

---

# APPLICATION DEVELOPER TRAINING

for

## LIFERAY 7.0

---



Copyright ©2016 Liferay, Inc.

All Rights Reserved.

No material may be reproduced electronically or in print,  
distributed, copied, sold, resold, or otherwise exploited  
for any commercial purpose without express written  
consent of Liferay, Inc.

7.3.0

---



# Contents

<b>1</b>	<b>Introduction to the Liferay Development Platform</b>	<b>5</b>
1.1	Course Topics . . . . .	5
1.2	Installing Developer Studio . . . . .	11
1.3	Liferay Core Concepts Review . . . . .	40
1.4	Developing For the Liferay Platform . . . . .	43
<b>2</b>	<b>Module Lifecycle</b>	<b>49</b>
2.1	Developing Applications . . . . .	49
2.2	What are Modules? . . . . .	60
2.3	What are Components? . . . . .	65
2.4	Applications Lifecycle . . . . .	73
2.5	Applications Lifecycle . . . . .	89
2.6	Debugging Applications . . . . .	101
<b>3</b>	<b>Building Modules</b>	<b>107</b>
3.1	Building a Module . . . . .	107
<b>4</b>	<b>Building Services</b>	<b>123</b>
4.1	Creating a Basic Service . . . . .	123
4.2	Implementing a Service . . . . .	130
<b>5</b>	<b>Java Standard Portlet</b>	<b>141</b>
5.1	Java Portlet Overview . . . . .	141
5.2	Portlet Lifecycle . . . . .	147
5.3	Interportlet Communication . . . . .	156
<b>6</b>	<b>Interacting with the Shell</b>	<b>165</b>
6.1	Shell Overview . . . . .	165
6.2	Bundle Installation . . . . .	175
6.3	Deployment Status . . . . .	182
<b>7</b>	<b>Building Portlet Modules</b>	<b>191</b>
7.1	Portlet Components . . . . .	191
7.2	Setting Attributes . . . . .	201
7.3	Creating the Presentation Layer . . . . .	213
7.4	The Controller Layer . . . . .	222

<b>8</b>	<b>Debugging Deployment</b>	<b>233</b>
8.1	Dependency Resolution . . . . .	233
8.2	Troubleshooting Deployment . . . . .	248
8.3	Discovering Services . . . . .	263
<b>9</b>	<b>Real World Application</b>	<b>279</b>
9.1	Project Overview . . . . .	279
9.2	Creating a Portlet . . . . .	288
9.3	Model and Persistence . . . . .	301
9.4	Service Layer . . . . .	316
9.5	View Layer . . . . .	322
<b>10</b>	<b>Liferay Utilities</b>	<b>337</b>
10.1	Common Utilities . . . . .	337
10.2	Tag Libraries . . . . .	344
<b>11</b>	<b>Feedback and Validation</b>	<b>353</b>
11.1	Validation . . . . .	353
11.2	Feedback . . . . .	367
<b>12</b>	<b>Liferay Services and Permissions</b>	<b>381</b>
12.1	How to Use . . . . .	381
12.2	Liferay Core Services . . . . .	389
12.3	Permissions . . . . .	394
12.4	Getting User Data . . . . .	413
<b>13</b>	<b>Integrating Liferay Frameworks</b>	<b>421</b>
13.1	Content Framework . . . . .	421
13.2	Assets . . . . .	432
13.3	Search and Indexing . . . . .	449
<b>14</b>	<b>Development Strategy</b>	<b>469</b>
14.1	Development Strategy . . . . .	469

## **Chapter 1**

# **Introduction to the Liferay Development Platform**



## COURSE SCHEDULE

Copyright ©2016 Liferay, Inc.  
All Rights Reserved.

No material may be reproduced electronically or in print,  
distributed, copied, sold, resold, or otherwise exploited  
for any commercial purpose without express written  
consent of Liferay, Inc.

### DAY 1

#### Introduction to Liferay

- » Setting Up a Development Environment
- » Concept Review: Liferay Core Concepts
- » Developing on the Liferay Platform

#### Module Lifecycle

- » Developing Applications in Liferay
- » What are Modules?
- » What are Components?
- » Application Lifecycle
- » Using the Shell

---

## DAY 1 CONTINUED

### Building Modules

- » Building Modules

### Building Services

- » Introduction to Service Architecture
- » Creating a Basic Service
- » Implementing a Service

---

WWW.LIFERAY.COM



---

## DAY 2

### Java Standard Portlet

- » Java Portlet Overview
- » Portlet Lifecycle
- » Interportlet Communication

### Interacting with the Shell

- » Shell Overview
- » Bundle Installation
- » Deployment Status

---

WWW.LIFERAY.COM



---

## DAY 2 CONTINUED

### Building Portlet Modules

- » Portlet Components
- » Setting Attributes
- » Creating the Presentation Layer
- » The Controller Layer

### Debugging Deployment

- » Dependency Resolution
- » Troubleshooting Deployment
- » Discovering Services

---

WWW.LIFERAY.COM



---

## DAY 3

### Real World Application

- » Project Overview
- » Creating the Portlet
- » Using Services to Model Data
- » The Model and Persistence Layer
- » Building the Service Layer
- » The View Layer

### Using Liferay Utilities

- » Common Utilities
- » Tag Libraries

---

WWW.LIFERAY.COM



---

## DAY 3 CONTINUED

### Feedback and Validation

- » Validation
- » Feedback and Localization

### Liferay Services

- » How to Use Liferay Services
- » Liferay's Core Services
- » Getting User Data

---

WWW.LIFERAY.COM



---

## DAY 3 CONTINUED

### Integrating with Liferay Frameworks

- » Content Framework
- » Assets
- » Search and Indexing

### Development Strategy

- » Upgrading Liferay

---

WWW.LIFERAY.COM



---

## PROVIDED SOFTWARE

The materials you have been provided include:

- » Liferay IDE
- » The Snippets plugin
- » The Solutions SDK
- » MySQL
- » The MySQL driver JAR
- » Java

---

## CLASS SCHEDULE

- » The class will divided up into sessions of approximately 90 minutes or less.
- » There will be shorter breaks at appropriate stopping points throughout the day, and a longer break for lunch.
- » We'll have 10 minutes for Q&A after each section. This is a good time to ask questions about recently-covered topics or how covered topics relate to each other.
- » We'll also have 30 - 60 minutes for Q&A at the end of each day. This is a good time to ask questions that are not covered in the course topics.



## INSTALLING DEVELOPER TOOLS

Copyright ©2016 Liferay, Inc.  
All Rights Reserved.

No material may be reproduced electronically or in print,  
distributed, copied, sold, resold, or otherwise exploited  
for any commercial purpose without express written  
consent of Liferay, Inc.

### OVERVIEW

- Liferay is tool-agnostic.
- Anything from a command prompt and text editor to a full-blown IDE can be used to develop on Liferay.
- Liferay IDE is a development environment provided by Liferay that has Liferay-specific features meant to simplify development.
- We will be using it for this training to streamline the setup process and help the exercises go more smoothly.
- You can use any tool you want to develop on Liferay, not just Liferay IDE.
- If you'd like to know how to set up Liferay with another development environment, our online Developer Guide covers this process:  
*[https://dev.liferay.com/develop/reference/-/knowledge\\_base/7-0/development-reference](https://dev.liferay.com/develop/reference/-/knowledge_base/7-0/development-reference)*

---

## LIFERAY IDE

- Liferay IDE is an Eclipse-based IDE that has been optimized for Liferay development, making it easy to get started developing apps.
- We have provided a copy of it with your training materials.

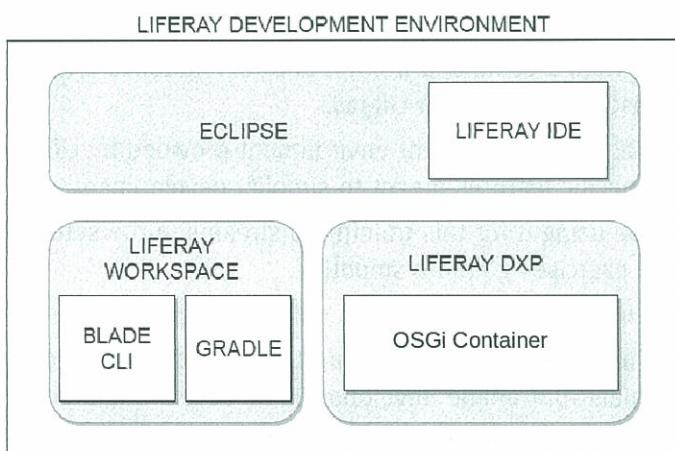
---

WWW.LIFERAY.COM



---

## LIFERAY DEVELOPER ENVIRONMENT STRUCTURE



---

WWW.LIFERAY.COM



---

## WHAT'S HAPPENING BEHIND THE SCENES?

- » Liferay IDE uses a *Liferay Workspace*.
- » A *Liferay Workspace* is a generated environment that manages your Liferay projects.
- » It provides a predictable folder structure, Gradle build scripts, and plugins to expedite the development process.
- » A *Liferay Workspace* can also be used outside of Liferay IDE.
- » A *Liferay Workspace* combines Gradle build scripts and plugins with a tool called *blade CLI*, all of which work together to make app development easier.
- » As you go through this course, you'll work your way up from no special tooling to a fully-configured environment in Liferay IDE.

---

## DO I HAVE TO USE ALL OF THESE?

- » We'll be using a number of different tools throughout the training.
- » You'll be able to see the wide range of options at your disposal.
- » Any toolchain you're comfortable with can be used to develop your Liferay applications.

---

## EXERCISE: SETTING UP LIFERAY IDE

- » Let's go ahead and install Liferay IDE.
  - 1. Find the *Liferay IDE* ZIP file for your platform.
    - » It will be clearly named, based on your OS, platform, and bitsize (32 or 64-bit).
  - 2. Create a directory called C:\liferay (or ~/liferay on a Unix-type system).
  - 3. Expand the contents of the archive in the directory you just created.
- ✓ Don't start Liferay IDE yet! We have some more configuration left, and starting Liferay IDE now would prevent this from being done correctly.

---

## EXERCISE: UNZIPPING THE BUNDLE

- » Liferay can come prepackaged with an application server. This prepackaging is known as a *bundle*.
  - » The bundle that we're going to be working with is Liferay bundled together with Tomcat.
1. Find the *liferay-dxp-digital-enterprise-tomcat-7.0-[version].zip* file.
  2. Expand the zip file into *C:\liferay* or */home/[your-user]/liferay* directory.

---

## EXERCISE: GETTING READY TO DEPLOY

1. Go to *liferay-dxp-digital-enterprise-[version]* once the unzipping process has finished.
    - As a brief side note, this area of the file structure where we see the *OSGi*, *data*, *tomcat-[version]* folder, and so on is known as *Liferay Home*.
  2. Create a folder called "deploy" inside Liferay Home.
  3. Copy the *activation-key-development-7.0de-liferaycom.xml* file from the materials provided to you.
  4. Paste the copy of the activation key into the deploy folder.
- ✓ The bundle is ready to be started, but we'll do that later. Let's move on to getting our exercise files for the training all squared away.

---

## EXERCISE: INSTALL EXERCISE FILES

1. Find an installer labeled *application-developer-install.jar* in your exercise materials.
2. Run the installer.
3. Double-click the file to run it (if you're running Windows or Mac).
4. Linux users should go to the file in the terminal and run the installer using this command:  
`java -jar application-developer-install.jar`

---

## TROUBLESHOOTING: JAVA NOT INSTALLED

- » If you're unable to run the above JAR file, you may not have Java or a full JDK installed.
- » You will need to install Java to go through the rest of the training.

---

WWW.LIFERAY.COM



---

## JAVA INSTALLER

- » For this course, you'll need to install JDK 8.
- » A fresh copy can be downloaded from <http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html>

---

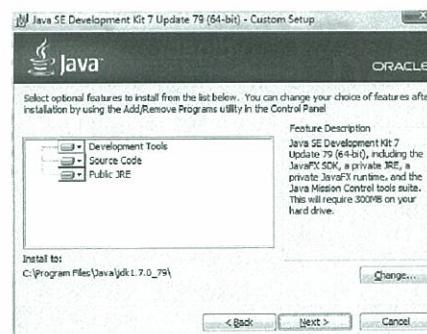
WWW.LIFERAY.COM



---

## EXERCISE: JAVA - RUN INSTALLER

1. Run the installer.
2. Click *Next*.
3. Click *Next* again, and leave all the default selections for the optional features.



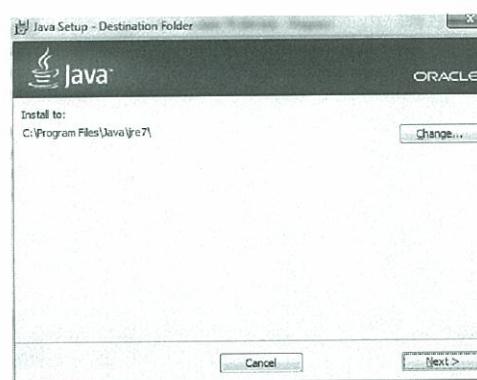
WWW.LIFERAY.COM

LIFERAY.

---

## EXERCISE: JAVA - JRE LOCATION

1. When prompted for a location to install the JRE, leave the default value.
2. Click *Next*.
3. Click *Close* once installation completes.



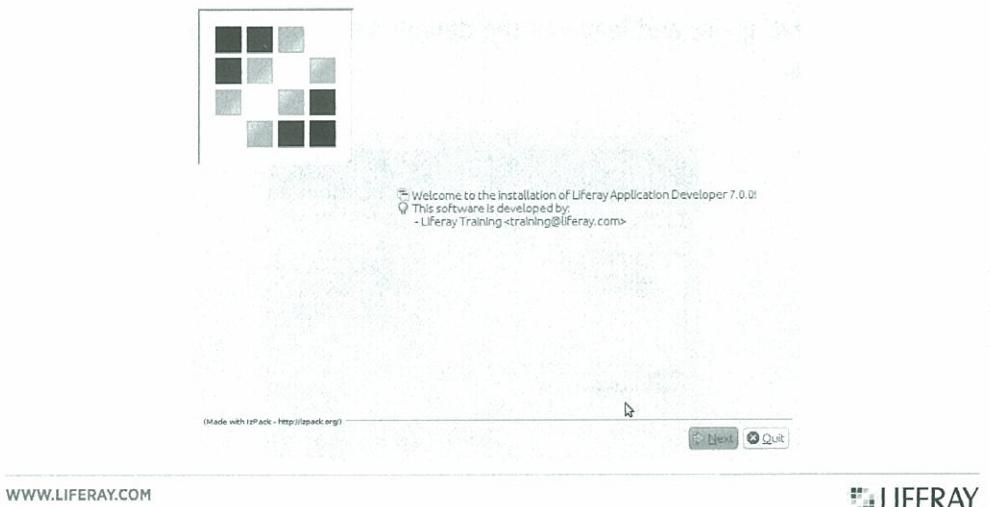
WWW.LIFERAY.COM

LIFERAY.

---

## EXERCISE: INSTALLATION INTRODUCTION

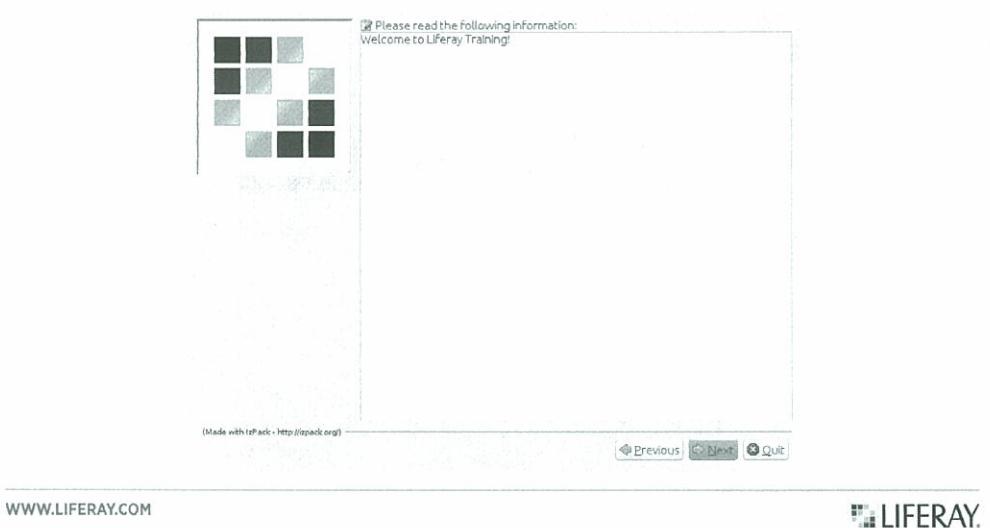
1. Click *Next* once the installer has opened.



---

## EXERCISE: WELCOME TO LIFERAY TRAINING

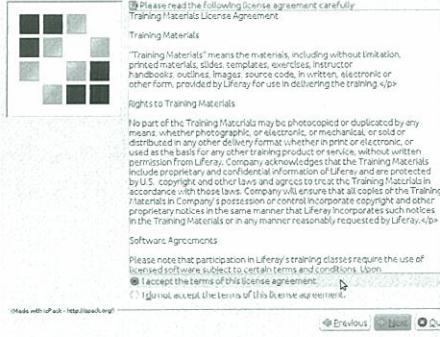
1. Click *Next*.



---

## EXERCISE: TERMS OF USE

1. Read through the Terms of Use.
2. Click “*I accept the terms of this license agreement*” when you’re done reading.
3. Click Next.



The screenshot shows a web page titled "Training Materials License Agreement". At the top, there is a decorative graphic of colored squares. Below it, a checkbox is checked with the text "Please read the following license agreement carefully". The main content is the "Training Materials License Agreement". It defines "Training Materials" as including不限于 (without limitation) printed materials, slides, templates, exercises, instructor handbooks, outlines, images, source code, in written, electronic or other form, provided by Liferay for use in delivering the training. It then discusses "Rights to Training Materials", stating that no part of the materials may be photocopied or duplicated by any means, whether photographic, or electronic, or mechanical, or sold or distributed in any other delivery format whether in print or electronic, or used in whole or in part, without prior written permission from Liferay. The company acknowledges that the Training Materials include proprietary and confidential information of Liferay and are protected by U.S. copyright and other laws. All rights are reserved. Any unauthorized copying or distribution is illegal. All ensure that all copies of the Training Materials in Company's possession or control incorporate copyright and other proprietary notices in the same manner that Liferay incorporates such notices in the Training Materials or in any manner reasonably requested by Liferay. The "Software Agreements" section notes that participation in Liferay's training classes require the use of licensed software subject to certain terms and conditions. Upon accepting the terms of this license agreement, the user agrees to the Software Agreements. A checkbox labeled "I agree to the terms of this license agreement" is checked. At the bottom, there are links for "Previous", "Next", and "Cancel".

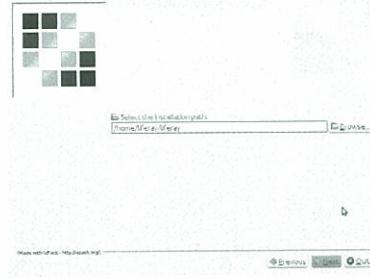
WWW.LIFERAY.COM

LIFERAY.

---

## EXERCISE: INSTALLATION PATHS

1. See the *Installation Path* text box:
  - **Mac:** should be `/Users/[your-user]/liferay`
  - **GNU/Linux:** should be `/home/[your-user]/liferay`
  - **Windows:** should be `C:\liferay`
  - In the image, the user happens to be called “liferay” as well.
2. Click Next.



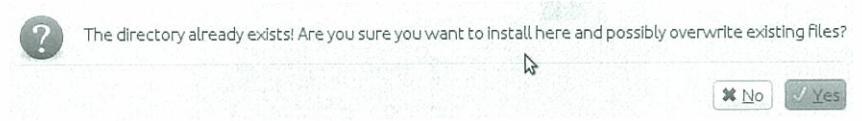
The screenshot shows a web page titled "Select Liferay installation path". It features a decorative graphic of colored squares at the top. Below it, there is a text input field with the placeholder "Select Liferay installation path" and the value "`/home/liferay/liferay`". At the bottom, there are links for "Previous", "Next", and "Cancel".

WWW.LIFERAY.COM

LIFERAY.

## EXERCISE: WARNING!

- » A warning will pop up, cautioning us that we may overwrite existing files.
- 1. Click Yes.
- » Don't worry; none of the things that we've done so far will be affected.
- » The installer will add all of our exercise files.



WWW.LIFERAY.COM

LIFERAY.

## EXERCISE: INSTALLING THE TRAINING MODULES

- » The next screen we see gives us a list of what is going to be installed and how much space we need.
- 1. See that *Training Modules* is selected.
- 2. Click *Next*.



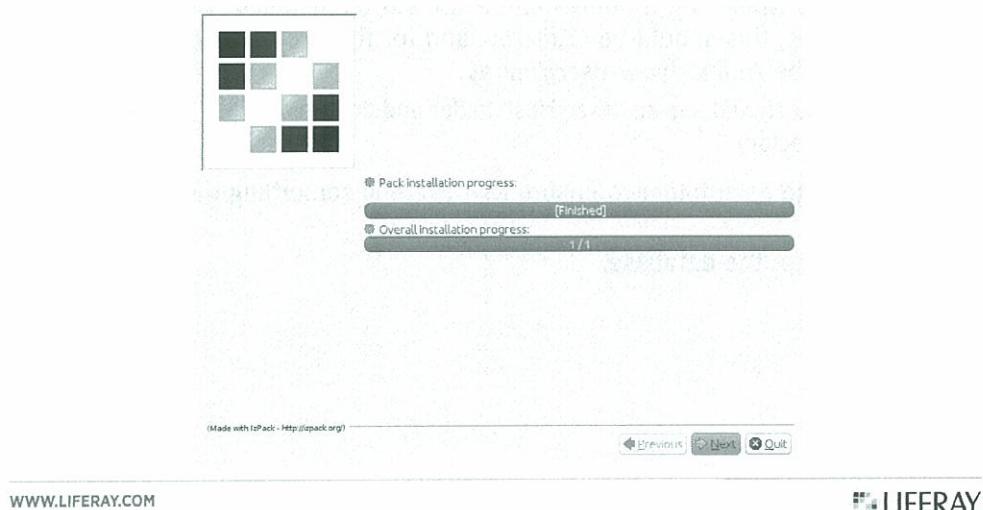
WWW.LIFERAY.COM

LIFERAY.

---

## EXERCISE: TRAINING MODULE INSTALLED

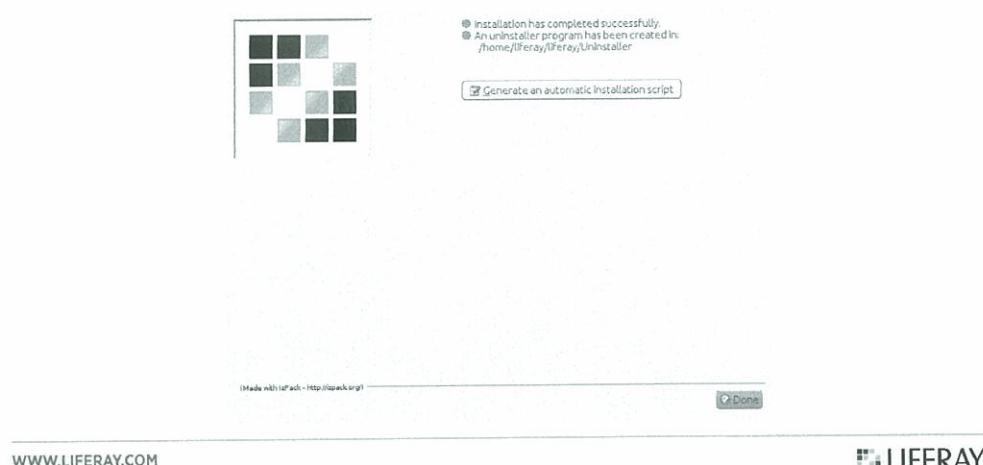
1. Click *Next* once the training module is installed.



---

## EXERCISE: ALL DONE

1. Click *Done*.
- ✓ We are now finished with the installation of the training module.



---

## EXERCISE: CONFIRMATION

- » Let's make sure everything that needed to happen, happened.
- 1. Go to where the training module should be installed. For those on Windows, this should be *C:\liferay*, and for those on Mac/Linux, this should be */home/[your-user]/liferay*.
  - » You should see an "Exercises" folder and an "Uninstaller" folder in the directory.
- 2. Click into each folder to ensure that there is something there.
  - » Next step, the database.

---

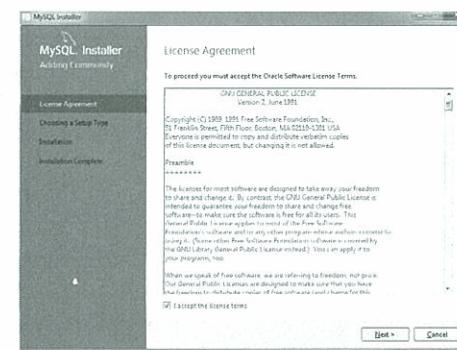
## MYSQL

- » MySQL is a leading open-source database.
- » It is widely used to power many websites.
- » It is small and fast, and its small footprint makes it ideal for training.

---

## EXERCISE: MYSQL - RUN INSTALLER

- ❖ A copy of MySQL can be found in the provided materials.
- ❖ If necessary, it can also be downloaded from <https://dev.mysql.com/downloads/mysql/5.6.html#downloads>.
- ❖ These instructions are for Windows. If you need Mac OS X instructions, skip down a few slides.
  1. Run the installer.
  2. Check “I accept the license terms”.
  3. Click Next.



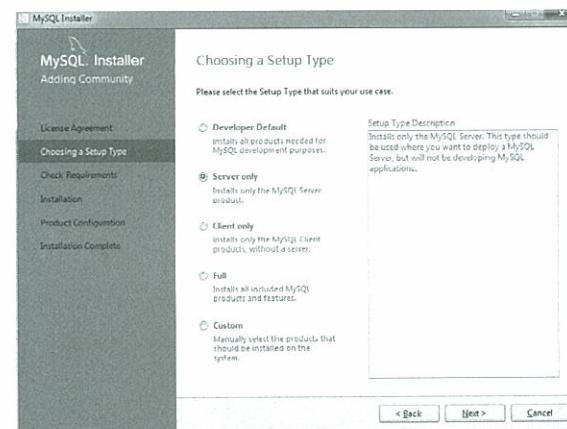
WWW.LIFERAY.COM

LIFERAY.

---

## EXERCISE: MYSQL - SETUP TYPE

1. Choose *Server only* on the next screen titled “Choosing a Setup Type”.
2. Click *Next*.



WWW.LIFERAY.COM

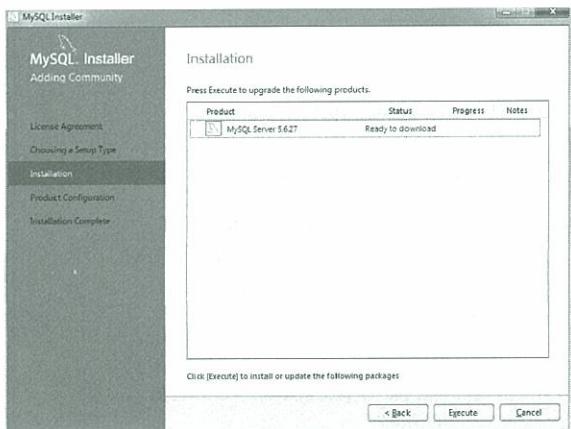
LIFERAY.

---

## EXERCISE: MYSQL - INSTALLATION

1. Click *Execute* on the next screen titled “*Installation*”.
2. Click *Next*.

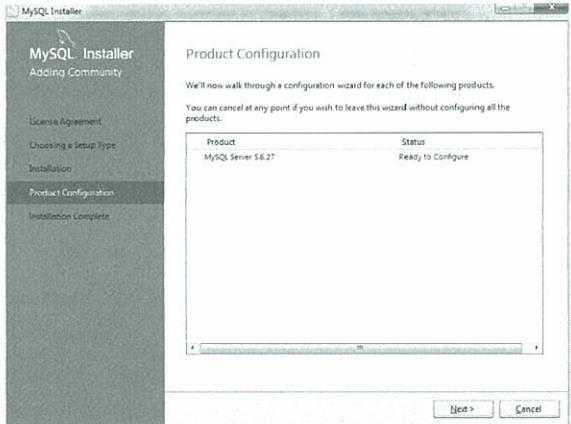
✓ Installation is complete.



---

## EXERCISE: MYSQL - PRODUCT CONFIGURATION

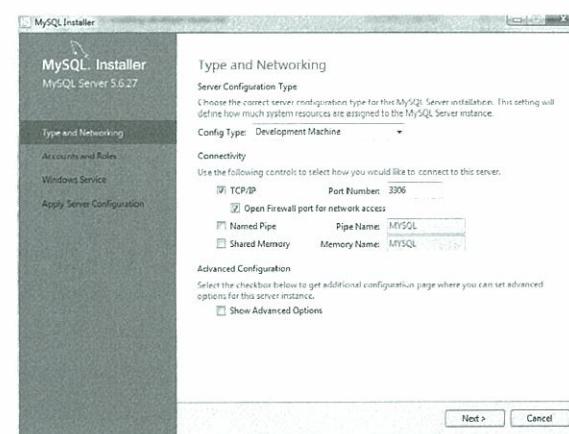
- » We will now configure the installed MySQL Server.
1. Click *Next*.



---

## EXERCISE: MYSQL - TYPE AND NETWORKING

1. See that the *Config Type* is set to *Development Machine*.
2. Click *Next*, leaving the default values for the rest of the fields.



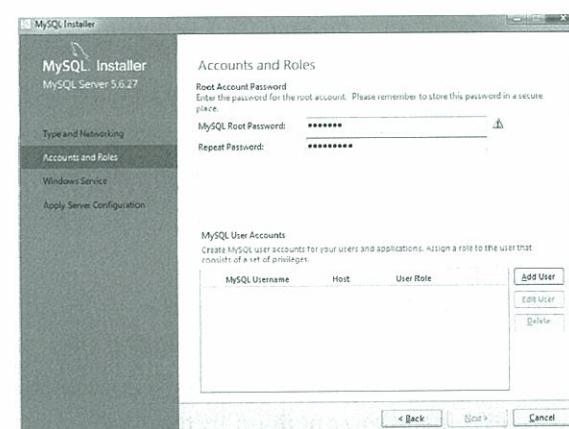
WWW.LIFERAY.COM

LIFERAY.

