groups only. For instance, the Book Street live group in the staging box will be mapped into the Book Street live group in the production boxes. Thus, the Book Street live group in the staging box could be called as a staging group of the Book Street live group in the production boxes.

All of the internal content management users are working in the staging box only. They can use CMS and WCM tools to manage the content and web content, for example, building a live web site. They can also apply workflow to approve or reject the content and web content. That is, the staging box is used only for the internal content management team. Once the pages are approved, content management team can publish these pages into the production boxes.

The production boxes are used only for end users (either public users or private customers). These boxes are set up with the same content and web content from the staging box at the beginning. Afterwards, they will accept updates via remote staging and publishing. And they also save specific data in the production boxes, such as users' ratings, comments, preferences, and so on.

In addition, you can have a set of production boxes – more than two boxes as stated in the above figure. These production boxes can be organized as a clustering environment. Further, these production boxes can be distributed geographically. For instance, it is possible that you can put one box in California, USA, and another box in Zurich, Switzerland. In order to show remote staging and publishing, we use the external IP 64.71.191.145 for the staging box I, and external IPs 69.198.171.104 and 69.198.171.105 for the production boxes II and III, respectively.

## How does it work?

How does remote staging and publishing work? Let's gain a deep insight on this feature. The entry point to publish to remote is specified in the Java file `EditPagesAction.java` in the `com.liferay.portlet.communities.action` package under the `/portal/portal-impl/src` folder. The following is a piece of code:

```
public void processAction(ActionMapping mapping, ActionForm form,
 PortletConfig portletConfig, ActionRequest actionRequest,
 ActionResponse actionResponse)
 throws Exception {
 String cmd = ParamUtil.getString(actionRequest, Constants.CMD);
 // ignore details
 else if (cmd.equals("publish_to_remote")) {
 StagingUtil.publishToRemote(actionRequest);
 }
}
```

The code above shows an implementation of the `processAction` method. When the `cmd` command is equal to the `publish_to_remote` value, it will publish the selected pages from the staging box to the production box.

## Importing and exporting

The details of remote publishing method are specified in the Java file `StagingUtil.java` in the `com.liferay.portlet.communities.util` package as follows:

```
public static void copyRemoteLayouts(long sourceGroupId, boolean
 privateLayout, Map<Long, Boolean> layoutIdMap,
 Map<String, String[]> exportParameterMap, String remoteAddress, int
 remotePort, boolean secureConnection, long remoteGroupId, boolean
 remotePrivateLayout,
 Map<String, String[]> importParameterMap, Date startDate,
 Date endDate) throws Exception {
 // ignore details
 HttpPrincipal httpPrincipal = new HttpPrincipal(
 url, user.getEmailAddress(),
 user.getPassword(),
 user.getPasswordEncrypted());
 try {
 GroupServiceHttp.getGroup(httpPrincipal, remoteGroupId);
 }
 // ignore details
 bytes = LayoutServiceUtil.exportLayouts(sourceGroupId,
 privateLayout, layoutIds, exportParameterMap, startDate,
 endDate);
}
LayoutServiceHttp.importLayouts(httpPrincipal, remoteGroupId,
 remotePrivateLayout, importParameterMap, bytes);
}
```

The code above shows a way to copy remote layouts. It first pings the remote host by the `GroupServiceHttp.getGroup` method and verifies whether or not the group exists. Then it exports the selected pages using the **LAR (Liferay Archive)** method `LayoutServiceUtil.exportLayouts`. Finally, it imports the selected pages into the remote host using LAR method `LayoutServiceHttp.importLayouts`. In short, it uses the LAR exporting and importing features.

## Using tunnel web

As stated above, LAR exporting and importing features are used for remote staging and publishing. At the same time, a **tunnel web** is used for the communication between the staging box and the production boxes. For example, the `GroupServiceHttp.getGroup` method uses tunnel web as follows:

```
public static com.liferay.portal.model.Group getGroup(
 HttpPrincipal httpPrincipal, long groupId)
throws com.liferay.portal.PortalException,
com.liferay.portal.SystemException {
```

```
try {
 Object paramObj0 = new LongWrapper(groupId);
 MethodWrapper methodWrapper = new MethodWrapper(
 GroupServiceUtil.class.getName(), "getGroup",
 new Object[] { paramObj0 });
 Object returnObj = null;
 try {
 returnObj = TunnelUtil.invoke(httpPrincipal, methodWrapper);
 }
 // ignore details
```

The code above shows how to use `TunnelUtil.invoke` to invoke the remote server. It first forms a method wrapper named `methodWrapper` with a group. Then it uses the `invoke` method of `TunnelUtil` in the `com.liferay.portal.service.http` package. Further, the `LayoutServiceUtil.exportLayouts` method also uses tunnel web.

```
public static void importLayouts(HttpPrincipal httpPrincipal, long
 groupId, boolean privateLayout, java.util.Map<String, String[]>
 parameterMap, byte[] bytes)
throws com.liferay.portal.PortalException,
com.liferay.portal.SystemException {
 try {
 Object paramObj0 = new LongWrapper(groupId);
 // ignore details
 MethodWrapper methodWrapper = new MethodWrapper(
 LayoutServiceUtil.class.getName(), "importLayouts",
 new Object[] {
 paramObj0, paramObj1, paramObj2,
 paramObj3
 });
 try { TunnelUtil.invoke(httpPrincipal, methodWrapper); }
 // ignore details
```

The code above used `TunnelUtil.invoke` and showed a way to invoke the remote server via tunnel web. It first formed a `methodWrapper` method wrapper with import layouts. Then it used the `invoke` method of `TunnelUtil`.

## Setting up tunnel web

In order to communicate with the remote server and, moreover, protect HTTP connection, we need to set up a tunnel web in `portal-ext.properties`. That is, we need to add the following lines at the end of `portal-ext.properties`:

```
tunnel.servlet.hosts.allowed=127.0.0.1, SERVER_IP, 69.198.171.104, 69.19
8.171.105, 64.71.191.145
tunnel.servlet.https.required=false
```

The preceding code shows a `tunnel.servlet.hosts.allowed` property with a list of allowed hosts, for example, `69.198.171.104`, `69.198.171.105`, and `64.71.191.145`. As stated above, we used these hosts as examples only. You can have your own real hosts. Meanwhile, it specifies the `tunnel.servlet.https.required` property. By default, it is set as a `false` value. You can set it as a `true` value if you want to use HTTPS.

## Using LAR to export and import

As stated above, the LAR exporting and importing features are used for remote staging and publishing. These features are implemented in the `PortletDataHandler` API. The intent of this API is to provide the portal and third-party portlets with a useful API for importing and exporting application content to and from the portal in a database agnostic fashion.

### Defining portlet-data-handler

The first step in using the API involves creating a class which implements `com.liferay.portal.kernel.lar.PortletDataHandler` in the `/portal/portal-service/src` folder. This interface defines the following methods:

```
public PortletDataHandlerControl[] getExportControls() throws
PortletDataException;
public PortletDataHandlerControl[] getImportControls() throws
PortletDataException;
public String exportData(PortletDataContext context, String portletId,
PortletPreferences prefs) throws PortletDataException;
public PortletPreferences importData(PortletDataContext context,
String portletId, PortletPreferences prefs, String data) throws
PortletDataException;
```

The code above shows a set of `getExportControls`, `getImportControls`, `exportData`, and `importData` methods in `PortletDataHandler`. The core functionality is drawn from the implementation of the `importData` and `exportData` methods. These are called when the import or export functions are executed by the users.

There are a set of classes which extend `com.liferay.portal.lar.BasePortletDataHandler`, implementing the `PortletDataHandler` interface. This includes `JournalContentPortletDataHandlerImpl`, `JournalPortletDataHandlerImpl`, `DLDisplayPortletDataHandlerImpl`, `DLPortletDataHandlerImpl`, `IGPortletDataHandlerImpl` among many.

The purpose of portal-data-handlers is to provide a pluggable way to handle data export and import, which generally revolves around the concept of storing data outside the portal permanently or temporarily. Liferay portal does this by handling the creation and interpretation of the **LAR** files. The LAR files are ZIP archives which contain, by default, a single file called `layouts.xml`. This XML file `layouts.xml` contains complete information about the community from which it was created, layout permissions, portlet permissions, portlet preferences, and so on.

## Configuring a portlet with portlet-data-handler

In order to call the portlet-data-handler, the portal must be informed of its existence. This is done by defining the `portlet-data-handler-class` tag in your portlet definition within the `liferay-portlet.xml` (for Liferay portal core and Plugins SDK) or the `liferay-portlet-ext.xml` (for Ext) files. The following is the configuration for the default data handler of the journal portlet:

```
<portlet>
 <portlet-name>15</portlet-name>
 <icon>/html/icons/default.png</icon>
 <!-- ignore details -->
 <portlet-data-handler-class>
 com.liferay.portlet.journal.lar.JournalPortletDataHandlerImpl
 </portlet-data-handler-class>
 <!-- ignore details -->
</portlet>
```

The code above shows that the Journal portlet uses `com.liferay.portlet.journal.lar.JournalPortletDataHandlerImpl` as the value of its `portlet-data-handler-class` portlet-data-handler class. Once portlet-data-handler is properly defined and configured, it will perform its actions when the import and/or export LAR functions are executed from the **Import/Export** tab of a community.

## Using portlet-data-handler

In general, there are different scenarios where we can use portlet-data-handler as well. First, we can use it for content development and staging. This involves developing content in a non-production portal instance, and then exporting that content to the production portal instance. We can also use portlet-data-handler for data archiving. This involves storing portal data as a safeguard against a complete system failure, or purely for record keeping.

Last but not the least, we can use it for versioning. This involves keeping a version of the content after each change has been made in order to satisfy some change in the management policy, to protect against mistakes, or to maintain several different content scenarios without requiring complete duplication of the staging environment.

## Using SCORM

Now it might be possible to create handlers that can import/export formats native to third-party applications without having to hack into the portal core. One significant example might be the **Sharable Content Object Reference Model (SCORM)** packaging format. If Liferay portal could implement a SCORM portlet, it might be possible to natively support SCORM packages with an associated `portlet-data-handler` in future. SCORM is a collection of standards and specifications for web-based e-learning. It defines communications between client-side content and a host system.

## Summary

This chapter discussed simple extension—how to build dynamic navigation and construct a customized site map. Then it addressed how to handle events and model listeners. Based on these features, this chapter further introduced local staging and publishing, and staging workflow. A way to schedule pages and assets was also discussed. Finally, it addressed how to publish web content remotely, where `portlet-data-handler` (for export and import via LAR) was addressed as well.

In the next chapter, we're going to introduce how to use common API.

# 12

## Using Common API

In the intranet web site bookpub.com of the enterprise "Palm Tree Publications", we have a plan to build dynamic, content-rich, and ad-serving Internet web sites. These additional features should be applied in order to manage the content and web content and, moreover, to smoothly build dynamic web sites. Thus, the common API of Liferay portal would be helpful for this purpose.

Liferay portal provides a feature called **Custom Attribute**, which allows extending the profile of users and organizations with fields to store custom information. Custom attribute is safely stored within the database and is fully indexed. Liferay portal also provides a capability to support **OpenSearch**. Using OpenSearch, we can get search results in the OpenSearch standard. Further, the portal provides **Web Services** where web services consumers can share the data of the portal from the outside. Web services are resources which may be called over the HTTP protocol to return the data. In addition, the Liferay services are implemented by **Spring Services**. Thus, we can provide a custom implementation of Liferay services by using Spring services.

This chapter will first introduce how to use custom attributes within both journal article templates and custom portlets. Then it will address how to build OpenSearch and how to use search capabilities in portlets. Later, this chapter will focus on the approaches about how to employ Spring services and construct web services. Finally, it will discuss the best practices, for example, using JavaScript portlet URL, customizing user and organization administration, speeding up portal, sharing UI Taglibs, producing and consuming WSRP, integrating with SharePoint and Terracotta DSO, and so on.

By the end of this chapter, you will have learned how to:

- Add custom attributes
- Build OpenSearch
- Employ Spring services
- Construct web services
- Enjoy best practices

## Adding custom attributes

Custom attributes allow extending the profile of users and organizations with fields to store custom information. At the same time, custom attributes are applicable to all entities generated by ServiceBuilder. Moreover, custom attributes are accessible from Velocity templates or in portlets. In this section, we're going discuss how to use custom attributes in both Velocity template and portlets.

## Building dynamic table with Velocity Expando template

First, we will build dynamic table with the Velocity **Expando** templates. In JavaScript, Expando means a way to attach additional properties to an object. Here, Expando means a way to attach additional attributes to an entity.

As shown in the following screenshot, we're going to define a collection of data dynamically – **BOOK TITLE LIST**. That is, we will build **BOOK TITLE LIST** without creating an additional database schema except custom attributes. For example, there are columns such as **Creator Name**, **Book Title**, **Price**, **Modified Date**, and actions such as **Edit** and **Delete**; and simple data for users, for example **Lotti Stein** and **David Berger**.



## Creating a journal structure

First, let's create a journal structure in the Journal portlet. This structure is required for any journal template as the connection between a journal article and a journal template is a function of the journal structures. Here we need a basic structure EXPANDO for Title, for example BOOK TITLES LIST, as follows:

```
<root>
 <dynamic-element name='title'
 type='text'
 repeatable='false'>
 </dynamic-element>
</root>
```

The code above shows a dynamic element title with a type text.

## Creating a journal template

Now we need to create a journal template EXPANDO with the following lines via the Web Content portlet. By the way, we need to disable template caching by deselecting **Cacheable**. Thus, we can receive context parameters, for example user, properly.

```
#set ($locale = $localeUtil.fromLanguageId($request.get("locale")))
#set ($dateFormatDateTime = $dateFormats.getDateTime($locale))
<h1>$title.Data</h1>
```

The code above shows the Velocity template's default variables: `localeUtil`, `dateFormats`, and `request`. It also shows a custom Velocity template variable `title`, which is specified in the journal structure EXPANDO. In order to get a view as shown in the code above, we should create a journal article EXPANDO in the Web Content portlet using the journal structure EXPANDO and the journal template EXPANDO. For the Title, you can type BOOK TITLES LIST. Later, click on the **Save and Approve** button. It means that you have to create a journal article EXPANDO using the journal structure EXPANDO and the journal template EXPANDO. Finally, we can see the results in the following steps:

1. Create a page named `Expando` in the public pages of the Guest community.
2. Add a Web Content Display portlet in the `Expando` page.
3. Choose the `Expando` journal article in the instance of the Web Content Display portlet.

Later, you will see the title BOOK TITLE LIST.

## Building Book Title List

Now let's build BOOK TITLE LIST completely in the Expando journal template. First, we can use Expando to define a table called BooksTable in the Expando journal template as follows. Expando allows us do this programmatically, and moreover, with very little code.

```
#set ($booksTableName = "BooksTable")
#set ($booksTable = $expandoTableLocalService.getTable(
 $booksTableName, $booksTableName))
#if (!$booksTable)
 #set ($booksTable = $expandoTableLocalService.addTable(
 $booksTableName, $booksTableName))
#set ($booksTableId = $booksTable.getTableId())
#set ($V = $expandoColumnLocalService.addColumn($booksTableId,
 "creatorName", 15))
#set ($V = $expandoColumnLocalService.addColumn($booksTableId,
 "bookTitle", 15))
#set ($V = $expandoColumnLocalService.addColumn($booksTableId,
 "price", 5))
#set ($V = $expandoColumnLocalService.addColumn($booksTableId,
 "modifiedDate", 3))
#endif
#set ($renderUrl = $request.get("render-url"))
#set ($namespace = $request.get("portlet-namespace"))
#set ($cmd = $request.get("parameters").get("cmd"))
#set ($creatorName = '') #set ($bookTitle = '')
#set ($price = 0.0)
```

The code above shows how to use the template variable `expandoTableLocalService` to create a table and columns. It checks whether the `BooksTable` table exists or not. If not, it creates it. Later, it builds the columns `creatorName`, `bookTitle`, `price`, and `modifiedDate` for this table to keep a track of the last time the book's title was updated. Meanwhile, this process need not be followed every time; so only do it when the table is created for the first time.

For each column, an integer is used as the last parameter. These integer constants are defined in `com.liferay.portlet.expando.model.ExpandoColumnConstants` and include the types. For instance, 15 represents `java.lang.String`, 5 represents `java.lang.Double`, and 3 represents `java.util.Date`. In addition, the template adds approaches to detect and handle the various operations of the application required by the CRUD operations. These request parameters include `renderUrl`, `namespace`, `cmd`, `creatorName`, `bookTitle`, and `price`.

Expando records have a primary key called `classPK`. To add, update, delete, and edit a Book Title of the `BooksTable` table, we need to add the following lines at the end of the journal template EXPANDO:

```
#set ($classPK = $getterUtil.getLong($request.get(
 "parameters").get("classPK")))
#if ($cmd.equals("add") || $cmd.equals("update"))
 /* ignore details */#elseif ($cmd.equals("delete"))
 /* ignore details */#elseif ($cmd.equals("edit"))
 /* ignore details */
#endif
>

#if (!$cmd.equals("edit"))
 <input type="button" value="Create Account"
 onClick="self.location = '${renderUrl}&${namespace}
 cmd=edit';" />

<table class="lfr-table">
 <tr>
 <th>Creator Name</th>
 <th>Book Title</th>
 <th>Price</th>
 <th>Modified Date</th>
 <th><!--empty --></th>
 </tr>
 #set ($rowsCount = $expandoRowLocalService.getRowsCount(
 $booksTableName, $booksTableName))
 #set ($rows = $expandoRowLocalService.getRows($booksTableName,
 $booksTableName, -1, -1))
 #foreach($row in $rows)
 #set ($currentClassPK = $row.getClassPK())
 <tr> #set ($currentCreatorName = $expandoValueLocalService.
 getData($booksTableName, $booksTableName,
 "CreatorName", $currentClassPK, ""))
 <td>${currentCreatorName}</td>
 /* ignore details */
 <td>
 Edit
 |
 Delete
 </td>
 #end
 </tr>
#end
</table>
```

```
 Delete

 </td>
 </tr>
#else /* ignore details */
#end /* ignore details */
</table>
#end


```

The code above shows a set of operations on book titles, including add, update, delete, edit, and view. It uses template variables to display Book Titles, for example `expandoRowLocalService` and `expandoValueLocalService`. The methods include `expandoRowLocalService.getRowsCount`, `expandoRowLocalService.getRows`, `expandoValueLocalService.getData`, `row.getRowClassPK`, and so on.