---

## EXERCISE: MYSQL - ACCOUNTS AND ROLES

1. Type *liferay* for the *Root Password*, and repeat it in the next field.
2. Click *Next*.

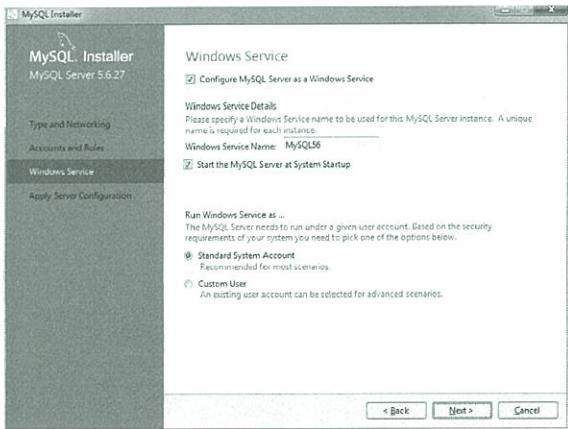


WWW.LIFERAY.COM

LIFERAY.

## EXERCISE: MYSQL - WINDOWS SERVICE

1. Click *Next*, leaving all of the default values.
  2. Click *Execute* on the next screen.
  3. Click *Finish* once that finishes.
- ✓ MySQL is now successfully installed!



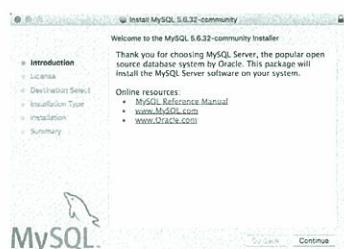
WWW.LIFERAY.COM

LIFERAY.

## EXERCISE: MYSQL INSTALLATION OS X - DOWNLOAD

➤ Although the principles remain the same, the specifics of the MySQL installation on Mac OS X are different from the installation on Windows.

1. Download the official DMG file for MySQL Community Server 5.6 here: <https://dev.mysql.com/downloads/mysql/5.6.html>.
2. Go to the download location in your file manager.
3. Double-click the `mysql-[version]-x86_64.dmg` file.
4. Double-click on the PKG file to launch the PKG installer contained in the DMG.



WWW.LIFERAY.COM

LIFERAY.

---

## EXERCISE: MYSQL INSTALLATION OS X - AGREE TO TERMS

1. Click *Continue* on the first page of the installer.
2. Click *Continue* once you've taken a look at the software agreement.
3. Click *Agree*.

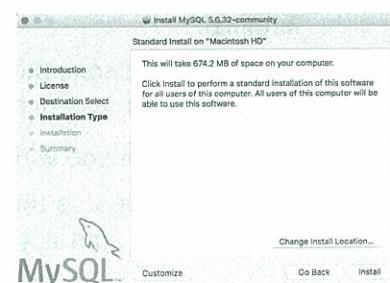
WWW.LIFERAY.COM

LIFERAY.

---

## EXERCISE: MYSQL INSTALLATION OS X - INSTALL

1. Click to accept the defaults.
2. Click *Install*.
3. Type your password if prompted.
4. Click *Close* when the installation completes.



WWW.LIFERAY.COM

LIFERAY.

---

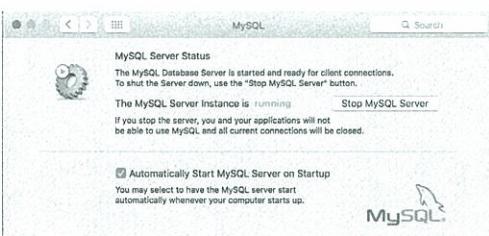
## EXERCISE: MYSQL - THE TRICKY PART

- » In order to use MySQL without having to reference the entire directory structure every time you want to run a command, you'll need to add its bin/ folder to your path.
  - » In order to add the bin/ folder to your path, you'll need to edit a hidden file called `.profile`, which is found in your home directory.
1. Open `~/.profile` using the Terminal app or third party file browser of your choice.
  2. If the file doesn't exist, create it.
  3. Add the line:  
`export PATH=/usr/local/mysql-5.6.32-osx10.11-x86_64/bin:$PATH`
  4. Restart all terminal apps to refresh the profile, making sure the folder name exactly matches the location where you installed MySQL.

---

## MANAGING MYSQL ON OS X

- » To test that MySQL installed properly, run `mysql --version` in your Terminal.
- » You should see something like:  
`mysql Ver 14.14 Distrib 5.6.32, for osx10.11 (x86_64) using EditLine wrapper`
- » MySQL provides you with some basic controls in System Preferences.
- » You can start or stop the server here, as well as configure whether or not MySQL will start automatically with your system.



---

## MAC OS X MYSQL PASSWORDS

- » Another thing to note is that by default, your MySQL will be configured with no root password.
- » You can leave this as is, but be sure to take note of instructions or snippets that use a password for MySQL.
- » Alternatively, you can set the password by running the following command from the Terminal:

```
mysqladmin -u root password liferay
```

---

## EXERCISE: LET THE CAT OUT OF THE JAR

- » Liferay also needs the MySQL connector JAR file, which can be downloaded at <https://dev.mysql.com/downloads/connector/j/>.
1. Download the connector JAR if you haven't done so already.
  2. Move the connector JAR into `/tomcat-[version]/lib/ext`.
- ✓ This will help us out when we start our Liferay bundle, so sit tight for now while we continue with our database creation.

---

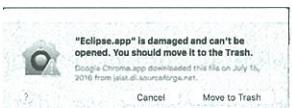
## EXERCISE: CREATING OUR DATABASE

- » Let's go ahead and create our database.
  1. Click on MySQL 5.6 Command Line Client in your *Start Menu*.
  2. Type your **Root Password** (*liferay*).
  3. Type the command `create database lportal character set utf8;`
    - » You should see a response that says `Query OK, 1 row affected (0.02 sec)`.
  4. Exit the Command Line Client.
- » Next, let's create our *Liferay Workspace*.

---

## TROUBLESHOOTING: ECLIPSE IS "DAMAGED"

- » There is a known issue on Mac, where the `Eclipse.app` is recognized as "damaged" and won't launch.

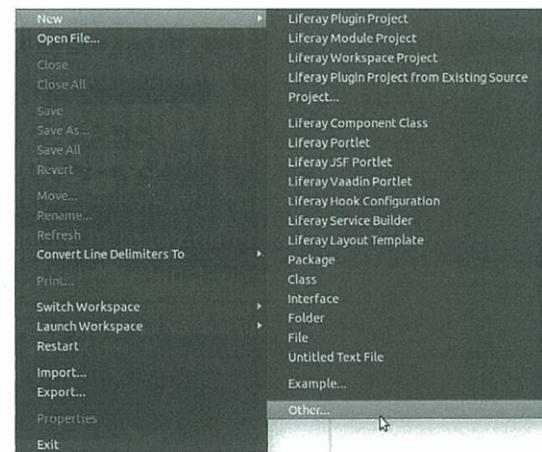


1. Go to the `/[user-home]/liferay` folder in your Terminal app to solve this problem.
  2. Run the command `xattr -d com.apple.quarantine Eclipse.app/`.
- ✓ Now you can start Liferay IDE with no further issues.

---

## EXERCISE: LIFERAY WORKSPACE CREATION

1. Start Liferay IDE.
2. Click *OK* when prompted for a workspace, leaving the default value.
3. Create a new *Liferay Workspace* once Liferay IDE has started.
4. Click on *File* → *New* → *Other....*



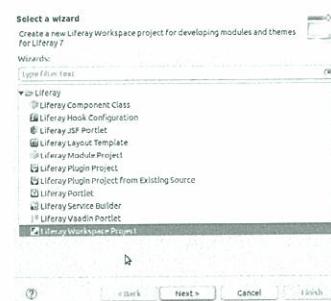
WWW.LIFERAY.COM

 LIFERAY.

---

## EXERCISE: SELECTING THE WORKSPACE OPTION

1. Find the Liferay Folder.
2. Expand it out.
3. Choose *Liferay Workspace Project*.
4. Click *Next*.

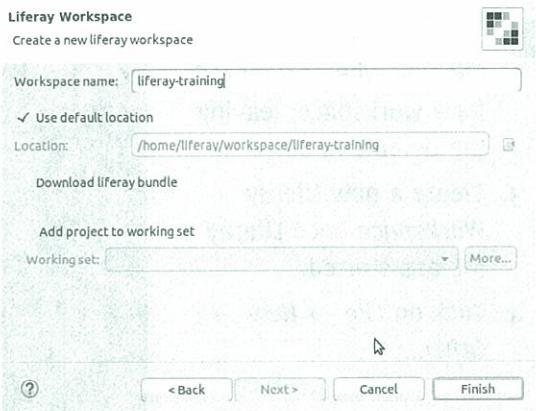


WWW.LIFERAY.COM

 LIFERAY.

## EXERCISE: LIFERAY WORKSPACE SETUP

1. Type `liferay-training` for the name of the workspace.
  2. Click *Finish*.
  3. Click *Yes* when the window asks about *Open Perspective*.
- ✓ Our *Liferay Workspace* is all set up.



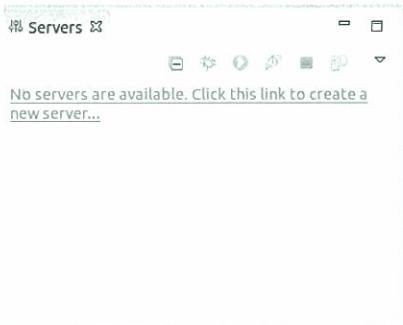
WWW.LIFERAY.COM

LIFERAY.

## EXERCISE: CONNECTING TO OUR BUNDLE

Remember that Liferay bundle we unzipped earlier? Now it's time to connect that to our Liferay IDE.

1. Click the *No servers are available. Click...* link at the bottom-left corner under the *Server* tab.



WWW.LIFERAY.COM

LIFERAY.

## EXERCISE: SELECTING OUR SERVER

1. Expand the *Liferay, Inc.* folder out once you've clicked the link either by double-clicking the folder or clicking on the drop-down arrow.
2. Choose the *Liferay 7.x* option and leave the defaults.
3. Click *Next*.



WWW.LIFERAY.COM

LIFERAY.

## EXERCISE: ADDING THE BUNDLE TO BE CONNECTED

1. Click on the *Browse...* button.
2. Go to where the bundle has been unzipped.
  - Windows users will find it at *C:\liferay*.
  - Linux and Mac users will find it at */home/[your-user]/liferay*.
3. Choose the bundle. You do not have to go to the bundle itself.
4. Click *Finish*.

The screenshot shows the 'Liferay Portal Runtime' configuration dialog. It asks to specify the installation directory of the portal bundle. The 'Name' field is set to 'Liferay 7.x'. The 'Liferay Portal Bundle Directory' field contains '/home/liferay/liferay/liferay-dxp-digital-enterprise-7.0-ga1'. The 'Detected portal bundle type' dropdown is set to 'tomcat'. The 'Select runtime JRE' dropdown is set to 'java-8-oracle'. At the bottom, there are buttons for '?', '< Back', 'Next >', 'Cancel', and 'Finish'.

WWW.LIFERAY.COM

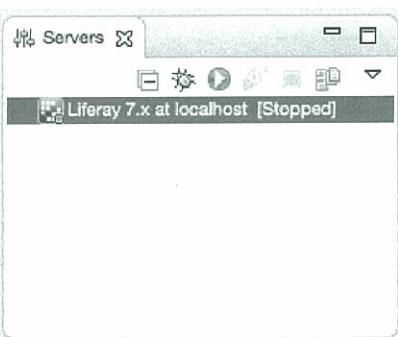
LIFERAY.

---

## EXERCISE: STARTING LIFERAY

» Now it's time to actually start Liferay.

1. Click the green "play" button to start the server inside Liferay IDE in the Server pane.



WWW.LIFERAY.COM

LIFERAY.

---

## SERVER STARTUP

» Once the server starts up (after you see some nice ASCII art), Liferay will open our default browser to the Basic Configuration screen.



WWW.LIFERAY.COM

LIFERAY.

---

## EXERCISE: SETUP WIZARD

1. Type *S.P.A.C.E* for the *Portal Name*.
2. Uncheck the *Add Sample Data* box.
3. Type the first name *Space*, the last name *Admin*, and the email address *space.admin@spaceprogram.liferay.com* for the *Administrator User*.
4. Click *Change* in the *Database* section.



WWW.LIFERAY.COM

LIFERAY.

---

## EXERCISE: MYSQL SETUP

1. Choose MySQL in the new section that opens up.
2. Type *root* for *User Name*.
3. Type *liferay* for *Password*.
4. See that the *JDBC URL* matches the database you created.

Database

Use Default Database

Database Type

MySQL

JDBC URL

com.mysql.jdbc.Driver

User Name

root

Password

Launch Configuration

WWW.LIFERAY.COM

LIFERAY.

---

## EXERCISE: FINISHING MY SQL SETUP

1. Click *Finish Configuration*.
2. Restart Liferay by clicking on the green "play" button used to start the server in the IDE.

---

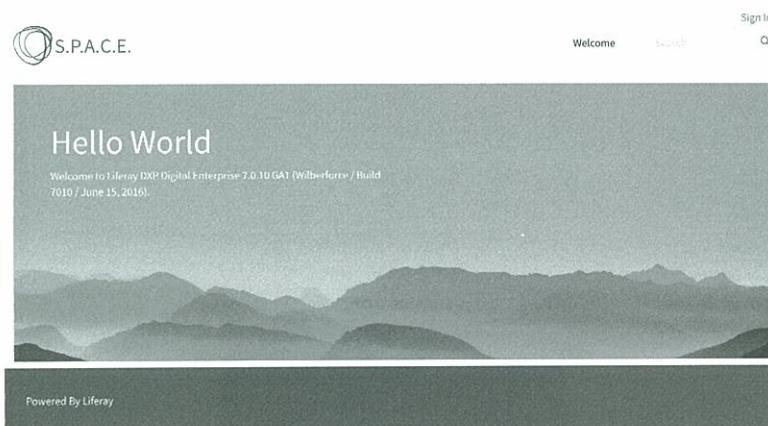
## EXERCISE: SIGN-IN WIZARD

1. Sign In using the login info below.
  - » Email Address: test@liferay.com
  - » Password: test
2. Read through the Terms of Use.
3. Click *Accept* after reading.
4. Choose a password reminder query.

---

## EXERCISE: THE FINISH LINE

- ✓ Now you're ready to go.



WWW.LIFERAY.COM

LIFERAY.

---

## EXERCISE: FINDING THE SNIPPETS

- › Before we get on with our fun coding adventures, let's make sure the snippets have imported correctly.
  1. Go to *Window* → *Show View* → *Other...*
    - › If you are one of the lucky ones, you may see "Snippets" as one of the default options under *Show View*.
  2. Type "snippets" in the search bar that pops up.
  3. Choose *Snippets*.
  4. Click *OK*.

WWW.LIFERAY.COM

LIFERAY.

---

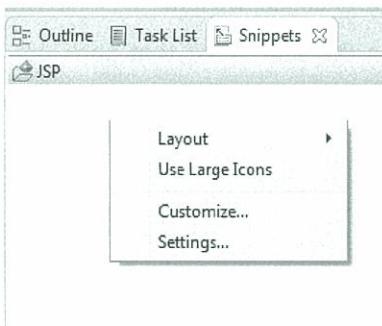
## EXERCISE: CHECKING THE SNIPPETS

1. Find the *Snippets* tab in the IDE.
2. Click on it.
3. See that the tab is populated with sections such as *01-Building a Bundle* and so on.

---

## EXERCISE: MANUALLY IMPORTING SNIPPETS

- » If the snippet plugin import failed, you can add the snippets manually.
  1. Right-click in the *Snippets view*.
  2. Click *Customize*.



---

## EXERCISE: SELECTING SNIPPETS TO IMPORT

1. Click *Import*.
2. Go to the directory containing the provided training materials.
3. Click the XML files from the snippets folder to import them.

Notes:



## CONCEPT REVIEW: LIFERAY CORE CONCEPTS

Copyright ©2016 Liferay, Inc.  
All Rights Reserved.

No material may be reproduced electronically or in print,  
distributed, copied, sold, resold, or otherwise exploited  
for any commercial purpose without express written  
consent of Liferay, Inc.

### CONCEPT REVIEW

- In order to develop applications for Liferay, you should have an understanding of Liferay's core concepts.
- If you've taken our *Liferay Fundamentals* training, you should have a thorough understanding of the basics of Liferay.
- If not, most of the concepts should be familiar to those with Web Development experience.

---

## CORE CONCEPTS: USERS AND PERMISSIONS

- » Liferay is a **Digital Experience Platform**. A Digital Experience Platform provides a way for **Users** to interact with **Content**.
- » Users can be members of **User Groups** and/or **Organizations**.
- » Users can be assigned functions via **Roles**.
- » Roles are collections of **Permissions** that can be created or customized to meet specific needs.
- » User Groups and Organizations can also be assigned to Roles, essentially granting the Role's permission set to every member.

---

## CORE CONCEPTS: SITES, SCOPE, AND CONTENT

- » All **Content** is created in a **Scope**.
- » The Scope can be **Global**, for a specific **Site**, or for a specific **Page**.
- » All **Pages** are created inside of **Sites**.
- » Content can only be displayed on a page.

---

## CORE CONCEPTS: APPLICATIONS

- » Applications allow Users to create and interact with content and perform functions.
- » In our course, we will work to understand how to create applications on the Liferay Platform and how to leverage all of Liferay's capabilities to meet the needs of your users.

---

WWW.LIFERAY.COM



Notes:



## LIFERAY DEVELOPMENT ENVIRONMENT

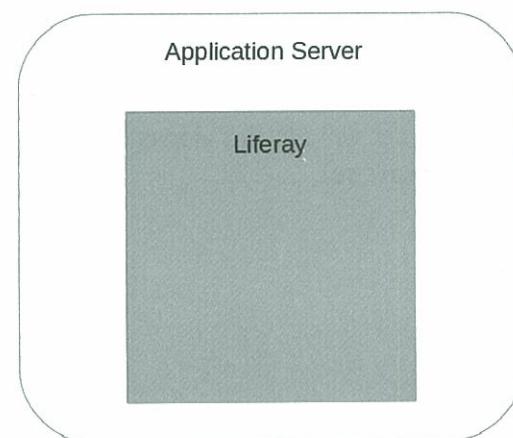
Copyright ©2016 Liferay, Inc.

All Rights Reserved.

No material may be reproduced electronically or in print,  
distributed, copied, sold, resold, or otherwise exploited  
for any commercial purpose without express written  
consent of Liferay, Inc.

### THE LIFERAY PLATFORM

- » Liferay is a web application.
- » Liferay can run on any Java EE application server or servlet container.
  - » Apache Tomcat
  - » Glassfish
  - » Jboss
  - » And many others



---

## WRITING CODE FOR LIFERAY

- Developing for Liferay only requires a standard Java EE development environment.
- You can use any tools/IDE that you are comfortable with:
  - Eclipse on Windows
  - Netbeans on a Mac
  - Emacs on Linux
- Liferay is tool-agnostic, meaning that there are no special tools required for development.

---

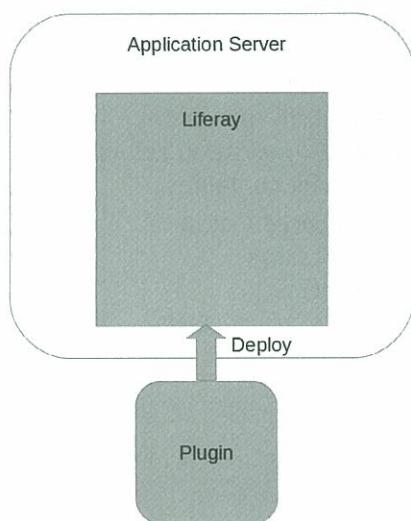
WWW.LIFERAY.COM



---

## CREATING PLUGINS

- We as developers are going to be developing plugins for Liferay.
- These plugins will be installed in Liferay, but will not run alongside Liferay (per se).



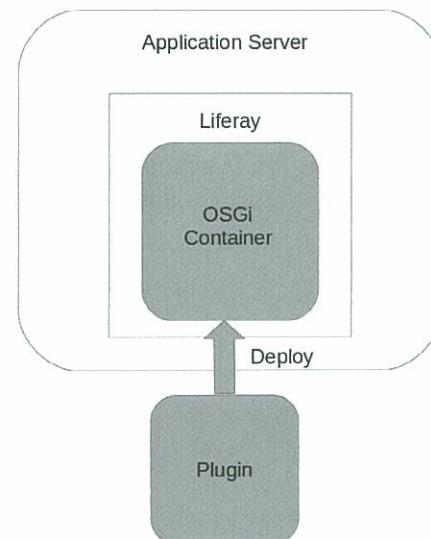
---

WWW.LIFERAY.COM



## THE OSGI CONTAINER

- » Within Liferay is the *OSGi Container*.
- » When we install plugins or *modules*, we are actually installing them into Liferay's *OSGi Container*.
- » The *OSGi Container* is responsible for managing our plugins (*modules*) in Liferay.



WWW.LIFERAY.COM

LIFERAY.

## TOOLS TO DEVELOP PLUGINS

- » Although you can create plugins from scratch, there are a number of tools that help make plugin creation easier.
  - » Liferay blade CLI: <https://github.com/liferay/liferay-blade-cli/>
  - » Bnd: <http://bnd.bndtools.org/>
  - » Bndtools in Eclipse: <http://bndtools.org/>
  - » Osmorc for IntelliJ: <https://www.jetbrains.com/idea/help/osmorc.html>
  - » PAX: <https://ops4j1.jira.com/wiki/display/ops4j/Pax>

WWW.LIFERAY.COM

LIFERAY.

---

## BUILD TOOLS

- » Liferay also allows developers to use build tools they are already familiar with, such as:
  - » Maven: <https://maven.apache.org/>
  - » Gradle: <http://gradle.org/>
  - » Ant: <http://ant.apache.org/>
  - » Ivy: <http://ant.apache.org/ivy/>

---

## WHAT WE ARE USING

- » First off, we're going to be running JDK 8.
- » Our database of choice will be MySQL.
- » For our IDE, we'll be using Liferay IDE GA2.
  - » Alternatively, we can use Eclipse Mars 2 JEE.
- » We will also be using blade CLI, Bndtools, and *Liferay Workspace*.

---

## LIFERAY WORKSPACE

- » *Liferay Workspace* is an all-in-one development environment built on Gradle, Bnd, and other Liferay Gradle plugins.
- » It uses a folder structure to hold all the plugins and adds them to subprojects so they can be referenced in larger projects.
- » *Liferay Workspace* helps build our plugins or *modules* using Liferay best practices.

---

## DEVELOPMENT THROUGHOUT THE TRAINING

- » We'll start developing plugins from scratch so we can really get an idea of how things work.
- » As we progress, we'll add different tools to show how easy it can be to develop plugins.
- » Eventually, we'll be going the Liferay route of development and using Liferay tools.
- » Again, you're not bound to any one toolset and are free to use what you want.

---

## NOTE TO PREVIOUS LIFERAY USERS

- » Previously, we've used a Plugins-SDK to develop the various Liferay plugins.
- » We are no longer developing using the Plugins-SDK.
- » Although the Plugins-SDK is no longer the primary way to develop plugins, for now, it can still be used.
- » All of our previous plugin types (portlets, themes, etc.) will be built as *modules*.

---

WWW.LIFERAY.COM



Notes:

## Module Lifecycle

### Chapter 2

## Module Lifecycle



## DEVELOPING APPLICATIONS IN LIFERAY

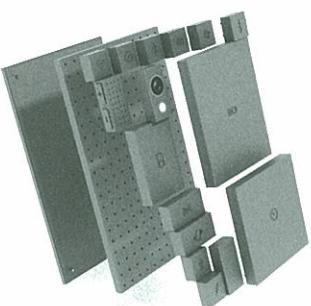
Copyright ©2016 Liferay, Inc.

All Rights Reserved.

No material may be reproduced electronically or in print,  
distributed, copied, sold, resold, or otherwise exploited  
for any commercial purpose without express written  
consent of Liferay, Inc.

### APPLICATIONS IN LIFERAY

- Liferay is a robust platform for running applications.
- Apps can range from simple commands in a shell to complex, multi-layered suites.
- The Liferay Platform is a *modular approach* to development.



## DEVELOPMENT PARADIGMS

- » Apps can be designed as single, unitary programs, built from the ground up and fully integrated.
- » Apps can also be designed as *modular* pieces.
- » Each piece adds functionality to the app.

## MONOLITHIC APPLICATIONS

- » Apps can be *monolithic*.
- » Apps have a flat structure, and all their functionality is self-contained.
- » Apps can have layers to logically separate functionality.



---

## MODULAR APPLICATIONS

- Apps can be *modular*.
- Their structure can be simple or complex.
- Their functionality is separated into individual *modules*.
- Each *module* is self-contained.



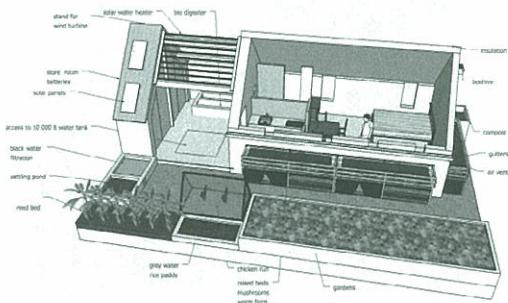
WWW.LIFERAY.COM

LIFERAY.

---

## A REAL MONOLITH

- Getting a handle on *monolithic* versus *modular* in the abstract is difficult.
- These development styles exist in the real world.
  - Example: houses, model buildings



WWW.LIFERAY.COM

LIFERAY.

---

## A SOLID BUILDING

- » We can build a model house or building from wood, paper, plaster, etc.



---

## MONOLITHIC FEATURES

- » Some benefits:
  - » Easy to build quickly
  - » Easy to make cosmetic changes
  - » Simple structure
- » Some drawbacks:
  - » Hard to change structure
  - » Difficult to expand/contract
  - » If part of it breaks, the whole thing is damaged

---

## A MODULAR BUILDING

- » We can build a model house or building from uniform, simple *modules*.



WWW.LIFERAY.COM

LIFERAY.

---

## MODULAR FEATURES

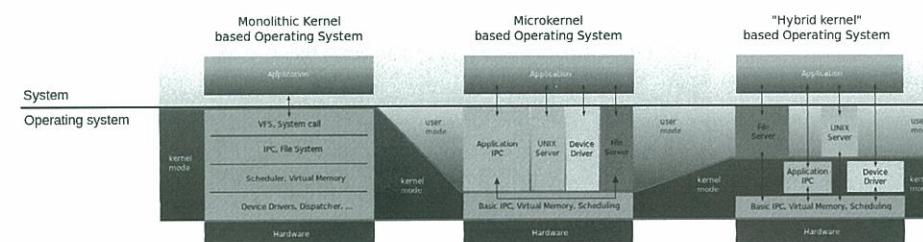
- » Some benefits:
  - » Easy to build
  - » Each module is self-contained
  - » Easy to repair
  - » Easy to make any change
  - » Can build large or small
  - » Can change structure dynamically
- » Some drawbacks:
  - » Simple structure more complex than monolithic
  - » Simple changes may need more effort
  - » More steps to build some structures

WWW.LIFERAY.COM

LIFERAY.

## A MONOLITH IN TECH

- » Monolithic development is also found in operating systems.
- » Can be built on a *monolithic* kernel
  - » DOS, Windows 9x, UNIX, Linux (0.01–1.2)
- » Or built on a *modular* micro or hybrid kernel
  - » Minix, QNX, Darwin (OS X, iOS), Linux (1.2+)

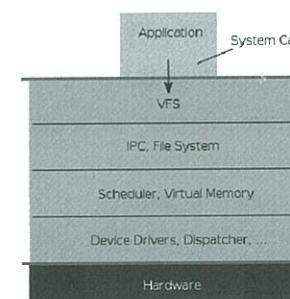


WWW.LIFERAY.COM

LIFERAY.

## MONOLITHIC KERNEL

- » A *monolithic* kernel controls the system, from low-level hardware to user-level services:

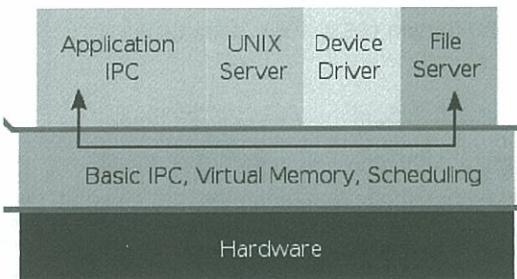


WWW.LIFERAY.COM

LIFERAY.

## MODULAR KERNEL

- » A *modular kernel* (*hybrid* or *micro kernel*) controls the low-level hardware and provides a **platform** for modules to run on:

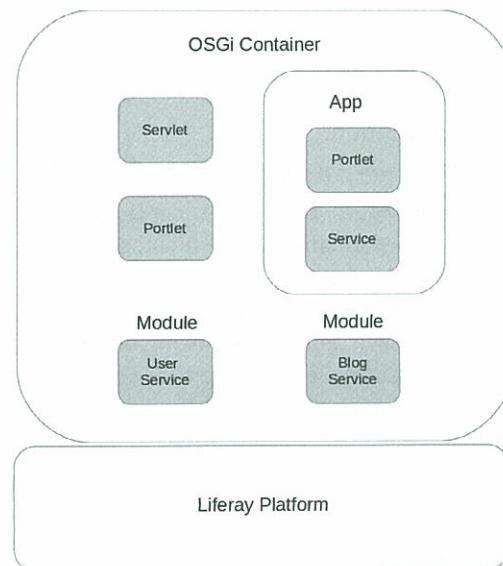


## LIFERAY: A MODULAR PLATFORM

- » Liferay is like an operating system:
  - » Provides a platform to run web applications on
  - » Interacts with low-level services (request processing, etc.)
  - » Simplifies development with additional useful services and features (User access, content management, etc.)
- » Liferay contains a solid platform to run on.
- » Liferay's features are implemented as separate modules.
- » Highly dynamic, flexible environment:
  - » Modules can be installed anytime.
  - » Code can be hot-swapped at runtime (upgrades, implementation changes, etc.)
  - » Modules can be removed or stopped at will.