In brief, there are a set of Velocity templates available to build Book Title List: `expandoTableLocalService`, `expandoColumnLocalService`, `expandoRowLocalService`, and `expandoValueLocalService`. Expando services (for example, Velocity templates) allow us to define a collection of data dynamically.

## What's happening?

Actually, Liferay portal has a predefined set of the Velocity template variables for the Expando services. Meanwhile, a set of models and services are also specified in the portal. Thus, we're going to take a detailed look at these features.

## The Expando Velocity template variables

Expando services are specified as the Velocity template variables in `com.liferay.portal.velocity.VelocityVariables` under the `/portal/porta-impl/src` folder. In `VelocityVariables`, you can find the following lines related to the Expando services and their Velocity template variables:

```
ServiceLocator serviceLocator = ServiceLocator.getInstance();
velocityContext.put("expandoColumnLocalService",
 serviceLocator.findService(ExpandoColumnLocalService.
 class.getName()));
velocityContext.put("expandoRowLocalService",
 serviceLocator.findService(ExpandoRowLocalService.
 class.getName()));
velocityContext.put("expandoTableLocalService",
 serviceLocator.findService(ExpandoTableLocalService.
 class.getName()));
velocityContext.put("expandoValueLocalService",
```

```
serviceLocator.findService(ExpandoValueLocalService.
 class.getName()));
```

The code above shows the Expando services in the Velocity templates. It specifies four services as the Velocity templates variables—`expandoColumnLocalService` for column service, `expandoRowLocalService` for row service, `expandoTableLocalService` for table service, and `expandoValueLocalService` for value service.

## Models and services

There are four models which are predefined in the portal. As shown in the following screenshot, these models include `expandotable`, `expandocolumn`, `expandorow`, and `expandovalue`. The `expandotable` model represents an entry point of the Expando services. It includes attributes: `tableId`, `classNameId`, `name`, and `companyId`. For instance, the above example defined the `BooksTable` name for the `expandotable` model. At the same time, it created a custom class `BooksTable` with a value of `classNameId`. As you can see, an Expando table is associated with `classNameId`. If the class name (what `classNameId` represents) has a value `com.liferay.portal.model.Layout`, it means that the Expando table is based on the layout pages. If the class name has a value `com.liferay.portlet.journal.model.JournalArticle`, it means that the Expando table is based on the web content. In short, Expando tables can be applied on any kind of entities, either generated by ServiceBuilder (for example, `com.liferay.portal.model.User` and `com.liferay.portal.model.Organization`), or custom entities (for example, `BooksTable`). In addition, you can find a default table name `DEFAULT_TABLE` in `com.liferay.portlet.expando.model.ExpandoTableConstants` under the `/portal/portal-service/src` folder.

<code>expandotable</code>	<code>expandocolumn</code>	<code>expandovalue</code>																																						
<table border="1"> <thead> <tr> <th><code>tableId</code></th><th><code>bigint(20)</code></th></tr> </thead> <tbody> <tr> <td><code>classNameId</code></td><td><code>bigint(20)</code></td></tr> <tr> <td><code>name</code></td><td><code>varchar(75)</code></td></tr> <tr> <td><code>companyId</code></td><td><code>bigint(20)</code></td></tr> </tbody> </table>	<code>tableId</code>	<code>bigint(20)</code>	<code>classNameId</code>	<code>bigint(20)</code>	<code>name</code>	<code>varchar(75)</code>	<code>companyId</code>	<code>bigint(20)</code>	<table border="1"> <thead> <tr> <th><code>columnId</code></th><th><code>bigint(20)</code></th></tr> </thead> <tbody> <tr> <td><code>tableId</code></td><td><code>bigint(20)</code></td></tr> <tr> <td><code>name</code></td><td><code>varchar(75)</code></td></tr> <tr> <td><code>type_</code></td><td><code>int(11)</code></td></tr> <tr> <td><code>defaultData</code></td><td><code>longtext(2147483647)</code></td></tr> <tr> <td><code>typeSettings</code></td><td><code>longtext(2147483647)</code></td></tr> <tr> <td><code>companyId</code></td><td><code>bigint(20)</code></td></tr> </tbody> </table>	<code>columnId</code>	<code>bigint(20)</code>	<code>tableId</code>	<code>bigint(20)</code>	<code>name</code>	<code>varchar(75)</code>	<code>type_</code>	<code>int(11)</code>	<code>defaultData</code>	<code>longtext(2147483647)</code>	<code>typeSettings</code>	<code>longtext(2147483647)</code>	<code>companyId</code>	<code>bigint(20)</code>	<table border="1"> <thead> <tr> <th><code>valueId</code></th><th><code>bigint(20)</code></th></tr> </thead> <tbody> <tr> <td><code>tableId</code></td><td><code>bigint(20)</code></td></tr> <tr> <td><code>columnId</code></td><td><code>bigint(20)</code></td></tr> <tr> <td><code>rowId_</code></td><td><code>bigint(20)</code></td></tr> <tr> <td><code>classNameId</code></td><td><code>bigint(20)</code></td></tr> <tr> <td><code>classPK</code></td><td><code>bigint(20)</code></td></tr> <tr> <td><code>data_</code></td><td><code>longtext(2147483647)</code></td></tr> <tr> <td><code>companyId</code></td><td><code>bigint(20)</code></td></tr> </tbody> </table>	<code>valueId</code>	<code>bigint(20)</code>	<code>tableId</code>	<code>bigint(20)</code>	<code>columnId</code>	<code>bigint(20)</code>	<code>rowId_</code>	<code>bigint(20)</code>	<code>classNameId</code>	<code>bigint(20)</code>	<code>classPK</code>	<code>bigint(20)</code>	<code>data_</code>	<code>longtext(2147483647)</code>	<code>companyId</code>	<code>bigint(20)</code>
<code>tableId</code>	<code>bigint(20)</code>																																							
<code>classNameId</code>	<code>bigint(20)</code>																																							
<code>name</code>	<code>varchar(75)</code>																																							
<code>companyId</code>	<code>bigint(20)</code>																																							
<code>columnId</code>	<code>bigint(20)</code>																																							
<code>tableId</code>	<code>bigint(20)</code>																																							
<code>name</code>	<code>varchar(75)</code>																																							
<code>type_</code>	<code>int(11)</code>																																							
<code>defaultData</code>	<code>longtext(2147483647)</code>																																							
<code>typeSettings</code>	<code>longtext(2147483647)</code>																																							
<code>companyId</code>	<code>bigint(20)</code>																																							
<code>valueId</code>	<code>bigint(20)</code>																																							
<code>tableId</code>	<code>bigint(20)</code>																																							
<code>columnId</code>	<code>bigint(20)</code>																																							
<code>rowId_</code>	<code>bigint(20)</code>																																							
<code>classNameId</code>	<code>bigint(20)</code>																																							
<code>classPK</code>	<code>bigint(20)</code>																																							
<code>data_</code>	<code>longtext(2147483647)</code>																																							
<code>companyId</code>	<code>bigint(20)</code>																																							
<table border="1"> <thead> <tr> <th><code>expandorow</code></th><th></th></tr> </thead> <tbody> <tr> <td> <table border="1"> <thead> <tr> <th><code>rowId</code></th><th><code>bigint(20)</code></th></tr> </thead> <tbody> <tr> <td><code>tableId</code></td><td><code>bigint(20)</code></td></tr> <tr> <td><code>classPK</code></td><td><code>bigint(20)</code></td></tr> <tr> <td><code>companyId</code></td><td><code>bigint(20)</code></td></tr> </tbody> </table> </td><td></td></tr> </tbody> </table>	<code>expandorow</code>		<table border="1"> <thead> <tr> <th><code>rowId</code></th><th><code>bigint(20)</code></th></tr> </thead> <tbody> <tr> <td><code>tableId</code></td><td><code>bigint(20)</code></td></tr> <tr> <td><code>classPK</code></td><td><code>bigint(20)</code></td></tr> <tr> <td><code>companyId</code></td><td><code>bigint(20)</code></td></tr> </tbody> </table>	<code>rowId</code>	<code>bigint(20)</code>	<code>tableId</code>	<code>bigint(20)</code>	<code>classPK</code>	<code>bigint(20)</code>	<code>companyId</code>	<code>bigint(20)</code>																													
<code>expandorow</code>																																								
<table border="1"> <thead> <tr> <th><code>rowId</code></th><th><code>bigint(20)</code></th></tr> </thead> <tbody> <tr> <td><code>tableId</code></td><td><code>bigint(20)</code></td></tr> <tr> <td><code>classPK</code></td><td><code>bigint(20)</code></td></tr> <tr> <td><code>companyId</code></td><td><code>bigint(20)</code></td></tr> </tbody> </table>	<code>rowId</code>	<code>bigint(20)</code>	<code>tableId</code>	<code>bigint(20)</code>	<code>classPK</code>	<code>bigint(20)</code>	<code>companyId</code>	<code>bigint(20)</code>																																
<code>rowId</code>	<code>bigint(20)</code>																																							
<code>tableId</code>	<code>bigint(20)</code>																																							
<code>classPK</code>	<code>bigint(20)</code>																																							
<code>companyId</code>	<code>bigint(20)</code>																																							

The `expandocolumn` model represents the columns of the expandable model. It has a set of attributes: `columnId`, `tableId`, `name`, `type_`, `defaultData`, `typeSettings`, and `companyId`. The `typeSettings` attribute specifies properties settings, for example input box (that is, dynamic UI) height, and width, and whether it is hidden/secret/index-able or not. For instance, if we want the attribute `Ext_Desc` to have an input box with height as 20 and width as 100, to be not hidden, to be not secret and to be index-able, we would have a value: `height=100 indexable=1 hidden=0 width=20 secret=0`. In addition, the attribute `type_` specifies the data type for column data. You can find all predefined types (for example, `BOOLEAN` 1, `DATE` 3, `DOUBLE` 5, `STRING` 15, `FLOAT` 7, `INTEGER` 9, `LONG` 11, and so on.) in `com.liferay.portlet.expando.model.ExpandoColumnConstants` under the `/portal/portal-service/src` folder.

The `expandrow` model represents rows of the expandable model. It also has a set of attributes: `rowId_`, `tableId`, `classPK`, and `companyId`. The `classPK` field is used as a primary key for the Expando records. The `expandvalue` model represents the actual data of the expandable model. It has a set of attributes: `valueId`, `tableId`, `columnId`, `rowId_`, `classNameId`, `classPK`, `data_`, and `companyId`. It is associated with the following tables: `expandable`, `expandocolumn`, and `expandrow` by the foreign keys `tableId`, `columnId`, and `rowId_`, respectively. `classNameId` and `classPK` are especially used to identify the target model and the target entity. If `classNameId` has a value for `com.liferay.portal.model.User`, it means that this data is applied on the `User` model and `classPK` will be used to identify specific user, for example, Lotti Stein. In addition, the `data_` field maintains the actual data of the expandable model. In brief, this data can be typed (Boolean, date, double, integer, long, short, string, and arrays of all these basic types), associated with a specific entity (for example, `com.liferay.portal.model.User`, `com.liferay.portal.model.Organization`), arranged into any number of columns, made available to plugins (for example, portlets, theme), accessed from the Velocity templates, and accessed via JSON API through AJAX.

Generally speaking, the Expando service is a generic service which allows us to dynamically define a collection of data. Some of these services are `com.liferay.portlet.expando.service.ExpandoTableLocalService`, `com.liferay.portlet.expando.service.ExpandoColumnLocalService`, `com.liferay.portlet.expando.service.ExpandoRowLocalService`, `com.liferay.portlet.expando.service.ExpandoValueLocalService`. The Expando service also provides CRUD methods.

## Extending custom attributes

As you can see, custom attributes support the types **primitives** (Boolean, int, string, short, and so on) and **presets** (text box, selection, and so on) only. It would be nice to extend the above types and let the custom attributes support the custom types, for example, Image Gallery images and Document Library documents. By this extension, the target objects (for example, User and Organization) can have references (called **associations**) with Image Gallery images and Document Library documents via custom attributes. Moreover, the target objects (for example, Document Library documents) can have self-references (for example, Document Library documents) via the custom attributes. There is no doubt that this feature will be implemented in the near future.

## Enhancing users and organizations

We have discussed how to use the Expando services in the Velocity templates. Now we will go further and see how to use the Expando services in the portlets. For instance, we may want to extend the profile of users and organizations using the Expando services. Fortunately, the Liferay portal provides UI for adding custom attributes to users and organization forms. Meanwhile, Liferay portal also provides a framework to add custom attributes to any ServiceBuilder entity at runtime where indexed values, text boxes, and selection lists for input and dynamic UI are available.

As shown in the following screenshot, you can add custom attributes for users. That is, you have the capability to apply custom attributes on the `com.liferay.portal.model.User` entity. Logically, there is no limitation to the number of custom attributes that you can add on the `User` entity. Similarly, you can add custom attribute for organizations. That is, you have the capability to apply custom attributes on the `com.liferay.portal.model.Organization` entity. For demo purposes, we're going to add the `Ext_Desc` and `Ext_Name` custom attributes to the profile of the users.



To do so, just click on the **Add Custom Attribute** button and provide input for attribute-key and attribute-type for both `Ext_Desc` and `Ext_Name`. Note that the custom attribute key is used to access the attribute programmatically through the `<liferay-ui:custom-attribute />` JSP tag. Meanwhile, choose the attribute type carefully as once it is defined, it can't be changed. When a custom attribute is created, you can manage it as well. Unquestionably, you can edit custom attributes, set up permissions on custom attributes, or delete custom attributes. In order to edit custom attributes, first locate a custom attribute (for example, `Ext_Desc`), then click on the **Actions** icon next to the custom attribute, and then choose one action from the actions icons **Edit**, **Permissions**, and **Delete**.

If you click on the **Edit** icon from the **Actions** icon next to the `Ext_Desc`, you will see the editing custom attribute. Here you can edit the `Ext_Desc` custom attribute, for example typing default value, resetting properties of the custom attribute, and so on.

The properties of custom attributes are dynamically configurable. You can set the **Hidden** property with a `True` value. It means that the attribute's value is never shown in any user interface besides this one. This allows the attribute to be used for some more obscure and advanced purpose such as acting as a placeholder for custom permissions. You can also set the **Searchable** property with a `True` value. It means that the value of attribute will be indexed when the entity (for example, User) is modified. Note that only the attributes that have the data type `java.lang.String` can be made searchable. When an attribute is newly made searchable, the indexes must be updated before the data is available to search. The other properties involve **secrete**, **Height**, and **Width**. The properties **Height** and **Width** provide help with the dynamic view of UI for text box input. Note that you can apply localization only for the attribute name. Thus, we can localize the names by adding the key to the locale files. The values themselves can't be localized.

Once the custom attribute has been created, it will appear in the user profile within **My Account** and the **User** administration UI in the Control Panel.

## What's happening?

Generally speaking, there are four parts related to the above custom attributes: entry point of custom attribute, adding the custom attribute, updating the custom attribute, and typing values for custom attributes. We're going to look at the custom attributes in detail based on the entity `User`. The entry point of the custom attribute is specified in the JSP file `view_users.jsp` under the `/portal/portal-web/docroot/html/portal/enterprise_admin` folder as follows:

```
<liferay-util:include page="/html/portlet/enterprise_admin/user/
toolbar.jsp">
 <liferay-util:param name="toolbarItem" value="view-all" />
</liferay-util:include>
```

The code above shows that a /user/toolbar.jsp toolbar is included. In order to add/update the custom attribute, we should double-check the JSP file toolbar.jsp in the /porta/portal-web/docroot/html/portal/enterprise\_admin/user folder. You will find the following lines related to the Expando portlet:

```
<div class="lfr-portlet-toolbar">
 <!-- ignore details -->
 <c:if test="<% RoleLocalServiceUtil.hasUserRole(user.getUserId(),
 user.getCompanyId(), RoleConstants.ADMINISTRATOR,
 true) %>">
 <liferay-portlet:renderURL windowState="<%=WindowState.
 MAXIMIZED.toString() %>">
 var="expandoURL" portletName="<% PortletKeys.EXPANDO %>">
 <portlet:param name="struts_action"
 value="/expando/view" />
 <portlet:param name="redirect"
 value="<% currentURL %>" />
 <portlet:param name="modelResource"
 value="<% User.class.getName() %>" />
 </liferay-portlet:renderURL>

 <a href="<% expandoURL %>">
 <liferay-ui:message key="custom-attributes" />

 <!-- ignore details -->
 </c:if>
</div>
```

The code above shows the **Custom Attributes** button and a URL for this button. It specifies a portlet named Expando with render URL in general. It specifies a portlet render URL with the window state `WindowState.MAXIMIZED`, variable `expandoURL`, and portlet name `PortletKeys.EXPANDO`. Moreover, it specifies portlet parameters, for example `struts_action` with a value `/expando/view`, `redirect` with a value `currentURL`, and `modelResource` with a value `User.class.getName()`. Finally, it displays the portlet render URL `expandoURL` as a link. As you can see, no change is required in the Expando portlet; just create a portlet render URL.