## THE OSGI CONTAINER

- » Liferay provides a modular framework for development.
- » Modularity is implemented using an *OSGi Container*.
- » The *OSGi Container* is a dynamic development environment in Liferay.



WWW.LIFERAY.COM

LIFERAY.

## DEVELOPING APPS AS MODULES

- » Making apps in the OSGi Container is simple:
  - » Lots of small apps can work together as an *app suite*.
  - » Each app contains any number of features – deployed as one or more *modules*.
  - » Modules can implement features using small, functional objects – *components*.
- » In this paradigm, simple to complex applications can be broken down similarly.

WWW.LIFERAY.COM

LIFERAY.

---

## A MODULAR CAMPUS

- » Let's apply this architectural concept to an app we might write in Liferay.
- » Developing at a University, we may need to write a course catalogue application.
- » A *course catalogue* can be built with different logical layers:
  - » **Model:** the Course object, containing name, description, code, and more information about a course, and storing and retrieving courses from the database
  - » **View:** the HTML, CSS, and JavaScript used to show the course information, such as a course listing or details
  - » **Controller:** The logic for controlling page flow, and sending course (*model*) information to the View
- » What would this look like as a modular app?

---

## APP ARCHITECTURE

- » The Course storage and retrieval, View displaying the course catalogue, and application business logic can run in independent *modules*:
  - » Each module can be added or removed at runtime.
  - » Modules talk to each other to pass information.
  - » Functionality is self-contained, but can be shared between modules.
- » For instance, we could build the app by creating these modules:
  - » **Service module:** contains everything for storing and retrieving *model* data (the Course)
  - » **Controller module:** contains all business logic for the application (switching views, getting a *model* and passing it to the View, etc.)
  - » **Webapp module:** contains HTML, CSS, images needed to display the course information and code necessary to build those displays

---

## A NEW DEVELOPER STORY

- » Breaking down apps into self-contained, independent modules
- » The *OSGi Container* simplifies development in modules.
- » Liferay's *OSGi Container* is implemented using the OSGi standard.
  - » *OSGi Core 6*
  - » *OSGi Compendium 5*
- » Walking through some real apps, you'll see how well modules work together.
- » Develop robust apps in a new paradigm.

---

WWW.LIFERAY.COM



Notes:



## WHAT ARE MODULES?

Copyright ©2016 Liferay, Inc.  
All Rights Reserved.

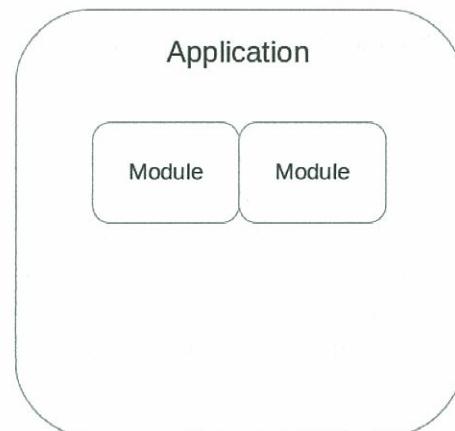
No material may be reproduced electronically or in print,  
distributed, copied, sold, resold, or otherwise exploited  
for any commercial purpose without express written  
consent of Liferay, Inc.

### WEB APPLICATIONS OUTSIDE OF LIFERAY

- If you've developed or deployed web applications before, you've seen them come packaged as WAR files or EAR files.
- Within a WAR file, for example, are all the relevant files of the web application.
- Most of the time, you'll only have one WAR file for your application.
- The WAR file is a container that allows a web application to be deployed to an application server.
- Apps we'll build for Liferay are often web applications.
- They'll contain many of the same files and even Java classes as standard webapps.
- They're just packaged a bit differently.

## APPLICATIONS INSIDE OF LIFERAY

- » In Liferay, when developing an application, we think in terms of *modules*.
- » An **application** is made from *one or more modules*.
- » A **module** is a self-contained unit that deploys within Liferay.
- » A **module** is the *container* for your app's features.

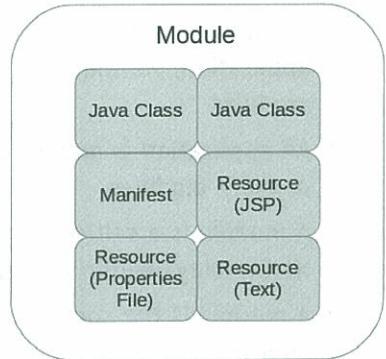


## WHAT IS A MODULE?

- » **Modules** are the *unit of deployment* in Liferay.
- » You'll also hear them referred to as *bundles*.
- » A **bundle** is the JAR file that *contains a module*.
- » Similarly to how the WAR file is the JAR file that contains a servlet, the *bundle* contains a *module*.
- » All *bundles* are standard JAR files.
- » A **bundle** contains:
  - » Java classes
  - » A manifest file
  - » Resources:
    - » JSPs
    - » Properties files
    - » Binary or text data

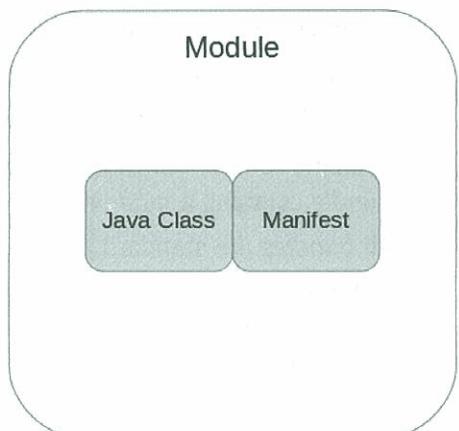
## WHAT'S IN A MODULE BUNDLE?

- » A *bundle* is the only type of file you deploy in Liferay and is the basic unit of development.
- » Bundles can contain any combination of Java classes, JSPs, properties files, images, or any other binary and text data that your application might need.
- » One bundle contains one module.
  - » The bundle is the physical representation of a module.
- » Modules are packaged in a bundle as basic JAR files and are deployed as JAR files.



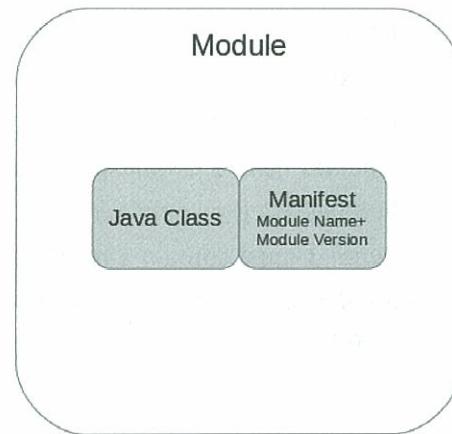
## THE ESSENTIALS OF A BUNDLE

- » In order for a bundle to be valid, the JAR file needs:
  - » A valid manifest file (MANIFEST.MF) with the correct bundle headers
- » Java classes and other binary data are not necessary to make a valid bundle.
- » Resources are not necessary for a bundle to be valid, but are used by your application.



## IDENTIFYING A MODULE

- Modules are deployed to the same container.
- Telling the difference between modules is essential for modularity.
- Modules provide a **unique identifier** in their headers.
- The *unique identifier* of a module is composed of:
  - Module Name
  - Module Version
- We declare the module name and module version headers in the module's *manifest file*.

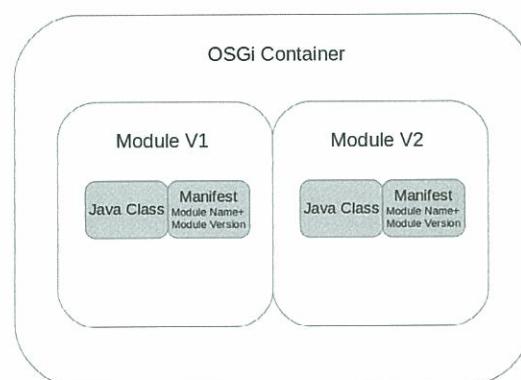


WWW.LIFERAY.COM

LIFERAY.

## ONE MODULE, TWO VERSIONS

- If a module is deployed with a different version number, the OSGi Container will see the module as a unique module.
- Since each module has a different *unique identifier*, they are seen as different modules.
- Unexpected problems can come up as a result.



WWW.LIFERAY.COM

LIFERAY.

---

## MODULES IN REVIEW

- » **Applications** can be monolithic or modular in structure.
- » **Modules** are the basic unit of modularity for modular *applications*.
- » *Modules* contain functionality for the *application*.
- » *Modules* have a *unique identifier*: name and version number.
- » **Bundles** are the container for a *module*.
- » *Bundles* must contain a manifest.
- » *Bundles* are the basic packaging (JAR) and unit of deployment for a *module*.



## DEVELOPING WITH COMPONENTS

Copyright ©2016 Liferay, Inc.  
All Rights Reserved.  
No material may be reproduced electronically or in print,  
distributed, copied, sold, resold, or otherwise exploited  
for any commercial purpose without express written  
consent of Liferay, Inc.

### INSIDE OUR MODULES

- Modules make up the architecture of our application.
- Features can be implemented in modules any way we want.
- Inside a module, we can have Java classes, resource files, libraries, and anything we need to provide functionality.
- Similar to the way we can organize our application into modules, we can also organize what's inside a module.
- Features we need can be implemented in a modular way *inside* a module.
- Functionality can be implemented using components.

---

## A COMPONENT BY ANY OTHER NAME

- » Many of us are already familiar with using *components*:
  - » Enterprise Java Beans
  - » Swing components
  - » CDI components
  - » Google Guice components
  - » JSF components
- » Using components to build out the features of an application is conceptually no different from using components to build out a web UI.

---

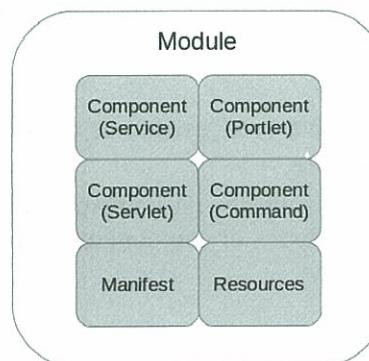
## WHAT IS A COMPONENT?

- » Applications are made of
- » Modules - *units of deployment* - and they contain
- » Components - *objects that provide functionality*
- » A component is an object that provides specific functionality – it implements a feature.
- » Some examples of a component:
  - » Service
  - » Servlet
  - » Portlet
  - » Shell Command
  - » Controller Action
- » Components provide features within an application.

---

## WHERE DO COMPONENTS LIVE?

- » Components are contained inside of *modules*.
- » Components structure the implementation of a module.
  - » Modules structure the implementation of an application.



---

## WHY COMPONENTS?

- » Components provide an easy, concrete way to build features in a module.
- » Each component is self-contained:
  - » Easy to test
  - » Easily replaced
  - » Reusable
- » Components with the desired features are snapped together, like building blocks, to create a module.
- » Components are not bound to a specific application and can be reused throughout any number of applications.
- » If I create a component that retrieves Course information from the databases, I can reuse that component in another module for a different application.

## CONCRETE COMPONENTS

- » You can use many different implementations of components, like a few familiar ones:
  - » Enterprise Java Beans
  - » Spring Components
- » The OSGi Container provides component solutions:
  - » Blueprint
  - » Declarative Services
- » We'll be using **Declarative Services (DS)** as our component implementation of choice.
- » You're not limited — use whichever implementation you like best. You can even use different types of components in the same module:
  - » DS Components
  - » JSF Components

## A TASTE OF DS

- » To show you how easy it is to make a component, let's consider a simple example:

```
public class MoJo {  
    public static String execute() {  
        return "MoJo POJO!";  
    }  
}
```

- » We want to make this POJO a *component*:

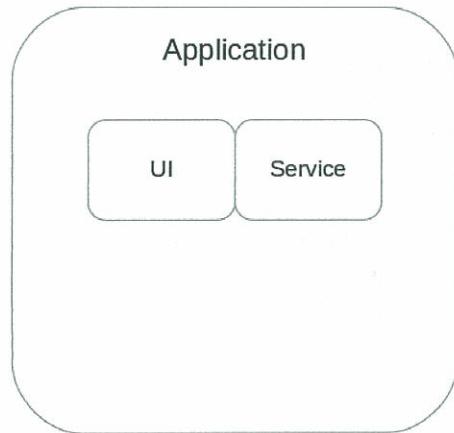
```
@Component  
public class MoJo {  
    public static String execute() {  
        return "MoJo POJO!";  
    }  
}
```

- » That's it! The @Component annotation does all the work for us.

- » As we make more components, we'll use more features that come with *Declarative Services*.

## BUILDING A SIMPLE APPLICATION

- » Suppose we want to build out a simple application to allow students to register for classes:
  - » A simple UI to view class schedules and click a *Register* button
  - » A service that retrieves the course information for display
  - » A service that allows the student to be added to the class roster

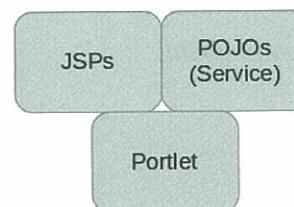


WWW.LIFERAY.COM

LIFERAY.

## PIECES OF THE APPLICATION

- » Let's go ahead and lay out everything we need to fulfill our requirements:
  - » The front end UI can be implemented using JSPs.
  - » We'll use POJOs to implement our services to retrieve course details and store student registration info.
  - » Finally, the Controller can be a Java standard portlet.
- » Now that we know what pieces are going to be used, how does this fit with modules and components?



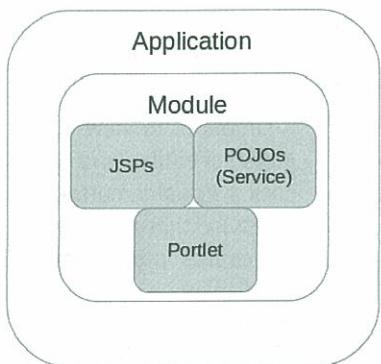
WWW.LIFERAY.COM

LIFERAY.

---

## DEVELOPING WITH EASE

- » We've identified the major areas of our app we might break into *modules*.
- » One way is to simply put everything into one *module* –
  - » A portlet class implements the basic Controller functionality and forwards to JSPs.
  - » JSPs are added as resources containing our UI.
  - » POJOs implement simple services called by our Controller.



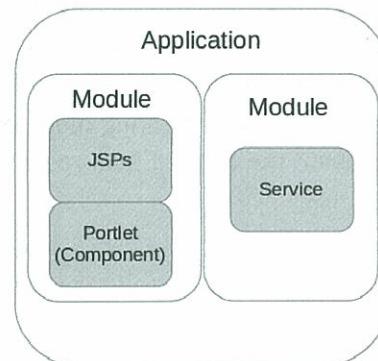
---

## DEVELOPING WITH COMPONENTS

- » We could also separate our app layers into modules –
  - » **Web module** – containing the portlet and JSPs, the View, and the Controller
  - » **Service module** – containing the Java interfaces of our services and the POJOs implementing the services
- » In both cases, we can create reusable, self-contained *components*:
  - » A *component* implementing each service
  - » A *component* implementing the portlet (*Controller*)
- » Each *component* can be started and stopped independently.
- » Each *component* can be replaced at runtime, if needed.
- » It's easy to test each *component*.

## DEVELOPING THE MODULAR WAY

- » By separating the service into its own module, it's easy to test, replace, and reuse.
- » It also makes it a bit easier to share functionality across applications.



WWW.LIFERAY.COM

LIFERAY.

## EXERCISE: DEPLOYING COMPONENTS

- » Just to see what a component looks like in the wild, we've built a simple one to get you started:
  1. Copy the `training-component.jar` from `exercises/application-developer-training-exercises/02-module-lifecycle` into the `/LIFERAY-HOME/deploy` directory.
  2. Go to the running Liferay instance via a web browser.
  3. Click on the *Add* button on the *Control Menu* at the top of the page.
  4. Add the Training Portlet Component to the page under the Sample Category.
- ✓ See that the component is used on the page:



WWW.LIFERAY.COM

LIFERAY.

---

## PUTTING IT ALL TOGETHER

- » Modules are the basic deployable container used in Liferay.
- » Modules contain the functionality for your application.
- » Modules can be built with *components*.
- » *Components* are simple objects that provide functionality.
- » The component provides the implementation of a feature for an application, such as business logic, lifecycle management, and services.
- » We can build out applications by creating modules containing components that provide the desired features and services.

Notes:



## SHARING FEATURES BETWEEN MODULES

Copyright ©2016 Liferay, Inc.  
All Rights Reserved.

No material may be reproduced electronically or in print,  
distributed, copied, sold, resold, or otherwise exploited  
for any commercial purpose without express written  
consent of Liferay, Inc.

### WHAT'S IN AN APP?

- » Building apps from modules means:
  - » Separating functionality
  - » Creating modules that encapsulate functionality
  - » Communicating between modules to form a *single unit*

---

## TALK TO ME

- Modules need to talk to each other in order to provide useful functionality.
- Apps can provide features to other apps.
- Modules need to:
  - *Provide* features to other modules
  - *Use* features from other modules
- Modules are packaged as *bundles* – simple JAR files.
- Bundles can tell Liferay what they want to share by:
  - **Exporting Java packages**
  - **Providing capabilities**

---

## SENDING OUT PACKAGES

- The most basic thing a module can share is, well, *Java classes*.
- Let's say I write some handy utility class for formatting Course information for display in a University catalog.
- I use that functionality in my own app, and think it may be useful to other apps.
- I'd like to share those Java objects with other modules.

---

## EXPORTING OUR GOODS

- » We can share Java classes between modules by **exporting** and **importing** packages.
- » My Course formatter is a handy object called `CourseFormatterUtil`.
- » It's located in the package `edu.diversity.platform.tools`.
- » My module can **export** this package in the bundle header:  
`Export-Package: edu.diversity.platform.tools`

---

## IMPORTING WHAT'S GOOD

- » Another team is building another app to display information about a random course on a University Department's homepage.
- » They'd love to use our Course formatter.
- » Since our module's installed and **exports** the *package*, they can easily use it.
- » The new module can **import** the package in the bundle header and use the functionality:  
`Import-Package: edu.diversity.platform.tools`

---

## SEPARATING CONCERNS

- » It's easy to share Java objects by exporting their packages.
- » When we're sharing functionality, we want to separate the *implementation* of the feature from the *declaration* of the feature.
- » We can build out a definition of the feature — its **contract** with other modules.
- » Then we can implement the feature **based on the contract**.
- » With a separate implementation, a module only needs to know the terms of the *contract*, not how its implemented.
- » This is easy to accomplish by building *services*.

---

WWW.LIFERAY.COM

LIFERAY.

---

## SERVICE-BASED ECONOMY

- » A basic way to communicate is through *services*.
- » Services can be easily made available in Liferay.
- » A *service* published in the OSGi Container is called an **OSGi Service**.
- » Services make new functionality available to modules.

---

WWW.LIFERAY.COM

LIFERAY.

---

## SERVICES PROVIDE CAPABILITIES

- New functionality made available by *services* can be described as **Capabilities**.
- A module can **provide Capabilities** to other modules.
- Bundles can declare what *packages* we want to **export**:  
`Export-Package: com.liferay.training.test`
- Bundles can also declare what *Capabilities* we want to **provide**:  
`Provide-Capability: osgi.service`
- Packages describe a range of classes and interfaces other modules can use.
- Capabilities describe functionality provided to other modules.

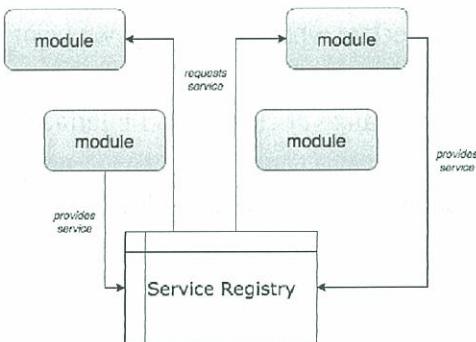
---

## SERVICE-ORIENTED FRAMEWORK

- Modules can *declare* what functionality they provide as an **OSGi Service**.
- Modules can *ask* for specific functionality in an **OSGi Service**.
- The container provides:
  - A dynamic environment for publishing services
  - Free interaction between modules
  - Registering and unregistering modules and their services anytime

## MODULE TRAFFIC COP

- » An environment where services are added and removed easily needs to be managed.
- » The *OSGi Container* manages this interaction between modules using the Service Registry.



WWW.LIFERAY.COM

LIFERAY.

## CALLING ALL SERVICES!

- » The **Service Registry** is like a switchboard that connects modules with **OSGi Services**:
  - » Modules can call up the registry and ask to be connected to a specific service.
  - » Modules can also call up the registry and add a new **OSGi Service**.
- » Liferay handles this complexity, so modules can simply plug in and be connected.

WWW.LIFERAY.COM

LIFERAY.

---

## DESIGNING FUNCTIONALITY

- » What does it take to create a new OSGi Service?
- » We *design* a service API.
- » We **export the package** of the Java interfaces that make up the API — our *contract*.
- » Then, *implement* the API in an OSGi Service.
- » Next, *call* the Service Registry.
- » Your information is now available to any *module* as an OSGi Service.
- » They can **import the package** of the Java interfaces that define the service *contract*.
- » But how can we *use* an OSGi Service, once it's registered?

---

WWW.LIFERAY.COM



---

## I JUST DESIGNED YOU

- » Like all good explanations, it begins with a story:
- » In a lonely area of Moduleville (the OSGi Container), a lone *module* is in search for someone.
- » He only has a name (the service API name).
- » He doesn't know where to find this service.
- » What is our poor module to do?

---

WWW.LIFERAY.COM



---

## AND THIS IS CRAZY

- » He finds a famous private investigator who specializes in finding services.
- » The PI takes the (service API) name, and goes looking for the service.
- » Where can our intrepid investigator find the service?

---

## HERE'S MY INTERFACE

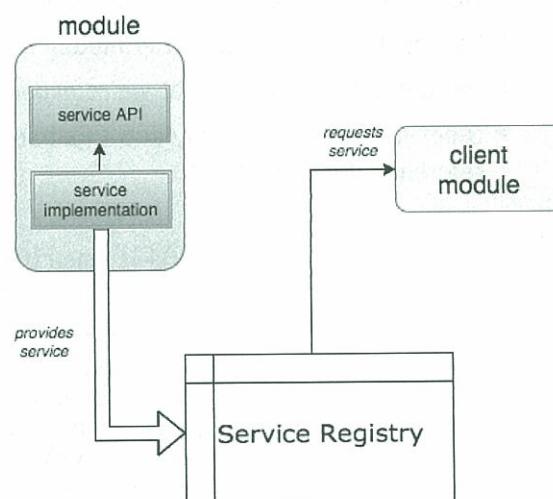
- » The PI heads over to the switchboard (the Service Registry) with the name (the **service API**).
- » Leaning over, he *listens* for the name he needs – and grabs the *number* (the **service implementation**).

## GIVE ME A SERVICE, MAYBE?

- » The PI runs back to our waiting module, and quickly passes him the *number* (the service implementation).
- » Our module is now free to use this service at will!
- » Our service mystery is solved!

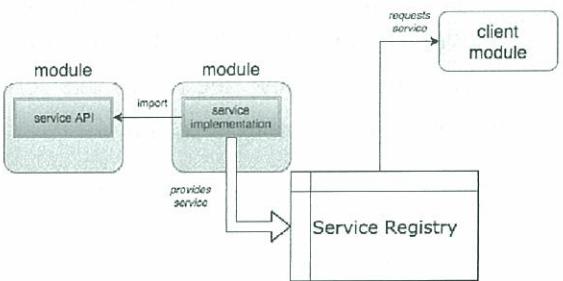
## THE REAL DEAL

- » Fun stories aside, what does *creating an OSGi Service* look like in a real bundle?
- » All services consist of:
  - » API
  - » Implementation
- » We can build one module that contains both.
- » This module will:
  - » Export the interfaces
  - » Implement the interface in a POJO



## SEPARATING CONCERNS

- » To be more modular, we can build two modules: an **API module** and an **service module**:
  - » The API module contains all of the necessary interfaces to *define* the service API.
  - » The implementation module:
    - » *implements* the interfaces in a POJO
    - » *publishes* the OSGi Service to the Service Registry



WWW.LIFERAY.COM

LIFERAY.

## DEFINE THE API

- » We need to define the API for our new service – DogeService.
- » Encapsulate the API in an **API module**.
- » Tell the OSGi Container that we're *exporting* the API package.
- » Other modules can simply add this bundle (JAR) to their build path to reference the service API.

### API module

```
«interface»  
DogeService  
+very(String): String  
+such(String): String
```

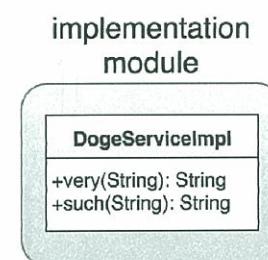
WWW.LIFERAY.COM

LIFERAY.

---

## FILL IN THE BLANKS

- To implement the service, we'll create a new module – the **service module**.
- Tell the framework we need to *import* the API package.
- We can also tell the framework we're *providing* a new Capability.
- Create objects that implement the API in the module.



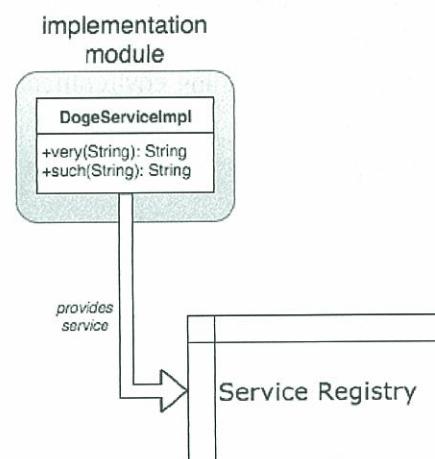
WWW.LIFERAY.COM

LIFERAY.

---

## CONNECT THE DOTS

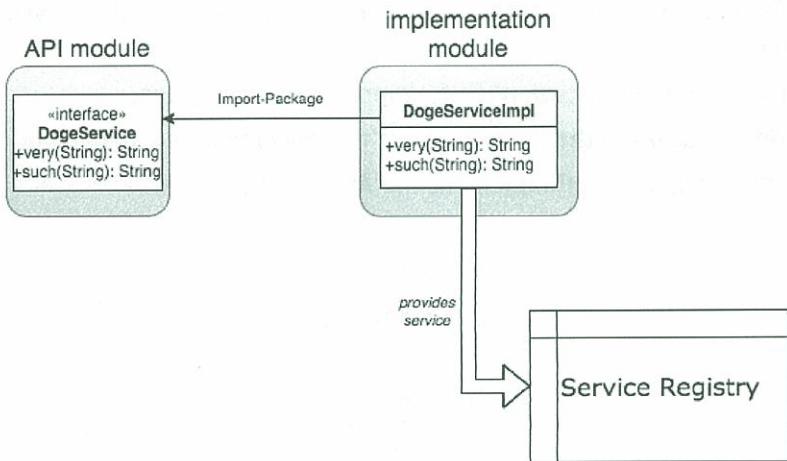
- When our module is *started*:
  - *call* the Service Registry.
  - *publish* the implementation.



WWW.LIFERAY.COM

LIFERAY.

## THE COMPLETE PICTURE



## TAKE IT OR LEAVE IT

- It's easy to generate new OSGi Services and consume existing services in Liferay.
- As a living, breathing environment, modules can come and go as they please.
- What effect does this have on our modules?

---

## LEAVING THE FRAMEWORK

- » When a module decides to pack up and go, and is *providing* a service:
  - » The service is *unregistered* from the Service Registry.
  - » If another implementation is available, a *different* module's implementation will be used.

---

## LIFE WITHOUT YOU, SERVICE

- » If your module is *consuming* a service, generally:
- » If the service is *available*, the module is started, active, and provided with the implementation.
- » When the implementation is *removed*, your module may be provided with a different implementation.
- » If there are *no implementations available* when the service is removed, your module may be *stopped*.
- » There are some advanced settings where you can control how your module reacts, but that's beyond what we need in most cases.

---

## SERVICES, HERE WE COME!

- » This simple service architecture is how all OSGi Services are registered and used.
- » Modules that provide a service *publish* the **OSGi Service** to the *Service Registry*.
- » Modules that consume services *ask for the implementation* from the *Service Registry*.
- »
- » There are some basic ways to implement an OSGi Service.
  - » Great tooling exists to simplify the process.
  - » You can use tools or not – it's entirely up to you!

---

WWW.LIFERAY.COM

 LIFERAY.

---

## COMPONENTS, AHOY!

- » Implementing a *service* is as easy as writing a POJO.
- » But we also need to make it available to other modules via the *Service Registry*.
- » Well, in our modules, we like to implement our features as *components*.
- » We're using standard *DS components* to simplify our code:

```
@Component
public class WhatAPOJO {
    ...
}
```
- » If we're implementing functionality by making components, is there anything special about making *services*?

---

WWW.LIFERAY.COM

 LIFERAY.

---

## AT YOUR SERVICE

- **No!** DS Components are fully integrated with the Service Layer of the OSGi Container.

- Let's implement our DogeService as a POJO:

```
public class DogeServiceImpl implements DogeService {  
    ...  
}
```

- To make it a component, we just add the Component annotation:

```
@Component  
public class DogeServiceImpl implements DogeService {  
    ...  
}
```

- Our component is *now available in the Service Registry!*

- It's just as easy to *use services*:

```
@Reference  
protected DogeService dogeService;
```

---

## SHARING IS CARING

- Modules can **share** information with each other by:
  - Exporting packages
  - Publishing services
  - Providing capabilities
- Packages can be *imported* and *exported* through headers in the bundle.
- Bundles can declare what *capabilities* they **provide** or **require** through headers in the bundle.
- Services are published on the *Service Registry*.
- Implementations are provided by the *Service Registry* to consumers.
- Services can be implemented with *components* (@Component).
- Services can be used as *components* (@Reference).
- Modules share functionality and work together to build a robust set of apps.

Notes:



## APPLICATION LIFECYCLE

Copyright ©2016 Liferay, Inc.  
All Rights Reserved.

No material may be reproduced electronically or in print,  
distributed, copied, sold, resold, or otherwise exploited  
for any commercial purpose without express written  
consent of Liferay, Inc.

### OVERVIEW

- Any platform that allows you to run applications will have a mechanism to install, start, and stop applications, among other operations.
- This is commonly known as that platform's application lifecycle.
- On most platforms, this lifecycle can be as simple as *Installed*→*Running*→*Uninstalled*.

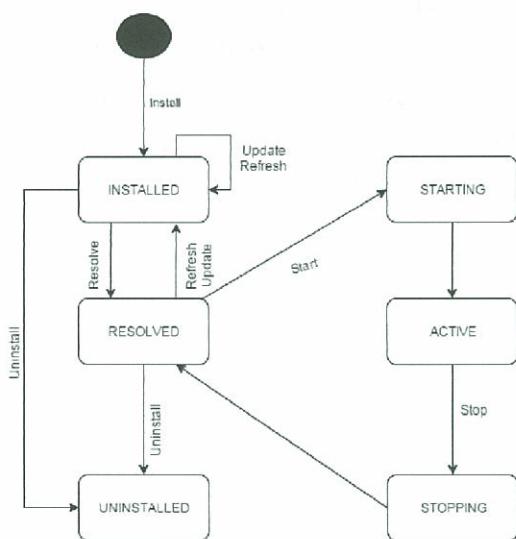
## THE MODULE LIFECYCLE

- » The OSGi Container also has a corresponding lifecycle for any modules that are deployed to it.
- » In Liferay, this is called the Module Lifecycle.

WWW.LIFERAY.COM

LIFERAY.

## THE MODULE LIFECYCLE, VISUALIZED



WWW.LIFERAY.COM

LIFERAY.

---

## MODULE LIFECYCLE STATES

- » The OSGi Container allows you to dynamically manage a module's lifecycle while Liferay is running.
- » The Module Lifecycle has six states:
  - » Installed
  - » Resolved
  - » Starting
  - » Active
  - » Stopping
  - » Uninstalled

---

## WHAT ABOUT COMPONENTS?

- » *Modules* are the primary container for app features, but they also contain *Components*.
- » Since *components* are independent functional objects, they also have a lifecycle.
- » Like *modules*, *components* have dependencies (such as the *interfaces* they implement) and provide new functionality.
- » The *component* lifecycle is simple:
  - » **Activate:** when a component is being started – additional code can run here if needed.
  - » **Active:** the component is started and available.
  - » **Deactivate:** the component is being stopped – additional code can be added here if necessary.
- » How does this relate to *modules*?

---

## MODULE-COMPONENT LIFECYCLE

- » The entire *component* lifecycle takes place in a *module's active* state:
  - » Installed
  - » Resolved
  - » Starting
  - » Active
    - » (*Components in the module:*)
    - » Activate
    - » Active
    - » Deactivate
  - » Stopping
  - » Uninstalled
- » We'll revisit this idea when we work with some common Components like Portlets.

---

## WHAT DOES THIS MEAN FOR ME?

- » As a module developer, you want to be able to easily debug your applications.
- » Part of that involves being able to tell which state of the lifecycle your component is in.
- » The easiest way to do that is by using the Gogo Shell.

---

## GOGO SHELL? WHAT'S THAT?

- » The Gogo Shell is a lightweight, text-based interface into the OSGi Container.
- » To connect to Liferay's Gogo Shell, enter the command

```
telnet localhost 11311
```
- » Let's take a look at some of the more common commands.

---

## SHELL COMMANDS

- » `lb`: lists all of your modules. The important thing to note here is your module ID, which will be a number like `431`. This also lists your module's status.
- » `start`: starts a module. For example, `start 431` will start the module with the ID `431`, and move it from *resolved* to *starting* to *active*.
- » `stop`: stops a module. For example, `stop 431` will stop the module, and move it from *active* to *stopping* to *resolved*.

---

## LET'S TRY THIS OUT

- » Let's take a look at how the Module Lifecycle works by installing a module and watching it go through the various states of the lifecycle.

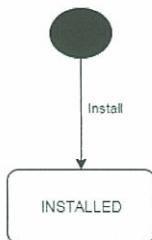
WWW.LIFERAY.COM

LIFERAY.

---

## INSTALLED

- » This state of the lifecycle is where a module starts as soon as it has been installed into Liferay.
- » If all of a module's dependencies are in place, a module will not stay in this state for very long and should shortly after transition to the *resolved* state.



WWW.LIFERAY.COM

LIFERAY.

---

## EXERCISE: INSTALLING A MODULE

- » We have created a module that will allow us to step through the different states of the lifecycle.
1. Run the command `telnet localhost 11311` to install this module and connect to the Gogo Shell.
  2. Drop the `lifecycledemomodule.jar` file into the `deploy` folder of your Liferay Home folder.
  3. Type the command `lb` in the Gogo Shell.
- ✓ See the following:

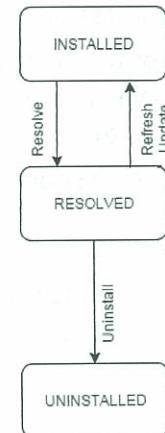
```
506|Active | 1|lifecycledemomodule (0.0.0.201605170710)
```

- » 506 in the above line is the module ID.

---

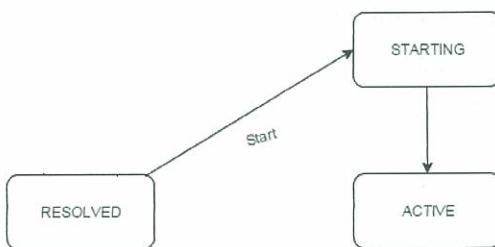
## RESOLVED

- » After going through the `installed` state successfully, the module enters the `resolved` state.
- » This state is when all of the module's dependencies were able to be resolved by the OSGi Container.
- » At this point, your module has all the dependencies it needs and is ready to be started.



## STARTING

- » This state is when the module is being activated, or started.
- » This state normally occurs during the process of activating a module, as a transition between the *resolved* state and the *active* state.
- » If a module is able to successfully start, it will go into the *active* state.

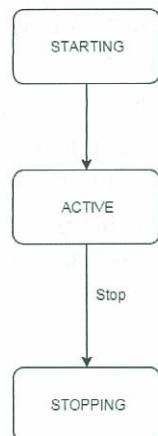


## EXERCISE: ACTIVE

- » If a module is in this state, it has been successfully activated and is now running.
  - » For this to happen, all the module's dependencies have to have been resolved, and it has to have been able to activate successfully.
  - » Let's test our current bundle.
1. Type the command `life` in the Gogo Shell.

✓ See the following output:

```
[Lifecycle Module] Liferay: Enterprise. Open Source. For  
[Lifecycle Module] This module is active and running!
```



---

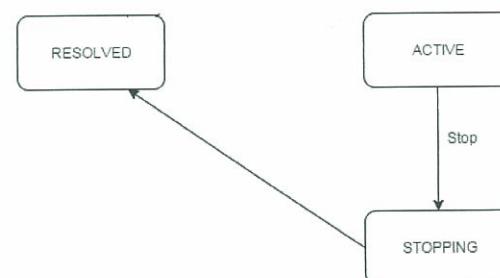
## LIFERAY STARTS A BUNDLE

- You may be wondering how our module is already at the *active* state.
- We never saw it go through the *install* state or the *resolve* state.
- We also didn't do anything ourselves to get the bundle to the *active* state, so what's going on?
- Liferay is starting our module for us.
- Assuming that a module was able to properly install and resolve, Liferay will automatically start the bundle for us.

---

## STOPPING

- Now if we want to stop a bundle, there is of course a state for that.
- This state occurs when the module is being stopped. This will send it back to the *resolved* state.
- This state normally occurs during the process of stopping a module, as a transition between the *active* state and the *resolved* state.



---

## EXERCISE: STOPPING A BUNDLE

» Now, let's see what it's like to stop a bundle.

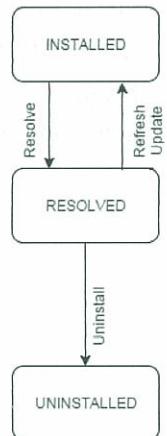
1. Run the command `stop 506` in the Gogo Shell, replacing 506 with your module ID.
- ✓ See the following line:
2. Type in 1b.
- ✓ See the following output:

```
506|Resolved | 1|lifecycledemomodule (0.0.0.201605170710)
```

---

## RESOLVED

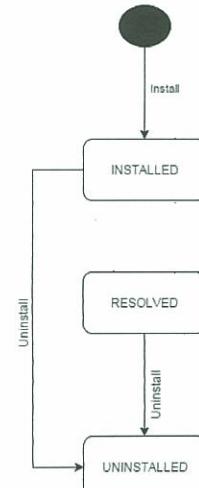
- » Just as we've seen within the shell, when we stop a bundle it goes back to the *resolved* state.
- » This state is when all of the module's dependencies were able to be resolved by the OSGi Container.
- » At this point, your module has all the dependencies it needs to start, but has not been started yet.



---

## EXERCISE: UNINSTALLED

- » The last state we need to address is the *uninstalled* state.
- » This state means that the module has been uninstalled from Liferay.
  1. Delete the `lifecycledemomodule.jar` file from `osgi/modules` in your Liferay Home folder to try this out.
  2. Run 1b.
- ✓ See that the module is no longer listed.



---

## THE MODULE LIFECYCLE SUMMARIZED

- » As you can see, the Module Lifecycle has a number of states.
- » One key difference is that due to the nature of the OSGi Container, the lifecycle has a separate stage for dependency resolution, the *resolved* state.
- » Having this additional state makes things easier for Liferay and for the module developer.
- » It allows for distinguishing between a module that has its dependencies available and one that doesn't.

Notes:



## SHELL

Copyright ©2016 Liferay, Inc.  
All Rights Reserved.

No material may be reproduced electronically or in print,  
distributed, copied, sold, resold, or otherwise exploited  
for any commercial purpose without express written  
consent of Liferay, Inc.

### CONNECTING TO THE GOGO SHELL

- To interact with the OSGi Container, you can use Felix Gogo shell.
- To access it, use a telnet client to connect to port 11311 of your Liferay server's machine.
- Since the shell is accessible through telnet, it is only available on the local network interface.
- Mac OSX and most Linux distributions include a command line Telnet client, and you can easily install one on Windows (see next slide.)

---

## TROUBLESHOOTING: INSTALLING A WINDOWS TELNET CLIENT

1. Open a command prompt window.
  - 1.1 Click Start.
  - 1.2 Type cmd in the Start Search box, and then press ENTER.
2. Type the following command: pkgrmgr /iu:"TelnetClient"
3. If the User Account Control dialog box appears, confirm that the action it displays is what you want, and then click Continue.
4. When the command prompt appears again, the installation is complete.

---

## RUNNING THE TELNET CLIENT

1. Use the following command with Liferay running: telnet localhost 11311
2. Once you connect, you should see a message that says Welcome to Apache Felix Gogo.
  - » Here are some useful Gogo shell commands:
    - » help: lists all the available Gogo shell commands. Notice that each command has two parts to its name, separated by a colon. The first part is the command scope while the second part is the command function.
    - » help [command name]: lists information about a specific command.
    - » lb: lists all of the bundles installed in Liferay.
    - » b [bundle ID]: lists information about a specific bundle
    - » headers [bundle ID]: lists metadata about the bundle from the bundle's MANIFEST.MF file

---

## MORE HELPFUL SHELL COMMANDS

- » `diag [bundle ID]`: lists information about why the specified bundle is not working (e.g., unresolved dependencies, etc.)
- » `packages [package name]`: lists all of the named package's dependencies
- » `scr:list`: lists all of the components registered in the OSGi Container. (`scr` stands for service component runtime.)
- » `services`: lists all of the services that have been registered in Liferay.
- » `install [path to JAR file]`: installs the specified bundle into Liferay.

---

## MORE HELPFUL SHELL COMMANDS (CONT.)

- » `start [bundle ID]`: starts the specified bundle
- » `stop [bundle ID]`: stops the specified bundle
- » `uninstall [bundle ID]`: uninstalls the specified bundle from Liferay.

For more information about the Gogo shell, please visit  
<http://felix.apache.org/documentation/subprojects/apache-felix-gogo.html>.

---

## BUNDLE DETAILS

- » The displayed bundle details include:
  - » Its symbolic name
  - » Its version
  - » Its location on the machine running Liferay
  - » Imported and exported packages and their versions
  - » Used and provided services and their IDs

---

## SERVICE DETAILS

- » You can view the details of each registered service.
- » These details include:
  - » The name of the service bean
  - » The symbolic name of the bundle providing the service
  - » The version of the bundle providing the service
  - » The bundles using the service (if any)
- » This information can be very useful when developing Liferay bundles that consume or provide services.

Notes:



## Chapter 3

# Building Modules



## BUILDING A MODULE

Copyright ©2016 Liferay, Inc.  
All Rights Reserved.

No material may be reproduced electronically or in print,  
distributed, copied, sold, resold, or otherwise exploited  
for any commercial purpose without express written  
consent of Liferay, Inc.

## MODULE REVIEW

- *Modules* are plugins that are deployed into Liferay.
- Modules are packaged as JAR files.
- A module just needs to have a valid manifest file.

---

## WHAT ARE BUNDLES?

- » In the OSGi world, what Liferay is based on, modules are known as *bundles*.
  - » OSGi *bundle* = Liferay *module*
- » Because Liferay already has the term *bundle* to refer to Liferay *bundle* with an application server, we use the term *module* to differentiate.
- » In the OSGi world, what the OSGi Container is based on, modules are known as *bundles*.
  - » Example:
    - » Liferay *bundle* = Liferay Platform + Tomcat
    - » OSGi *bundle* = Liferay *module*

---

## LOOKING AT OUTSIDE SOURCES

- » If you do additional research on OSGi, keep in mind that you will not find the term *module* but rather *bundle*, as *bundle* is the standard term outside of Liferay.
- » In Liferay, we'll always refer to OSGi bundles as modules to avoid confusion.

---

## ARTISAN MODULE DEVELOPMENT

- » Now that we've clarified our use of the term module, let's go ahead and actually build one from scratch.
- » There are two key parts to a module:
  - » Java class
  - » Manifest file
- » Though there are tools that help in module building, let's look at how to build a module without them.

---

## THE BLUEPRINT

- » The steps we'll be taking to create our module:
  - » Create a project
  - » Write a Java class
  - » Create the manifest file
  - » Package the module as a JAR file
  - » Deploy

## EXERCISE: CREATING THE PROJECT

1. Click on *File* → *New* → *Project* → *Java Project* in Liferay IDE.
2. Type *hello-api* in *Project name*.
3. Click *Finish*, leaving the default settings.

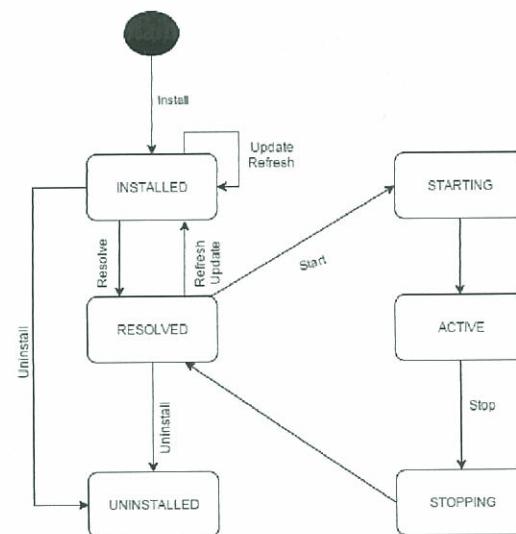


WWW.LIFERAY.COM

LIFERAY.

## LIFE OF A MODULE

- » A module goes through a lifecycle.

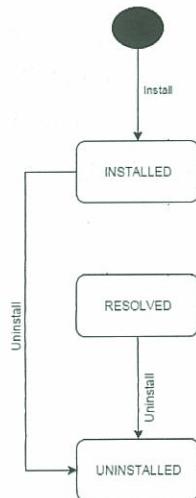


WWW.LIFERAY.COM

LIFERAY.

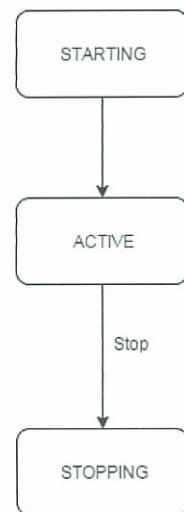
## LIFE IN THE OSGI CONTAINER

- » The OSGi Container handles:
  - » Installed State
  - » Resolve State
  - » Uninstalled State



## DOING LIFE IN THE MODULE

- » A module is responsible for:
  - » Starting (Start)
  - » Stopping (Stop)



## EXERCISE: CREATING THE PACKAGE

1. Right-click on the project in the *Package Explorer*.
  2. Click *New → Package*.
  3. Name the package `com.liferay.training.service.hello`.
- ✓ Click Finish.

## EXERCISE: CREATING THE SERVICE API

1. Right-click on the project in the *Package Explorer*.
  2. Click *New → Interface*.
  3. Name the interface `HelloService`.
- ✓ Click Finish.



---

## EXERCISE: CREATING THE SERVICE CLASS

1. Add snippet *o1-HelloAPI* to the body of the class.

✓ Save the file.

**TROUBLESHOOTING NOTE:** On Windows and Linux, you can double-click snippets or drag them from the snippet view into the code. On Mac OS X, you must right-click on snippets and select "Insert" to add snippets.

---

## MANIFESTATION OF THE MANIFEST

- Our first requirement, a Java class, is fulfilled. The next requirement for our module is a *manifest*.
- The *manifest* describes:
  - **Symbolic Name:** The module's name (*Bundle-SymbolicName*)
  - **Module Version:** The module's version (*Bundle-Version*)
  - **Name:** The module's readable name (*Bundle-Name*)
- There are other attributes within the manifest, but these three will be the main focus in this section.

---

## EXERCISE: CREATING THE FOLDER FOR THE MANIFEST

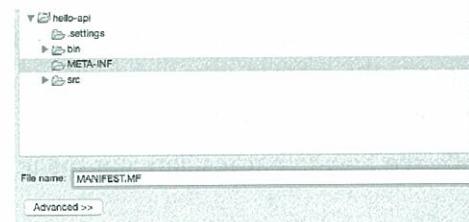
1. Right-click on the project in the *Package Explorer*.
2. Click on *New → Folder*.
3. Create a new folder called **META-INF**.



---

## EXERCISE: CREATING THE MANIFEST

- » Right-click the **META-INF** folder.
- » Choose *New → File*.
- » Create a new file called **MANIFEST.MF**.
- » Add snippet *o2-Manifest File*.



---

## EXPORTING PACKAGES

- » In some cases, your module will be very selfish. It will stand alone and not provide anything for any other modules.
- » In other cases, like ours, more altruistic modules will provide a service class or some other method which is going to be used by other modules.
- » In order for other modules to be able to access classes in your module, you need to add the Export-Package directive to specific which packages are available for use.
- » Our case is very simple, we are exporting a single package, which also happens to be the only package in our class.
- » In more complex cases, you can specify multiple packages, as well as the specific version of those packages to be exported.

---

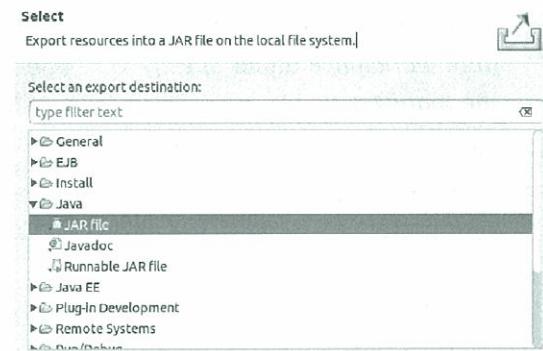
## PACK IT UP, WE'RE DONE HERE

- » Now that we've created the two essential files, our next step is to package the module.
- » All modules are packaged as JAR files.
- » Let's see how to encapsulate our module in a JAR file.

---

## EXERCISE: PACK IT UP, WE'RE DONE HERE

1. Right-click on the project.
2. Click on *Export*.
3. Choose *Java → JAR file* as the export type.
4. Click *Next*.



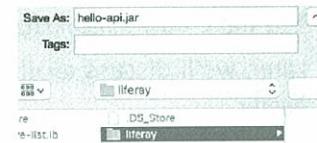
WWW.LIFERAY.COM

LIFERAY.

---

## EXERCISE: LOCATION OF EXPORTED JAR FILE

1. Choose where the JAR File will be exported. You can export it to whatever location is convenient, or create /build folder under your Liferay folder.
2. Name the JAR file `hello-api.jar`.
3. Click *Next*, and leave the defaults.
4. Click *Next*.

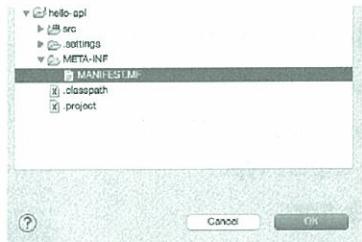


WWW.LIFERAY.COM

LIFERAY.

## EXERCISE: CHOOSING THE MANIFEST

1. Choose *Use existing manifest* from workspace under *Specify the manifest*.
2. Click *Browse...* next to *Manifest*.
3. Find the **MANIFEST.MF** file in your project.
4. Click *Finish* to export the JAR.



## SEND IT OFF

- » You have now successfully created a module from scratch!
- » Later, we'll discuss available tools that can make our development easier.
- » For now, let's make sure that our module works properly by deploying it.

---

## EXERCISE: DEPLOY IT

1. Copy the hello-api.jar.
2. Go to /LIFERAY-HOME/deploy/.
3. Paste the copied JAR file in the /deploy/ folder.
4. See STARTED com.liferay.training.service.hello\_1.0.0 in Liferay IDE's console.

---

## EXERCISE: MODULE ACTIVE

1. Open a terminal or Command Prompt window.
  2. Open a telnet connection to the shell using: telnet localhost 11311.
  3. Type lb to see a list of modules once connected.
- ✓ See the Hello API module in the *Active* state at the bottom of the list.

503 Active	10 lcs-portlet (7.0.10.2)
504 Active	10 user-profile-theme (1.0.2)
506 Active	10 Hello API (1.0.0)

---

## EXERCISE: STOP IT

1. Type `stop [module_number]` in the Shell to stop the Hello API Module.
2. See STOPPED `com.liferay.training.service.hello_1.0.0`
3. Type `lb` to confirm that the Hello API Module is in the resolved state.

```
503|Active      | 10|lcs-portlet (7.0.10.2)
504|Active      | 10|user-profile-theme (1.0.2)
506|Resolved    | 10>Hello API (1.0.0)
```

---

## EXERCISE: START IT AGAIN

1. Type `start [module_number]` to start the Hello API Module again.
2. See the message STARTED  
`com.liferay.training.service.hello_1.0.0`

```
503|Active      | 10|lcs-portlet (7.0.10.2)
504|Active      | 10|user-profile-theme (1.0.2)
506|Active      | 10>Hello API (1.0.0)
```

---

## BRINGING IT ALL BACK TOGETHER

- To create a module, we first created a Java interface defining our methods.
- We then created a manifest file that contained the symbolic name, version, and name of the module.
- Finally, we packed our module as a JAR and deployed it into Liferay.

Notes:



## **Chapter 4**

# **Building Services**



## CREATING A BASIC SERVICE

Copyright ©2016 Liferay, Inc.  
All Rights Reserved.

No material may be reproduced electronically or in print,  
distributed, copied, sold, resold, or otherwise exploited  
for any commercial purpose without express written  
consent of Liferay, Inc.

### SERVICES IN LIFERAY

- » When we create services in Liferay, we are publishing an OSGi Service.
- » An OSGi Service consists of 2 modules:
  - » An API module
  - » An implementation module
- » In this section, we'll focus on the API module.

---

## THE API MODULE

- An API is a contract between the service and the consumer of the service.
- In the OSGi Container, the API is implemented as interfaces in a module.
- Just like any other module, we still need:
  - Java Classes or Interfaces
  - Manifest File (MANIFEST.MF)

---

## FLASHBACK COMPONENT

- Let's take a journey back to the component since they are essential to the idea of services.
- Components:
  - Implement functionality
  - Are self contained in a module.
- Components can also be reused among any number of applications.

---

## WHAT MAKES A COMPONENT A COMPONENT

- » So what does make a component a component?
- » All it takes is a POJO, an annotation and a dream (well maybe not the a dream).
- » Let take for example a dream class.

```
public class MyDreamClass{  
    \\lots of dream code  
}
```
- » To make it a component all we need to do is add an *@Component* annotation right above the class declaration.

```
@Component  
public class MyDreamClass{  
    \\So much sleepy dream code  
}
```

---

## WHAT'S MINE IS YOURS

- » Components can be shared among many different apps for their use.
- » This in turn allows for functionality to be shared.
- » The main tool we'll be using is called *declarative services*.
- » By using the annotation *@Component* we are already using declarative services.

---

## SETTING PROPERTIES

- There are times where the functionality we are implementing needs to have properties set.
- We are able to do this within the annotation.

```
@Component(  
    property={  
        "temperature = 68F",  
        "bed.firmness = 95"  
    },  
)  
public class MyDreamClass{  
    \\Good Night Code  
}
```

---

## MAY I TAKE YOUR HAT SIR?

- So the question that may be in the air, "How does a component become a service?" .
- With a line of code `service = Dream.class` we have a service.

```
@Component(  
    service = Dream.class  
)  
public class MyDreamClass{  
    \\So much sleepy dream code  
}
```

- Of course you can also be implementing an API.

---

## EXPOSING OUR SERVICE

- » As we know, the API has to be called by an application one way or another.
- » So, how do we expose the API or Interface to the outside world?
- » The answer is the MANIFEST.MF.

---

## HELLO, OUTSIDE WORLD!

- » In MANIFEST.MF we saw a number of manifest headers that we were able to set.
  - » Manifest Headers:
  - » Bundle-Name
  - » Bundle-Version
- » One of the manifest headers we are able to set is EXPORT-PACKAGE.
- » By exporting the package containing the interface, we make it available to modules in the OSGi Container.

---

## SIMPLIFY THE MANIFEST

- › The manifest controls many of the configurations of a module, such as its name, version, and the packages a module imports and exports.
- › With all of the different manifest headers to keep track of, along with which packages a module is importing and exporting, there has to be a simpler way.

Notes:



## IMPLEMENTING THE SERVICE

Copyright ©2016 Liferay, Inc.  
All Rights Reserved.

No material may be reproduced electronically or in print,  
distributed, copied, sold, resold, or otherwise exploited  
for any commercial purpose without express written  
consent of Liferay, Inc.

### TOOLS OF THE TRADE

- In the previous section, we left off talking about our manifest file.
- There has to be a better way to manage all that is going on with the manifest file.
- There is a better way and that way is Bndtools.

---

## BNDTOOLS

- » *Bndtools* is a front end of Bnd, the base development tool that aids in the creation of a module's manifest file.
- » *Bndtools* creates a file called `bnd.bnd` which helps streamline the creation of the manifest file.
- » All we have to do is, with the aid of a GUI, provide all of the relevant information that we want in our manifest and Bndtools will handle the rest for us.
- » Before we get into using Bndtools, let us recap OSGi services.

---

## THE STORY SO FAR

- » An OSGi Service published in the OSGi Container consists of two modules.
  - » The API Module
  - » The Implementation Module
- » In the previous section, we created the API module which publishes the Interface.
- » In this section, we will create the Implementation Module using Bndtools.

---

## THE IMPLEMENTATION MODULE

- » The Implementation Module will be implementing the API module that we created previously.
- » We'll then import the package from the API Module.
- » Typically, next the Implementation Module will publish the Implementation of the service into the Service Registry.

---

## THE SERVICE REGISTRY

- » The Service Registry is the 411 of Liferay.
- » A module can "call" the Service Registry for a specific implementation of a service.
- » A module can also let the Service Registry know of an implementation so that it can be called upon.

---

## IMPLEMENTING THE IMPLEMENTATION OF THE IMPLEMENTATION MODULE

- We'll first create the implementation class for the API module.
- Next, we'll register our service to the Service Registry.
- Finally, the Implementation Module will import the package from the API module.
- That's it! Let's do... or do not... there is no try.
- NOTE: During this exercise you may be asked if you want to switch to the BND Tools Perspective in Eclipse. Choosing to change perspectives or not won't have any effect on completing the exercise.

---

## EXERCISE: BND CONFIGURATION PROJECT

- In order to use BND Tools we'll need to create a configuration project named 'cnf'.
  - This is so Bndtools can cache all of your projects' dependencies for faster builds later.
1. Go to *File* → *New* → *Other* → *BndTools* → *BND OSGi Workspace*.
  2. Click *Next*, leaving the default settings.
  3. Choose *bndtools/workspace*.
  4. Click *Next*.
  5. Click *Finish* after viewing the list of workspace changes.

## EXERCISE: CREATING A NEW PROJECT

1. Click *File* → *New* → *BND OSGi Project*.
2. Choose *Component Development*, followed by *Next*, under *OSGi Standard Template*.
3. Name the project `hello-service`.

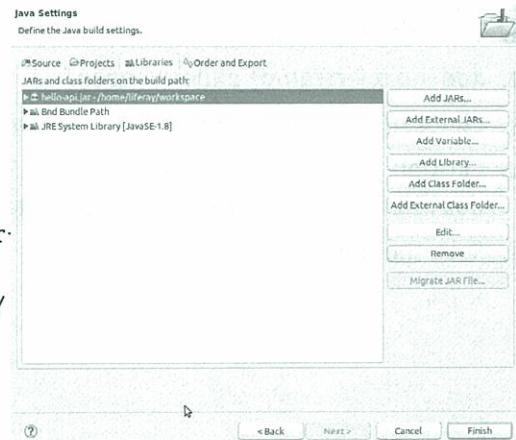


## EXERCISE: NAMING THE PACKAGE

1. Uncheck the box for *Derive from project name*.
2. Name the package `com.liferay.training.service.hello.impl`
3. Click *Next*.

## EXERCISE: IMPORTING LIBRARIES

1. Click on the *Libraries* tab.
2. Click on *Add External JARs...*
3. Add the `hello-api.jar` found in `/exercises/application-developer-training-exercises/04-building-services/hello-api/generated`.
4. Click *Finish*.

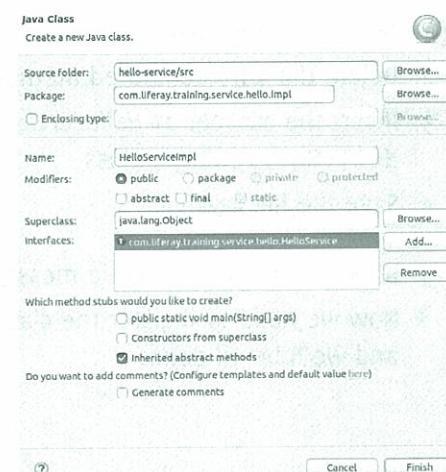


WWW.LIFERAY.COM

LIFERAY.