In order to type values for a custom attribute, we should double-check the JSP file `custom_attributes.jsp` in the `/porta/portal-web/docroot/html/portal/enterprise_admin/user` folder. This JSP file will appear in the user profile within **My Account** and the **User** administration UI in the Control Panel. You will find the following lines related to typing values:

```
<% User selUser = (User)request.getAttribute("user.selUser"); %>
<h3>
 <liferay-ui:message key="custom-attributes" />
 AAA
</h3>
<fieldset class="block-labels">
 <liferay-ui:custom-attribute-list
 className="com.liferay.portal.model.User"
 classPK="<%=(selUser != null) ? selUser.getUserId() : 0 %>"
 editable="<%=true %>" label="<%=true %>" />
</fieldset>
```

The code above shows the JSP tag `custom-attribute-list` for typing values of custom attributes. This JSP tag uses `className`, `classPK`, `editable`, and `label` as its parameters.

## Sharing the Expando portlet

As stated above, the Expando portlet can be applied for custom attributes. Now we will see how to share this portlet. You can find the following lines related to the definition of the Expando portlet in the XML file `/portal/portal-web/docroot/WEB-INF/portlet-custom.xml`:

```
<portlet>
 <portlet-name>139</portlet-name>
 <display-name>Expando</display-name>
 <portlet-class>com.liferay.portlet.StrutsPortlet</portlet-class>
 <expiration-cache>0</expiration-cache>
 <supports><mime-type>text/html</mime-type></supports>
 <resource-bundle>
 com.liferay.portlet.StrutsResourceBundle
 </resource-bundle>
</portlet>
```

The code above shows a portlet with the portlet name 139 and the display name as Expando. The portlet class is `com.liferay.portlet.StrutsPortlet`. This means that this portlet is a Struts portlet. This portlet has been registered by using the following lines in `/portal/portal-web/docroot/WEB-INF/liferay-portlet.xml`:

```
<portlet>
 <portlet-name>139</portlet-name>
 <icon>/html/icons/default.png</icon>
 <struts-path>expando</struts-path>
 <use-default-template>false</use-default-template>
 <restore-current-view>false</restore-current-view>
 <private-request-attributes>false</private-request-attributes>
 <private-session-attributes>false</private-session-attributes>
 <render-weight>50</render-weight>
 <header-portlet-css>
 /html/portlet/expando/css.jsp
 </header-portlet-css>
 <css-class-wrapper>portlet-expando</css-class-wrapper>
 <add-default-resource>true</add-default-resource>
 <system>true</system>
</portlet>
```

The code above shows that the Expando portlet has been registered with the portlet name as 139 and Struts path as expando. More specifically, it sets `header-portlet-css` with a value `/html/portlet/expando/css.jsp`. `header-portlet-css` is used to set the path of CSS, which will be referenced in the page's header relative to the portlet's context path. It also sets the `system` value to `true`. It means that the portlet is a system-level portlet that users cannot manually add to any page.

The portlet action is specified in the Java file `com.liferay.portlet.expando.action.EditExpandoAction` under the `/portal/portal-impl/src` folder. It extends `com.liferay.portal.struts.PortletAction`. Moreover, the Struts action uses this portlet action as shown in the following lines in `/portal/portal-web/docroot/WEB-INF/struts-config.xml`:

```
<action path="/expando/edit_expando"
 type="com.liferay.portlet.expando.action.EditExpandoAction">
 <forward name="portlet.expando.edit_expando"
 path="portlet.expando.edit_expando" />
 <forward name="portlet.expando.error"
 path="portlet.expando.error" />
</action>
<action path="/expando/view" forward="portlet.expando.view" />
```

The preceding code shows the `expando/edit_expando` and `/expando/view` action paths of the Expando portlet. The page flows are defined in `/portal/portal-web/docroot/WEB-INF/tiles-defs.xml`:

```
<definition name="portlet.expando" extends="portlet" />
<definition name="portlet.expando.edit_expando"
 extends="portlet.expando">
 <put name="portlet_content"
 value="/portlet/expando/edit_expando.jsp" />
</definition>
<definition name="portlet.expando.view"
 extends="portlet.expando">
 <put name="portlet_content"
 value="/portlet/expando/view.jsp" />
</definition>
```

The code above shows the page flow of the Expando portlet. It forwards `portlet.expando.edit_expando` to `/portlet/expando/edit_expando.jsp`, and `portlet.expando.view` to `/portlet/expando/view.jsp`. In addition, you can find the JSP files `view.jsp`, `edit_expando.jsp` and other related files; for example, `css.jsp`, `error.jsp`, `init.jsp` in the `/portal/portal-web/docroot /html/portlet/expando` folder.

In short, Liferay portal provides the capability to manage custom attributes. Custom attributes can be applied to any entity generated by ServiceBuilder, whereas the Expando portlet<sup>139</sup> can be used to manage them. No changes are required to the Expando portlet; just create `portletURL` as we can find it in the User and Organization management. Custom attributes are implemented using Expando services, for example `ExpandoTableLocalService`, `ExpandoColumnLocalService`, `ExpandoRowLocalService`, and `ExpandoValueLocalService`. This means that all the data introduced in custom attributes is safely stored within the database and fully indexed using the Lucene search engine.

## Building OpenSearch

Liferay portal provides OpenSearch capability via search portlet. OpenSearch is a collection of simple formats for sharing search results. As shown in the following screenshot, enter the keyword 'alfresco' and you will see search results coming from the Document Library, Image Gallery, Blogs, Bookmarks, Directory, Message Boards, Wikis, and Web Content portlets:

The screenshot shows a search interface with a search bar containing 'alfresco'. Below the search bar is a button 'Add Liferay as a Search Provider'. To the right is a 'Return to Full Page' link. The results are divided into two sections: 'Document Library' and 'Blogs'. The 'Document Library' section has a header '# Summary Tags' and contains two items: '1. activision cms alfresco' and '2. activision cms alfresco result'. The 'Blogs' section has a header '# Summary Tags' and displays the message 'No results were found that matched the keywords: alfresco.'

As you can see, search results share a simple format. That is, search results with the same formats are against Document Library, Image Gallery, Blogs, Bookmarks, Directory, Message Boards, Wikis, and Web Content. How do you add the OpenSearch capability on custom portlets? In this section, we're going to answer these questions related to search and OpenSearch.

## What's happening?

In fact, Liferay portal supports OpenSearch standard (or called federated search). That is, it returns search results from multiple content sources including Liferay portlets and external integrated applications. The **OpenSearch** interface has been specified in `com.liferay.portal.kernel.search.OpenSearch` under the `/portal/portal-kernel/src/` folder with the following code:

```
public interface OpenSearch {
 public boolean isEnabled();
 public String search(HttpServletRequest request, String url)
 throws SearchException;
 public String search(HttpServletRequest request, String keywords,
 int startPage, int itemsPerPage, String format) throws
 SearchException;
}
```

The code above shows an interface `OpenSearch`. It specifies the `isEnabled` method to either enable or disable the OpenSearch capability first. Then it specifies two search methods with different parameters. Thus, we can use the search method either by the URL parameter or by the parameters `keywords`, `startPage`, `itemsPage`, and `format` based on different requirements. OpenSearch is a collection of simple formats for the sharing of search results, which is suitable for syndication and aggregation, and a way for web sites and search engines to publish search results in a standard and accessible format. Refer to <http://www.opensearch.org> for more information.

As you can see, the abstract class `com.liferay.portal.search.BaseOpenSearchImpl` in the `/portal/portal-impl/src/` folder implements the `com.liferay.portal.kernel.search.OpenSearch` interface. The `com.liferay.portlet.directory.util.DirectoryOpenSearchImpl` class goes further to extend `com.liferay.portal.search.BaseOpenSearchImpl`. For this reason, the Search portlet includes the search results from the Directory portlet.

The `com.liferay.portal.search.HitsOpenSearchImpl` abstract class extends the `com.liferay.portal.search.BaseOpenSearchImpl` abstract class. It adds the `getHits`, `getSearchPath`, and `getTitle` methods and, moreover, it overrides the `search` method. In addition, a set of classes extend the `com.liferay.portal.search.HitsOpenSearchImpl` abstract class some of which are `com.liferay.portal.wiki.util.WikiOpenSearchImpl`, `com.liferay.portlet.blogs.util.BlogsOpenSearchImpl`, `com.liferay.portlet.journal.util.JournalOpenSearchImpl`, and so on. Note that `HitsOpenSearchImpl` is different from `BaseOpenSearchImpl` as it extends `BaseOpenSearchImpl` and adds search hits (for example, start locations, search time, length, scores, and so on). You can refer to `com.liferay.portal.kernel.search.Hits` in the `/portal/portal-kernel/src` folder.

Further, the `OpenSearch` implementation must be registered via the `open-search-class` tag in `liferay-portlet.xml` under the `/portal/portal-web/docroot/WEB-INF` folder. The `open-search-class` value must be a class that implements `com.liferay.portal.kernel.search.OpenSearch` and is called to get search results in the `OpenSearch` standard. The following code shows the `OpenSearch` registration of the Web Content portlet:

```
<portlet>
 <portlet-name>15</portlet-name>
 <icon>/html/icons/default.png</icon>
 <struts-path>journal</struts-path>
 <configuration-action-class>
 com.liferay.portlet.journal.action.ConfigurationActionImpl
 </configuration-action-class>
 <indexer-class>
 com.liferay.portlet.journal.util.Indexer
 </indexer-class>
 <open-search-class>
 com.liferay.portlet.journal.util.JournalOpenSearchImpl
 </open-search-class>
 s<!--ignore details -->
</portlet>
```

The code above describes the `open-search-class` tag with the `com.liferay.portlet.journal.util.JournalOpenSearchImpl` value, thus implementing `com.liferay.portal.kernel.search.OpenSearch` indirectly.

Similarly, you can find the OpenSearch registration of the Wikis, Blogs, Document Library, Image Gallery, Bookmarks, and Message Boards portlets in `liferay-portlet.xml`.

## Adding the OpenSearch capability on custom portlets

How to add the OpenSearch capability on custom portlets? In general, there are two steps to add the OpenSearch capability on custom portlets:

1. Prepare a class that implements the OpenSearch interface.
2. Register the class as the value of the `open-search-class` tag in `liferay-portlet.xml`.

Let's take a look at the OpenSearch capability using an example:

`alfresco-content-portlet`. Check out the `alfresco-content-portlet` portlet from `svn://svn.liferay.com/repos/public/plugins/trunk/portlets/alfresco-content-portlet` to `$PLUGINS_SDK_HOME/portlets/alfresco-content-portlet`. As mentioned earlier, all portlets stay in the `$PLUGINS_SDK_HOME/portlets` folder, and `$PLUGINS_SDK_HOME` is the current plugins SDK home.

Refresh the `$PLUGINS_SDK_HOME` project and you will see the `alfresco-content-portlet` folder in `$PLUGINS_SDK_HOME/portlets/`. You will see that a `com.liferay.alfrescocontent.util.AlfrescoOpenSearchImpl` class in the `$PLUGINS_SDK_HOME/portlets/alfresco-content-portlet/docroot/WEB-INF/src` folder implements `com.liferay.portal.kernel.search.OpenSearch` as follows:

```
public class AlfrescoOpenSearchImpl implements OpenSearch {
 // ignore details
 public boolean isEnabled() { return ENABLED; }
 public String search(HttpServletRequest request, String url)
 throws SearchException {
 String xml = StringPool.BLANK;
 if (!ENABLED) {
 if (_log.isDebugEnabled()) {
 _log.debug("Search is disabled");
 }
 return xml;
 }
 if (_log.isDebugEnabled()) {
 _log.debug("Search with " + url);
 }
 try {
 xml = HttpUtil.URLtoString(url, HOST, PORT, REALM,
```

```
 USERNAME, PASSWORD) ;
 }
 catch (IOException ioe) {
 _log.error("Unable to search with " + url, ioe);
 }
 return xml;
}
public String search(
 HttpServletRequest request, String keywords, int startPage,
 int itemsPerPage, String format) throws SearchException {
 String url =
PROTOCOL + ":" + SEARCH_URL + "?q=" +
 HttpUtil.encodeURL(keywords) + "&p=" +
 startPage + "&c=" +
 itemsPerPage +
 "&guest=&format=" + format;
 return search(request, url);
}
}
```

The code above describes the `AlfrescoOpenSearchImpl` class which implements `OpenSearch`. The methods which have been specified and implemented in the `OpenSearch` interface are `isEnabled` and `search`. Then, you need to add the `AlfrescoOpenSearchImpl` class as the value of the `open-search-class` tag in `liferay-portlet.xml` under the `$PLUGINS_SDK_HOME/portlets/alfresco-content-portlet/docroot/WEB-INF` folder. Eventually, you are not required to add this, as it already exists in `liferay-portlet.xml` as follows:

```
<portlet>
 <portlet-name>l</portlet-name>
 <icon>/icon.png</icon>
 <configuration-action-class>
 com.liferay.alfrescocontent.action.ConfigurationActionImpl
 </configuration-action-class>
 <open-search-class>
 com.liferay.alfrescocontent.util.AlfrescoOpenSearchImpl
 </open-search-class>
 <layout-cacheable>true</layout-cacheable>
 <instanceable>true</instanceable>
 <render-weight>1</render-weight>
 <header-portlet-css>/portlet.css</header-portlet-css>
 <css-class-wrapper>alfresco-content-portlet</css-class-wrapper>
</portlet>
```

The preceding code displays the registration of the alfresco-content-portlet portlet, including the portlet name as 1, instanceable as true, the open-search-class tag with a value com.liferay.alfrescocontent.util.AlfrescoOpenSearchImpl, and so on.

## Adding search capabilities in portlets

How to add search capabilities to a portlet? Let's take the Blogs portlet as an example. Generally speaking, there are three steps in which you can add search capabilities to the Blogs portlet. First, create a com.liferay.portlet.blogs.util.Indexer class in the /portal/portal-impl/src folder, which implements the com.liferay.portal.kernel.search.Indexer interface under the /portal/portal-kernel/src folder. Here you do not need to create the Indexer class, as it exists by default. For your custom portlet, you need to create an Indexer class in a similar way. This class is responsible for adding, updating, and deleting documents to the index. It uses com.liferay.portal.kernel.search.SearchEngineUtil, which is an abstraction to the underlying search engine.

Then, to register the Indexer class, just add the indexer-class tag in liferay-portlet.xml pointing to the com.liferay.portlet.blogs.util.Indexer class. Again, you do not need to add the Indexer class eventually, as it exists by default. Whenever a blog entry is added, updated, or deleted from the database, the Indexer class is called to update the index accordingly. Note that web content can be added, updated, or removed from the index. Thus, you are able to make search requests to SearchEngineUtil in com.liferay.portlet.blogs.util.Indexer as follows:

```
public static void updateEntry(long companyId, long groupId, long
 userId, String userName, long entryId, String title, String content,
 Date displayDate, String[] tagsEntries, ExpandoBridge expandoBridge)
throws SearchException {
 Document doc = getEntryDocument(companyId, groupId, userId,
 userName, entryId, title, content, displayDate,
 tagsEntries, expandoBridge);
 SearchEngineUtil.updateDocument(companyId,
 doc.get(Field.UID), doc);
}
```

The code above exhibits a mechanism for re-indexing search indexes. When re-indexing is active, this code calls SearchEngineUtil.updateDocument to update the search indexes of the search documents.

Finally, you need to add the search method in the JSP file `search.jsp` under the `/portal/portal-web/docroot/html/portlet/blogs` folder. This JSP file calls the `search` method and iterates over the `com.liferay.portal.kernel.search.Hits` class, which is a collection of documents. It represents the result of the search. The `Hits` class gives the score (that is, the relevance) of each document, tells us how long the search took, and how many documents were found.

## Using Solr for enterprise search

Liferay portal supports pluggable search engines and the current implementation uses the open source search engine **Solr**. This allows us to use a completely separate product which can be installed on any application server. The search engine then operates completely independently of the portal nodes in a clustered environment, and acts as a search service for all of the nodes simultaneously.

This will solve the problem in a cluster when sharing **Lucene** indexes. As you can see, you can share one search index among all of the nodes of the cluster without worrying about putting it in a database, or maintaining separate search indexes on all of the nodes. Each portal node will send requests to the search engine in order to update the search index when in need, and then these updates are queued and handled automatically by the search engine.

In brief, Solr is an Apache open source search server based on Lucene, but with a more advanced feature set. It is an enterprise search server based on the Lucene Java search library with XML/HTTP and JSON APIs, caching, replication, a web administration interface, hit highlighting, faceted search, and so on. Refer to <http://lucene.apache.org/solr> for more information.

As Solr is a standalone search engine, we're going to use it as plugin webs (web applications). To do so, first check out webs `solr-web` from `svn://svn.liferay.com/repos/public/plugins/trunk/webs/solr-web` to `$PLUGINS_SDK_HOME/webs/solr-web`, while `$PLUGINS_SDK_HOME` is the current Plugins SDK home. Refresh the `$PLUGINS_SDK_HOME` project and you will see the `solr-web` folder under the `$PLUGINS_SDK_HOME/webs` folder.

Locate the `solr-web` folder and you will find a file called `solr-spring.xml` in the `/docroot/WEB-INF/src/META-INF` folder. Open this file and you will see that there is an entry which defines where the Solr server can be found by Liferay portal:

```
<bean id="solrServer"
 class="org.apache.solr.client.solrj.impl.CommonsHttpSolrServer">
 <constructor-arg type="java.lang.String"
 value="http://localhost:8080/solr" />
</bean>
```

The preceding code shows where the Solr server is running. You should modify this value so that it points to the server upon which you are running Solr. You can also find the XML file schema.xml in the \$PLUGINS\_SDK\_HOME/webs/solr-web/docroot/WEB-INF/conf folder. This file describes how the fields will be indexed to the Solr index. You can go further and find the classes, for example com.liferay.portal.search.solr.SolrSearchEngineUtil and com.liferay.portal.search.solr.messaging.SolrReaderMessageListener in the \$PLUGINS\_SDK\_HOME/webs/solr-web/docroot/WEB-INF/src folder.