## EXERCISE: CREATING THE IMPL CLASS

1. Delete the `example.java` class and `test` folder.
2. Right-click the project `com.liferay.training.service.hello.impl`.
3. Click *New → Class*.
4. Name the class `HelloServiceImpl`.
5. Add the `HelloService` as an interface.
6. Click *Finish*.



WWW.LIFERAY.COM

LIFERAY.

---

## EXERCISE: KEEP THINGS HIDDEN (JUST IN CASE)

1. Open the `bnd.bnd` file and make sure that the content tab is selected.
2. Add `1.0.0.${tstamp}` under Version.
3. Choose *OSGi DS Annotations* for Declarative Services.
4. Click on the *green plus* next to *Private Packages* to add our package, `com.liferay.training.service.hello.impl`.
5. Save the file.

---

## EXERCISE: MUST HAVE CALLED A THOUSAND TIMES

1. Go to to the `HelloServiceImpl` class.
  2. Delete the auto-generated methods.
  3. Insert the snippet *o1-HelloService Implementation* into the body of the `HelloServiceImpl` class.
  4. Save the file again.
- If we call `say()`, we get a message. It's very simple.  
► Now we need to register the class as a Declarative Services Component, and we'll be all done.

---

## EXERCISE: REGISTERING THE SERVICE WITH DECLARATIVE SERVICES

1. Insert snippet *o2-DS-Component* at the top of the class, between the imports and the class declaration.
2. Resolve imports using CRTL/COMMAND + SHIFT + O.
3. Save the file.
4. Close all the editor windows.
5. Click on the *hello-service* project.
6. Refresh by pressing *F5* on your keyboard.
  - This may seem extensive, but this ensures that all of our files are refreshed and up to date.
  - That way our JAR file is not outdated.

---

## IMPORTING THE API MODULE

- The last and final step is to import the API Module.
- This is done within the *bnd.bnd* file.
- The *bnd.bnd* file is the file that BndTools uses to make the *MANIFEST.MF* for us.

---

## EXERCISE: OFF YOU GO, INTO THE OSGI CONTAINER

- » Now that we have finished creating our Implementation Module, let's send the JAR file off to the OSGi Container.
  1. Copy the JAR file from the generated folder.
    - » Found in workspace/hello-service/generated
  2. Paste the JAR file in the /deploy/ folder.

---

## EXERCISE: MUST HAVE CALLED A THOUSAND (MORE) TIMES

1. Deploy the module `hello-client.jar` found in the exercises folder to test the service.
2. Open a terminal/Command Prompt and telnet into the shell.
  - » `telnet localhost 11311`
3. See that the API and Implementation Module have been deployed using `lb`.
4. Type `say` to test the service call.

```
g! say  
Hello...
```

---

## TO GO OVER EVERYTHING

- The Implementation Module is the second module that makes up the OSGi Service.
- Like any service architecture, it implements an API, in this case the API Module.
- To register a service in the Service Registry, we use method call `registerService()` on the `BundleContext` Object within the `BundleActivator`.
- The Implementation Module has to import the API Module.

Notes:



# Chapter 5

## Java Standard Portlet



## WHAT ARE PORTLETS?

Copyright ©2016 Liferay, Inc.  
All Rights Reserved.

No material may be reproduced electronically or in print,  
distributed, copied, sold, resold, or otherwise exploited  
for any commercial purpose without express written  
consent of Liferay, Inc.

### OVERVIEW

- In Liferay, applications are composed of modules that can interact with each other.
- Modules contain components, each of which has different responsibilities and functions.
- For example, the MVC design pattern can be implemented using a separate module for each of the three different layers.
- The View layer can be implemented using one module, the Controller using another module, and the Model using a third module.

---

## PORLETS AND COMPONENTS

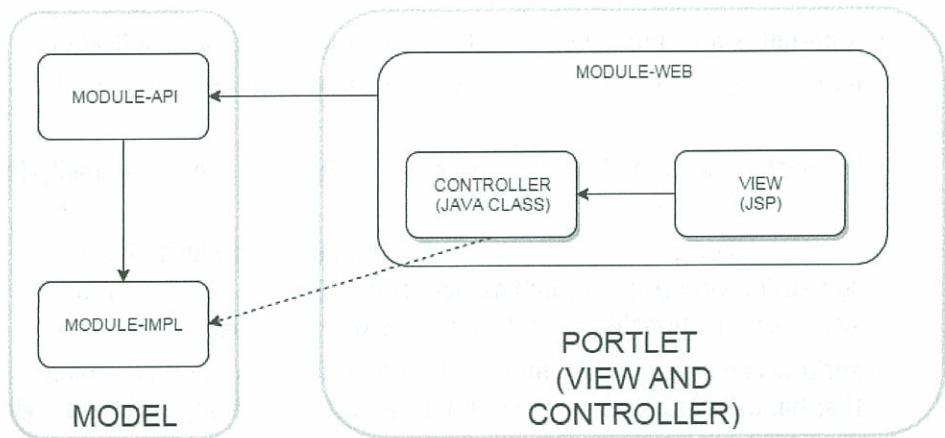
- A portlet is a component that acts as the front-end of an application.
- Portlets are the primary method of integrating your application with Liferay's UI.
- In the traditional MVC design pattern, a portlet adds the functionality for the Controller and the View layer.
- The Controller provided by a portlet can range in complexity from something very basic (switching between pages of a portlet) to more advanced functionality (handling actions or processing data).
- Portlets can also take the middle ground of having a main Controller that hands off to a separate class for the data processing/business logic.

---

## PORLETS IN A MODULAR WORLD (I)

- In addition to being an OSGi Container, Liferay also includes a portlet container.
- The presence of this portlet container allows applications to integrate with Liferay's UI and to be displayed on its web interface.
- Therefore, any application you want to integrate with Liferay's UI needs to have a portlet component.
- Modular MVC-based applications are typically made up of three modules:
  - `<module-name>-api` - This module contains the service API.
  - `<module-name>-impl` - This module contains the service implementation.
  - `<module-name>-web` - This module contains the portlet component, with the View and Controller layers of your application.

## PORTLETS IN A MODULAR WORLD (II)



## DO I NEED A PORTLET?

- You need a portlet if you want your application to have a UI, and show up as an application in Liferay's web interface.
- Since most applications do require a UI it's very likely that your application will require a portlet component as well.

---

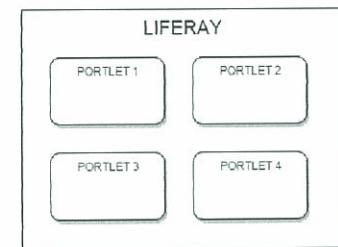
## EXAMPLE OF A PORTLET

- Let's take a look at an application that could use a portlet.
- For our example, let's consider an MVC application that needs to keep track of a collection of books.
- The model would be implemented as a service that would connect to the database to store/retrieve the information about the books in your collection.
- Following the convention we've looked at, the model would be implemented as two modules, book-api and book-impl.
- We also want to be able to view our collection, which is where a portlet comes into play.
- This would be implemented as a module called book-web, and would contain the view and controller of our application.

---

## HOW DO PORTLETS WORK?

- Portlets are meant to coexist with other portlets on the same page.
- This means that portlets can interact with each other, like any other components. It also means that a portlet may be run due to an action a user has performed in another portlet.
- Just as modules have a series of states they go through (called a lifecycle), portlets have their own lifecycles as well.
- This helps portlets better manage different needs for the application, such as displaying content, processing data, etc.



---

## THE JAVA STANDARD

- Portlets are governed using a Java standard called JSR-286 (Portlet 2.0)
- Developing your portlet using this standard allows you to easily move your portlet to any other portlet container that complies.
- Portlets developed for Liferay, while not strictly standards-compliant, use the same concepts as are provided by the standard and are structured very similarly.
- Because of its importance, we will be taking a look at how the Java standard works.
- If you're interested, more information on the standard can be found at <https://jcp.org/en/jsr/detail?id=286>

Notes:



## THE PORTLET LIFECYCLE

Copyright ©2016 Liferay, Inc.

All Rights Reserved.

No material may be reproduced electronically or in print,  
distributed, copied, sold, resold, or otherwise exploited  
for any commercial purpose without express written  
consent of Liferay, Inc.

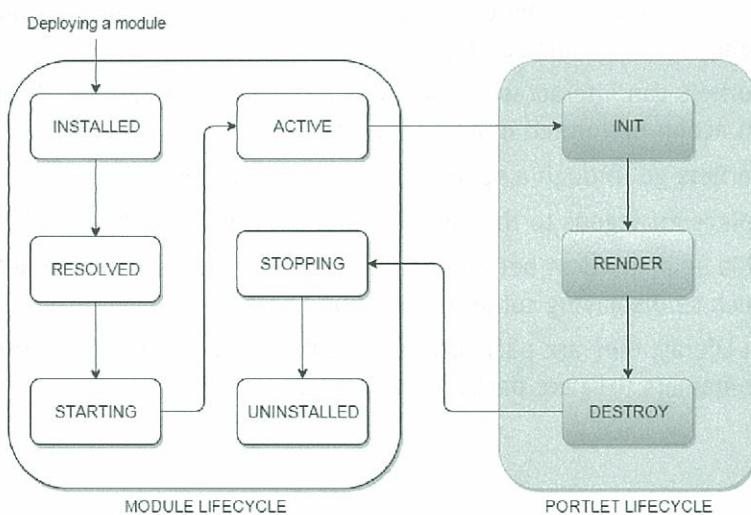
### OVERVIEW

- ❖ Portlets are components that are meant to coexist with other portlet components on the same page.
- ❖ Portlets can interact with each other, and a portlet may be affected by an action performed in another portlet.
- ❖ Portlets go through a series of states called a lifecycle.
- ❖ This is analogous to the module lifecycle, which has similar phases
- ❖ This helps portlets better manage different needs for the application, such as displaying content, processing data, etc.
- ❖ In Liferay, they are packaged as modules; however, in other portlet containers they are packaged as WARs.

## ANOTHER LIFECYCLE

- The portlet lifecycle kicks in once the module is activated.
- Remember that portlets are components that are packaged in modules.
- This means that, like other modules, they would have to go through the initial module lifecycle.
- To be activated, a module needs to be in the Active phase.
- This means all of the module's dependencies have to be available and resolved.
- Once the module reaches the Active phase, the portlet starts its lifecycle, beginning at the init phase.

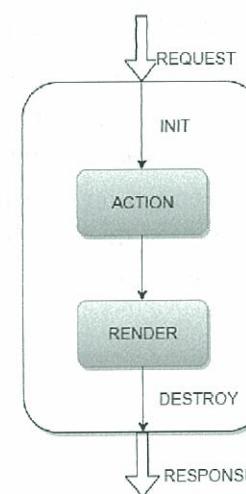
## LIFECYCLE INTERACTION



## STAGES OF THE LIFECYCLE

- » Portlets are governed by a Java standard called JSR-286 (Portlet 2.0).
- » This standard provides for six phases in the portlet lifecycle:
  - » init
  - » render
  - » action
  - » resource serving
  - » event
  - » destroy

## PORTLET LIFECYCLE OVERVIEW



---

## INIT PHASE

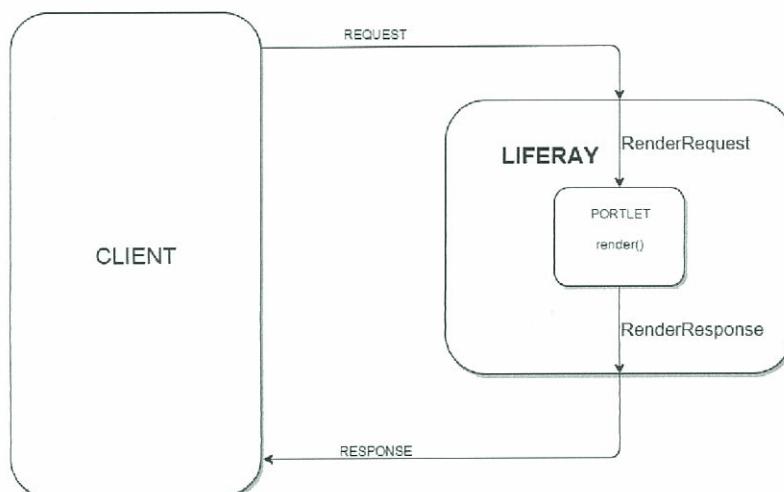
- The *init* phase is the first phase a portlet goes through when it's deployed.
- In the Java standard, this phase is tied to the `init()` method.
- During this phase, portlets typically initialize any back-end resources or perform any one-time activities that they need.

---

## RENDER PHASE

- The *render* phase is the phase a portlet uses to display its content.
- In the Java standard, this phase is tied to the `render()` method.
- It is normally called when a page is first loaded, and gets called whenever the portlet needs to display some content.
- It is also called whenever a portlet on the same page has to display some content, as the whole page needs to be redisplayed.
- This phase allows us to separate out the portlet display code from the rest of the portlet.

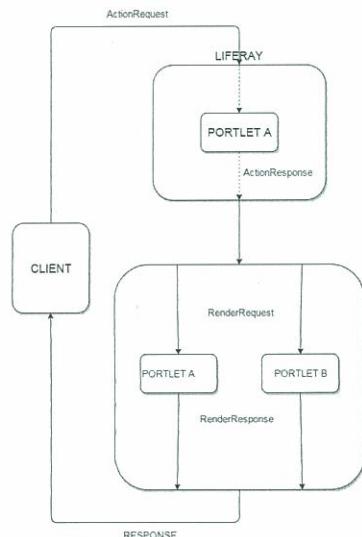
## RENDER PHASE VISUALIZED



## ACTION PHASE

- » The *action* phase is the phase a portlet uses to perform an action or to process some data.
- » In the Java standard, this phase is tied to the `processAction()` method.
- » This phase occurs when a user interacts with a portlet in some way (e.g., submitting a form).
- » It allows us to separate the data processing code from the display code.
- » Only one portlet can go through the *action* phase during a single request/response cycle.
- » Once the *action* phase finishes, the portlet sends out any events that may have been triggered by the action.
- » After this has been done, the portlet switches to the *render* phase.

## ACTION PHASE VISUALIZED



WWW.LIFERAY.COM

LIFERAY

## DESTROY PHASE

- › The *destroy* phase is called by the portlet container when the portlet is uninstalled.
- › In the Java standard, this phase is tied to the `destroy()` method.
- › This phase is designed to allow the portlet to release any resources it needs to and to save its state if necessary.

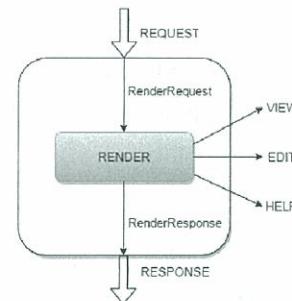
WWW.LIFERAY.COM

LIFERAY

---

## PORLET MODES

- » The *render* phase can be further subdivided into different modes, which gives users an easy way to access a particular page in a portlet.
- » The Java standard allows for three modes: *View*, *Edit*, and *Help*.
- » The *View* mode is the default view for a portlet.
- » The *Edit* mode is normally used to customize a portlet.
  - » For example, a weather portlet may use the *Edit* mode to change the zip code whose weather is being displayed.
- » The *Help* mode is normally used to display information about how to use the portlet.



---

## WINDOW STATES

- » The Java standard also allows for three possible window states during the *render* phase.
- » Portlets can be either minimized, maximized, or normal.
- » Minimizing a portlet collapses it, leaving only the titlebar.
- » Maximizing a portlet makes it take up the whole page.
- » Normal is the default window phase portlets use when added to a page.

---

## EXERCISE: LIFECYCLE PORTLET - INIT

- » We have created a sample portlet that will allow you to see exactly when the different states of a portlet's lifecycle are used.
  - » This portlet will also demonstrate how the various portlet modes are used.
1. Copy the `LifecyclePortlet.jar` file into the `LIFERAY_HOME/deploy` folder.
  - ✓ In the console, you should see "... *Init Phase ...*".

---

## EXERCISE: LIFECYCLE PORTLET - RENDER

- » Add the portlet to a page:
  1. Click on the *add* button on the *Control Menu*.
  2. Click on the the *Application* drop-down.
  3. Add the *Lifecycle Portlet* application under the *Sample* category on the page.
- ✓ In the console, you should see "... *Render Phase ...*".

---

## EXERCISE: LIFECYCLE PORTLET - ACTION AND RESOURCE SERVING

1. Click on the button that says "*Click to invoke Action Phase*" in your *Lifecycle Portlet*.
  2. See "... Action Phase ..." followed by "... Render Phase ..." in the console.
  3. Click on the button that says "*Click to invoke Resource Serving Phase*" in your portlet.
- ✓ In the portlet, you should see some text appear that says "Resource served successfully!".
- ✓ In the console, you should see "... Serve Resource Phase ...".

Notes:



## INTERPORTLET COMMUNICATION

Copyright ©2016 Liferay, Inc.  
All Rights Reserved.

No material may be reproduced electronically or in print,  
distributed, copied, sold, resold, or otherwise exploited  
for any commercial purpose without express written  
consent of Liferay, Inc.

### OVERVIEW

- So far we've looked at how to create some basic applications using the OSGi Container.
- As your applications get more complex, you will most likely be creating more and more modules.
- While we've already seen how to compose an application made up of multiple modules, we've not yet seen any ways to have the modules actually talk to each other.
- This can be necessary as your application grows in size and complexity.

---

## MESSAGING BETWEEN MODULES

- » Being able to send messages between different modules in your application allows them to pass data back and forth.
- » This allows development of larger and more complex applications and makes collaboration between your developers easier as well.

---

## HOW TO MAKE MODULES TALK TO EACH OTHER

- » There are a number of different ways to allow your modules to talk to each other.
- » Some of the more common ones are:
  - » Enterprise Service Bus (ESB)
  - » Java Message Service (JMS)
  - » Liferay's Message Bus API
  - » AJAX
- » If you're creating portlet modules, the Java standard also allows for *Interportlet Communication*, which allows communication between portlets.

---

## WHAT IS IPC?

- » So far, we have looked at some portlets that provide both a UI layer (implemented using JSPs), and a Controller (the portlet classes).
- » We still haven't, however, had our portlets pass information to each other.
- » Portlets built using the Java standard have a built-in standard method for doing so, known as IPC (Interportlet Communication).
- » IPC gives us a standard way of having portlets talk to each other and provides two mechanisms for doing so.

---

## PUBLIC RENDER PARAMETERS

- » These are the simplest methods for standard IPC.
- » Public render parameters are parameters that are declared by each portlet.
- » These parameters can also be uniquely identified using a namespace.
- » Liferay then decides which parameters of two or more different portlets should map to the same information, based on the parameters' name.
- » These parameters are available through the entire lifecycle (`processAction`, `render`, `processEvent` and `serveResource`) of the portlet.

---

## HOW TO USE PUBLIC RENDER PARAMETERS (I)

- For each portlet that will be using a particular parameter, that parameter needs to be added into the portlet component's declaration:

```
@Component(  
    service = Portlet.class,  
    property = {  
        "javax.portlet.supported-public-render-parameter=tag";  
        "http://my.prp.tld/ns"  
    }  
)  
public class MyPortlet implements Portlet {  
    ...  
}
```

---

## HOW TO USE PUBLIC RENDER PARAMETERS (II)

- When you want to use a particular parameter, the method call will look identical to that used for any other parameter. For example, from a method in your portlet class, you can access the above parameter using:  
`request.getParameter("tag");`
- From a JSP, you can access the above parameter using:  
`String assetTagName = ParamUtil.getString(request, "tag");`
- ParamUtil is a helper utility provided by Liferay that simplifies getting values from request parameters and automatically checks for blank or null return values.

---

## EVENTS

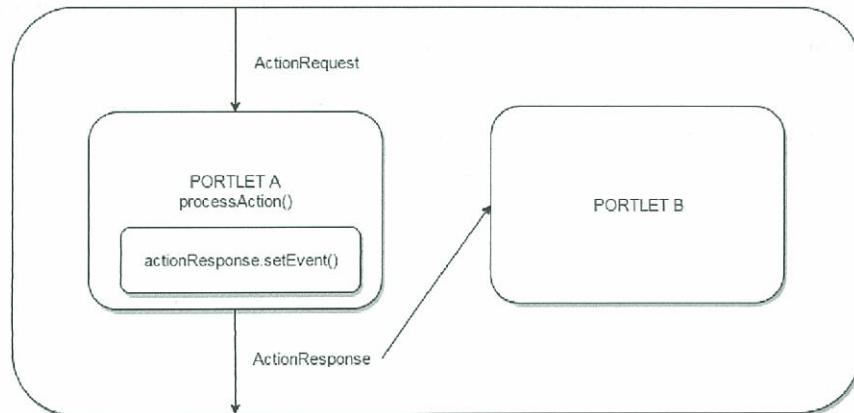
- » Events are a more powerful method of IPC.
- » They are very loosely coupled and allow for greater flexibility.
- » They follow a Producer-Listener pattern.
  - » One portlet produces an event, which zero or more portlets may be listening to.

---

## EVENT PHASE

- » The *event* phase of the portlet lifecycle is the phase a portlet uses to handle incoming events.
- » In the Java standard, this phase is tied to the `processEvent()` method.
- » Events that are set in the `processAction` phase are processed during the event phase, in the `processEvent` method of any registered listeners.
- » Once all events have been processed, the portlet container will render all of the portlets on the page, calling the `render()` method on all of them.

## EVENT PHASE - SENDING



## HOW TO PRODUCE AN EVENT

- » For each event that you want a portlet to use, you need to decide whether that portlet is producing or consuming said event.
- » If the portlet is producing a particular event with the name `ipc.beammeup` and the namespace `http://my.event.tld/ns`, the following code needs to be added into the portlet component's declaration:

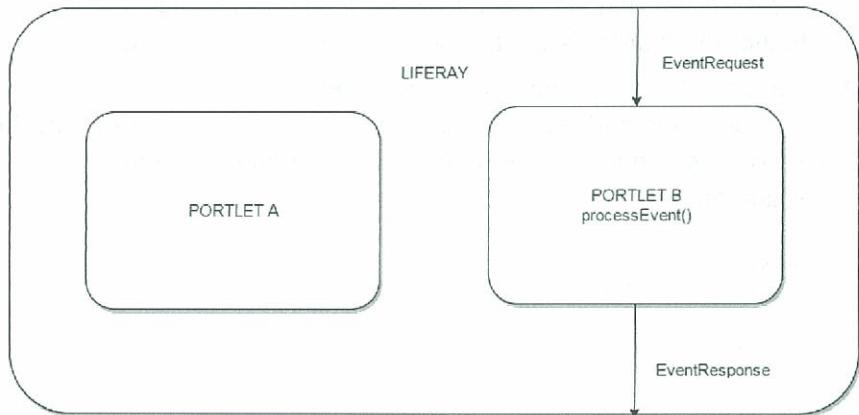
```
@Component(  
    service = Portlet.class,  
    property = {  
        "javax.portlet.supported-publishing-event=ipc.beammeup;  
        http://my.event.tld/ns"  
    }  
)  
public class MyProducingPortlet implements Portlet {  
    ...  
}
```

## HOW TO SEND AN EVENT

- » To actually send the event from the producer, the following code would need to be added to the `processAction` method:

```
QName qName = new QName("http://my.event.tld/ns", "ipc.beammeup");
response.setEvent(qName, "Event message here!");
```

## EVENT PHASE - RECEIVING



---

## HOW TO CONSUME AN EVENT

- » A portlet component consuming the same event would have the following code in its declaration:

```
@Component(  
    service = Portlet.class,  
    property = {  
        "javax.portlet.supported-processing-event=ipc.beammeup;  
        http://my.event.tld/ns"  
    }  
)  
public class MyConsumingPortlet implements Portlet {  
    ...  
}
```

---

## HOW TO RECEIVE AN EVENT

- » This will result in the `processEvent` method being called for any consumers of the event, which can get the event information using something like this:

```
@ProcessEvent(qname="{http://my.event.tld/ns}ipc.beammeup")  
public void arrivalDestination(EventRequest request, EventResponse response) {  
    Event event = request.getEvent();  
    String beammeup = (String)event.getValue();  
    response.setRenderParameter("beammeup", beammeup);  
}
```

- » The `@ProcessEvent` annotation allows us to specify that this method will act as the event processor for an event with a particular name.

## REASONS FOR USING IPC

- » Liferay provides the Message Bus API to allow your modules to pass information to each other.
- » If your modules are portlets, however, they come with support for IPC, which gives you two built-in methods of communication.
- » IPC is a powerful and decoupled way to easily send messages from one portlet to another that lets you use any type of Java objects you want.
- » Using these methods also makes your portlets more portable, as they will work on any JSR-286 compliant portlet container.

Notes:

## Chapter 6

# Interacting with the Shell



## OVERVIEW OF THE SHELL

Copyright ©2016 Liferay, Inc.  
All Rights Reserved.

No material may be reproduced electronically or in print,  
distributed, copied, sold, resold, or otherwise exploited  
for any commercial purpose without express written  
consent of Liferay, Inc.

### INTRODUCTION TO THE SHELL

- ❖ Now that we have an idea of what's inside the OSGi Container and how it works, we need a way to be able to launch, configure, and control the modules within Liferay.
- ❖ Let's take a look.

---

## FEATURES OF THE SHELL

- The shell is an interface in which we are able to interact with the OSGI Container and the modules within it.
- The shell is lightweight, and it's easy to add new commands.
- It acts like a POSIX shell and has similar BASH/SH syntax and commands.
- The specification of the Shell can be found in RFC-132:  
<https://osgi.org/download/osgi-4.2-early-draft.pdf>

---

## CONNECTING TO THE SHELL

- The Shell is composed of four parts:
  1. The Command Processor
  2. The ThreadIO
  3. The Converter
  4. The Command Provider
- The Command Processor is what allows outside access to the shell through telnet or other means.
- We first encountered the Command Processor when telnetting into the shell using the command `telnet localhost 11311`.

---

## ENTERING A COMMAND

- Once connected to the shell, the Command Processor is also responsible for the commands that we enter, making sure the right command is executed.
- Everything in the shell is Object based manipulation rather than String based
  - The commands we execute are method calls to particular services which implement the command.
- When we type a command such as `start 431`, it's actually a method call on a specific service that corresponds to the command `start`.
- The command services and their implementations are found within the Command Provider.

---

## USER INTERACTION

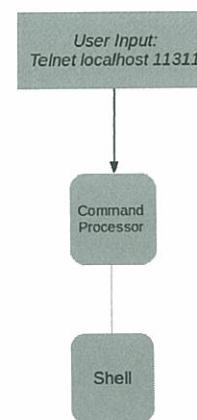
- If a specific command needs to interact with the User, that's where the ThreadIO comes into play.
- The ThreadIO allows a command to use `System.In`, `System.Out` and `System.Err` to prompt for an input or send messages to the end user.
- If ThreadIO prompts the User for an input or there is already a parameter following the command (e.g. `431` in the command `start 431`), the parameter is taken and passed to the last part of the shell.

## FROM TEXT TO OBJECTS

- » The Converter is what takes the parameter from the command and converts it to a specific object-type and back to text.
- » The shell is implemented using object-based manipulation rather than String (text) based.
- » The commands, such as `lb`, `start`, `install` correspond to methods of a service.

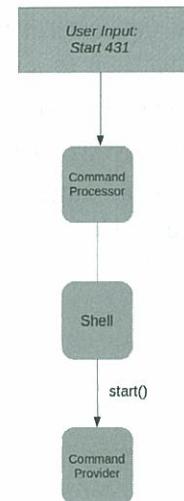
## TELNET THROUGH THE COMMAND PROCESSOR

- » Let's take a look at everything all put together.
- » We first telnet into the shell through the Command Processor.



## COMMAND PROCESSOR PROCESSES THE COMMAND

- When we type a command such as `start 431`, the Command Processor takes the command and invokes the method that corresponds to that command found in the Command Provider.

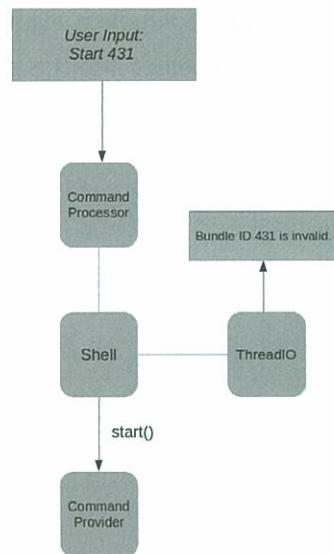


WWW.LIFERAY.COM

LIFERAY.

## TAKE AN INPUT OR SEND A MESSAGE

- At this point if the command needs an input from the User or needs to display a message, the ThreadIO provides that for the command.

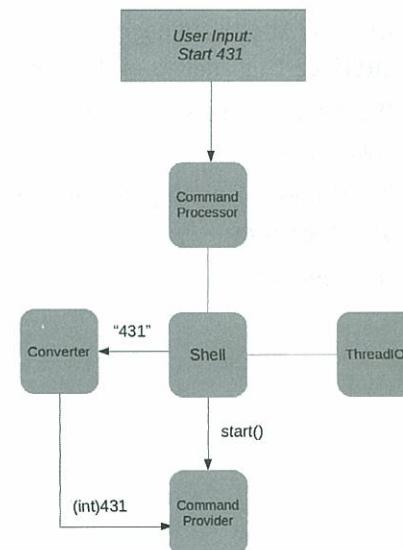


WWW.LIFERAY.COM

LIFERAY.

## CONVERT PARAMETERS

- Since our command `start 431` has the parameter of `431`, the Converter will typecast our parameter into the specific object type it needs to be in order to run our command (method).

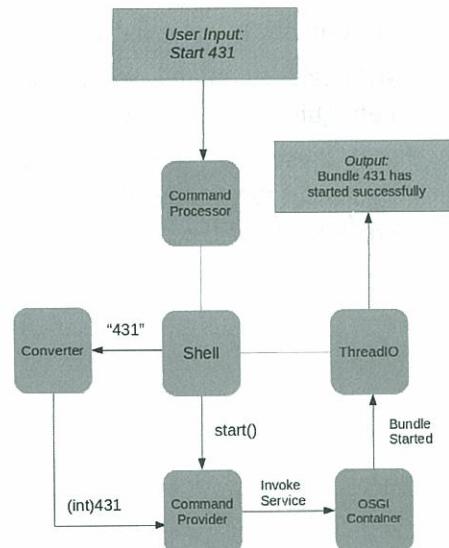


WWW.LIFERAY.COM

LIFERAY

## EXECUTE

- Finally, the command is executed, and, if applicable, feedback is sent back to the end user.