Afterwards, you can drop the build.xml file from \$PLUGINS\_SDK\_HOME/webs/solr-web to the Ant view; and then double-click on the deploy target under web in the Ant view.

## Overriding the Spring services

Liferay portal uses the Spring framework and keeps the Spring configuration. In this section, we're going to introduce how to provide a custom implementation of a Liferay service by using Spring and how to apply the extension's ready-to-use mechanisms to ease this task.

The Spring framework is a support structure in which another software project can be organized and developed. It minimizes dependency of the application components by providing a plugin architecture. The Spring framework is very powerful as it combines **Dependency Injection (DI)**, **Inversion of Control (IoC)**, and **Aspect Oriented Programming (AOP)**. The DI approach provides an instance of the dependent object at runtime by an external process. Spring implements this via an XML file, which defines the dependencies between objects. IoC occurs via **Dependency Injection**, and these two terms are interchangeable. AOP is the ability to modularize functionality. AOP is generally used for Transaction management and logging—code litter, things unrelated to code.

Spring handles the middle tier—the Liferay portal service layer—using IOC and AOP. Spring also handles Liferay portal business and data service layers (for example, implementation classes and Hibernate injections). The portal utilizes Spring for its transaction management: org.springframework.orm.hibernate3.HibernateTransactionManager.

## Overriding method validation

Let's consider the use case *Overriding method validate*. `OrganizationLocalService` has its own validation method for creating or updating organizations. Now, we're to invoke the parent to keep the existing validation and add a new validation condition—saying that the names of organizations must be in uppercase.

How to implement this use case? We should implement this by overriding the implementation of `OrganizationLocalService` to change the validation code for organizations. The following are main steps of the process.

First, locate a package named `com.ext.portal.service.impl` in the `/ext/ext-impl/src` folder. This package will hold our custom implementation of `OrganizationLocalService`. Of course, you can have a different package name; we're using this just for ease of reference. Then create a custom implementation class named `OrganizationLocalServiceImpl` in the `com.ext.portal.service.impl` package, which extends `com.liferay.portal.service.impl.OrganizationLocalServiceImpl` as follows:

```
public class OrganizationLocalServiceImpl extends
 com.liferay.portal.service.impl.OrganizationLocalServiceImpl {
 protected void validate(long companyId, long organizationId,
 long parentOrganizationId, String name, String type,
 long countryId, int statusId) throws PortalException,
 SystemException {
 if (_log.isDebugEnabled()) {
 _log.debug("Executing my custom organization validation!");
 }
 super.validate(companyId, organizationId, parentOrganizationId,
 name, type, countryId, statusId);
 if (!name.equals(name.toUpperCase())) {
 throw new OrganizationNameException("The name must be upper
 case");
 }
 }
 private static Log _log = LogFactory.getLog(
 OrganizationLocalServiceImpl.class);
}
```

The preceding code shows how to override the implementation of `OrganizationLocalService` to change the validation code for organizations. It first inherits the default implementation of `com.liferay.portal.service.impl.OrganizationLocalServiceImpl`. Then it overrides the `validate` method, invokes the parent to keep the existing validation, and adds a new validation condition. Finally, we need to update the `com.liferay.portal.service.OrganizationLocalService.impl` bean definition. By default, the `com.liferay.portal.service.OrganizationLocalService.impl` bean has been defined in the XML file `portal-spring.xml` under the `/portal/portal-impl/src/META-INF` folder as follows:

```
<beans>
 <!-- ignore details -->
 <bean id="com.liferay.portal.service.OrganizationLocalService.impl"
 class="com.liferay.portal.service.impl.
 OrganizationLocalServiceImpl" />
</beans>
```

The code above displays the `com.liferay.portal.service.OrganizationLocalService.impl` bean default definition. The class of the bean is set as `com.liferay.portal.service.impl.OrganizationLocalServiceImpl`. Now we need to override the definition of the `com.liferay.portal.service.OrganizationLocalService.impl` bean. To do so, locate the XML file `ext-spring.xml` in the `/ext/ext-impl/src/META-INF` folder and add the following bean definition before the `</beans>` tag of `ext-spring.xml`:

```
<bean id="com.liferay.portal.service.OrganizationLocalService.impl"
 class="com.ext.portal.service.impl.
 OrganizationLocalServiceImpl" />
```

The code exhibits how the bean for `OrganizationLocalService` implementation has been re-defined with custom class `com.ext.portal.service.impl.OrganizationLocalServiceImpl`.

That's it. You can verify what you have done above. Stop Tomcat, run targets `clean`, `deploy in Ext`, and `restart Tomcat`. Then log in the portal, and using Control Panel create a new organization and verify that the new condition is enforced properly. In the same way, you can override the other Spring services, for example `UserLocalService`, `JournalArticleLocalService`, `DLLocalService`, `IGIImageLocalService`, `BlogsEntryLocalService`, `WikiPageLocalService`, and `MBMessageLocalService`.

## Changing model name via ServiceBuilder

Now let's go further. Consider the use case *Changing model name*. As mentioned earlier, we have applied the XML file `service.xml` in the `com.ext.portlet.tasks` package that is in the `/ext/ext-impl/src` folder to represent the workflow model. Inside the XML file `service.xml`, the code tells the ServiceBuilder what the package-path is. For example, we have package-path for the workflow model as follows:

```
<!-- ignore details -->
<service-builder package-path="com.ext.portlet.tasks">
 <namespace>Tasks</namespace>
 <!-- ignore details -->
</service-builder>
```

The code above shows the definition of a workflow model, which is called `tasks` here. It specifies the package path as `com.ext.portlet.tasks` and namespace as `Tasks`.

After running the `build-service-portlet-extTasks` target under `ext-impl` of the Ant view, ServiceBuilder will generate models (both local and remote services) and implementations. In general, ServiceBuilder can be used to build Java services that can be accessed in a variety of ways, including local access from Java code, remote access using web services, and so on. As you can see, the `com.ext.portlet.tasks`, `com.ext.portlet.tasks.model`, `com.ext.portlet.tasks.service`, and `com.ext.portlet.tasks.persistence` packages, with models and service interfaces, are generated in the `/ext/ext-service/src` folder. The `com.ext.portlet.tasks.model.impl`, `com.ext.portlet.tasks.service.base`, `com.ext.portlet.tasks.service.http`, `com.ext.portlet.tasks.service.impl`, and `com.ext.portlet.tasks.service.persistence` packages, with implementation of models and services, are generated in the `/ext/ext-impl/src` folder. Moreover, Hibernate object-relationship mapping is specified in the XML file `ext-hbm.xml` under the `/ext/ext-impl/src/META-INF` folder, where you can find model implementation with package name as `com.ext.portlet.tasks.model.impl`. Similarly, you can find model hints in the XML file `ext-model-hints.xml` under the `/ext/ext-impl/src/META-INF` folder where you can find models with package name as `com.ext.portlet.tasks.model`. Further, you can find beans definition in the XML file `ext-spring.xml` under the `/ext/ext-impl/src/META-INF` folder where you can find Dependency Injection among models, services, and their implementation.

Now, for some reason, let's suppose we have to change the model name from the current tasks to the workflow target. For example, we have the package-path target and namespace for the workflow model in the XML file service.xml as follows:

```
<!-- ignore details -->
<service-builder package-path="com.ext.portlet.workflow">
 <namespace>Workflow</namespace>
 <!-- ignore details -->
</service-builder>
```

The code above shows the definition of workflow model, here called `Workflow`. It specifies the package path as `com.ext.portlet.workflow` and namespace as `Workflow`. Eventually, you can use a different target name for this use case. How to do it? Here we list the required steps according to best practices:

1. Delete the packages `com.ext.portlet.tasks`, `com.ext.portlet.tasks.model`, `com.ext.portlet.tasks.service`, and `com.ext.portlet.tasks.persistence`, with models and service interfaces, in the `/ext/ext-service/src` folder.
2. Delete the packages `com.ext.portlet.tasks.model.impl`, `com.ext.portlet.tasks.service.base`, `com.ext.portlet.tasks.service.http`, `com.ext.portlet.tasks.service.impl`, and `com.ext.portlet.tasks.service.persistence`, with implementation of models and services, in the `/ext/ext-impl/src` folder.
3. Remove Hibernate object-relationship mapping of tasks specified in the XML file `ext-hbm.xml` under the `/ext/ext-impl/src/META-INF` folder.
4. Remove the model hints of tasks in the XML file `ext-model-hints.xml` under the `/ext/ext-impl/src/META-INF` folder, and remove the beans definitions of tasks from the XML file `ext-spring.xml` under the `/ext/ext-impl/src/META-INF` folder.

Afterwards, you may run the `build-service-portlet-extTasks` target under `ext-impl` of the Ant view to generate both local and remote services, and implementations of `Workflow`. To find out why we need the above processes, you can refer to next section.

## What's happening?

Liferay portal keeps the Spring configuration in the following files under the /portal/portal-impl/src/META-INF folder:

- portal-spring.xml: This contains all of the definitions of the service implementations. It also contains the transaction properties of such services.
- jcr-spring.xml: This specifies the factory that selects the JCR (JSR-170) implementation. The default factory uses Jackrabbit.
- dynamic-data-source-spring.xml: This defines the data source and the transaction manager.
- Others: base-spring.xml, counter-spring.xml, document-library-spring.xml, hibernate-spring.xml, infrastructure-spring.xml, lock-spring.xml, mail-spring.xml, management-spring.xml, messaging-spring.xml, migrate-spring.xml, portlet-container-spring.xml, scheduler-spring.xml, search-spring.xml, util-spring.xml, wsrp-spring.xml, and so on.

Liferay portal's default Spring configuration files should not be modified directly. Instead, any bean definition can be overridden in a special file called ext-spring.xml in the /ext/ext-impl/src/META-INF folder. The overriding order is specified in portal.properties under the /portal/portal-impl/src folder as follows:

```
spring.configs=\
META-INF/base-spring.xml,META-INF/hibernate-spring.xml, \
META-INF/infrastructure-spring.xml,
META-INF/management-spring.xml, \
META-INF/util-spring.xml,META-INF/jcr-spring.xml, \
META-INF/messaging-spring.xml,META-INF/scheduler-spring.xml, \
META-INF/search-spring.xml,META-INF/counter-spring.xml, \
META-INF/document-library-spring.xml, \
META-INF/lock-spring.xml,META-INF/mail-spring.xml, \
META-INF/portal-spring.xml, \
META-INF/portlet-container-spring.xml, \
META-INF/wsrp-spring.xml,META-INF/mirage-spring.xml, \
META-INF/ext-spring.xml
```

The code above displays the Spring configuration in the portal. The above XML files will be loaded via the contextClass parameter with the PortalApplicationContext value after the bean definitions specified in the contextConfigLocation parameter in web.xml. By default, the value of contextConfigLocation is empty. As you can see, the last XML file is META-INF/ext-spring.xml. This is the reason we can override the validate method of OrganizationLocalService for the use case *Overriding method validate* in the XML file ext-spring.xml.

Liferay portal also keeps the Hibernate configurations in the following files under the /portal/portal-impl/src/META-INF folder:

- counter-hbm.xml: This contains Hibernate object-relationship mappings for the model com.liferay.counter.model.Counter
- mail-hbm.xml: This contains Hibernate object-relationship mappings for the com.liferay.mail.model.CyrusUser and com.liferay.mail.model.CyrusVirtual models
- portal-hbm.xml: This contains Hibernate object-relationship mappings for portal models, for example com.liferay.portal.model.impl.ContactImpl, com.liferay.portal.model.impl.UserImpl, and so on

Liferay portal default Hibernate configuration files should never be modified. Instead, any custom Hibernate object-relationship mappings can be specified in ext-hbm.xml in the /ext/ext-impl/src/META-INF folder. The loading order is specified in portal.properties under the /portal/portal-impl/src folder as follows:

```
hibernate.configs=\nMETA-INF/counter-hbm.xml,\nMETA-INF/mail-hbm.xml,\nMETA-INF/portal-hbm.xml,\nMETA-INF/ext-hbm.xml
```

The code above makes the Hibernate configuration in the portal visible. Obviously, custom Hibernate object-relationship mappings are specified in the XML file ext-hbm.xml.

Finally, Liferay portal also keeps model hints configurations in the XML file portal-model-hints.xml under the /portal/portal-impl/src/META-INF folder. portal-model-hints.xml contains model hints for portal models, for example com.liferay.portal.model.Contact, com.liferay.portal.model.User, and so on. Custom model hints in Ext can be specified in ext-model-hints.xml in the /ext/ext-impl/src/META-INF folder; whereas custom model hints in Plugins SDK can be specified in the XML file portlet-model-hints.xml, which is in the /docroot/WEB-INF/src/META-INF folder. For example, for the WOL portlet mentioned earlier, you can find the XML file portlet-model-hints.xml in the \$PLUGINS\_SDK\_HOME/portlets/wol-portlet/docroot/WEB-INF/src/META-INF folder.

The model hints that configuration has been specified in portal.properties under the /portal/portal-impl/src folder as follows. Note that model hints are used to adjust database column sizes during the build phase.

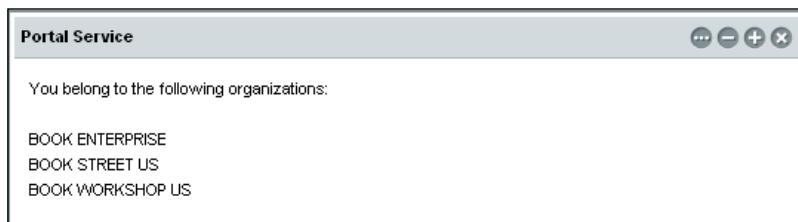
```
model.hints.configs=\nMETA-INF/portal-model-hints.xml,\nMETA-INF/ext-model-hints.xml,\nMETA-INF/portlet-model-hints.xml
```

The preceding code exhibits the model hints configuration in the portal. As you can see, custom model hints are specified in the XML file `ext-model-hints.xml`.

Obviously, we have an answer for the use case *Changing model name*. In order to change a model name, we need to delete the previous models, services, and their implementation, on the one hand. On the other hand, we need to remove the Hibernate object-relationship mapping of previous model name from the XML file `ext-hbm.xml`, the beans definitions from the XML file `ext-spring.xml`, and model hints from `ext-model-hints.xml`.

## Consuming Liferay services in portlets

Here we will go further to show how to consume Liferay services through regular Java calls. As shown in the following screenshot, we plan to show the organizations that the current user belongs to through regular Java calls. Suppose the current user is "Test Test"—an admin of the portal. We also created an organization **BOOK ENTERPRISE**, and two locations **BOOK STREET US** and **BOOK WORKSHOP US** with the parent organization **BOOK ENTERPRISE**. Now assign the user "Test Test" to this organization and these locations. Using regular Java calls `OrganizationServiceUtil` and `Organization` model, we can easily find the organizations the current user belongs to in the **Portal Service** portlet.



## How does it work?

First of all, we could get the **Portal Service** portlet as a plugin portlet, via checking out the portlet `portal-service-portlet` from `svn://svn.liferay.com/repos/public/plugins/trunk/portlets/sample-portal-service-portlet` to `$PLUGINS_SDK_HOME/portlets/portal-service-portlet`. Refresh the `$PLUGINS_SDK_HOME` project, and you will see the `portal-service-portlet` folder under the `$PLUGINS_SDK_HOME/portlets` folder.

Then we need to change display name and portlet info as stated in the above screenshot. To do so, locate the XML file `portlet.xml` in the `/portal-service-portlet/docroot/WEB-INF` folder. Open it, set the display name, title, short title, and keywords as `Portal Service`, and save it.

Finally, we need to deploy this portlet to the portal—locate the XML file `build.xml` in the `/portal-service-portlet` folder and drop it to the Ant view. You can see `portlet` in the Ant view. Then expand `portlet` and you will see the targets `clean`, `compile`, `deploy`, and so on. Double-click on the target `deploy` and you can get the Portal Service portlet hot-deployed. In addition, you can view details of regular Java calls in the JSP file `view.jsp` under the `/portal-service-portlet/docroot` folder. As shown in the following lines, you will see model and service, which are used for the Portal Service portlet:

```
<%@ taglib uri="http://liferay.com/tld/theme" prefix="liferay-theme"
%
<%@ page import="com.liferay.portal.model.Organization" %
<%@ page import="com.liferay.portal.service.OrganizationServiceUtil" %
%
<%@ page import="java.util.List" %> <liferay-theme:defineObjects />
You belong to the following organizations:

<% List organizations = OrganizationServiceUtil.getUserOrganizations(
 themeDisplay.getUserId());
for (int i = 0; i < organizations.size(); i++) {
 Organization organization = Organization)organizations.get(i); %
 <%= organization.getName() %>

<% } %>
```

The code above shows how to consume Liferay services through regular Java calls. These services include `com.liferay.portal.service.OrganizationServiceUtil` and the model involves `com.liferay.portal.model.Organization`. Similarly, you can use other services, for example `com.liferay.portal.service.UserServiceUtil`, `com.liferay.portal.service.GroupServiceUtil`; and models, for example `com.liferay.portal.model.User`, `com.liferay.portal.model.Group`. Of course, you can find other services and models—locate the `com.liferay.portal.service` package in the `/portal/portal-service/src` folder, and you will find services. In the same way, locate the `com.liferay.portal.model` package in the `/portal/portal-service/src` folder and you will find models.

What's the difference between `*LocalServiceUtil` and `*ServiceUtil`? `*` represents models, for example `Organization`, `User`, `Group`, and so on. Generally speaking, `*Service` is the remote service interface, which defines the service methods available to remote code. `*ServiceUtil` is a facade class that combines the service locator with the actual call to the service `*Service`. While `*LocalService` is the internal service interface, `*LocalServiceUtil` is a facade class that combines the service locator with the actual call to the service `*LocalService`.