WWW.LIFERAY.COM

LIFERAY

---

## TINY SHELL LANGUAGE

- The Tiny Shell Language (TSL) is the command syntax of the shell interface that makes all this possible.
- TSL is what allows text we input to be interpreted as method calls on Java Objects.
- TSL allows for piping, closures, variable setting and referencing, and other features.
- For now, we'll keep things simple.

---

## COMMANDS OF THE SHELL

- Starting off, to get a list of commands simply type `help`.
- With each command, you can type `man <command>` to get the manual or help guide of a specific command (assuming that one is written for it).
- With these two commands, you can read about each command and figure out what all of them do, but we'll spare you the reading and show some of the common commands.

---

## LIST OF MODULES AND PIPING

- » To show a list of all the *modules* installed within Liferay, type `lb`.
  - » `lb` stands for list *bundles*, Liferay modules are known as bundles in OSGi.
- » If you are looking for a specific module you can use `lb <bundle name>`.
- » Commands can be chained or piped using `|` allowing one command's input to be the input of another command.

```
_g! lb | grep Polls_
305|Active      | 1|Liferay Polls Service (1.0.0)
329|Active      | 1|Liferay Polls Web (1.0.0)
398|Active      | 1|Liferay Polls API (1.0.0)
true
```

---

## VARIABLE ASSIGNMENT

- » We can assign values to a variable just like we do in Java.

```
variable1 = 'Here is a Sentence'
```
- » We can also reference our variables using `$<variable-name>`

```
g! echo $variable1
Here is a Sentence
```
- » There are other commands which will be covered in later sections.

Notes:



## BUNDLE INSTALLATION

Copyright ©2016 Liferay, Inc.  
All Rights Reserved.

No material may be reproduced electronically or in print,  
distributed, copied, sold, resold, or otherwise exploited  
for any commercial purpose without express written  
consent of Liferay, Inc.

### BUNDLES AND MODULES

- At a minimum, modules are composed of just a `manifest file`.
- OSGi bundles are referred to as modules within Liferay and the terms can be used interchangeably.
  - OSGi bundle = Liferay module
- How do we install these modules in Liferay?

---

## DEPLOYING A MODULE IN LIFERAY

- » When installing a module, there are a number of states the module goes through.
- » The first state is *installed*, where Liferay checks the manifest for errors, resolves dependencies, then moves on to the next state.

---

## RESOLVING A MODULE

- » After *installed*, the next state is *resolved*.
- » Once a module is in the *resolved* state, it means that the OSGi Container was able to resolve all of the module's dependencies, whether they are external Java classes, services or other declared dependencies.
- » A module is ready to be started once it is in the resolved state.

---

## STARTING A MODULE

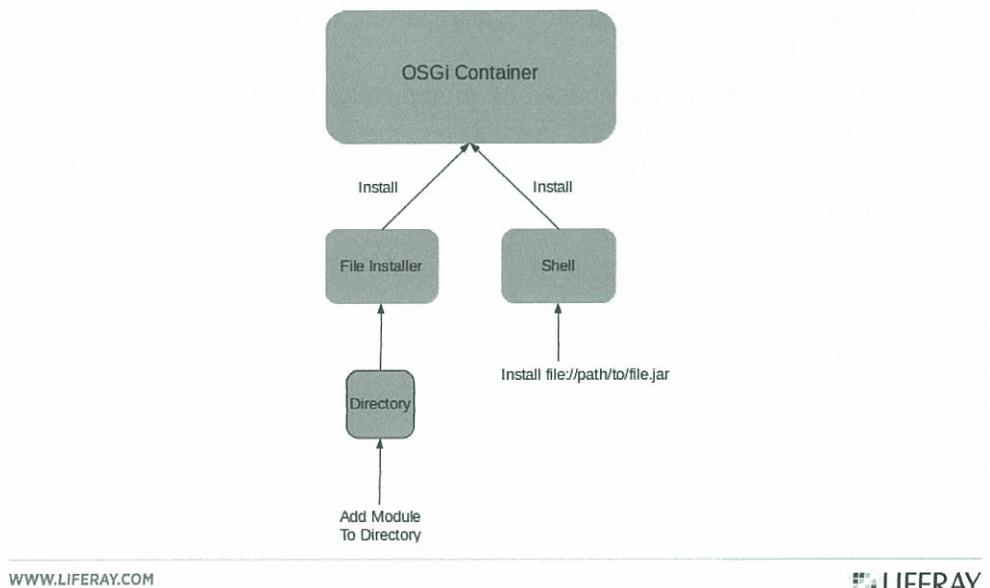
- Some modules may be automatically started, while others may be started manually.
- A module is either lazy loaded or immediately loaded.
  - *Lazy loaded* will install a module up to the resolved state.
  - *Immediate loading* will start the module after it enters the *resolved* state.
- Liferay modules are by default ~~lazy~~ loaded.  
*immediate*

---

## TWO WAYS OF INSTALLATION

- In order to install a module in Liferay, methods from a standard Java API are invoked to install and start the module.
- We can use other modules to help and manage the installation of our new modules.
- Two implementations of installing a module are the File Install and the Shell.

## TWO WAYS OF INSTALLATION VISUALIZED



## INSTALLING MODULES WITH THE SHELL

- To install a module using the Shell, use the command `install` followed by the URI of the module that you are installing.
- Some examples of installing a module using a URI:

```
install file:/Users/liferay/Desktop/my.module.jar  
install ftp://ftp.liferay.com/my.module.jar  
install http://www.liferay.com/downloads/fake-module-example.jar
```

```
g! install file:/home/liferay/Desktop/lifecycle-demo.jar  
Bundle ID: 446
```

---

## CONFIRMING OUR MODULE INSTALLATION

- » Once the module has been installed, we can check our list of modules to ensure it has been installed using `lb <module_name>`.
- » When installing through the Shell, the JAR file is not placed in a specific folder, but retrieved and installed based upon the URI provided.

---

## INSTALLING MODULES WITH FILE INSTALL

- » The *File Install* works by having a directory or directories designated for module management.
- » When a module is placed in a specified directory, it is deployed.
- » Upon being deployed, the module first goes through the *Installed* state and then the *Resolved* state.
  - » On install, the module is set to the *installed* state, the manifest is validated, and the module is registered in the OSGi container.
  - » On resolve, dependencies are resolved, and if successful, the module is set to the *resolved* state.

---

## LIFERAY DESIGNED DIRECTORIES

- » Installing using the File Install is very straight forward.
- » Take the module and place it in either of the following directories:

LIFERAY\_HOME/deploy/  
LIFERAY\_HOME/osgi/modules/

- » From there, Liferay will handle where to place the module.
- » Once placed in either folder, the module will then be installed into the OSGi Container.

---

## WHICH INSTALL METHOD TO USE

- » Because the *File Installer* and the Shell are separate modules, they initiate the lifecycle of the modules independently.
- » If you install a module using the File Installer, the Shell has no control over the module's installation, and vice versa.
- » Mixing the methods may cause issues and unexpected behavior when updating the module.
- » To ensure consistency, only use one installer - preferably the File Installer.

## WHICH INSTALLER TO USE

- Liferay provides two ways to install a module - the *File Installer* and using the Shell.
- Liferay recommends using the *File Installer* to install modules.
  - When a module is placed in either of the File Installer's designated directories, the module will be installed.

Notes:



## DEPLOYMENT STATUS

Copyright ©2016 Liferay, Inc.  
All Rights Reserved.

No material may be reproduced electronically or in print,  
distributed, copied, sold, resold, or otherwise exploited  
for any commercial purpose without express written  
consent of Liferay, Inc.

## DEPLOYMENT STATUS

- File Install is our primary way of installing modules into Liferay.
- Once installed, a module is easily managed through the Shell.

## EXERCISE: LIST OF MODULES

1. Open a *terminal* or *command prompt* window.
2. Connect to the Shell:  
`telnet localhost 11311`
3. Type the command `lb` to see a list of all the modules installed in Liferay.
  - » We'll compare this list of current modules installed to a new list of modules after we install our module.

ID State	Level Name
0 Active	0 OSGi System Bundle (3.10.200.v20150831-0856)
1 Active	6 Apache Felix Configuration Admin Service (1.8.8)
2 Active	6 Liferay Portal Configuration Persistence (1.0.0)
3 Active	6 org.osgi.org.osgi.service.metatype (1.3.0.201505202024)
4 Active	6 Meta Type (1.4.200.v20150715-1528)
5 Active	6 Apache Felix EventAdmin (1.4.4)
6 Active	6 Apache Aries JMX API (1.1.1)
7 Active	6 Apache Aries Util (1.0.0)
8 Active	6 Apache Aries JMX Core (1.1.3)
9 Active	6 Apache Felix Declarative Services (2.0.2)
10 Active	6 Apache Felix Bundle Repository (2.0.2)

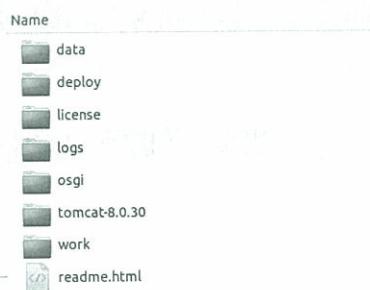
WWW.LIFERAY.COM

LIFERAY

## EXERCISE: DEPLOYING OUR MODULE

- » We're first going to be installing our module using File Install:

1. Go to the `application-developer-training-exercises-[version]/06-interacting-with-the-shell` folder.
2. Copy the file `com.liferay.training.lifecycle.command.jar`.
3. Paste the JAR in the `LIFERAY_HOME/deploy` / folder.

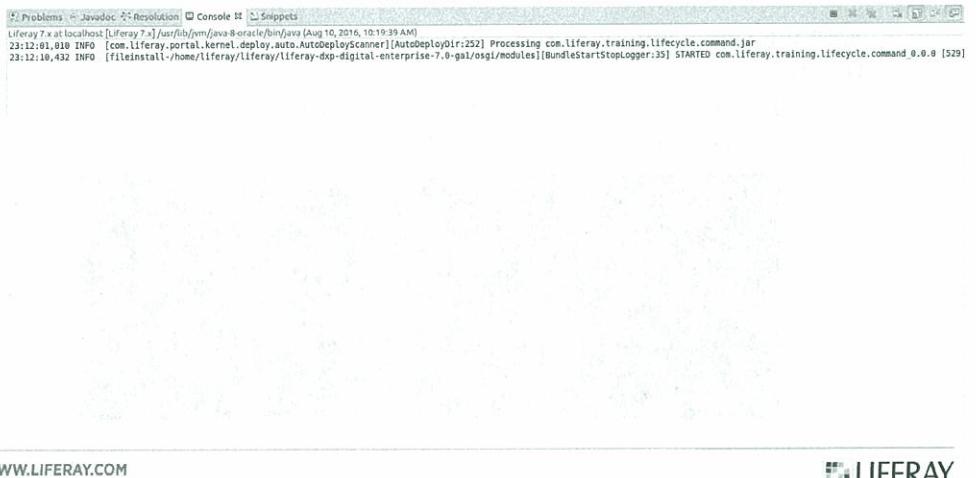


WWW.LIFERAY.COM

LIFERAY

## CONFIRMING OUR DEPLOYMENT

- » You should see something like:



A screenshot of a Java IDE's console tab. The log output shows the deployment of a command module named 'liferay-training-lifecycle-command'. It includes timestamped INFO messages from the 'AutoDeployScanner' and 'fileinstall' components, indicating the processing of the 'lifecycle.command.jar' file and its successful start.

```
Liferay 7 at localhost [Liferay 7.1] [User:lb] [env:java-8-oracle] [Java] [Aug 10, 2016, 10:19:39 AM]
23:12:01,010 INFO [com.liferay.portal.kernel.deploy.auto.AutoDeployScanner][AutoDeployDir:252] Processing com.liferay.training.lifecycle.command.jar
23:12:10,432 INFO [fileinstall-/home/liferay/liferay-dsp-digital-enterprise-7.0-gal/osgi/modules][BundleStartStopLogger:35] STARTED com.liferay.training.lifecycle.command_0.0.0 [529]
```

WWW.LIFERAY.COM

LIFERAY.

## EXERCISE: CHECK MODULE STATUS

1. Go to the Gogo Shell.
2. Type the command `lb` to see the list of modules.

- » Towards the bottom of the list, you should see something like this:

```
505|Active    | 10|lifecycle.portlet (1.0.0)
507|Active    | 10|com.liferay.training.hello.portlet (0.0.0.201608172132)
508|Active    | 10|hello-service (1.0.0.201607281731)
509|Active    | 10|hello-api (0.0.0)
510|Installed | 10|com.liferay.training.deployment.impl (0.0.0.2016031120)
513|Active    | 10|Hello Service (1.0.0)
514|Active    | 10|com.liferay.training.lifecycle.command (0.0.0)
```

- » At the top of the list of modules you'll also find what each column represents:

ID	State	Level	Name
----	-------	-------	------

WWW.LIFERAY.COM

LIFERAY.

---

## MODULE NUMBER

- Let's break down each column in more detail:  
444|Active |10|com.liferay.training.lifecycle.command (0.0.0)
- 444 refers to the module number: each module is assigned a unique ID upon install.
  - Your module number for the `lifecycle.command` may be different from what is above.
- Previous modules that are installed will retain the module number assigned to them, if they are installed again.

---

## STATE

- In our example:  
444|Active |10|com.liferay.training.lifecycle.command (0.0.0)
- Active refers to which part of the lifecycle the module is in.
- Upon installation, modules enter various phases of the lifecycle.
- By seeing Active, we know that the module has already gone through the *installed*, *resolved* and *starting* state in order to reach **active**.
- In the *active* state:
  - Components are available
  - Services can be called
  - Applications run

---

## START LEVEL

- » *Start levels* determine the start order of modules.
- » Start levels can be defined on a module.
- » A start level of 0 is associated with a system module and can't be changed.
- » Start levels can vary from system to system for a single module.
- » In general, modules should not be developed with specific start levels in mind unless absolutely necessary.

---

## NAME

- » Name refers to the name of the module as well as the version number of the module.
- » Both name and module version are set within the *manifest*.

---

## EXERCISE: STOPPING A MODULE

- » Now that we know what everything means, let's actually see how to start and stop our modules.
  - » To stop a module, type `stop` in the Shell.
  - » Go ahead and stop our *lifecycle.command* module using the command above and the appropriate module number:
    1. Find the module's *ID* from the list.
    2. Type the command `stop 444`.
      - 2.1 Be sure to replace 444 with your module's ID.
- ✓ See that the module is stopped with the command `lb`.

```
509|Active    | 10|hello-api (0.0.0)
510|Installed | 10|com.liferay.training.deployment.impl (0.0.0.201)
)
513|Active    | 10|Hello Service (1.0.0)
514|Resolved  | 10|com.liferay.training.lifecycle.command (0.0.0)
```

WWW.LIFERAY.COM

LIFERAY.

---

## EXERCISE: STATUS OF A STOPPED MODULE

- » Before *starting* the module, let's take a look at what state our module is in.
  1. Type the command `lb lifecycle`.
- » We see that the status of the module is **resolved**, meaning dependencies have been resolved, outside classes are available, and any needed services are available.

WWW.LIFERAY.COM

LIFERAY.

---

## EXERCISE: STARTING A MODULE

- » When a module is resolved, that means that the module is ready to be started:

1. Type the command *start* to start the module:

```
start [module number]
```

- » You should see the following:

```
[BundleStartStopLogger:35] STARTED com.liferay.training.lifecycle.command
```

---

## REMOVING A MODULE

- » To *uninstall* a module from Liferay, there are two ways:
- » If you installed a module from the Shell, you can uninstall the module from the Shell using the command *uninstall [module number]*.
- » If the module was installed with File Install, remove the JAR file from the `LIFERAY_HOME/osgi/modules` directory.

---

## EXERCISE: REMOVING A MODULE

- Since we installed our module using the File Install, we'll uninstall using File Install.
1. Go to the directory LIFERAY\_HOME/osgi/modules.
  2. Delete the file com.liferay.training.lifecycle.command.jar from the modules/ directory.
  3. Go to the Shell again if needed.
  4. See that the module has been removed from the list with 1b.

---

## BUNDLING EVERYTHING TOGETHER

- We've seen how to deploy a module in the OSGi Container.
- Once deployed, we focused on how to interpret the output of the command 1b.
- From there, we explored a list of commands used to manage our deployed module.

Notes:

# Chapter 7

## Building Portlet Modules

Portlets are Java components designed to be used in a modular way. They can be combined to build complex user interfaces. This chapter will introduce you to portlets and show you how to build them.

### What is a Portlet?

A portlet is a Java component that is designed to be used in a modular way. It can be combined with other portlets to build complex user interfaces. Portlets are typically used in web applications, but they can also be used in desktop environments.

### How do Portlets Work?

Portlets work by providing a standard interface for other components to interact with them. This interface is defined by the Java Portlet API. Portlets can be deployed as separate modules or as part of a larger application. They can be used to display data, perform actions, or both.

### Building a Portlet Module

To build a portlet module, you will need to follow these steps:

- Create a new Java project.
- Add the Java Portlet API dependency to your project.
- Create a new portlet class that implements the `Portlet` interface.
- Configure the portlet in the deployment descriptor.
- Deploy the portlet module to a web server.
- Test the portlet in a web browser.



## PORTLET COMPONENTS

Copyright ©2016 Liferay, Inc.  
All Rights Reserved.

No material may be reproduced electronically or in print,  
distributed, copied, sold, resold, or otherwise exploited  
for any commercial purpose without express written  
consent of Liferay, Inc.

### OVERVIEW

- Code is deployed into Liferay through modules.
- Services provide additional functionality.
- Modules “talk” to each other using services.
- By themselves, modules are not visible in Liferay’s web interface.
- Any apps you want to appear in Liferay’s web interface will need a controller and a view.
- These are provided by portlets, implemented as components in the *OSGi Container*.

---

## COMPONENTS

- » A *component* is simply an object that provides specific functionality.
- » Components are contained within a module.
- » They are meant to be reusable and thus highly modular.
- » The other modules in our application all publish services to the OSGi Container.
- » This allows us to develop these components independently, while allowing them to talk to each other using these services.

---

## PORTLET COMPONENTS

- » A portlet component is meant to provide a user interface (UI) for an application.
- » This means that it requires a view, as well as a controller to connect it to the other modules that comprise the application.

## HOW DO WE PUBLISH SERVICES?

- » In Liferay, the Service Registry looks up *registered* services.
- » Services can be manually registered using the `registerService()` method in a `BundleActivator` class.
- » However, this can be messy, and leads to a lot of boilerplate code.
- » Instead, *Declarative Services* provides a way to simply *declare* what services we provide.
- » It allows us to handle service registration without having to write any `BundleActivator` code.

## TRADITIONAL SERVICE REGISTRATION

- » Traditionally, service registration would look like this:

```
public void start(BundleContext bundleContext) throws Exception {
    Dictionary properties = new Hashtable<>();
    properties.put("com.liferay.portlet.display-category",
                  "category.sample");
    properties.put("com.liferay.portlet.instanceable", "true");
    properties.put("javax.portlet.display-name", "My OSGi API
                  Portlet");
    properties.put("javax.portlet.security-role-ref",
                  new String[] {"power-user", "user"});
    _serviceRegistration = bundleContext.registerService( Portlet.class,
                                                       new OSGiAPIPortlet(), properties);
}

public void stop(BundleContext bundleContext) throws Exception {
    _serviceRegistration.unregister();
}

private ServiceRegistration _serviceRegistration;
```

## DECLARATIVE SERVICES

- » Using Declarative Services, the same thing could be done like this:

```
@Component(  
    immediate = true,  
    property = {  
        "com.liferay.portlet.display-category=category.sample",  
        "com.liferay.portlet.instanceable=true",  
        "javax.portlet.display-name=My DS Portlet",  
        "javax.portlet.security-role-ref=power-user,user"  
    },  
    service = Portlet.class  
)
```

- » This doesn't require a separate annotation class, and keeps your code a lot cleaner.
- » Declarative Services also allows you to skip writing XML, and instead configure your component within your Java class.

## PORLET COMPONENTS USING DECLARATIVE SERVICES

- » As seen above, portlet components can be created using Declarative Services.
- » At a minimum, you need the following to declare a portlet component:

```
@Component (  
    service=Portlet.class  
)
```

- » This provides a *component* that implements a *Portlet* service.
- » You'll likely want to set more options than this.

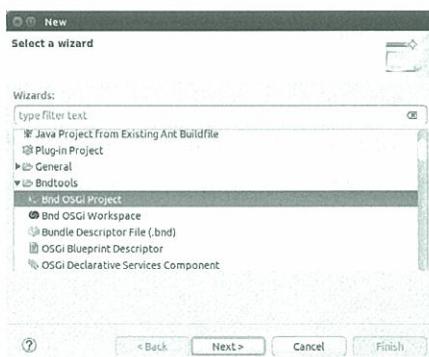
## NAVIGATING THE CODE

```
@Component (  
    immediate=true,  
    service=Portlet.class  
)
```

- » **@Component:** This is the annotation required to tell the OSGi Container we have a *component*.
- » **immediate=true:** By default, Components aren't created as soon as the module is installed. Set this to **true**, and the portlet will be immediately started upon installation. This will make it available in the web interface immediately.
- » **service=Portlet.class:** This indicates that we want to create a Portlet Component. In other words, we are declaring our intention to implement the Portlet API.

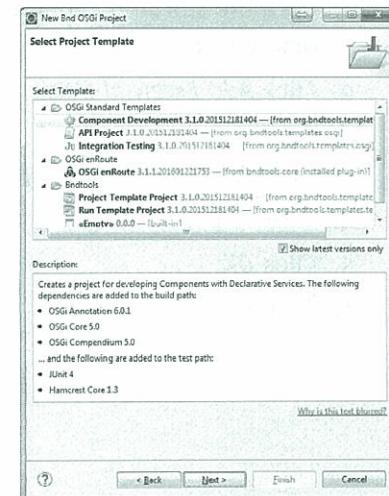
## EXERCISE: CREATING A BND OSGI PROJECT

1. Click *File* → *New* → *Other* in Liferay IDE.
2. Click on *Bnd OSGi Project* under *Bndtools*.
3. Click *Next*.



## EXERCISE: COMPONENT DEVELOPMENT

1. Find the *OSGi Standard Templates* heading under *Select Template*.
2. Choose *Component Development*.
3. Click *Next*.

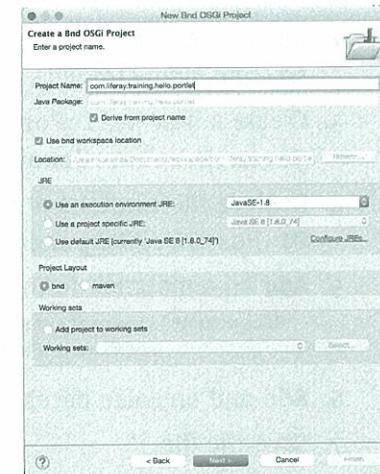


WWW.LIFERAY.COM

LIFERAY.

## EXERCISE: NEW BND PROJECT

1. Type *com.liferay.training.hello.portlet* for the project name, leaving the rest of the default values.
2. Click *Next*.

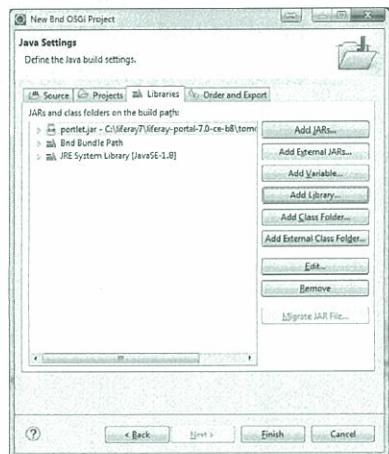


WWW.LIFERAY.COM

LIFERAY.

## EXERCISE: JAVA SETTINGS

1. Click the *Libraries* tab in the *Java Settings* dialog box.
2. Click *Add External JARs...*
3. Go to  
[LIFERAY\_HOME]/tomcat-[version]/lib/ext.
4. Choose portlet.jar.
5. Click *OK*.
6. Click *Finish*.



WWW.LIFERAY.COM

LIFERAY.

## EXERCISE: CREATING A PORTLET

1. Delete the existing autogenerated source folder test/.
2. Delete the existing autogenerated class Example.java found in src/com/liferay/training/hello/portlet.
3. Create a new HelloPortlet class in the package that extends javax.portlet.GenericPortlet.
4. Replace the contents of the class with snippet 01-Portlet-Components-Class.
5. Add an annotation to the class to make it a *Component*:

```
@Component(immediate=true,  
           service = Portlet.class)
```
6. Add and organize necessary imports using [ctrl]-[shift]-[o].
7. Save the file.

WWW.LIFERAY.COM

LIFERAY.

---

## EXERCISE: VIEW THE PORTLET

- Bndtools has now built our portlet component and produced the module JAR.
  - This JAR gets created in the project's generated folder.
  - Time to let the OSGi Container take over.
1. Copy the JAR from the module's generated folder into Liferay's deploy folder to deploy it.
  2. Sign in to Liferay.
  3. Add the portlet to the page.
    - It can be found in the *Undefined* category, and currently has no title.

---

## CHECKPOINT

- Portlets provide a UI and controller for our apps.
- They are how we can interact with our apps using Liferay's web interface.
- They are implemented as components.
- Components are services that serve a specific purpose.
- They are declared using *Declarative Services* annotations.

Notes:



## SETTING PORTLET ATTRIBUTES

Copyright ©2016 Liferay, Inc.

All Rights Reserved.

No material may be reproduced electronically or in print,  
distributed, copied, sold, resold, or otherwise exploited  
for any commercial purpose without express written  
consent of Liferay, Inc.

## CONFIGURING YOUR PORTLET

- » We've just created a very basic portlet using a standard Java GenericPortlet class.
- » Calling the portlet "generic" is really selling it short.
- » There are actually a lot of things we can do with this portlet, but we're probably going to need more than a single portlet class.
- » In order to harness the power we have here, let's take a look at how to configure various aspects and properties of our portlet.

---

## APPLICATION STRUCTURE

- » As you may recall, we're developing an application using a Portlet Component.
- » Components are a kind of service, registered through Declarative Services, that performs a discrete task.
- » This Component's task is to implement a Portlet class to provide a Controller for our application.
- » This means that we have to look at the settings of both the Component and the Portlet to provide the proper configuration in our portlet class.

---

## COMPONENT DECLARATION BREAKDOWN

- » Components have specific elements that go inside of the @Component declaration.
- » Let's take a look at what we do there.
  - » `-immediate = [true/false]`: determines whether the component will activate automatically upon installation or not.
  - » `-service = [classname]`: provides the class the component is registered to as a service.
  - » `-property = [a list of properties, each being a key=value pair]`: we're primarily interested in attributes that we can set on the component.

## PORLET PROPERTIES

- » In addition to the component's configuration, a Portlet Component will also contain the configuration for a portlet.
- » In a Java Standard Portlet, these would be set in a `portlet.xml` file.
- » In a Portlet Component, these are set inside of the `-property` element.
- » With that in mind, let's talk about configuring the portlet and its properties to meet your needs.
- » There are a number of properties from basic configuration to more advanced features which can be configured via properties under the `@Component` annotation in the portlet class.

## CLOTHES SHOPPING

- » To get a handle on what's happening with attributes, let's look at it in terms of shopping for clothes.
- » Let's say you're building three portlets for a project. We'll look at them as a pair of shoes, a pair of pants, and a shirt.
- » You don't need the same color or size for all three. In fact, you want a large blue shirt, size 34 grey pants, and size 10 black shoes. Notice that the sizes don't make sense in reference to each other.
- » Similarly, your three portlets will fulfill different purposes, and you'll need different considerations for each.
- » The "size" that makes sense for one portlet won't make sense for another, and the attributes, like "color", that you configure for one might not be configured for another.

---

## WHAT WE CAN CONFIGURE WITH PORTLET ATTRIBUTES

- » Like we've said, we can describe a broad range of things using the properties available in our component.
- » We can provide a category for the application to appear in when you're adding content to your site.
- » There are also configurations for standard portlet settings and Liferay-specific portlet enhancements.
- » We can also create custom properties and define their settings here.

---

## PORTLET.XML AND LIFERAY-PORTLET.XML

- » From a general portlet perspective, anything that you can find in the portlet specification standard, `portlet.xml`, can be declared in the component properties.
- » There is a difference in syntax. For example:
  - » The `portlet.xml` property `<portlet-name>` appears in the component declaration as "`javax.portlet.portlet-name`".
  - » Or `<portlet-mode>` would be "`javax.portlet.portlet-mode`".
- » We can declare any Liferay-specific settings that would be set in `liferay-portlet.xml` as well.
- » Liferay specific portlet properties follow the same pattern.
  - » The `<icon>` property becomes "`com.liferay.portlet.icon`"
  - » Or `<show-portlet-access-denied>` would be "`com.liferay.portlet.show-portlet-access-denied`".

## PORTLET.XML VS. COMPONENT CONFIGURATION (I)

- » Let's take a look at a `portlet.xml` file and an equivalent configuration in a Portlet Component.

### portlet.xml

```
<portlet>
    <portlet-name>sample-portlet</portlet-name>
    <display-name>Sample Portlet</display-name>
    <portlet-class>
        com.liferay.training.sample.portlet.SamplePortlet
    </portlet-class>
    <init-param>
        <name>view-template</name>
        <value>/html/view.jsp</value>
    </init-param>
    <resource-bundle>content.Language</resource-bundle>
    <security-role-ref>
        <role-name>user</role-name>
    </security-role-ref>
</portlet>
```

WWW.LIFERAY.COM



## PORTLET.XML VS. COMPONENT CONFIGURATION (II)

### Component Configuration

```
property = {
    "javax.portlet.display-name=Sample Portlet",
    "javax.portlet.security-role-ref=user",
    "javax.portlet.init-param.view-template=/html/view.jsp",
    "javax.portlet.resource-bundle=content.Language"
}
```

- » We can see two different ways of presenting the same data. We can also see that there is a clear migration path from one to the other
- » If you look at the hierarchy, you can see `<portlet>` = `javax.portlet`, `<init-param>` = `javax.init-param`, and so on.
- » Where in `portlet.xml`, each level is divided by XML hierarchy, in the component configuration, you simply delimit each level in the hierarchy with a `"."`

WWW.LIFERAY.COM



---

## INIT-PARAM

- The parameters set when a portlet is initialized are some of the most important sets of properties we can define.
- In `portlet.xml`, the `<init-param>` element specifies initialization parameters to create an initial state inside your portlet class.
- In our configuration, the properties will appear as  
"javax.portlet.init-param..."
- In our application, we'll use `init-param` properties to define the location of resources, like JSPs and properties files, that our portlet will need.
- Initialization parameters are also important because they exist outside of the database and so can be quickly changed with an edit and a redeploy to provide flexibility.