## Customizing friendly URL mappings

We developed Social Office portlets earlier. One of them was the Profiles portlet. For this portlet, we defined Friendly URL Mapping `com.book.so.profiles.portlet.ProfilesFriendlyURLMapper`, which extended `com.liferay.portal.kernel.portlet.BaseFriendlyURLMapper`. A method named `populateParams` was overridden, where a custom mapping from Friendly URL to portlet URL was provided in the `ProfilesFriendlyURLMapper` class for the profiles portlet as follows:

```
public void populateParams(String friendlyURLPath, Map<String, String[]> params) {
 int x = friendlyURLPath.indexOf("/", 1);
 int y = friendlyURLPath.indexOf("/", x + 1);
 if (y == -1) { y = friendlyURLPath.length(); }
 String jspPage = friendlyURLPath.substring(x + 1, y);
 if (Validator.isNull(jspPage)) { return; }
 addParam(params, "p_p_id", _PORTLET_ID);
 addParam(params, "p_p_lifecycle", "2");
 addParam(params, "p_p_state", WindowState.NORMAL);
 addParam(params, "p_p_mode", PortletMode.VIEW);
 addParam(params, "jspPage", "/jsp/profiles/" + jspPage + ".jsp");
}
```

The code above exhibits a method named `populateParams` which tells us how to transfer the parameters of friendly URL into a portlet URL. The purpose of this method is to take the information out of the friendly URL Path and place it into the `Map params` parameter for using later. At the same time, we have registered the profiles portlet with the `friendly-url-mapper-class` tag as follows:

```
<portlet>
 <portlet-name>4</portlet-name>
 <friendly-url-mapper-class>
 com.book.so.profiles.portlet.ProfilesFriendlyURLMapper
 </friendly-url-mapper-class>
 <header-portlet-javascript>
 /javascripts/javascript.js
 </header-portlet-javascript>
 <css-class-wrapper>so-portlet-profiles</css-class-wrapper>
</portlet>
```

As shown in the code above, the profiles portlet (with ID 4) has a value `com.book.so.profiles.portlet.ProfilesFriendlyURLMapper` for the `friendly-url-mapper-class` tag. The logic behind the ability to convert back and forth between a portlet URL and a friendly URL has been built by the `ProfilesFriendlyURLMapper` class.

## What's happening?

Eventually, the portal provides an ability to convert back and forth between a portlet URL and a friendly URL via an interface `com.liferay.portal.kernel.portlet.FriendlyURLMapper`. You can find this interface in the `/portal/portal-kernel/src` folder:

```
public interface FriendlyURLMapper {
 public String buildPath(LiferayPortletURL portletURL);
 public String getMapping();
 public boolean isCheckMappingWithPrefix();
 public void populateParams(String friendlyURLPath, Map<String,
 String[]> params);
}
```

The code above shows the logic of being able to convert back and forth between portlet URL and friendly URL. The `buildPath` method takes in a portlet URL as input and returns a friendly URL; whereas the `populateParams` method takes in a friendly URL path as input and manipulates it into a portlet URL.

In `liferay-portlet.xml`, there is a mapping for the `friendly-url-mapper-class` tag. The `friendly-url-mapper-class` value must be a class that implements `com.liferay.portal.kernel.portlet.FriendlyURLMapper`. Use this if the content inside a portlet needs to have a friendly URL.

Liferay portal currently uses this interface for many of out of the box portlets, including Blogs, Journal, Asset Publisher, Message Boards, Wiki, Software Category, and Tags. The `com.liferay.portal.kernel.portlet.BaseFriendlyURLMapper` abstract class implements the above `com.liferay.portal.kernel.portlet.FriendlyURLMapper` interface. An individual portlet will extend the `BaseFriendlyURLMapper` abstract class. For example, the Blogs portlet has the `com.liferay.portal.blogs.BlogsFriendlyURLMapper` class, which extends `com.liferay.portal.kernel.portlet.BaseFriendlyURLMapper`. Then in `liferay-portlet.xml`, set the `friendly-url-mapper-class` tag as a value `com.liferay.portal.blogs.BlogsFriendlyURLMapper`.

Note that for the out of the box portlets, the mapping for the `friendly-url-mapper-class` tag is specified in `liferay-portlet.xml` under the `/portal/portal-web/docroot/WEB-INF` folder. For the portlets in Ext, the mapping for the `friendly-url-mapper-class` tag is specified in `liferay-portlet-ext.xml` under the `/ext/ext-web/docroot/WEB-INF` folder. For the portlets in Plugin SDK, the mapping for the `friendly-url-mapper-class` tag is specified in `liferay-portlet.xml` under the `/docroot/WEB-INF` folder.

The following is a sample code for the Tags portlet, which is abstracted from the `liferay-portlet.xml` file in the `/portal/portal-web/docroot/WEB-INF` folder:

```
<portlet>
 <portlet-name>103</portlet-name>
 <icon>/html/icons/default.png</icon>
 <struts-path>tags_compiler</struts-path>
 <friendly-url-mapper-class>
 com.liferay.portlet.tagscompiler.TagsCompilerFriendlyURLMapper
 </friendly-url-mapper-class>
 <!-- ignore details -->
</portlet>
```

The code above shows the registration of the Tags Compiler portlet (with an ID 103). The `friendly-url-mapper-class` tag has a value `com.liferay.portlet.tagscompiler.TagsCompilerFriendlyURLMapper`.

In brief, you can add custom-friendly URL mappings on custom portlets. There are two steps you need to follow:

1. Create a class which extends `com.liferay.portal.kernel.portlet.BaseFriendlyURLMapper`, and provide custom functions for the `buildPath` and `populateParams` methods.
2. Map this class with the `friendly-url-mapper-class` tag in the registration of custom portlets.

## Constructing web services

In general, web services are resources which 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 the database entries are generated by ServiceBuilder, web services can be generated as well based on Apache Axis. As a result, nearly all of the backend API calls can be made using web services. Java clients may be generated from **Web Service Definition Language (WSDL)** using any number of tools (Axis, Xfire-CXF, JAX-RPC, and so on). Apache Axis is essentially a **Simple Object Access Protocol (SOAP)** engine—a framework for constructing SOAP processors such as clients, servers, gateways, and so on. Refer to <http://ws.apache.org/axis/> for more details.

## Building custom web services

We can build custom web services in Ext as well. For demo purposes, we take the model workflow as an example. The model workflow was represented as three entities: `LayoutHistory`, `TasksProposalHistory`, and `TasksCommentsHistory`. Models and services, either local services or remote services, have been generated by ServiceBuilder. Now we have a question—how to build web services based on workflow model?

To activate web services for workflow, you just have to double-click on the target `build-wsdd` under `ext-impl` of the Ant view. That's it. In fact, these three SOAP services are generated in `server-config.wsdd` under the `/ext/ext-web/docroot/WEB-INF` folder, according to these entities: `LayoutHistory`, `TasksProposalHistory`, and `TasksCommentsHistory`. For example, for the `LayoutHistory` entity, it generated the following code:

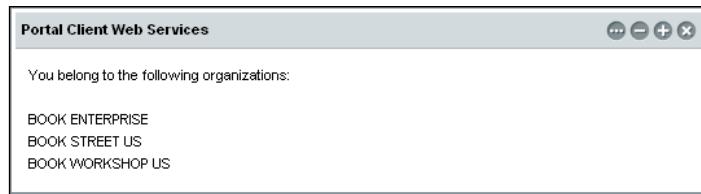
```
<service name="Portlet_Tasks_LayoutHistoryService"
 provider="java:RPC" style="rpc" use="encoded">
 <!-- ignore details -->
 <parameter name="className"
 value="com.ext.portlet.tasks.service.
 http.LayoutHistoryServiceSoap"/>
 <!-- ignore details -->
</service>
```

The code above displays a SOAP service for the `LayoutHistory` entity. The SOAP service (**RPC – Remote Procedure Call**) is named as `Portlet_Tasks_LayoutHistoryService`. The class name is `com.ext.portlet.tasks.service.http.LayoutHistoryServiceSoap`, generated by ServiceBuilder in the `/ext/ext-impl/src` folder.

If you stop Tomcat, deploy Ext, restart Tomcat, and type `http://127.0.0.1:8080/tunnel-web/axis` in your browser, you will see a list of SOAP services with a link to WSDL. These SOAP services include default portal services (with prefix `Portal_`), default portlet service (with prefix `Portlet_`), and custom portlet services (for example `Portlet_Tasks_LayoutHistoryService`, `Portlet_Tasks_TasksProposalHistoryService`, and `Portlet_Tasks_TasksCommentsHistoryService`). Note that it is the IP `127.0.0.1`, not the domain name `localhost`. Why not? Refer to the following section.

## Consuming web services in portlets

Here we will go further to show how to use `portal-client.jar` to access Liferay's services using SOAP. As shown in the following screenshot, we plan to show the organizations which the current user belongs to through SOAP. As mentioned earlier, the current user is "Test Test" (username `test`, email address `test@liferay.com`, and password `test`), an admin of the portal. We have created an organization **BOOK ENTERPRISE**, and two locations **BOOK STREET US** and **BOOK WORKSHOP US** with the parent organization **BOOK ENTERPRISE**. We have assigned the user "Test Test" to this organization and these locations. Using regular Java calls `OrganizationServiceSoap`, `OrganizationServiceSoapServiceLocator` and `OrganizationSoap` model, we can easily find the organizations the current user belongs to in the **Portal Client Web Service** portlet.



## How does it work?

First, we can get the Portal Client Web Services portlet as a plugin portlet. To do so, check out portlet `portal-client-portlet` from `svn://svn.liferay.com/repos/public/plugins/trunk/portlets/sample-portal-client-portlet` to `$PLUGINS_SDK_HOME/portlets/portal-client-portlet`. Refresh the `$PLUGINS_SDK_HOME` project and you will see the `portal-client-portlet` folder under the `$PLUGINS_SDK_HOME/portlets` folder.

Then we need to change the `display-name` and `portlet-info` as stated in the above screenshot. To do so, locate the XML file `portlet.xml` in the `$PLUGINS_SDK_HOME/portlets/portal-client-portlet/docroot/WEB-INF` folder. Now open it, set the display name, title, short title, and keywords as `Portal Client Web Services` and save it. Afterwards, we need to deploy this portlet to the portal: Locate the XML file `build.xml` in the `/portal-client-portlet` folder, and drop it to the Ant view. You can see `portlet` in the Ant view. Then expand `portlet` and you will see the targets `clean`, `compile`, `deploy`, and so on. Double-clicking on the `deploy` target can get the `Portal Client Web Services` portlet hot-deployed.

Finally, we need to update the JSP file `view.jsp` under the `/portal-client-portlet/docroot` folder with IP as `127.0.0.1` and user password as `test`. As shown in the following lines, you will see the model and service, which are used for the `Portal Client Web Services` portlet:

```
<%@ page import="com.liferay.client.soap.portal.
 model.OrganizationSoap" %>
<%@ page import="com.liferay.client.soap.portal.
 service.http.OrganizationServiceSoap" %>
<%@ page import="com.liferay.client.soap.portal.
 service.http.OrganizationServiceSoapServiceLocator" %>
<%@ page import="com.liferay.portal.kernel.util.GetterUtil" %>
<%@ page import="java.net.URL" %>
You belong to the following organizations:

<% String remoteUser = request.getRemoteUser();
 long userId = GetterUtil.getLong(remoteUser);
 OrganizationServiceSoapServiceLocator locator = new Organization
 ServiceSoapServiceLocator();
 OrganizationServiceSoap soap = locator.getPortal_
 OrganizationService(_getURL(remoteUser,
 "Portal_OrganizationService"));
 OrganizationSoap[] organizations = soap.getUserOrganizations(
 userId);
 for (int i = 0; i < organizations.length; i++) {
 OrganizationSoap organization = organizations[i];
 }>
<%= organization.getName() %>
 <% } %>
<%! private URL _getURL(String remoteUser, String serviceName)
throws Exception {
 // Unauthenticated url
 String url = "http://127.0.0.1:8080/tunnel-web/axis/" +
 serviceName;
 // Authenticated url
 if (true) {
 String password = "test";
 url = "http://" + remoteUser + ":" + password +
 "@127.0.0.1:8080/tunnel-web/secure/axis/" +
 serviceName;
 }
 return new URL(url); } %>
```

The code above shows how to consume services through SOAP.

These services include `com.liferay.client.soap.portal.service.http`, `OrganizationServiceSoap` and `OrganizationServiceSoapServiceLocator`, and the model involves `com.liferay.client.soap.portal.model`. `OrganizationSoap`. The user must already have the permission to access whatever resources will be accessed via the web services. The credentials need to be passed on to the URL `http://test:test@127.0.0.1:8080/tunnel-web/axis/Portal_OrganizationService`.

Similarly, you can use other services, for example `com.liferay.client.soap.portal.service.http.UserServiceSoap`, `com.liferay.client.soap.portal.service.http.GroupServiceSoap`; and models, for example `com.liferay.client.soap.portal.model.UserSoap`, `com.liferay.client.soap.portal.model.GroupSoap`. Of course, you may be interested in other services and models: Locate the JAR file `portal-client.jar` in the `/portal/portal-client` folder. Open it (for example, using **WinZip**) and you will find all services and models.

## What's happening?

The portal uses Apache Axis to generate web services. The default Axis configuration is specified in the `server-config.wsdd` file under the `/portal/tunnel-web/docroot/WEB-INF` folder. When you double-click on the target deploy in the Ant view, the file `server-config.wsdd` under the `/portal/tunnel-web/docroot/WEB-INF` folder will be merged with the `server-config.wsdd` file under the `/ext/ext-web/docroot/WEB-INF` folder. Therefore, when you type `http://127.0.0.1:8080/tunnel-web/axis` in your browser, you will see a list of SOAP services.

To access a service remotely, the host must be allowed via the `portal-ext.properties` properties file. After that, the user must have permission to access the portal resources. The default settings to access a service remotely are specified in the `portal.properties` file as follows:

```
axis.servlet.hosts.allowed=127.0.0.1,SERVER_IP
axis.servlet.https.required=false
```

The code above shows the IPs to access the Axis servlet. You can input a blank list to allow any IP to access this servlet. `SERVER_IP` will be replaced with the IP of the host server. By default, `127.0.0.1` is the IP for local host. This is the reason that you can access web services only by the IP `127.0.0.1`, and not `localhost`. Of course, you can use a domain name such as `www.bookpub.com` if you had set the mapping between `127.0.0.1` and `www.bookpub.com` in the hosts file.

## Enjoying best practices

Now let's share the best practices from the Liferay community. These best practices will be useful to customize portal systems on top of Liferay portal. There are a lot of examples for best practices, but we will just list some of them, including JavaScript portlet URL, user and organization administration, speeding up portal, UI taglibs, WSRP, and integration with SharePoint and Terracota DSO.

## Using JavaScript Portlet URL

Liferay portal provides functionality to build Portlet URLs using JavaScript. Obviously, there are many places where you need to generate Portlet URLs. Imagine a huge list of links on a portlet. Instead of downloading repeated hundreds of URLs from the server site, you can simply create a JavaScript function that returns a `PortletURL` instance and set the parameters you need on the fly. From now on, you can simply integrate pure JavaScript files with `PortletURLs`, without passing it as a parameter for JavaScript constructor. The usage of this functionality using JavaScript is simple. It is as follows:

```
<script> var portletURL = Liferay.PortletURL.createResourceURL();
portletURL.setPortletId(133);
portletURL.setParameter("widgetURL", widgetURL);
</script>
```

The code above shows how to create Portlet URL with a specific portlet 133 – Portlet Sharing. Or you can simply wrap it into a JavaScript function like this:

```
<script>
function createURL(widgetURL) {
 var portletURL = Liferay.PortletURL.createResourceURL();
 portletURL.setPortletId(133);
 portletURL.setParameter("widgetURL", widgetURL);
 return portletURL.toString();
}
</script>
```

You can find the details of `Liferay.PortletURL` in the `portlet_url.js` file under the `/portal/portal-web/docroot/html/js/liferay` folder. As the `p_l_id: themeDisplay.getPlid()` parameter is used in `PortletURL`, we could differentiate `portletURL` for the same portlet, but located on a different page, via `p_l_id`. The following is a list of some of the functions:

```
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) ;
```

The code above defines a set of JavaScript functions. For instance, the `setParameter: function(key, value)` method specifies a parameter with a pair (key, value). As an example, `portletURL.setParameter("widgetURL", widgetURL)` means that a parameter is set with a key "widgetURL" and a value `widgetURL`.

## Customizing user and organization administration

The **User Administration** tool offers a complete set of options while keeping it usable. The form used for adding or updating users can be completely customized. The customization can range from configuring which sections of form navigation menu are displayed to adding custom ones.

The form sections can be configured through properties in the `portal-ext.properties` file. The following properties determine which sections are available when creating a new user account:

```
users.form.add.main=details,organizations
users.form.add.identification=
users.form.add.miscellaneous=
```

The code above displays a list of sections that will be included as part of the user form when adding a user. After the user account has been created, the form is expanded to show several additional sections. These could be configured through properties in the `portal-ext.properties` file too, as follows:

```
users.form.update.main=details,password,organizations,communities,
user-groups,roles,categorization
users.form.update.identification=addresses,phone-numbers,
additional-email-addresses,websites,
instant-messenger,social-network,sms,open-id
users.form.update.miscellaneous=announcements,
display-settings,comments,custom-attributes
```

The code above shows a list of sections that will be included as a part of the user form when updating a user.

## Creating a new section

Of course, you can add custom new sections to the user form in Ext. The following are the main steps to do so:

1. Choose a name for the section and add it to the desired section in the `portal-ext.properties` properties file.
2. Create a JSP file which contains the form section. Note that the file must be located in Ext under the `/ext/ext-web/docroot/html/portlet/enterprise_admin/user` folder and the name must be the one chosen in the previous step, but with underscores instead of dashes.
3. Add a new entry in `Language-ext.properties` in the `/ext/ext-impl/src/content` folder using the name chosen as the key and the desired label as the value.
4. Repeat this for any languages that are supported in the portal.

## Customizing fields of form section

As you can see, each section of the user form has been implemented as a separate JSP template with no business logic so that it can be safely overridden. To modify any of these files, first just copy it to the equivalent directory in Ext: `/ext/ext-web/docroot/html/portlet/enterprise_admin/user`, and then perform any desired changes.

## Customizing columns of the list

The list of users can also be customized in Ext. To do so, copy the file from `/portal/portal-web/docroot/html/portlet/enterprise_admin/user/search_columns.jspf` into `/ext/ext-web/docroot/html/portlet/enterprise_admin/user` first, and then perform any changes desired. Note that the `search_columns.jspf` file doesn't contain any business logic. It has a taglib invocation for each of the columns. Therefore, it is pretty easy to change the order or add additional columns.

Similarly, the **Organization Administration** tool follows the same design patterns employed by User Administration to achieve better usability and more flexibility. In short, the portal has full customizability with the ability to hide or add certain parts of the forms for both the users and the organizations.

## Speeding up the portal

The golden rule to speed up the portal is to optimize the frontend performance, where 80% or more of the end user response time is spent. There is a set of options to speed up the portal. Here we will just list some of them.

First, we should use **CSS Sprites** for multiple images. CSS Sprites is used to group multiple images into one composite image and to display it using CSS background positioning. That is, it should combine the background images into a single image and use the CSS background-image and background-position properties to display the desired image segment.

The portal adds a property called `theme.images.fast.load` with the default value `true` for a fast loading in the production phase. It means that when the server starts up, two files are automatically created on each image folder of the theme—`.sprite.png` and `.sprite.properties`. The taglibs are programmed to automatically read `packed.sprite` and display that relative file on `packed.png` if this feature is enabled. By the way, set this property with the value `false` for easier debugging in the development phase.

Secondly, the portal should use cache filter. The `com.liferay.portal.servlet.filters.cache.CacheFilter` class extends the abstract class `com.liferay.portal.servlet.filters.BasePortalFilter`, which implements `com.liferay.portal.kernel.servlet.BaseFilter`.

Thirdly, we should put scripts at the bottom. The problem caused by scripts is that they block parallel downloads. Thus, use the `footer-portlet-javascript` tag instead of the `header-portlet-javascript` tag in custom `liferay-portlet.xml`. In this way, the portal will download unnecessary JavaScript as late as possible. For example, we have used the `header-portlet-javascript` tag in `liferay-portlet.xml` of the `ipc-faq-portlet` portlet. Instead, we should use the `footer-portlet-javascript` tag.

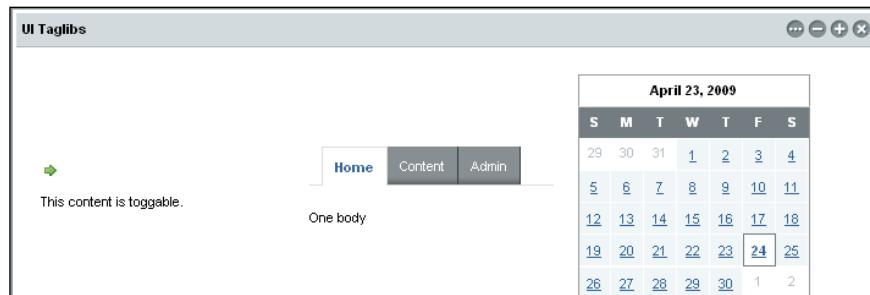
Finally, we should use the `Never Expire` header for static components, and the `Cache-Control` header for dynamic components. Implement the `Never Expire` policy by setting `Far Future Expires Header` for static components; whereas use an appropriate `Cache-Control` header to help the browser with conditional requests for dynamic components.

## Sharing UI Taglibs in portlets

In this section, we're going to show how to use UI tags in a portlet. Liferay portal uses Taglibs extensively. The standard **JSTL (JavaServer Pages Standard Tag Library)** library is used mainly for tags, for example `c:if` and `c:when/c:otherwise`.

As shown in the following screenshot, it uses the `liferay-ui:calendar` UI taglib to represent a calendar view – month titles and days of the month. The `liferay-ui:calendar` taglib only requires the inputs `month`, `day`, `year`, and `headerPatterns`. Moreover, it uses the UI taglib `liferay-ui:tabs` to represent tabs. There are three tabs in this example: `Name`, `Content`, and `Admin`. Clicking on a tab will show the content of that tab. The `liferay-ui:tabs` UI taglib requires the inputs `names`, `refresh` and the `liferay-ui:section` taglib.

Further, it uses the `liferay-ui:toggle` UI taglib to represent the toggleable content. The preference for this content is persisted based on the specified ID. If the user is a guest, the preference is persisted for the session only. If the user is authenticated, the preference is persisted in the database for all of the future requests. The `onImage` and `offImage` attributes are optional and default to the images in this sample. You can also customize the default images by replacing them in a custom theme:



## How does it work?

First, we can get the UI Taglibs portlet as a plugin portlet by checking out `portlets/ui-taglibs-portlet` from `svn://svn.liferay.com/repos/public/plugins/trunk/portlets/sample-ui-taglibs-portlet` to `$PLUGINS_SDK_HOME/portlets/ui-taglibs-portlet`. Refresh the `$PLUGINS_SDK_HOME` project and you will see the `ui-taglibs-portlet` folder under the `$PLUGINS_SDK_HOME/portlets` folder.

Now we need to change the display name and portlet info as stated in the above screenshot. To do so, locate the XML file `portlet.xml` in the `/ui-taglibs-portlet/docroot/WEB-INF` folder. Open it, set the display name, title, short title, and keywords as UI Taglibs; and save it. Afterwards, we need to deploy this portlet to the portal. So locate the XML file `build.xml` in the `/ui-taglibs-portlet` folder, and drop it to the Ant view. You can see `portlet` in the Ant view. Expand `portlet` and you will see the targets `clean`, `compile`, `deploy`, and so on. Double-click on the `deploy` target, you can get the UI Taglibs portlet hot-deployed.

In addition, you need to add the following lines before the line `<%@ page import="java.util.Calendar" %>` in `init.jsp` under the `/ui-taglibs-portlet/docroot` folder:

```
<%@ page import="com.liferay.portal.kernel.util.CalendarFactoryUtil"
%
<%@ page import="com.liferay.portal.kernel.util.CalendarUtil" %> <%@
page import="com.liferay.portal.kernel.util.DateFormats" %>
<%@ page import="com.liferay.portal.kernel.util.DateUtil" %>
<%@ page import="java.text.DateFormat" %>
```

Moreover, overwrite the JSP file `view.jsp` under the `/ui-taglibs-portlet/docroot` folder with the following lines:

```
<%@ include file="/init.jsp" %>
<% Calendar curCal = CalendarFactoryUtil.getCalendar(
 timeZone, locale);
int curMonth = curCal.get(Calendar.MONTH);
int curDay = curCal.get(Calendar.DATE);
int curYear = curCal.get(Calendar.YEAR); %>

| |
|--|
| <div> <liferay-ui:toggle id="toggle_id_sample_ui_taglibs" showImage='<%= themeDisplay.getPathThemeImages() + "/arrows/01_down.png" %>' hideImage='<%= themeDisplay.getPathThemeImages() + "/arrows/01_right.png" %>' defaultShowContent="true" /> </div> <div >;="" 10px;">="" <="" content="" div="" id="toggle_id_sample_ui_taglibs" is="" padding-top:="" style="display: <liferay-ui:toggle-value id=" this="" toggable.="" toggle_id_sample_ui_taglibs"=""> </div> |
|--|


```

```
</div>
</td>
<td style="padding: 10px; width: 200px; align: top; ">
 <liferay-ui:tabs names="Home,Content,Admin"
 refresh="<%=_false %>">
 <liferay-ui:section> One body </liferay-ui:section>
 <liferay-ui:section> Two body </liferay-ui:section>
 <liferay-ui:section> Three body </liferay-ui:section>
 </liferay-ui:tabs>
</td>
<td style="padding: 10px; width: 200px; align: top; ">
 <liferay-ui:calendar month="<%=_curMonth %>">
 day="<%=_curDay %>">
 year="<%=_curYear %>">
 headerFormat="<%=_DateFormat.getDateInstance(
 DateFormat.LONG, locale) %>" />
 </liferay-ui:calendar>
</td>
</tr>
</table>
<div class="separator"></div>
<‐;
Back
```

The code above exhibits how to use the UI taglibs `liferay-ui:calendar`, `liferay-ui:tabs` and `liferay-ui:toggle`. Of course, you may be interested in other UI taglibs, for example `liferay-ui:custom-attribute-list`, `liferay-ui:custom-attribute`, `liferay-ui:search`, `liferay-ui:search-iterator`, `liferay-ui:ratings`, `liferay-ui:discussion`, and so on. You can simply find the code that implements taglibs in the `com.liferay.taglib.ui` package under the `/portal/util-taglib/src` folder, the JSP files of taglibs in the `/portal/portal-web/docroot/html/taglib/ui` folder, and the `liferay-ui` parameters in the `liferay-ui.tld` file under the `/portal/util-taglib/src/META-INF` folder.

## Consuming WSRP

In general, WSRP defines a set of interfaces and related semantics. This standardizes the interactions with components that provide user-facing markup, including the processing of user interactions with that markup. It allows the portal to consume components such as providing a portion of the overall custom application without writing unique code for interacting with each component. Refer to <http://www.oasis-open.org> for more details.

Liferay portal supports full WSRP 1.0 and 2.0 specifications. It can act as both WSRP 2.0 Consumer and Producer. In fact, WSRP Producer is a web service server that offers one or more portlets and this implements the interfaces and operations defined in the specification. It supplies the environment for deploying and managing portlets. WSRP Consumer is a web service client that invokes a Producer and supplies the environment in which users interact with portlets from Producers.

As shown in the following screenshot, you can find the **WSRP Consumer** portlet in the category Server position 4 . 0. Through the WSRP consumer, you can connect to a producer, and then install portlets.

The screenshot shows the Liferay Admin interface. On the left, there's a sidebar with categories: Content, Portal, Server, and WSRP Consumer. Under Server, there are sub-options: Server Administration, Portal Instances, Plugins Installation, and WSRP Consumer. The WSRP Consumer option is selected. The main content area has a title "Server" and a sub-section "WSRP Consumer". Below that, there are two tabs: "Producers" (selected) and "Portlets". A button "Connect to Producer" is visible. A table lists producers with columns Name, ID, and Status. Two entries are shown: "IPC" (ID 11710, Enabled) and "ConsumerIPC" (ID 11714, Enabled). To the right of the table are "Actions" buttons for each row: Edit, Install Portlet, Update Service Description, Delete, and another Actions button.

As shown in the following screenshot, you can find the **WSRP Producer** portlet in the **Server** category and the position 5 . 0. Through the WSRP producer, you can add more producers, and can set consumer registration and registration properties.

The screenshot shows the Liferay Admin interface. On the left, there's a sidebar with categories: Content, Portal, Server, and WSRP Producer. Under Server, there are sub-options: Server Administration, Portal Instances, Plugins Installation, and WSRP Consumer. The WSRP Producer option is selected. The main content area has a title "Server" and a sub-section "WSRP Producer". Below that, there are two tabs: "Producers" (selected) and "Portlets". A button "Add Producer" is visible. A table lists producers with columns Name, Status, and Registration. One entry is shown: "IPC" (Enabled, Required). To the right of the table are "Actions" buttons for each row: Edit, Consumer Registrations, Registration Properties, Delete, and another Actions button.

## How do we get the WSRP portlets?

First of all, we can get the WSRP portlets as a plugin portlet by checking out portlets wsrp-portlet from `svn://svn.liferay.com/repos/public/plugins/trunk/portlets/wsrp-portlet` to `$PLUGINS_SDK_HOME/portlets/wsrp-portlet`. Refresh the project `$PLUGINS_SDK_HOME` and you will see the `wsrp-portlet` folder under the `$PLUGINS_SDK_HOME/portlets` folder.

Then, locate the XML file `liferay-portlet.xml` in the `/wsrp-portlet/docroot/WEB-INF` folder, and open it. You will find the following code:

```
<portlet>
 <portlet-name>1</portlet-name>
 <control-panel-entry-category>server</control-panel-entry-category>
 <control-panel-entry-weight>4.0</control-panel-entry-weight>
 <css-class-wrapper>wsrp-portlet-consumer</css-class-wrapper>
</portlet>
<portlet>
 <portlet-name>2</portlet-name>
 <control-panel-entry-category>server</control-panel-entry-category>
 <control-panel-entry-weight>5.0</control-panel-entry-weight>
 <css-class-wrapper>wsrp-portlet-producer</css-class-wrapper>
</portlet>
```

The code above exhibits the WSRP portlets' registration. The Consumer portlet is displayed in the server category and the position 4.0. The Producer portlet is displayed in the server category and the position 5.0.

## How does it work?

Eventually, the WSRP portlets use four models: `WSRP_WSRPConfiguredProducer`, `WSRP_WSRPConsumerRegistration`, `WSRP_WSRPPortlet`, and `WSRP_WSRPProducer`. They are explained here:

- The `WSRP_WSRPConfiguredProducer` model specifies information about the configured producer, for example `configuredProducerId`, `name`, `portalId`, `namespace`, `producerURL`, `producerVersion`, `producerMarkupURL` status, and so on
- The `WSRP_WSRPConsumerRegistration` model specifies messages related to consumer registration, for example `consumerRegistrationId`, `consumerName`, `status`, `registrationHandle`, `registrationData`, and so on
- The `WSRP_WSRPPortlet` model has attributes `portletId`, `name`, `channelName`, `title`, `shortTitle`, `displayName`, `keywords`, `status`, `producerEntityId`, `consumerId`, `portletHandle`, `mimeType`, and so on
- The `WSRP_WSRPProducer` model has attributes `producerId`, `portalId`, `status`, `namespace`, `instanceName`, `requiresRegistration`, `supportsInbandRegistration`, `version`, `offeredPortlets`, and so on

Locate the `service.properties` file in the `$PLUGINS_SDK_HOME/portlets/wsdp-portlet/docroot/WEB-INF/src` folder and you will see the following code:

```
spring.configs=WEB-INF/classes/META-INF/base-spring.xml,
WEB-INF/classes/META-INF/hibernate-spring.xml,
WEB-INF/classes/META-INF/infrastructure-spring.xml,
WEB-INF/classes/META-INF/portlet-spring.xml,
WEB-INF/classes/META-INF/dynamic-data-source-spring.xml
```

The code above makes the Spring configuration for the WSRP portlets visible. These XML files will be loaded after the bean definitions specified in the `contextConfigLocation` parameter in `web.xml`. Of course, you can find these XML files in the `/wsdp-portlet/docroot/WEB-INF/src/META-INF` folder.

## Integrating with SharePoint

Liferay portal implements the SharePoint protocol, which allows saving documents to Liferay portal as if it were a SharePoint server. In `portal.properties` under the `/portal/portal-impl/src` folder, you will find the following configuration:

```
sharepoint.storage.tokens=document_library
sharepoint.storage.class [document_library]=com.liferay.portlet.
documentlibrary.sharepoint.DLSharepointStorageImpl
```

This code shows the integration of SharePoint in the portal. It sets the tokens for supported SharePoint storage paths first, and then it sets the class names for supported SharePoint storage classes, for example `com.liferay.portlet.documentlibrary.sharepoint.DLSharepointStorageImpl`. The `DLSharepointStorageImpl` class extends the `com.liferay.portal.sharepoint.BaseSharepointStorageImpl` abstract class, which implements the `com.liferay.portal.sharepoint.SharepointStorage` interface.

On the whole, there are these additional classes `com.liferay.portal.sharepoint.CompanySharepointStorageImpl` and `com.liferay.portal.sharepoint.GroupSharepointStorageImpl`, which extend `BaseSharepointStorageImpl`. In support of this, the packages `com.liferay.portlet.documentlibrary.sharepoint` and `com.liferay.portal.sharepoint` are available in the `/portal/portal-impl/src` folder. In any event, these packages and related Java classes will be useful when you customize SharePoint integration in Liferay portal.

## Integrating with Terracotta DSO

Liferay portal integrates closely with Terracotta DSO for increased scalability and performance. In the `portal.properties` file under the `/portal/portal-impl/src` folder, you will find the following configuration related to Terracotta DSO integration:

```
net.sf.ehcache.configurationResourceName=/ehcache/
hibernate-terracotta.xml
```

The code above shows cache configuration for a Terracotta environment. **Terracotta DSO (Distributed Shared Objects)** is an open source technology created by Terracotta, which is meant to provide clustering to Java at the virtual-machine level. It layers in the sharing and coordinating of data. Refer to <http://www.terracotta.org> for more details.

In addition, you can find the XML file `hibernate-terracotta.xml` in the `/portal/portal-impl/src/ehcache` folder. This configuration is optimized for Terracotta DSO. The `maxElementsInMemory` attribute has been increased to account for the capabilities that Terracotta provides.

## Summary

This chapter first introduced how to use custom attributes for both journal article templates and custom portlets. Then it addressed how to build OpenSearch and how to employ search capabilities. Later, it focused on approaches to employ the Spring services and to construct web services. Finally, it discussed the best practices, such as using JavaScript portlet URL, customizing the user and organization administration, speeding up portal, sharing UI Taglibs, producing and consuming WSRP, and integrating with SharePoint and Terracotta DSO.