---

## IMPORTANT PROPERTIES

- To demonstrate how we set these attributes, let's add a few important standard Java properties from `portlet.xml`.
- These will take care of some standard portlet needs:
  - `javax.portlet.display-name`= provides the display name for the portlet.
  - `javax.portlet.security-role-ref`= sets security information for the portlet, but can be overridden by platform permissions.
  - `javax.portlet.resource-bundle`= sets the location of the default file which contains keys for localization.

---

## EXERCISE: ADDING STANDARD PROPERTIES

1. Open HelloPortlet.java.
2. Add a comma after service = Portlet.class inside the @Component declaration.
3. Add snippet *o1-standard-attributes* below it.

✓ Your declaration will look like this:

```
@Component(  
    immediate = true,  
    service = Portlet.class,  
    property = {  
  
        "javax.portlet.display-name=Hello Portlet",  
        "javax.portlet.security-role-ref=power-user,user",  
        "javax.portlet.resource-bundle=content.Language"  
    }  
)
```

---

## EXERCISE: TESTING THE ATTRIBUTES

- ❖ The display name is easy to demonstrate.
1. Deploy the portlet again by copying the newly generated JAR from generated to the deploy folder.
  2. Go to the page where you have the portlet displayed, via localhost:8080 in your browser.
  3. Sign in to Liferay.
- ✓ See the portlet title on the page. It will match what you declared.



Hello Portlet

Hello World!

---

## EXERCISE: DEFAULT BEHAVIOR

» Before we add any attributes, let's look at the current default behavior.

1. Sign in.
2. Click on the *Options* menu for our portlet.
3. Choose *Permissions*.



WWW.LIFERAY.COM

LIFERAY

---

## EXERCISE: NOTHING TO SEE HERE

1. Uncheck the *View* permission for the *Guest* role on the *Permissions* page.
  2. Save the lightbox.
  3. Close the lightbox.
  4. Sign out of the administrative account.
- ✓ Take a look at the app on your page:

You do not have the roles required to access this portlet.

» That doesn't look great, but we can fix it.

WWW.LIFERAY.COM

LIFERAY

---

## EXERCISE: YOU CAN LOOK, BUT YOU CAN'T TOUCH

- » Let's look at one more default behavior before we continue.
  - 1. Sign in again as the admin user.
  - 2. Open the *Add → Applications* menu.
  - 3. Try to add another copy of the application to the page.
- ✓ You can see the new portlet name, but there is no option to add it to the page. The category also remains *Undefined*.

WWW.LIFERAY.COM

LIFERAY.



---

## LIFERAY-SPECIFIC ATTRIBUTES

- » The previous properties we added were standard portlet properties from `portlet.xml`.
- » Our application has a proper display name, but there are still some important pieces missing.
- » We'll add the following to help everything integrate the application into Liferay:
  - » `"com.liferay.portlet.display-category=` specifies the category where the application will appear.
  - » `"com.liferay.portlet.instanceable=` determines whether a portlet will display with an error message or not at all if the User doesn't have access to it.
  - » `"com.liferay.portlet.show-portlet-access-denied=` determines whether a portlet will display with an error message or not at all, if the user doesn't have access to it.

WWW.LIFERAY.COM

LIFERAY.

---

## EXERCISE: ADDING LIFERAY-SPECIFIC PROPERTIES

1. Go to HelloPortlet.java inside the @Component declaration.
2. Add snippet *o2-liferay-attributes* below the properties you just added, and add commas if necessary.
3. Save the file - the portlet will rebuild.
4. Copy the JAR from generated to deploy to redeploy it.

✓ Your properties section should look like this:

```
property = {  
    "javax.portlet.display-name=Hello Portlet",  
    "javax.portlet.security-role-ref=power-user,user",  
    "javax.portlet.resource-bundle=content.Language",  
    "com.liferay.portlet.display-category=category.sample",  
    "com.liferay.portlet.instanceable=true",  
    "com.liferay.portlet.show-portlet-access-denied=false"  
}
```

---

## EXERCISE: TESTING THE NEW ATTRIBUTES

- » Let's go back to the web browser and test the new behaviors.
  1. Refresh the page.
    - » Since we changed the portlet to "instanceable", our portlet is gone. Let's re-add it.
  2. Sign in and add the Hello Portlet, under the *Sample* category, to the page.
  3. Edit the permissions for the Hello Portlet so that the Guest role cannot view the portlet.
  4. Sign out to verify that the permissions are working as we expect.
    - » Can you see the portlet or error message as an unauthenticated User?
  5. Sign in.
    - » How does the category display now?
    - » Can you add multiple "instances" of the application to the page?

---

## USING ATTRIBUTES

- » There are, of course, cases where you might want an error message to display as well as cases where you wouldn't.
- » For an app that displays on a public-facing website that some Users may not have access to, you probably don't want to display an ugly message or let Users know there's something there that they can't see.
- » For an app displaying on an intranet page that may have some complicated internal permissions, you might want Users to see a message saying they can't view it so that they can request access.
- » This is why we have attributes. There's very little in software development that is "one size fits all", so we try to have as many sizes of pants, shirts, and shoes as are available to fit any app.

---

## WHERE ARE ALL THE ATTRIBUTES?

- » So all this attribute stuff is great, you're thinking, but where can I see all of them?
- » There are too many to list them all here, but you can find a complete list of all standard portlet attributes in the base `portlet.xml`.
- » A complete list of Liferay-specific attributes, with more information, is in `liferay-portlet-app_7_0_0.dtd`.
- » You can also look at Liferay's `PortletTracker.java` and `PortletPropertyValidator.java` to see what happens behind the scenes as attributes are processed.

Notes:



## PRESENTATION LAYER

Copyright ©2016 Liferay, Inc.

All Rights Reserved.

No material may be reproduced electronically or in print,  
distributed, copied, sold, resold, or otherwise exploited  
for any commercial purpose without express written  
consent of Liferay, Inc.

### PORTLET REVIEW

- So far, we've created a basic portlet and edited some of its attributes.
- The portlet class is a service produced by a component.
- The portlet's attributes are managed through properties inside of the @Component annotation.
- This portlet class represents the Controller in our application.
- The Controller connects the business logic to the View layer which is presented to users.
- In this exercise, we'll take a look at the View layer and how our portlet is presented to the user.

---

## VIEW LAYER

- » The View layer of our portlet is what the user will end up seeing and interacting with.
- » In some cases, this could be a simple interface in the Gogo Shell. In others, it will be a complex UI on a web page.
- » The key here is that an end user will need some way to interact with the application or receive data from it.



WWW.LIFERAY.COM

LIFERAY.

---

## SOMETIMES IT'S JUST A VIEW

- » In a very simple case, the View layer may be exclusively informational.
- » Imagine a restaurant that uses a screen at each table to display the menu.
  - » The restaurant customers (your users) can view the content display by the application and tell their order to a waitress.
  - » The View layer is simply informational and provides no form of interaction.



WWW.LIFERAY.COM

LIFERAY.

## SOMETIMES IT'S AN INTERACTION

- In other cases, you might need your users to interact with your application.
- Now, the same restaurant has decided that they want customers to order directly from the screens with no wait staff taking orders.
  - The restaurant customers can now use a touch screen to select the things that they wish to order, pay for the order, and receive feedback when their order is placed.
  - The View layer will now receive interaction and provide feedback to the users.



WWW.LIFERAY.COM

LIFERAY.

## THE BIG PICTURE

- Let's take a step back and imagine the whole app.
  - You have the Model, which provides the methods to connect to the database containing the menu items, prices, and so on.
  - You have the View, which is an interactive touch app that customers enter their orders and pay with.
  - You have the Controller, which is a portlet class which directs traffic and manages the various calls.
  - In this case, you might have a second application View which notifies the kitchen staff when an order is placed
- What would this application look like from the perspective of the OSGi Container?

WWW.LIFERAY.COM

LIFERAY.

---

## HOW DO WE BUILD A VIEW?

- So a View can be a complex UI or a text-only shell interface or a touch screen tablet interface on a tablet. How would you implement that?
- One of the most common ways to implement the View in Liferay is using JSPs (JavaServer Pages). This is generally the default for Liferay development.
- JSF (JavaServer Faces) and Facelets are another great option.
- If you're creating a mobile app, you might want to use Liferay Screens to connect the backend to a usable iOS or Android app.
- If all you need is text access through a shell, you can create a proxy class that defines your shell methods.

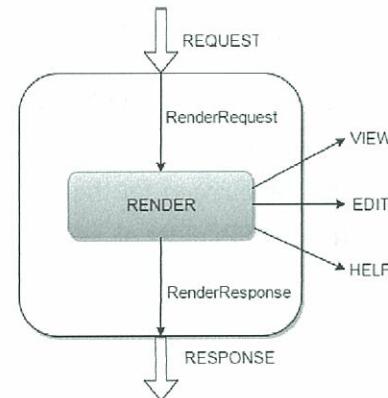
---

## PORTLET JSPs

- JSPs allow developers to dynamically create web pages.
- A JSP can contain HTML, XML, Javascript, and Java fragments, which are compiled at runtime.
- In some cases, a JSP might represent a complete page on a website. In our case, each portlet displayed on a page will be generated by a JSP.
- The most important function of a JSP as the View layer (other than being able to make it look pretty) is that we can communicate between it and the portlet class.

## JSPs AND PORTLETS WORKING TOGETHER

- » The first step in wiring all of this together is to define the location of the `view.jsp` and any additional JSPs in our `@Component`.
- » When the portlet enters the Render phase in View mode, we render our portlet display from the `view.jsp`.
- » Liferay's APIs wire all of this together so that the portlet class can exchange information with the JSP.



## CREATING THE PORTLET VIEW

- » There are a few steps we need to complete to create a View for our portlet:
  1. Create a folder structure for the application JSPs.
  2. Create a JSP which provides some basic functionality.
  3. Create a `doView()` method in our portlet class which will render `view.jsp` when our portlet enters its Render phase.

---

## EXERCISE: CREATING THE FOLDER

» First we'll create the folder structure and file.

1. Right-click on the project root.
2. Choose *New → Folder*.
3. Type resources for the *Folder name*.
4. Create a subfolder of resources named `html` next.



---

## EXERCISE: CREATING THE FILE

1. Right-click on the `html` folder.
2. Choose *New → File*.
3. Type `view.jsp` for the *File name*.
4. Insert snippet `o1-view-jsp` into the new file.
5. Save the file.

---

## EXERCISE: KNOWLEDGE IS POWER

- Before we modify our *HelloPortlet.java* we have to let our project know about the *resources* folder we just created.
- We'll do this in the the *bnd.bnd* file.
  1. Open the *bnd.bnd* file of our portlet project.
  2. Click on the Source tab if it doesn't automatically take you there.
  3. Add *-includeressource: resources/* below the *Private-Package: com.liferay.training.hello.portlet*.
- Now that our project knows about the *resources/* folder, we can create the code to render the JSPs in the *resources/* folder.

---

## EXERCISE: DO THE DOVIEW() METHOD

- Now, we'll create the *doView()* method to render the JSP.
- 1. Open *HelloPortlet.java* located in *src/com.liferay.training.hello.portlet*.
- 2. Replace the existing *doView* method with snippet *02-do-view*.
- 3. Type CTRL + SHIFT + O to resolve any import issues.
- 4. Save the file.
- 5. Copy the JAR file in the *generated/* folder - the portlet will regenerate automatically.
- 6. Paste the jar file in the *[LIFERAY-HOME]/deploy* folder to redeploy.

---

## EXERCISE: PORTLET VIEW RENDER

» Now that we have our portlet updated, let's see what it looks like.

1. Open your browser to localhost:8080.
2. Sign in.
3. Open the *Add* menu.
4. Go to *Sample*.
5. Drop *Hello Portlet* on the page.

✓ We now have a portlet which displays a simple View when it renders:

Hello Portlet  
Hello, is it me you're looking for?

WWW.LIFERAY.COM

LIFERAY.

---

## DO YOU DOVIEW()?

- » When the portlet enters the Render phase, it will look for the `doView()` method in the portlet class to provide a view for the portlet.
- » You can also create `doEdit()` and `doHelp()` methods, which would contain relevant code for providing the Edit and Help modes in the standard portlet.
- » In the `doView()` for a portlet using a JSP as its View, like ours, we provide the dispatcher the path to the JSP.

```
getPortletContext().getRequestDispatcher(viewJSP).include(request, response);
```
- » The JSP is then compiled and rendered by the Java servlet engine.

WWW.LIFERAY.COM

LIFERAY.

---

## NEXT STEPS

- Now that we've seen how to create a View layer with a JSP and connect it to our portlet class, what's next?
- Our current setup might work if we just needed to display a static object, like a restaurant menu, but we need to do more than that.
- We need to provide a way for the JSP to retrieve information from the portlet class and for the portlet class to receive information back from the JSP.
- Next, we'll connect our JSP to the portlet as the Controller for our application and make them work together.

Notes:



## THE CONTROLLER LAYER

Copyright ©2016 Liferay, Inc.  
All Rights Reserved.

No material may be reproduced electronically or in print,  
distributed, copied, sold, resold, or otherwise exploited  
for any commercial purpose without express written  
consent of Liferay, Inc.

### THE PREQUEL

- » Bndtools and Declarative Services are tools that simplify module development.
- » Using component annotations, we are able to set the properties of our portlet component, eliminating the need to modify an XML file.
- » In the MVC design pattern, the View layer (or presentation layer) is used as a means for us to display something to the end User.
  - » The View is implemented using JSPs.

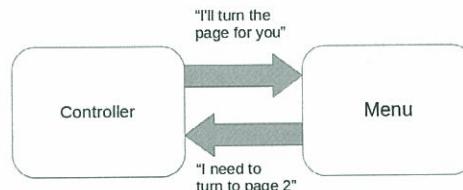
## PLUGGING IN THE CONTROLLER

- Following the MVC design pattern, the Controller can serve two purposes:
  - Controlling what is seen in the View layer (known as page flow)
  - Act as the middle man between the View Layer and Model Layer, allowing the View to pass data to the Model via the Controller and vice versa
- The controller layer is made up of Java classes.
- The methods and logic found in our Java classes are what make the link between the View and Model Layer for our applications.



## READING THE INSTRUCTIONS

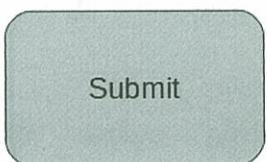
- Suppose that, in our example of the restaurant, you're looking at a huge menu that spans many pages.
- The Controller layer is like the digital hand that turns the pages of the menu.
- The View Layer asks the Controller to help it display another page of the menu by displaying another JSP.
- The Controller, being the good friend it is, gets the right page of the menu to display in the View layer.



---

## PRESS A TO SELECT

- » After you flip through a few menu pages, you find what you want to eat.
- » You tap on the picture of that Chicken and Tomato Risotto you've been craving since last Tuesday, add an appetizer of Mozzarella Sticks, and get ready to submit your selection.
- » You hit the green submit button, and your order is sent! But what was going on behind the scenes?



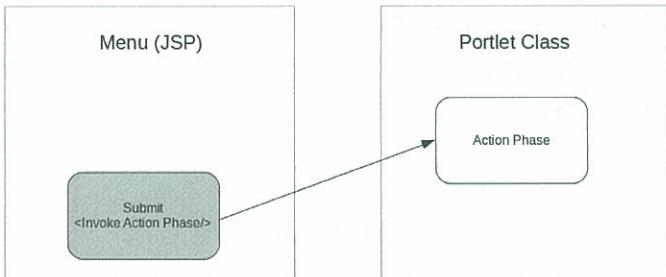
WWW.LIFERAY.COM

LIFERAY.

---

## PRESS START TO PAUSE THE GAME

- » Each page of the menu is implemented using JSPs, which display the text and images of the menu.
- » Tied to the green submit button you just pressed is a tag that will invoke the action phase of the portlet.

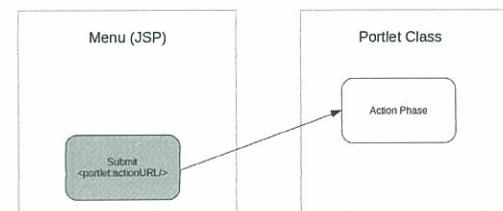


WWW.LIFERAY.COM

LIFERAY.

## PRESS B TO PERFORM AN ACTION

- The action method contains the logic that will take your order form and process it.
- The action method contain the logic that will take our order form and process it accordingly.
- To invoke the action phase of the portlet component from a JSP, we use the tag <portlet:actionURL name="">.



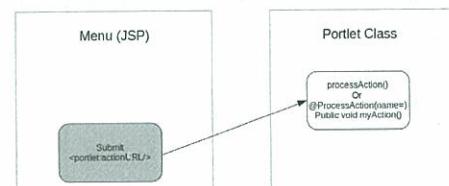
WWW.LIFERAY.COM

LIFERAY.

## PRESS SELECT TO CHOOSE WHICH ACTION

- Once invoked, if there is no name associated with the actionURL, the method called in the portlet will be `processAction()`.
- If there is a name associated with the actionURL, the portlet class there will have an annotation as follows:

```
@processAction(name="insertnamehere")
public void myAction(){ ... }
```
- The action method with the annotation's name that matches with the name in the JSP will be called.



WWW.LIFERAY.COM

LIFERAY.

## THE DIRECTION PAD

- » Ultimately, we're using a tag to invoke the action phase of the portlet component through the JSP.
- » The action phase is one of the phases in the portlet lifecycle.
- » It's through the action phase that we are able to call action methods within our portlet class to do something useful.

## A DIFFERENT WAY TO PLAY

- » Traditionally, the various action methods of the Controller were contained within the portlet class.
- » In the OSGi Container, we have the option of putting the Controller actions into one module and the View and portlet component into another.
- » This helps mitigate the complexity of managing many actions while giving us the ability to change how actions work without affecting other areas.
- » For now, to keep things simple, we'll just include the action methods in the Controller class.

---

## READY PLAYER ONE

- » We're going to build off of the portlet component from the previous section by generating the "Hello" text and displaying it in the JSP.
- » Though the Model Layer will not be involved, we'll see the interaction between the View Layer and the Controller Layer with this simple example.
- » Our example won't implement our action in another module, but that will be covered in a later section.

---

## THE STRATEGY GUIDE

- » We're going to take the following approach:
  - » Create a link tied to the `<actionURL name="">` tag that will invoke the action phase.
  - » The method that is called will take our input and call the service, returning a response that will be stored as a render parameter.
  - » After the action phase is complete, the render phase is initiated, displaying the value of the render parameter in the JSP.

---

## EXERCISE: LEVEL ONE - THE JSP

1. Open the view.jsp.
  2. Add snippet 01-actionURL-tag.
  3. Save the file.
- » The actionURL tag will set the portlet component into its action phase and call the processAction() method in the Controller.  
» The method will process our text and display the correct value in our JSP.

---

## EXERCISE: LEVEL TWO - THE CONTROLLER

1. Open HelloPortlet.java.
  2. Add snippet 02-process-action.
  3. Add snippet 03-service-reference below the doView().
  4. Type Ctrl-Shift-O to organize imports.
  5. Save the file.
- » Here we see the text generated and stored with the render parameter.  
» Remember that, after the action phase of a portlet component, the render phase is invoked.  
» We also added the necessary reference to the previously created service to get the text for us to display.

---

## EXERCISE: LEVEL TWO - SECRET AREA

- » The last snippet we added will have some unresolved imports.
  - » We need to add the Hello Service JAR to the path and then reorganize the imports to add the correct imports.
1. Right-click on the project root in Eclipse.
  2. Choose *Properties*.
  3. Click *Java Build Path*.
  4. Choose *Libraries*.

---

## EXERCISE: A SECRET RESOLVE

1. Click *Add External JARs*.
  2. Find the `hello-api.jar` in the generated folder for the service project.
  3. Click *Open*.
- ✓ The imports will resolve.

---

## EXERCISE: LEVEL THREE - BOSS FIGHT

- » To utilize the action command we need to have the correct imports for the tag libraries that we reference.
1. Open your `bnd.bnd` file.
  2. Click on the *Source* tab.
  3. Insert snippet `o4-taglib-import` into the bottom of your `bnd.bnd`.
  4. Save the file.

---

## EXERCISE: LEVEL FOUR - WARP ZONE DEPLOY

1. Copy the `com.liferay.training.hello.portlet.jar` from the generated folder.
2. Paste the `com.liferay.training.hello.portlet.jar` into the `/deploy/` folder.
3. Replace the previous portlet with the updated application.

---

## EXERCISE: FINAL LEVEL - SAY HELLO!

1. Type "hello" to test the updated application.
2. Click on the *Say it!* link.
  - » In the application, you should see a response like *I don't think you understand*.

---

## REPLAY

- » The Controller Layer is represented by Java classes.
  - » The Controller is responsible for page flow and connecting the View Layer with the Model Layer.
  - » The View Layer triggers the action phase of the portlet component by using the tag `<portlet:actionURL name="">`.
  - » In the Java class, there is a `@ProcessAction` annotation. When you set the `name` element on the annotation:  
`@ProcessAction(name="addAssignment")`
- It matches an `ActionURL` with the same name:
- ```
<portlet:actionURL name="addAssignment" />
```

## Notes:

Notes: [View](#) [Edit](#) [Delete](#) [Import](#)

## Chapter 8

# Debugging Deployment



## DEPENDENCY RESOLUTION

Copyright ©2016 Liferay, Inc.  
All Rights Reserved.

No material may be reproduced electronically or in print,  
distributed, copied, sold, resold, or otherwise exploited  
for any commercial purpose without express written  
consent of Liferay, Inc.

### PART OF YOUR APP

- Modules combine to form apps with complex needs.
- Modules can be added and removed at any time.
- Modules export and import classes from across the OSGi Container.
- Modules provide and require capabilities from other modules.
- The framework installs, starts, and stops modules.
- Modules that declare dependencies on others need to be *resolved*.

---

## LOOK AT THESE MODULES

- The Module lifecycle provides a perfect setting to resolve external dependencies.
- After a module is *installed*, it goes to the *resolver*.
- The *resolver* satisfies dependencies in each module.
- Once dependencies are satisfied, the module is marked as *resolved*.

---

## HMS RESOLVER

- Knowing how our modules find dependencies prevents simple errors.
- Understanding the *resolver* and where it looks makes designing large apps easier.
- Modules declare dependencies through:
  - **Required packages:** These are Java packages that are *necessary* for our module to execute.
  - **Required bundles:** These are modules that provide *necessary* exports for our module.
  - **Required capabilities:** These are Java interfaces with attributes that describe functionality *necessary* for our module to execute.
- The *resolver* works in the *resolve* phase of the lifecycle to match dependencies with what's available in Liferay.

---

## TO PACKAGE OR TO BUNDLE?

- » Declaring dependencies in modules is simple: ask for Java *packages*, other installed *modules*, or predefined *capabilities*.
- » *Capabilities* are a powerful way to express dependencies, but is more complex.
- » Dealing with *packages* is the first way you'll express external dependencies.
- » One modular practice is to ask for *packages* (*Import-Package*).
  - » Doesn't matter which module provides the package
  - » Implementations can be changed at runtime
  - » Maintains separation between implementation and interface
- » Depending on a *module* (*Required-Bundles*) gives access to all of the module's exports.

---

## AN EXPORT/IMPORT BANK

- » The *resolver* spends some of its time matching imports with exports.
- » A module *cannot* start without its dependencies being present.
  - » Prevents simple “class not found” errors
- » This guarantees the availability of classes the developer needs.
  - » Classloading is simpler and predictable.

---

## A DAY IN THE LIFE OF A RESOLVER

- » What's it like to walk in one day to find an inbox full of modules?
- » Well, with the modules installed, the resolver looks to the *import packages* of each module:
  - » The resolver attempts to locate the packages as *export packages* of other modules.
  - » When it finds a match, the resolver *wires* the export module to the import module.
  - » If the resolver can't match a package, the module has an *unmet dependency*.
- » When the resolver finds modules that have *required modules*:
  - » The resolver locates the required modules in Liferay.
    - » When found, the resolver *wires* the exports of the required module to the original module.
  - » If a required module is not found, the module has an *unmet dependency*.

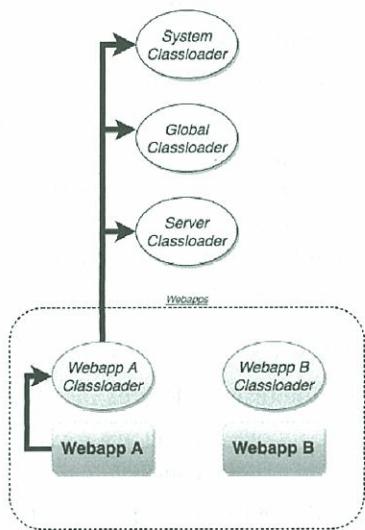
---

## WHAT'S IN A CLASSLOADER?

- » The resolver is a key part of classloading in *Liferay*.
- » The framework uses a very simple classloading architecture.
- » Modules have quick access to the correct classloader for each class.
- » What we know about Java EE classloading is a bit different.

## A TALE OF TWO CLASSLOADING ARCHITECTURES

- » Classloading in a typical J2EE server relies on *hierarchy*.
- » There are multiple levels of classloaders, from the system-level down.
- » Each webapp gets its own classloader, at the bottom of chain.
- » When a class is needed, classloaders *delegate* up the hierarchy until the class is found.

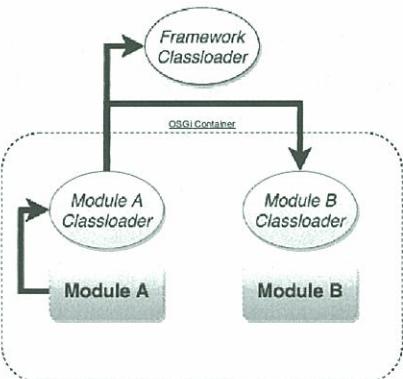


WWW.LIFERAY.COM

LIFERAY.

## A NEW KIND OF CLASSLOADER

- » In the *OSGi Container*, classloading is simple.
- » Each *module* has its own classloader.
- » If a class is needed from a `java.*` package, it's delegated to the parent classloader.
- » Modules have direct access to each others' classloaders to instantiate *imported classes*.
- » There is no long delegation chain.



WWW.LIFERAY.COM

LIFERAY.

## MEETING THE DEPENDENCIES

- The *resolver* wires together export–import bundles, so there is no searching necessary to instantiate classes.
- If a class is available in a module in the OSGi Container, you can *import* it.
- If a class needs to be available in the OSGi Container, you can *export* it.
- If you try to use a class you haven't imported, the *resolver* will fail.
- This prevents some bad code from running.
- The *resolver* also wires together *Capabilities*, so that a bundle doesn't start if it's missing key functionality.

## FULLY CAPABLE

- Modules provide functionality through services and additional OSGi integration.
  - Some of these include *Trackers* and *Extenders* – the OSGi Alliance has great info (<http://enroute.osgi.org/doc/218-patterns.html>).
- **Capabilities** are a robust way to describe *what your module can do*, or *what your module needs*.
- Bundles can declare Java packages they want to reference or share.
- Bundles can also declare the type of functionality the need or provide using *Capabilities*.

---

## WHAT IS A CAPABILITY?

- » Bundles can provide a *Capability*, much like they can export a package.
- » A *Capability* describes functionality you want to provide to the OSGi Container.
- » It uses a simple description with two major pieces:
  - » **Namespace:** a unique identifier, like a package name
  - » **Attributes:** a list of objects that describe the capability
- » What if we build a module that provides an LMS training module?
- » We could provide a *description* of this Capability in the bundle we deploy.
- » This description exists in the Manifest using the header **Provide-Capability**:

```
Provide-Capability: com.liferay.training.module;type:String=LaTeX
```

---

## A SPACE FOR NAMES

- » Each *Capability* starts off with a unique **namespace**:  
`com.liferay.training.module`
- » Attached to the *namespace*, we can identify **attributes**.
- » This is an extremely powerful way to provide an accurate description.
- » Attributes have the form **[name]:[type]=[value]** –  
`type:String=LaTeX`
- » In our example:
  - » the **name** of the *attribute* is `type`
  - » the **type** of the *attribute* is `String`
    - » Always use a Java type: `Long`, `String`, `List<String>`, etc.
  - » the **value** of the *attribute* is `LaTeX`
- » You can have many attributes, separated by a semicolon (`;`).

## DESCRIBING CAPABILITIES

- » We could build out a fuller description of this *Capability*, too:

```
Provide-Capability: com.liferay.training.module;type:String=LaTeX;  
    numsections:Integer=3;  
    sections>List<String>="Awesomeness, Great, Delight"
```

- » This simply describes what your module *can do*, much like an interface (API) can.

- » In fact, this could be visualized the same way:

```
package com.liferay.training.module; //Namespace  
  
public interface Capability {  
    public String type;  
    public Boolean active;  
    public List<String> sections;  
}
```

- » Each module that **provides** this *Capability* provides a value for each attribute.

## REQUIRING FUNCTIONALITY

- » Just like **exporting packages** means that you can **import packages**, **providing Capabilities** means you can **require** them, too.

- » **Requiring** a *Capability* is also done through Manifest headers:

```
Require-Capability: com.liferay.training.module
```

- » Just like you can *describe* what you provide through attributes, you can **filter** based on attributes:

```
Require-Capability: com.liferay.training.module;filter="(type=LaTeX)"
```

- » This filter is an *LDAP filter*, so we can search using the same syntax:

```
Require-Capability: com.liferay.training.module;filter=  
    "(&(type=LaTeX)(active=true))"
```

- » The *resolver* wires these **requirements** against available **capabilities**.

---

## A MASTERFUL EXAMPLE

- » Let's imagine we're in a new Liferay environment.
- » We've installed a module, `com.liferay.training.foo`.
- » The *Foo* module has the following *headers* in the manifest:

```
Import-Package: com.liferay.training.shi  
Require-Capability: com.liferay.training.bake;filter="(oven=bread)"
```

- » *Foo* uses a class, `MasterBaker`, from the *Shi* module.
- » *Foo* also needs to use some functionality called `com.liferay.training.bake`.