As expected, you can find common API related to external services in both the `/portal/portal-kernel/src` and `/portal/portal-service/src` folders. These external services are not only useful for portlets, hooks, and webs that are developed in Plugins SDK, but are also helpful for customization and extension of portlets that are developed in Ext. Further, we could also find all common API related to internal services in the `/portal/portal-impl/src` folder. These internal services are useful only for customization and extension of out of the box portlets which are developed in Ext.

For the reasons discussed above, this book mainly introduced customization and extension of portlets in both Ext and Plugins SDK. It presented JSR-286 portlets, ServiceBuilder, and development environments first. Then it exhibited how to use Struts framework to construct portlets with actions and permissions in Ext. Later, it made use of real examples – how to manage pages, how to customize WYSIWYG editor, how to customize CMS and WCM, and how to build My Community. It also revealed how to set up Plugins SDK and how to use it to build layout templates and themes. Moreover, it also explained how to utilize Plugins SDK for building My Social Office together with themes, portlets, and hooks. Finally, it addressed how to apply, customize, and extend the features of staging, workflow, scheduling, and publishing both locally and remotely. In short, we can say that both Ext and Plugins SDK are powerful and useful environments to develop Liferay portal systems.

# Index

## A

### **action class, Struts portlet**

- action, creating 109
- action, defining 110
- adding 108
- errors pages, creating 111, 112
- form, adding in JSP page 111
- success pages, creating 111, 112

### **actions, Struts portlet**

- adding 121
- entry\_action.jspf file, creating 123, 124
- existing JSP files, updating 124, 125
- methods, creating 122
- updating 123

### **addBook method 116**

### **Ant**

- about 59
- ANT\_HOME variable, setting up 59
- downloading 59
- installing 59

### **AOP 487**

### **Apache Axis 498**

### **Apache Struts**

- about 93
- features 134
- need for 134

### **application framework, supported by Liferay**

- Liferay
- DAO 91
- Hibernate 91
- JSF 91
- JSON 91
- JSP 91
- LAR 91
- Lazzlo 91

PHP 91

Python 91

Ruby 91

Spring MVC 91

Struts 91

Tapestry 91

Wicket 91

### **application servers, Ext set up**

- about 60

Tomcat 61

### **application servers, Liferay portal**

- Apache Geronimo 60

Borland ES 60

JBoss 60

Jetty 60

JOnAS 60

JRun 60

OracleAS 60

Orion 60

Resin 60

Sun GlassFish 60

SUN JSAS 60

Tomcat 60

WebLogic 60

WebSphere 60

### **architecture, Liferay portal**

- ESB 17

SOA 16

### **article template, building for journal articles**

- article template, setting up 291

article templates, updating 291

default article type, setting up 290

### **Aspect Oriented Programming. See AOP**

### **Asset Publisher portlet**

- about 246

customizing 246

Ext Asset Publisher portlet, building  
249-251  
large-size image, adding 246, 247  
medium-size image, adding 246, 247  
view, extending 252  
**auto-login hooks** 417

## B

**BPMS | Intalio** 453  
**Breadcrumb portlet** 420  
**business benefits, Liferay portal**  
high adaptability to the demands of a fast-changing market 15  
highest values 15  
single point of access 14  
user experience 13

## C

**caching features, JSR-286**  
about 46  
multiple cached views 46  
public caching scope 46  
validation-based caching 46  
**category attribute** 50  
**chat portlet, My Social Office**  
about 393  
deploying 393  
setting up 393  
**clustering** 15  
**CMS and WCM, extending**  
about 271  
articles, employing 271  
content, tagging 273, 274  
Document Library portlet, extending 275  
Image Gallery portlet, extending 275  
journal template, using 272  
structures, employing 271  
template, employing 272  
velocity template, using 272  
velocity templates, adding in Asset Publisher portlet 275  
Web Content search portlet, features 273  
**communities and layout page**  
comments, using 177  
communities, using 177  
community, employing 176

extending 178  
group, employing 176  
layout pages, using 177  
permissions, employing 176  
ratings, using 177  
using 176  
**Communities portlet**  
about 135  
extending 136  
**Communities portlet, extending**  
about 136, 137  
Ext Communities portlet, building 137  
Ext Communities portlet, setting up 144, 149  
**community** 177  
**Computer Associate's (CA) SiteMinder** 417  
**container runtime options** 42  
**content, sharing with friends**  
applications, sharing on any web site 280, 281  
emails, sending with sharing links to friends 282  
print button 278  
print preview, building as article template 279, 280  
send message button 278  
send to friend button 278  
**content-rich flashes, inserting into web content**  
about 207  
advanced search functions, adding for slideshow 211, 213  
advanced search functions, adding for SWF 211, 213  
advanced search functions, adding for videos 211, 213  
advanced search views, adding for slideshow 211  
advanced search views, adding for SWF 211  
advanced search views, adding for videos 211  
flashes, querying 208, 209  
flash objects, adding for viewing slideshow 213  
flash objects, adding for viewing SWF 213  
flash objects, adding for viewing video 213

games, adding to journal articles 218-220  
games, playing while reading text 220  
play-list, employing 220  
playlists, adding to journal articles 218- 220  
RESTful services, preparing 221  
single flash SWF, adding to journal articles 210, 211  
slideshows, adding to journal articles 210, 211  
video list, adding to journal articles 215, 216  
video list, putting in journal articles 216, 217  
video queue, adding to journal articles 215, 216  
video queue, setting up in journal articles 217, 218  
videos, adding to journal articles 210, 211

**content articles, featured content**  
about 238  
building 238, 239  
images, preparing 238

**Control Panel**  
about 369, 370  
Content category 372  
customizing 377  
features 371, 372  
groups 372  
My category 372  
Portal category 372  
Server category 372  
working 373

**Control Panel, customizing**  
about 377  
customized portlets, configuring 379, 380  
edit page, updating 378  
theme, changing 377  
view page, updating 378

**Control Panel, working**  
about 373  
Control Panel settings, employing 374, 375  
Control Panel theme, using 373  
portlets, configuring 376, 377

**CSS 3 348**

**CSS Sprites 506**

**custom attributes**  
about 468

dynamic table, building with Velocity  
Expando template 468  
Expando portlet, sharing 478, 480  
extending 475

**customization 13**

**customized theme, creating**  
about 340  
differences of themes, building 343  
theme project, setting up 340-342

**customized Velocity templates**  
drop-down menu use case, implementing 361-363  
navigation bar use case, implementing 364, 365

## D

**database interaction, Struts portlet**  
about 112  
database structure, creating 113-115  
existing files, updating 116, 117  
methods, creating 115, 116  
records, retrieving from database 118

**databases, Ext set up**  
about 59  
MySQL 59

**databases, Liferay portal**  
Apache Derby 59  
Firebird 59  
Hypersonic 59  
IBM DB2 59  
Informix 59  
InterBase 59  
JDataStore 59  
MySQL 59  
Oracle 59  
PostgresSQL 59  
SAP 59  
SQL Server 59  
Sybase 59

**DebugBar 355**

**default Velocity templates**  
about 356  
default Velocity variables, experiencing 356, 357, 358  
Velocity variables, customizing 358, 360

**delete method 122**

**Dependency Injection.** *See DI*

**destroy method** 27

**development environments, JSR-286 portlets**

- Ext 57
- Ext, using 91
- Ext, working 91
- Plugins SDK 57
- using 90

**DI** 487

**document library hooks** 416

**Document Object Model** 348

**Document Tracking** 360

**doFilter method** 41

**dynamic articles, building with polls**

- about 266
- journal articles, associating with polls 270, 271
- template node poll, adding 267, 268
- Web Content portlet, updating with template node poll 269, 270

**dynamic articles, building with recently added content**

- about 255, 256
- journal articles, displaying through asset ID 257
- recently added content, listing 262, 263
- related content, exhibiting 264, 266
- touts, displaying with article ID 258

**dynamic navigation**

- building 420
- custom navigation, constructing 421
- street navigation, constructing 422
- views, building 423, 424

**dynamicQuery API** 290

**dynamic table, building with Velocity Expando templates**

- book title list, building 470, 472
- journal structure, creating 469
- journal template, creating 469

**E**

**Eclipse IDE**

- about 62, 63
- downloading 63
- installing 63

**Eclipse plugin development tools**

**CSS** 65

**EJB** 65

**HTML** 65

**Java** 65

**JSP** 65

**Struts** 65

**XML** 65

**Enterprise Service Bus.** *See ESB*

**ESB** 14, 17

**escapeXml attribute** 48

**event handlers**

- custom cookie, creating with login 429, 431
- customizing 427
- events, handling 427
- global shutdown actions, configuring 428, 429
- global startup action, configuring 428
- global startup actions, configuring 429

**event model** 348

**EventPortlet interface** 52

**Expando** 468

**Expando services, in portlets**

- about 475
- organization profiles, extending 475
- user profiles, extending 475

**Expando Velocity template variables**

- about 472
- models 473
- services 473

**Ext**

- about 58
- building 67
- deploying 74
- setting up 58
- using 91
- working 91

**Ext, building**

- about 67
- Ant targets 69
- Ext structures, navigating 73
- portal source code, getting 68
- properties, customizing 71, 72
- source structures 69
- Tomcat, updating to support Ext development 70, 71
- via Ant 73

**Ext, deploying**

about 74  
Ant deploy, used 75  
database, configuring 74, 75  
fast-deploy 77  
portal structures, viewing in Tomcat 76

**Ext, setting up**  
about 58  
application servers 60  
databases 59  
IDE 62  
portal source code 67  
required tools 58

**Ext Comment portlet**  
building 300

**Ext Comment portlet, building**  
email notification, setting up 303, 304  
permissions, adding 301, 302  
UI tag, updating 302

**Ext Communities portlet**  
about 137  
actions, setting up 140, 141  
constructing 137, 138, 139  
JSP files, preparing 143, 144  
page flow, setting up 141, 142  
page layout, setting up 141, 142  
setting up, in backend 144  
setting up, in frontend 149

**Ext Communities portlet, setting up in backend**  
action class, updating 148  
database, creating 145, 146  
delete method, creating 147, 148  
retrieve method, creating 147, 148  
update method, creating 147, 148

**Ext Communities portlet, setting up in frontend**  
community-customized columns, deleting 149  
community-customized columns, retrieving 150  
community-customized columns, updating 149

**extension environment** 18

**Ext layout management portlet**  
action, setting up 153, 154  
constructing 152, 153  
JSP files, preparing 155

page flow, setting up 155  
page layout, setting up 155

**Ext layout management portlet, setting up in backend**  
action class, updating 160, 162  
database structure, creating 156, 157  
delete methods, creating 158, 159  
retrieve methods, creating 158, 159  
update methods, creating 158, 159

**Ext Web Content Display portlet**  
creating 232, 233

## F

**Facebook** 393

**FCKeditor**  
about 180  
Ant target, extending 180, 181  
customized icons, adding 182  
default configuration, employing 183, 184  
features 222  
file browser connector, configuring with Liferay portal services 191  
file browser connector, customizing with RESTful services 195  
file browser connector, extending 222  
images, inserting from different services 190  
links, inserting from different services 190  
setting up 182  
styles, adding 184, 185  
styles and formats, constructing 186  
templates, adding 184, 185  
templates, building 189  
upgrading 181

**featured content**  
about 230  
constructing 230  
content articles, building 238  
features 231  
implementing 231  
structure, setting up 235  
template, setting up 235  
Web Content Display portlet, customizing 231

**file browser integration** 223

**file browser connector** 190

**file browser connector, configuring with Liferay portal services**

images and links, browsing 192  
Liferay portal services, preparing 193  
services, configuring for documents 191  
services, configuring for images 191  
services, configuring for pages 191

**file browser connector, customizing with RESTful services**

advanced search functions, adding to images and links 198- 205  
advanced search view features, adding 195, 196, 197  
RESTful services, preparing 205, 206

**Firebug 355**

**Floating Div pop-up 321**

**fragment 25**

## G

**getAll method 116**  
**getBook method 122**  
**getCacheability method 44**  
**getFile method 296**  
**getImage method 295**  
**getPortalContext method 38**  
**getScheme() method 47**  
**getServerName() method 47**  
**getServerPort() method 47**  
**getter method 297**  
**getWindowID() 47**  
**GIMP 330**

## H

**hooking 367**

**hooks**

auto-login hooks 417  
document library hooks 416  
general usage 414  
JSP hooks 414  
language properties hooks 414  
mail hooks 417  
parameters 414  
portal properties hooks 414  
special usage 416  
using 414  
WOL example 415

**hooks, building**

about 407  
model listeners, employing 409, 410  
portal event handlers, applying 408

**HttpSessionBindingListener 50**

## I

**IDE**

about 62  
Eclipse IDE 62, 63  
IntelliJ IDE 62  
NetBeans IDE 62  
Subclipse 64  
Tomcat plugins 65  
workspace 63

**IE Developer Toolbar 355**

**Inflight Data Tracking (IDT) 361**

**init method 27**

**insert method 115**

**Intalio 15**

**IntelliJ IDE 62**

**Inter-Portlet Communication. See IPC**

**Inversion of Control. See IoC**

**IoC 487**

**IPC**

about 369  
building 381  
IPC portlets, constructing 382  
portlet process actions, specifying 386  
portlet project, creating 381, 382  
portlet views, specifying 387, 388

**IPC portlets**

constructing 382, 383  
events, defining 384  
portlets, defining 384  
registering 384, 385

## J

**Jackrabbit 417**

**JAVA 5 features**

about 45  
enum 45  
generics 45

**JavaMail 391**

**Java Portlet Specification 2.0. See also JSR-286**

**JavaScript Portlet URL** 503  
**Java Server Pages** 94  
**Java Specification Request.** *See JSR-286*  
**jBPM** 15  
**jBPM workflow** 452  
**JCR** 417  
**JDK**  
    about 58  
    downloading 58  
    installing 58  
    JAVA\_HOME variable, setting up 58  
**journal articles, setting up**  
    about 286  
    article template, building 290  
    view counter, adding in Ext Web Content  
        Display portlet 287  
    view counter, handling for assets 292  
    VM service, setting up 289, 290  
**jQuery widget** 280  
**JSP portlet**  
    configuring 94  
    defining 94  
    developing 94  
    employing 99  
    jsp\_portlet portlet, configuring in liferay-portlet-ext.xml 95  
    view 98  
    view page, deploying 98  
    view page, updating 98  
**JSP portlet, defining**  
    files, deploying into Tomcat 96  
    the JSP page view.jsp, creating 96  
**JSP portlet, developing**  
    about 94  
    category, changing 97  
    JSP portlet, defining 94  
    JSP portlet, using 98  
    title, changing 97  
**JSR-286**  
    about 23, 24  
    caching features 46  
    features 27  
    JAVA 5 features 45  
    Liferay portal support 23  
    resource URLs 43  
    runtime ID, sharing 47  
    taglib, utilizing 47  
**JSR-286 portlets**  
    about 27  
    cookies, setting 34  
    development environment 57  
    document head section elements, setting 34  
    extending 40  
    features 45  
    HTTP headers, setting 34  
    need for 27  
    portlet life cycle 27  
    portlet modes, utilizing 29  
    RENDER\_HEADERS part, adding 34  
    RENDER\_MARKUP part, adding 34  
    servlet-portlet relationship 33, 34  
    uses 27  
    window states, employing 31  
**JSR-286 portlets, extending**  
    container runtime options, utilizing 42  
    portlet-managed modes, utilizing 42  
    portlet filters, utilizing 40  
**JSTL** 132  
**JSTL library** 507

## L

**LAR import and export**  
    about 464  
    portlet, configuring with portlet data handler 465  
    portlet data handler, defining 464  
    portlet data handler, using 465  
    SCORM, using 466  
**layout templates**  
    about 324  
    book\_street\_home layout template, building 328  
    Book Street Home example 324  
    Book Workshop Home example 324  
    building, in Ext 324, 325  
    customized layout templates, adding 328, 329  
    custom layout templates 326, 327  
    custom layout templates, constructing 326  
    default layout templates, experiencing 326  
    developing, in Plugins SDK 332  
    DTD 331  
    Palm Tree Publications example 325

registering 330, 331  
standard layout templates 326

**layout templates, developing in Plugins**

- SDK**
- about 332
- layout templates, building 334, 335
- layout templates, creating 336-338
- Product Home use case 333

**layout templates, page management**

- adding 171, 172
- applying 170, 171
- displaying, by sections 173, 174
- layout templates, setting up 171
- pages, setting up 171
- portlet mappings, setting up 171

**License Key Request 360**

**Liferay CMS and WCM**

- about 11
- features 11

**Liferay collaboration and social networking software**

- about 12
- features 12

**Liferay portal**

- about 9, 23
- architecture 16
- Asset Publisher portlet 246
- business benefits 13
- community, building 277
- custom attributes 467
- development strategies 18- 21
- experiencing 24
- featured content 230
- features 9, 10
- framework 16
- journal articles, setting up 286
- local staging 434
- need for 13
- OpenSearch 467
- portal 24
- portlet 25
- portlet container 26
- ServiceBuilder 57
- speeding up 506
- Spring framework 17
- spring services 467
- terms of use 228

user comments, personalizing 297  
Web Content List portlet 239  
web services 467

**Liferay portal, best practices**

- about 502
- JavaScript Portlet URL, using 503
- portal, speeding up 506
- SharePoint, integrating with 512
- Terracota DSO, integrating with 513
- UI Taglibs, sharing in portlets 507
- user administration, customizing 504
- WSRP, consuming 509

**local staging**

- about 434
- activate staging checkbox, deselecting 436
- activate staging checkbox, selecting 436
- Book Street community example 435
- staging, activating 435
- staging environment 437

**local staging and publishing 441**

**location 177**

**LogicLibrary's Logiscan 15**

**login view**

- my account portlet, locating 306, 307
- overriding 307, 308

**M**

**mail hooks 417**

**mail portlet, My Social Office**

- about 392
- setting up 391

**Manage Pages portlet**

- customizing 150

**Manage Pages portlet, custmomizing**

- about 150
- Ext layout management portlet, building 152
- Ext layout management portlet, setting up in backend 156
- Ext layout management portlet, setting up in frontend 160

**Model-View-Controller (MVC) architecture 93**

**model listeners**

- about 431
- custom model listener, creating 432, 433

**Mule 452****my account**

customized account on fly, creating 309, 310  
customizing 304  
login view, customizing 305

**MyEclipse IDE**

about 65  
Eclipse plugin development tools 65  
server connectors 65

**My Profile**

about 369

**My Social Office**

building 389  
chat portlet, setting up 393  
mail portlet, setting up 391

**My Social Office, building**

about 407  
hooks, building 407, 408  
JSP hooks, employing 412, 413  
model listeners, employing 409  
portal event handlers, applying 408  
portal properties, erecting 410, 411

**MySQL**

about 59  
downloading 59  
installing 59  
MYSQL\_HOME variable, setting up 60

**my street example, personal community**

about 311  
my street portlet, building 314, 315  
user model, customizing 312, 313

**my street portlet, building**

games, adding 317  
my street theme, sharing 316  
playlists, adding 317  
Struts view page, adding 316  
videos, adding 317

**N****Navigation portlet 420****NetBeans IDE 62****O****OpenPortal Portlet Container 26****OpenSearch**

about 480

adding, on custom portlets 483, 485  
building 480-482  
search capabilities, adding in portlets 485  
Solr, used for enterprise search 486, 487

**organization administration**

about 505  
customizing 505

**P****page management**

customizing 162

**page management, customizing**

about 162  
layout templates, applying 170  
localized feature, adding 162  
pages, tracking 175  
tabs, employing 169

**page management, localized feature**

adding 162  
language properties, customizing 164, 165  
model, extending 163, 164  
multiple languages, displaying 166-168

**permissions, Struts portlet**

deploying 132  
setting up 126, 127  
setting up, in backend 128-130  
setting up, in frontend 130, 131

**personal community**

building 278  
content, sharing with friends 278  
dynamic query API, using 320, 321  
my account, managing 319, 320  
my community, setting 319  
my street example 311  
popups, using 321  
user account, extending 318  
user preferences, extending 318  
using 318

**personalization 13****Plugins SDK**

about 86, 367  
fast development, with Tomcat 89, 90  
features 367  
plugins, deploying 88  
project, building 87

setting up 86  
using 91, 368  
working 367

**Plugins SDK environment** 19

**popups**  
about 321  
Floating Div pop-up 321  
Floating Div pop-up, employing 321  
Window pop-up 321  
Window pop-up, employing 322

**portal** 24

**PortletContext object** 38

**portal development strategies**  
about 18, 20  
extension environment 18  
Plugins SDK environment 19

**portlet** 25

**PortletConfig interface** 37

**PortletConfig object** 37

**portlet configuration**  
border, hiding 35  
edit controls, hiding 36  
employing 35  
title, changing 35  
using 37

**portlet container** 26

**portlet context**  
employing 38

**PortletContext interface** 38

**portlet development structures**  
ext-impl/ 79  
ext-service/ 79  
ext-web/docroot/html/ 79  
ext-web/docroot/WEB-INF 80  
ext-web/tmp/ 80  
viewing 79

**portlet filters**  
about 40  
ActionFilter 40  
EventFilter 40  
RenderFilter 40  
ResourceFilter 40

**Portlet interface** 27, 28

**portlet life cycle, JSR-286 portlets**  
about 27  
destroy method 27  
diagrammatic representation 28

end of service 29  
initialization 28  
init method 27  
loading 28  
Portlet interface 27  
processAction method 27  
render method 27  
request handling 28

**PortletMode class** 30

**portlet modes, JSR-286 portlets**  
edit mode 29, 30  
help mode 29, 30  
utilizing 29  
view mode 29, 30

**portlet preference**  
working with 39

**PortletPreferences interface** 39

**portlet reports**  
customizing 35

**portlet Reports, specifying**  
liferay-display.xml 85  
liferay-portlet-ext.xml 84  
portlet-ext.xml 84  
struts-config.xml 85  
tiles-defs.xml 85

**portlet request**  
utilizing 38

**portletRequest.getWindowID method** 47

**PortletRequest interface** 38

**portlet response**  
utilizing 39

**PortletResponse interface** 39

**portlets**  
coordinating 48  
customizing 36  
customizing, via programming 36, 37

**portlets coordination**  
about 48  
example 48  
portlets events, utilizing 52  
public render parameters, utilizing 54  
sharing via the session 50

**portlets events, portlets coordination**  
events, receiving 53  
events, sending 52  
utilizing 52

**portlet URLs** 29

**processAction** method 27  
**processEvent** method 38, 52  
**propertyTag** tag 48  
**public render parameters**  
    employing 54, 55  
**publishing feature**  
    about 438  
    copy from live 439, 440  
    publish from live 440, 441

## R

**redirection mechanism Struts portlet**  
    about 118  
    action, updating 119  
    action paths, updating 120  
    existing JSP files, updating 120  
**remote staging and publishing**  
    about 441, 458, 460, 461  
    LAR exporting 464  
    LAR importing 464  
    setting up 459  
    using 458  
    working 461  
**remote staging and publishing, working**  
    exporting 462  
    importing 462  
    tunnel web, setting up 463  
    tunnel web, using 462, 463  
**render method** 27  
**ResourceServingPortlet interface** 43  
**ResourceURL interfaces** 43  
**resource URLs, JSR-286**  
    about 43  
    AJAX data, serving 45  
    levels, caching 44  
    parameters, setting on URL 43  
    serveResource calls, caching 44  
    utilizing 43, 44  
**resourceURL tag** 48  
**response.getNamespace** method 47  
**retrieve method** 115  
**role** 176  
**role-based staging workflow** 442  
**runtime ID, JSR-286**  
    about 47  
    sharing 47

## S

**SCORM** 466  
**send to friend button, implementing**  
    article template, building 286  
    email, setting up 282, 283  
**jQuery service, preparing** 285  
**share portlet, building** 282  
**view action, setting up** 282, 283  
**view page, setting up with jQuery** 283, 285  
**server connectors, MyEclipse IDE**  
    BejyTiger 65  
    JBoss 65  
    Jetty 65  
    Jonas 65  
    JRun 65  
    Oracle 65  
    Orion 65  
    Resin 65  
    Sun 65  
    Tomcat 65  
    Weblogic 65  
    WebSphere 65  
**serveResource** method 38  
**ServiceBuilder**  
    about 57  
    using, in Ext 78  
**ServiceBuilder, using in Ext**  
    about 78  
    portlet development structures, viewing 79  
    portlet Reports, developing 78  
    portlet specification, navigating 84  
    services, building 80  
**ServiceMix** 452  
**Service Oriented Architecture.** See SOA  
**services, ServiceBuilder**  
    building 80, 82, 83  
    building, Ant target used 83  
    service XML, creating 81, 82  
**setCacheability** method 44  
**setter** method 297  
**Sharable Content Objecct Reference Model.**  
    See SCORM  
**SharePoint**  
    integrating, with Liferay portal 512  
**share portlet**  
    building 282

**sharing via the session, portlets coordination**  
about 50  
page parameters, utilizing 51, 52  
PortletContext, utilizing 51  
PortletSession, utilizing 50, 51

**site map**  
establishing 424  
portlet view, building 425  
street site map portlet, constructing 425

**SOA** 16

**Social Office** 369

**Social Office, building with portlets**  
activities portlet 395  
building 394  
invite-members portlet 395  
JavaScript functions, raising 399, 400  
members portlet 395  
portlets project, rearing 396  
profiles portlet 395  
social portlets, assembling 396, 398  
social views, erecting 400, 401

**Social Office portlets**  
about 402  
social activity tracking, adding 405, 406  
social models, experiencing 402  
social services, experiencing 403

**Social Office theme**  
about 389  
developing 389  
differences of so-theme, constructing 390  
theme project, setting up 390

**Software Maintenance Renewal** 360

**Solr** 486

**Spring configuration** 492

**Spring framework**  
about 17, 487  
AOP 487  
DI 487  
IoC 487

**Spring services**  
about 487  
friendly URL mappings, customizing 496-498  
Liferay services, consuming in portlets 494, 495  
model name, changing via ServiceBuilder 490, 491

overriding 487-494  
overriding method validation 488, 489

**staging environment**  
group model 437  
layout model 437, 438  
layoutset model 438

**staging workflow**  
about 442  
activating 442  
BPMS | Intalio, using 453  
customizing 448  
jBPM workflow 452  
journal article workflow, employing 452  
MANAGE\_LAYOUTS permission 442  
MANAGE\_STAGING permission 442  
mechanism 444, 446, 447  
models, extending 448, 449  
proposal, creating 443  
role-based staging workflow 442  
standalone workflow portlet, building 449, 450

**StateAwareResponse methods** 52

**StateAwareResponse.setEvent method** 52

**structure, featured content**  
about 235  
building 236  
icon images, preparing 236

**Struts**  
uses 133  
using 134

**Struts portlet**  
building 108  
constructing 100  
defining 101, 102, 103  
need for 93

**Struts portlet, building**  
action, adding 108  
database, interacting with 112  
more action, adding 121  
permissions, setting up 126  
redirect mechanism 118

**Struts portlet, constructing**  
category, changing 107  
JSP pages, creating 105, 106  
page flow, specifying 103-105  
page layout, specifying 103-105  
title, changing 107

**styles and formats, FCKeditor**  
constructing 186  
CSS styles, preparing in themes 186, 187  
customized CSS styles, employing from themes 187  
stles, customizing 188

**Subclipse**  
about 64  
installing 64  
uses 65

**SWF 207**

**Sysdeo**  
about 66  
installing 66  
plugin, configuring 66

**Sysdeo Eclipse Tomcat Launcher plugin 66**

**T**

**taglib, JSR-286**  
about 47  
portletPreferencesValues variable 48  
portletPreferences variable 48  
portletSessionScope variable 48  
portletSession variable 48

**template, featured content**  
about 235  
building 237, 238

**templates, FCKeditor**  
building 189

**terms of use**  
dynamic terms of use, building 229, 230  
managing 228  
static terms of use, customizing 228, 229

**Terracota DSO**  
about 513  
integrating, with Liferay portal 513

**themes**  
building, in Plugins SDK 339  
CSS, applying 354  
debugging tools 355  
developing tools 355  
JavaScript, employing 354  
using 353

**themes, building in Plugins SDK**  
about 339  
Book Street Theme use case 339  
Book Workshop Theme use case 339

customized theme, creating 340

**themes, deploying**  
about 343  
color schemes, adding 351, 352  
CSS, experiencing 346, 347  
HTML, putting to use 345  
images, experiencing 346, 347  
jQuery JavaScript library, using 347  
runtime portlets, adding 353  
theme settings, employing 350, 351  
WAP themes 352

**tiles**  
need for 134

**TinyMCE**  
about 183  
configuring 184

**Tomcat**  
\$LIFERAY\_PORTAL, setting 61  
about 61  
installing, under \$Liferay Portal 61

**Tomcat plugins**  
about 65  
MyEclipse IDE 65  
need for 65  
Sysdeo 65

**tools, Ext set up**  
Ant 59  
JDK 58

**tout**  
about 258  
article touts, building 262  
structure, building 260, 262  
template, building 260, 262  
velocity services, adding 258, 259, 260

**U**

**UI tag**  
updating 302

**UI Taglibs**  
about 507  
sharing, in portlets 507, 508, 509

**Universal Description and Discovery Information.** See **UDDI**

**update method 122**

**user 176**

**user administration**  
about 504

columns of list, customizing 505  
customizing 504  
fields of form section, customizing 505  
new section, creating 505  
**user comments, personalizing**  
about 297  
Ext Comment portlet, building 300  
user comments model, creating 298, 299  
**user group** 177

## V

**Velocity templates, customizing in themes**  
about 355  
customized themes, setting up 366  
customized Velocity templates, adding 360  
default Velocity templates, using 356  
layout templates, setting up 366  
**view, Asset Publisher portlet**  
default tags, setting up 252, 253  
extending 252  
tags, configuring 252  
updating 253, 254  
**view counter, handling for assets**  
journal article tokens, using 292  
view counter, setting up on Document library documents 296  
view counter, setting up on Image Gallery images 295  
view count on blog entries, getting 294  
view count on Wiki articles, getting 293  
views on message boards threads, getting 294  
visits getting, on Bookmark entries 296  
**view page, Web Content List portlet**  
custom article types, adding 243  
custom article types, consuming 243, 245

## W

**WAP** 328  
**WCM and CMS.** *See CMS and WCM*  
**web content**  
layouts, scheduling 455, 456  
pages, scheduling 454  
scheduler class, configuring 457  
scheduler engine, setting 455  
scheduling 453, 454, 455

**Web Content Display portlet, featured content**

about 231  
customizing 231  
Ext Web Content Display 231  
view action, building 234, 235

**Web Content List portlet**

customizing 239  
Ext Web Content List portlet, constructing 240, 241  
view action, building 242  
view page, setting up 243

**weblog. See blog**

**web services**

about 498  
constructing 498  
consuming, in portlets 500, 502  
custom web serices, building 499

**Window pop-up** 322

**WindowState class** 32

**window states, JSR-286 portlets**

about 31  
maximized 32  
minimized 32  
normal 32

**WOL example, hooks**

about 416  
portlets 416

**Workflow engine** 15

**workspace** 63, 64

**WSRP**

about 509  
consuming 509, 510  
WSRP portlets, getting 510, 511  
WSRP portlets, working 511

**WYSIWYG editor**

about 179  
adding, in custom portlet 224, 225  
Ant target, extending 180  
configuring 179, 180  
employing, in portlets 223  
employing, in web content portlet 223  
FCKeditor 180  
features 225  
Liferay user-interface tag, using 224  
upgrading 181, 182  
using 225



## Thank you for buying Liferay Portal 5.2 Systems Development

### Packt Open Source Project Royalties

When we sell a book written on an Open Source project, we pay a royalty directly to that project. Therefore by purchasing Liferay Portal 5.2 Systems Development, Packt will have given some of the money received to the Liferay project.

In the long term, we see ourselves and you—customers and readers of our books—as part of the Open Source ecosystem, providing sustainable revenue for the projects we publish on. Our aim at Packt is to establish publishing royalties as an essential part of the service and support a business model that sustains Open Source.

If you're working with an Open Source project that you would like us to publish on, and subsequently pay royalties to, please get in touch with us.

### Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to [author@packtpub.com](mailto:author@packtpub.com). If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

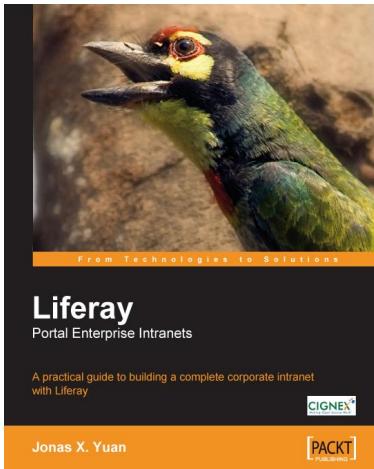
We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

### About Packt Publishing

Packt, pronounced 'packed', published its first book "Mastering phpMyAdmin for Effective MySQL Management" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution-based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our web site: [www.PacktPub.com](http://www.PacktPub.com).



## Liferay Portal Enterprise Intranets

ISBN: 978-1-847192-72-1      Paperback: 408 pages

A practical guide to building a complete corporate intranet with Liferay

1. Install, set up, and use a corporate intranet with Liferay – a complete guide
2. Discussions, document management, collaboration, blogs, and more
3. Clear, step-by-step instructions, practical examples, and straightforward explanation



## Alfresco Enterprise Content Management Implementation

ISBN: 978-1-904811-11-4      Paperback: 356 pages

How to Install, use, and customize this powerful, free, Open Source Java-based Enterprise CMS

1. Manage your business documents: version control, library services, content organization, and search
2. Workflows and business rules: move and manipulate content automatically when events occur
3. Maintain, extend, and customize Alfresco: backups and other admin tasks, customizing and extending the content model, creating your own look and feel

Please check [www.PacktPub.com](http://www.PacktPub.com) for information on our titles

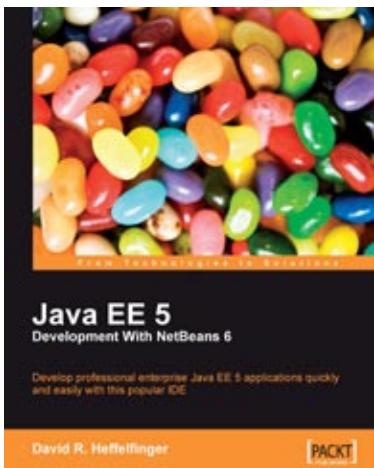


## Spring Web Flow 2 Web Development

ISBN: 978-1-847195-42-5      Paperback: 272 pages

Master Spring's well-designed web frameworks to develop powerful web applications

1. Design, develop, and test your web applications using the Spring Web Flow 2 framework
2. Enhance your web applications with progressive AJAX, Spring security integration, and Spring Faces
3. Stay up-to-date with the latest version of Spring Web Flow



## Java EE 5 Development with NetBeans 6

ISBN: 978-1-847195-46-3      Paperback: 400 pages

Develop professional enterprise Java EE applications quickly and easily with this popular IDE

1. Use features of the popular NetBeans IDE to improve Java EE development
2. Careful instructions and screenshots lead you through the options available
3. Covers the major Java EE APIs such as JSF, EJB 3 and JPA, and how to work with them in NetBeans
4. Covers the NetBeans Visual Web designer in detail

Please check [www.PacktPub.com](http://www.PacktPub.com) for information on our titles