---

## NO OVEN

- » Without the *Shi* module installed:
  - » The *resolver* looks for:
    - » The Capability `com.liferay.training.bake` in all the modules.
    - » `com.liferay.training.shi` in all the modules.
  - » When neither is found, the *resolver* marks the *Foo* module: **installed**.
  - » The module *Foo* doesn't resolve.

---

## I FOUND SOMETHING

- » When we install the *Shi* module:
  - » The *resolver* checks the manifest for *Shi*.
  - » It has no dependencies, so the resolver marks the module: **resolved**.
  - » OSGi Container starts the module, and the service component is available.

---

## A CHANGE OF MIND

- » When the new module is installed, the *OSGi Container* checks to see if any modules haven't been resolved.
- » The *Foo* module is picked up, and the *resolver* tries again:
  - » The *resolver* checks the available modules for `com.liferay.training.shi`.
  - » It finds it is the *Shi* module.
  - » The *resolver* creates a *Shi*—*Foo* wire.
    - » *Foo* can now instantiate classes directly through the *Shi* classloader.
  - » There's one more dependency, the *Capability*:
    - » The *resolver* checks the available modules for the *Capability* `com.liferay.training.bake`.
    - » It doesn't find it.
  - » The *Foo* module is not available.

---

## WHERE'S MY OVEN?

- » When we install a new *Oven* module:
  - » The *resolver* checks the manifest for *Oven*.
  - » The new module provides the Capability `com.liferay.training.bake`.
  - » It has no dependencies, so the resolver marks the module: **resolved**.
  - » OSGi Container starts the module, and is available.

---

## A CHANGE OF MIND

- » Again, with the new module installed, the *OSGi Container* checks again to see if any modules haven't been resolved.
- » The *Foo* module is picked up, and the *resolver* tries again:
  - » The *resolver* checks the available modules for the Capability `com.liferay.training.bake`.
  - » It finds the *Oven* module.
  - » The *resolver* checks the **filter**, and all attributes match.
  - » The *resolver* creates a Shi–Oven wire.
    - » The *Foo* module can now use the functionality from the *Oven* module.
  - » The *Foo* module is now available.

## I KNEAD YOU NOW

- *Foo* can now use classes from the *Shi* module.
- When *Foo* instantiates the *MasterBaker* class:
  - *Foo* uses its wiring to *Shi* to *delegate* to *Shi*'s classloader.
  - The new instance is successfully created.
- The object calls `knead()` and behaves as expected.
- There's no class to find, because the resolver has satisfied that dependency at installation!
- When the `bake()` method is run, the Capability provided by the *Oven* method can then act on the *Foo* module.

## SIMPLE MANAGEMENT

- Modules declare their intent to use Java packages with *imported packages* in the manifest.
- Modules can likewise *require modules* to be present, and use their Java packages.
- Modules can declare certain Capabilities that are *required*.
- When the dependencies are satisfied, that means:
  - Any Capabilities requested are available in the container.
  - The classloader for the imported packages is made available to the module.
  - A module can use those classes immediately, delegating to other classloaders.
- Classes can't be used unless the packages are installed and available.
- Modules are *stopped* and marked *unresolved* if any of their dependencies are removed.

---

## TROUBLES AHEAD

- » Issues can arise in two major areas: installation and development.
- » Problems during install:
  - » No installed modules export the needed packages.
  - » A required module is not present.
  - » A required Capability is not present.
- » Problems from development:
  - » Using a class that isn't in an explicitly imported package
  - » Manipulating the classloaders to gain access to classes that aren't accessible through the parent classloader

---

## HERE'S LOOKING FOR YOU, MODULE

- » Good modular development includes declaring dependencies.
- » Only use Java packages that are imported into your module.
- » Try not to look for and instantiate classes you haven't imported at runtime.
- » Use tools that automate declaring dependencies (such as Bnd).
- » Require Capabilities you'll need from other modules that provide them.

Notes:



## TROUBLESHOOTING DEPLOYMENT

Copyright ©2016 Liferay, Inc.  
All Rights Reserved.

No material may be reproduced electronically or in print,  
distributed, copied, sold, resold, or otherwise exploited  
for any commercial purpose without express written  
consent of Liferay, Inc.

## MODULE INSTALLATION

- Modules can be built using a variety of tools (Bndtools or Gradle, for instance).
- Once compiled and packaged as a JAR, modules are installed into the OSGi Container by deploying them to Liferay or by installing them from the Gogo shell.
- Modules installed into the OSGi Container are inspected from the shell.
- With the increased modularity of applications, and considering that the framework keeps unresolved modules from ever being started to avoid runtime errors, deployment is an important event when developing Liferay modules.

## COMMON TROUBLESHOOTING AREAS

- When starting to discern what's going on with your modules, you'll want to consider a few areas:
  - **Module Lifecycle:** Knowing whether or not your bundle is *active* or *installed* can tell you a lot about potential problems.
  - **Required Packages:** When packages you need aren't provided in the container, your modules won't be able to resolve.
  - **DS Components:** If your module contains components you need, it's possible your DS Components aren't active, or encountered problems.
  - **Requirements and Capabilities:** If you require certain capabilities that aren't met by the container, your module may be unable to function.
- We'll walk through a number of these areas, to help you figure out what's going on when something's wrong.

## DEPLOYMENT

- The deployment process introduces the module to the OSGi Container, which takes the module from there and runs it through the module lifecycle until it (hopefully) reaches the *active* state.
- This makes deployment of modules the most likely place to run into errors (and really, that's a better situation than discovering them at runtime!).
- Here, we'll learn about common deployment problems and the tools at your disposal to diagnose and fix them.

---

## TROUBLESHOOTING DEPLOYMENT (I)

- » So, you've successfully built your project (compiled your module, generated a JAR).
- » The next step is to deploy it into the container for installation and management of its lifecycle.
- » You deploy the module (or install it from the shell), and everything appears to work properly! There are no error messages in the log, and it appears that installation was a success!

---

## TROUBLESHOOTING DEPLOYMENT (II)

- » Following your apparently successful installation, you log in to Liferay to test the behavior of your application.
- » Wait, it's not where it's supposed to be. You can't add it to a page. What now? How do you track down the problem?
  - » Your familiarity with the shell will come in handy as you debug deployment issues.

```
lb [module id]
diag [module id]
```
  - » Liferay's log will also help guide you to the problem with your module.

## DEPLOYMENT AND THE MODULE LIFECYCLE

- Because deployment of your plugin is the start of the module lifecycle, most errors you run into will occur during transition through the various lifecycle states:
  - *installed*: Installation into the container
  - *resolved*: Dependency resolution
  - *starting*: Intermediate state while moving from *resolved* to *active*
  - *active*: Ready to use
    - DS Components now activate as well:
    - *activate*
    - *active*
    - *deactivate*

## COMMON DEPLOYMENT PROBLEMS

- You're not responsible for doing anything in your plugins that relates to stopping the module and uninstalling it.
- You'll really have problems in three places most of the time:
  - Resolving dependencies
  - Starting the module
  - Activating DS Components
- There, are, of course, runtime errors as well, but most of these will be the result of faulty code.

---

## FAILURE TO RESOLVE A MODULE (I)

- » A common problem to have is a module not resolving.
- » As you might recall, this is due to an unmet dependency.
- » Remember that a JAR in the OSGi Container has a MANIFEST.MF file which has information that describes the JAR's contents?
- » There are headers in the file that declare the module's dependencies, and inspecting them will be beneficial to debugging dependency resolution issues.
- » The question, then, is how best to inspect these headers?

---

## FAILURE TO RESOLVE A MODULE (II)

- » Where might you look to fix resolution problems? Inspecting the headers as specified in the `bnd.bnd` file would be a good place to start, as the `MANIFEST.MF` is generated from this file.
- » You can also inspect the module's headers from the shell.
- » Additionally, if you're developing in LDS or Liferay IDE, you probably have Liferay running from the IDE, and that means you have the log output at your fingertips.
- » If dependencies are not met, you'll sometimes get a `Could not resolve module...` error in the Liferay log.

`org.osgi.framework.BundleException: Could not resolve module:`

---

## EXERCISE: WHAT'S WRONG HERE?

- » To fix a problem, you need to know there *is* a problem.
- 1. Deploy the `com.liferay.training.deployment.impl` module to the Liferay instance.
- 2. See a [AutoDeployDir:213] Processing... message in the Liferay log.
- » If you look at your running Liferay log, you probably don't see an exception.
- » While that's good news, it's not the whole story.
- » There's nothing necessarily wrong with a module that doesn't resolve, if no other module is trying to access its contents.

---

## UNDERSTANDING RESOLUTION PROBLEMS

- » Since this JAR's packages aren't being imported by another module that's trying to start, you'll get no log errors related to the failure to resolve.
- » If a module tries to start but can't because it requires the packages from your module, then it will throw an exception that's printed in the log.

```
14:42:21,569 WARN [fileinstall-/home/russell/Documents/code/bundles/master/osgi/configs][org.apache.felix.fileinstall:102] Error while starting bundle: file:/path/liferay-portal/osgi/configs/com.liferay.training.web-1.0.0.jar  
org.osgi.framework.BundleException: Could not resolve module:  
com.liferay.training.inventory.web [473]  
Unresolved requirement:  
Import-Package: com.liferay.training.model; version="[1.0.0,2.0.0)"  
at org.eclipse.osgi.container.Module.start(Module.java:429)  
...
```

---

## INSPECTING MODULES: DIAG

- » There's a great shell command for diagnosing resolution issues with your module: `diag`.
- » This command takes one parameter: the module ID you want to diagnose. If you leave this blank, it will diagnose all the modules in Liferay.
- » The `diag` command prints the module name and ID, and the specific dependency resolution problem.
- » If there's no resolution problem, you'll learn that, too.

```
com.liferay.training.service.hello [470]
  No resolution report for the bundle.
```

- » It's important to note that the `diag` command only reports dependency resolution errors that relate to its imports.
- » For more info, inspect the *requirements* and *capabilities* of a module.

---

## INSPECTING CAPABILITIES

- » Since we can explicitly define *Capabilities* and *require* them during *Resolve*, it is useful to be able to check what the *required* and *provided* Capabilities are.
- » To check a module's *provided* Capabilities:  
`inspect cap [capability namespace] [bundle ID]`
- » To check a module's *required* Capabilities:  
`inspect req [capability namespace] [bundle ID]`
- » For instance, bundles with OSGi Services can provide a Capability with the namespace `osgi.service`.
- » We can check that a bundle *provides* OSGi Services:  
`inspect cap osgi.service 470`
- » We can also check if a bundle *requires* OSGi Services:  
`inspect req osgi.service 470`

---

## EXERCISE: INSPECTING DIAG MODULES

1. Type `diag [module id]`.

- The Module ID is evident from the shell if you already ran the `lb` command.

- ✓ Look at that! The `impl` module is missing a required import.

```
com.liferay.training.deployment.impl [466]
Unresolved requirement: Import-Package: com.liferay.training.deployment.api
```

---

## FAILURE TO RESOLVE: CONSEQUENCES

- If your module doesn't resolve, it can't possibly be activated.
- If another module depends on your module's packages, that's a problem.
- For example, recall our efforts to develop a service API and implementation earlier?
- The implementation imports the API.
- What if the API isn't installed at all?
- The implementation depends on the API being active.
- If the API had been installed and active, then was removed, the modules depending on it would need to be stopped.
- They would move back to the *installed* state if their dependency could no longer be met.
- Let's see this in action.

---

## EXERCISE: API INSTALLATION

- » You've already seen Liferay hold a module from being activated if it can't resolve its dependencies.
  - » Let's install the API and see what happens.
1. Deploy the module `com.liferay.training.deployment.api.jar`. It's provided in your exercises folder.
  2. Type `lb` in the Shell.
- ✓ After the `api` module is installed and activated, the dependencies of the `impl` module are resolved, and now they're both *active*.

---

## EXERCISE: TESTING API REMOVAL

- » Now let's uninstall the `api` module and see what happens.
1. Delete `com.liferay.training.deployment.api.jar` from Liferay's `/osgi/modules` folder.
  2. Type `lb` in Gogo Shell.
- ✓ After the `api` module is uninstalled, the dependencies of the other module can no longer be met, so it's stopped and moved back to *installed*.
- » The OSGi Container makes necessary updates to the lifecycle of all the installed modules.

---

## FAILURE TO START

- » Your module might resolve, but still not start.
  - » Why would this be?
  - » One very common reason would be that the module is using a lazy activation policy.
  - » Lazy activation is a feature that's used most often in cases where lots of modules are installed into the runtime at once. For performance reasons, developers might choose not to allow activation until the module is needed.
  - » A module is only activated if its code is called, or if another module requires one of its exports to satisfy its dependencies.

---

## VERIFYING ACTIVATION

- » If your application installs properly, it gets moved through the lifecycle all the way to the *active* state.
- » This means that Liferay knows your app is there, its dependencies were resolved, the module(s) were started, and the application is activated.
- » More specifically, the bundle has been activated, and any Declarative Services Components have been loaded.
- » To check a module's state, start the shell and enter *lb [fragment of the module's name]* (for example, *lb kaleo*). Assuming your module was the last one installed, it will be the last one displayed, and you'll see something like:

```
245|Installed | 10|com.liferay.random.api (0.0.0.201507072329)
```

---

## EXERCISE: FAILURE TO START

1. Deploy the `com.liferay.training.service.hello_1.0.0.jar` module to the Liferay instance.

2. See a message like this in the Liferay log:

```
-22:25:59,622 INFO [com.liferay.portal.kernel.deploy.auto.AutoDeployScanner]-[AutoDeployDir:213] Processing com.liferay.training.service.hello_1.0.0.jar
```

3. Start the shell (`telnet localhost 11311`).

4. Type `lb`.

```
466|Starting | 10|Hello Service (1.0.0)
```

- ✓ The module is *Starting*, which means its dependencies were met, but it couldn't be activated.

---

## INSPECTING MODULES: HEADERS

- » Manual inspection of the `bnd.bnd` file is possible, but it's very useful to be able to see what got generated in the `MANIFEST.MF` file.
- » Use `headers [module id]` to inspect all of the headers.
- » It's a simple command that simply prints the contents of the manifest into the shell for easy inspection and debugging.

```
headers [module id]
```

---

## EXERCISE: FIXING FAILURE TO START PROBLEMS

- So how do you begin to diagnose why your module didn't successfully start?
- A developer might specify a *lazy* activation policy, which could prevent the bundle from starting.
- Liferay tries to start all bundles automatically, so all bundles should start.
- Another reason would be that the module does not contain a component (for example, a JSP fragment).

1. Type *headers [module id]* from the Shell to look at the content of the MANIFEST.MF file.

```
Hello Service (488)
-----
Export-Package = com.liferay.training.service.hello
Bundle-ActivationPolicy = lazy
...
```

WWW.LIFERAY.COM

LIFERAY.

---

## EXERCISE: GET TO WORK, MODULE

- The module has been designed to activate lazily, so it won't start until it's needed.
- You can start the module yourself right from the Shell.

1. Type *start [module id]*.

- If the module shouldn't start lazily, you can delete the entire line from your bnd.bnd file.
- If you don't specify a Bundle Activation Policy, it defaults to eager activation, which means it will activate immediately after resolving.

```
[Hello Service] Starting module...
```

WWW.LIFERAY.COM

LIFERAY.

---

## INSPECTING SERVICES

- » The Gogo Shell is pretty useful for these basic deployment investigations. Can it do anything else?
  - » How about investigating the services registered, used, and available for use in the OSGi Container? Yes!
  - » Check service dependencies and versions? Yes!

```
{com.liferay.portal.kernel.service.BaseLocalService,  
 com.liferay.training.inventory.service.ManufacturerLocalService,  
 com.liferay.portal.kernel.module.framework.service.IdentifiableOSGiService}  
     ={service.id=3985, service.bundleid=469, service.scope=singleton}  
"Registered by bundle:" com.liferay.training.svc_1.0.0.201603072158 [469]  
"Bundles using service"  
     com.liferay.training.inventory.api_1.0.0.201603041743 [468]
```

### Notes:



## DISCOVERING SERVICES

Copyright ©2016 Liferay, Inc.  
All Rights Reserved.

No material may be reproduced electronically or in print,  
distributed, copied, sold, resold, or otherwise exploited  
for any commercial purpose without express written  
consent of Liferay, Inc.

### OVERVIEW

- Liferay allows modules to publish services.
- Services are also how modules communicate with each other.
- Consuming a service in Liferay is as simple as asking the OSGi Container for the appropriate service.
- Let's take a look at how you can examine services, and how modules use them.

---

## WHEN DO I NEED TO EXAMINE SERVICES?

- » There are a few reasons why you may need to examine the services available within Liferay:
  - » To discover the services available for your modules to consume
  - » If a module is publishing a service no one is using
  - » If a module is trying to consume a service that isn't currently available
  - » If two modules are publishing different implementations of the same service

---

## WHAT'S AVAILABLE TO ME?

- » Services can be inspected and discovered live using the shell.
- » Some of the core commands you'll use for interacting with services include:
  - » services
  - » dm
- » We can also combine features of the shell to retrieve and call services:
  - » Dynamic variables
  - » Calling methods on objects
  - » Access to the bundleContext object
- » Each one of these can be used to help discover running services and diagnose problems live.

---

## EXERCISE: LISTING SERVICES

- » If a module publishes a service and you'd like to verify the service is available in Liferay, we can use services:

1. Connect to the shell using telnet localhost 11311.

2. Run the services command to list all of the available services:

```
{org.eclipse.osgi.framework.log.FrameworkLog}={service.ranking=2147483647,  
service.pid=0.org.eclipse.osgi.internal.log.EquinoxLogFactory,  
service.vendor=Eclipse.org - Equinox, service.id=3, service.bundleid=0,  
service.scope=bundle}  
    "Registered by bundle:" org.eclipse.osgi_3.10.200.v20150831-0856 [0]  
    "No bundles using service."  
{org.eclipse.osgi.service.datalocation.Location}={service.ranking=2147483647,  
type=osgi.user.area, service.pid=  
0.org.eclipse.osgi.internal.location.BasicLocation,  
service.vendor=Eclipse.org - Equinox, service.id=5, service.bundleid=0,  
service.scope=singleton}  
    "Registered by bundle:" org.eclipse.osgi_3.10.200.v20150831-0856 [0]  
    "No bundles using service."
```

---

## EXERCISE: NARROWING THE SEARCH

- » If we know the fully-qualified name of the service, we can narrow the results:

1. Run the command services

```
com.liferay.training.service.hello.HelloService to find our  
Hello service:
```

```
{com.liferay.training.service.hello.HelloService}={service.id=9992,  
service.bundleid=454, service.scope=singleton}  
    "Registered by bundle:" com.liferay.training.service.hello_1.2.0 [454]  
    "No bundles using service."
```

- » You can also use se for shorthand:

```
se com.liferay.training.service.hello.HelloService
```

---

## WHAT WE SEE

- » This shows us:
  - » The fully qualified name of the service  
(com.liferay.training.service.hello.HelloService)
  - » The module that provides the service (454)
  - » Which modules are using this service (No bundles using service.)

---

## EXERCISE: SEARCH THE UNKNOWN

- » If we don't know the full package path for the service, we can use wildcards for the name:
- ✓ Run the command `se *.HelloService`:  
`{com.liferay.training.service.hello.HelloService}={service.id=9992,  
service.bundleid=454, service.scope=singleton}`
- » This shows us:
  - » the fully qualified name of the service  
(com.liferay.training.service.hello.HelloService)
  - » the module that provides the service (454)

---

## FILTERING RESULTS

- In many shell commands, we can narrow the results by adding a *search filter*.
- Search filters in the shell have the same syntax as **LDAP search filters**.
- A basic *expression* is surround by parentheses ( ): (property=value)
- Searching for something with a **bundleid** of 121 looks like:  
(bundleid=121)
- If we know which module (bundleid) provides a service (service):  
(service.bundleid=121)

---

## EXERCISE: VIEWING SERVICES BY MODULE

- Let's limit our services search to our specific module:
1. Type **lb** in the Shell to locate our *Hello Service* module:  
454|Active | 10|hello-service (1.0.0.201607281731)
  2. Type **se** to filter the published services to match our module (454):  
se "(service.bundleid=454)"
  3. See what services the module provides:  
{com.liferay.training.service.hello.HelloService}={service.id=9992,...}  
"Registered by bundle:" com.liferay.training.service.hello\_1.2.0 [454]  
"Bundles using service"  
hello-client\_1.0.0.201604111104 [522]
- You can also use the command **b 454** to view all capabilities of the bundle.

---

## EXERCISE: INSPECTING MODULE SERVICES

- » There is a simpler way to *discover* what services your module has to offer, using the `inspect` command:

1. Type `lb` to find your module number:

```
454|Active | 10|hello-service (1.0.0.201607281731)
```

2. Run the `inspect` command to list all services published in this module:

```
inspect capability service 454
```

3. See this module providing the `HelloService` service:

```
com.liferay.training.service.hello_1.2.0 [454] provides:
```

```
-----  
service; com.liferay.training.service.hello.HelloService with properties:  
  service.id = 9992  
  service.bundleid = 454  
  service.scope = singleton
```

---

## MY MODULE CAN'T FIND ITS SERVICE

- » If your module cannot be resolved, one of the first things to check is whether it's missing a dependency.

- » Gogo shell includes a dependency manager, which is meant to help us resolve situations like this.

- » The command to do this in the Gogo shell is `dm wtf` ("Where's the Failure"). Running this will give you the following output:

```
2 missing dependencies found.
```

```
-----  
The following service(s) are missing:
```

```
* com.liferay.training.helloworld is not found in the service registry
```

- » This command allows you to discover the missing dependencies that are preventing your application from loading.

---

## TESTING THE SERVICE

- » The shell is powerful enough to use to directly invoke service calls.
- » Using the shell in this advanced way requires:
  - » Variables
  - » Method calls
  - » Basic API calls
- » These features can be used and combined in limitless ways for live testing and debugging.

---

## STORING VALUE IN THE SHELL

- » The fundamental need for *variables* is to store values.
- » A very familiar assignment syntax is used:

```
variable = value
```
- » This can be done with any *value* of any type:

```
g! hello="Hello, world!"
```
- » Variables are referenced using \$: \$hello
- » We can test the value of a variable using echo:

```
g! echo $hello
Hello,world!
```

---

## CALLING ALL FUNCTIONS

- » The shell allows us make method calls.
  - » The most basic method calls are the shell commands, like `lb`, `bundle`, or `headers`.
- » The full notation for function call is an *expression*: (`function` `parameter`).
- » For example, using the function `bundle` to retrieve the module for `HelloService` (454):  
`(bundle 454)`
- » In a simple expression like this, the parentheses (( )) can be omitted:  
`bundle 454.`
- » Functions can have *multiple parameters*.

---

## FULLY OBJECTIVE

- » Functions are *objects*, and the object's *methods* are invoked by calling the *name* of the method on the object:  
`(foo get 454)`
- » The *object* returned by `foo` has a *method* called `get`, which takes a *parameter* (454).
- » Functional expressions can be *recursive*:  
`(echo (foo get 454))`
- » Here, we're using an *expression* (`bundles get 454`) to use as the *parameter* of another function (`echo`).

## MODULES, IN CONTEXT

- » With the full object-based notation at our disposal, we have the ability to use some APIs to interact with our services.
- » The *OSGi Container* provides a basic API for dealing with services.
- » Modules that register their services with the *service registry* provide:
  - » a fully-qualified *name* for the service
  - » an implementation of the service
- » Modules that *ask* for a service get a *reference* to the service.
- » The service reference is used to *locate* an implementation to use.
- » Two important methods available to us for services:
  - » `serviceReference`: returns a `ServiceReference` object given the fully-qualified name of the service
  - » `service`: returns an instance of the service object based on a given `ServiceReference`

## EXERCISE: CALLING COLLECT

1. Type the following to test the service reference for the *Hello Service*:

```
serviceReference "com.liferay.training.service.hello.HelloService"
```

2. See that the service has a valid reference:

```
UsingBundles null
PropertyKeys [objectClass, service.id, service.bundleid, service.scope]
Registration {com.liferay.training.service.hello.HelloService}=
    {service.id=9992, service.bundleid=454, service.scope=singleton}
Bundle 454|Active | 10|com.liferay.training.service.hello (1.2.0)
```

---

## EXERCISE: GETTING A REFERRAL

1. Save the service reference in a variable:

```
helloReference=(serviceReference  
"com.liferay.training.service.hello.HelloService")
```

2. See that the reference variable contains the reference:

```
echo $helloReference
```

✓ The output should match the serviceReference call you made earlier.

---

## EXERCISE: HOLDING ON TO THE SERVICE

1. Type the following to request the service object from the container:

```
service $helloReference
```

2. See that the service implementation is set:

```
com.liferay.training.service.hello.impl.HelloServiceImpl@168f0d4f
```

3. Save the service object in a variable for easy calling:

```
helloService=(service $helloReference)
```

4. We now have an instance of the service and are ready to use it!

---

## EXERCISE: SAYING HELLO

1. Type the following to call one of the service methods using our service object:

```
$helloService say
```

2. See that the output is what we expect from our service:

```
g! $helloService say  
Hello...
```

3. Type the following to verify the output of other methods:

```
g! $helloService say hello  
It's me...
```

---

## MULTIPLE MODULES PUBLISHING THE SAME SERVICE

- Any module can publish an implementation of a service.
- Multiple modules can publish the same service.
- Which service implementation is in use depends on:
  - Load time (first-loaded services generally take priority)
  - Module settings (modules can elect implementations based on criteria)
- It may be necessary to verify which implementation your module is using.

## INSPECTING SERVICE IMPLEMENTATIONS

- By using the services command, we can see which bundles provide an implementation:

```
services com.liferay.training.service.hello.HelloService
```

- We can then verify multiple modules are providing the service:

```
{com.liferay.training.service.hello.HelloService}={service.id=9993,  
service.bundleid=454, service.scope=singleton}  
"Registered by bundle:" com.liferay.training.service.hello_1.2.0 [454]  
"No bundles using service."  
{com.liferay.training.service.hello.HelloService}={service.id=9994,  
service.bundleid=456, service.scope=singleton}  
"Registered by bundle:" com.liferay.training.service.redux_1.0.0 [456]  
"No bundles using service."
```

## SERVICES IN USE

- When another module uses the service, we can see which one takes effect:

```
{com.liferay.training.service.hello.HelloService}={service.id=9993,  
service.bundleid=454, service.scope=singleton}  
"Registered by bundle:" com.liferay.training.service.hello_1.2.0 [454]  
"Bundles using service"  
org.apache.felix.gogo.runtime_0.10.0 [11]  
{com.liferay.training.service.hello.HelloService}={service.id=9994,  
service.bundleid=456, service.scope=singleton}  
"Registered by bundle:" com.liferay.training.service.redux_1.0.0 [456]  
"No bundles using service."
```

## STOPPING A SERVICE IMPLEMENTATION

- » We can force the modules to use one of the implementations by issuing a stop to the module with the active implementation:

```
g! stop 454  
[Hello Service] Stopping module...
```

- » If we inspect the available services, we can see that the old implementation is gone:

```
{com.liferay.training.service.hello.HelloService}={service.id=9994,  
service.bundleid=456, service.scope=singleton}  
"Registered by bundle:" com.liferay.training.service.redux_1.0.0 [456]  
"No bundles using service."
```

## TAKING EFFECT

- » We may need to refresh other modules using the service in order to force the implementation change:

```
{com.liferay.training.service.hello.HelloService}={service.id=9994,  
service.bundleid=456, service.scope=singleton}  
"Registered by bundle:" com.liferay.training.service.redux_1.0.0 [456]  
"Bundles using service"  
org.apache.felix.gogo.runtime_0.10.0 [11]
```

---

## USING A SERVICE IMPLEMENTATION

- Once we've forced the other implementation to take effect, any module asking for the service after that point will get the *new* implementation:

```
g! $hService say  
Allo...  
g! $hService say hello  
C'est moi...
```

---

## SWITCHING IMPLEMENTATIONS (I)

- If we re-activate the old implementation, it likewise does not immediately take effect:

```
{com.liferay.training.service.hello.HelloService}={service.id=9994,  
service.bundleid=456, service.scope=singleton}  
"Registered by bundle:" com.liferay.training.service.redux_1.0.0 [456]  
"Bundles using service"  
org.apache.felix.gogo.runtime_0.10.0 [11]  
{com.liferay.training.service.hello.HelloService}={service.id=9995,  
service.bundleid=454, service.scope=singleton}  
"Registered by bundle:" com.liferay.training.service.hello_1.2.0 [454]  
"No bundles using service."
```

## SWITCHING IMPLEMENTATIONS (II)

- » But by issuing `stop` to the currently active implementation, and a refresh to modules we want to force the change on:

```
{com.liferay.training.service.hello.HelloService}={service.id=9995,  
service.bundleid=454, service.scope=singleton}  
    "Registered by bundle:" com.liferay.training.service.hello_1.2.0 [454]  
    "Bundles using service"  
        org.apache.felix.gogo.runtime_0.10.0 [11]  
{com.liferay.training.service.hello.HelloService}={service.id=9996,  
service.bundleid=456, service.scope=singleton}  
    "Registered by bundle:" com.liferay.training.service.redux_1.0.0 [456]  
    "No bundles using service."
```

- » Not only can we check the implementation in use, we can switch between them and turn them on and off.

## IT'S ALL IN THE SHELL

- » Determining what services are available and used by our modules helps with troubleshooting.
- » Common commands for diagnosing missing services:
  - » `diag`
  - » `dm`
  - » `dm wtf`
  - » `services`
- » Additionally, we can test the implementation of a service with common API functions:
  - » `serviceReference`
  - » `service`
- » Using the full syntax of the shell, including *variables* and *function calls*, we can retrieve and use services at runtime.

**Notes:**

1. The following notes apply to all the tables in this section.

2. The data presented in the tables are based on the following definitions of variables:

3. The data presented in the tables are based on the following definitions of variables:

4. The data presented in the tables are based on the following definitions of variables:

5. The data presented in the tables are based on the following definitions of variables:

6. The data presented in the tables are based on the following definitions of variables:

7. The data presented in the tables are based on the following definitions of variables:

8. The data presented in the tables are based on the following definitions of variables:

9. The data presented in the tables are based on the following definitions of variables:

10. The data presented in the tables are based on the following definitions of variables:

11. The data presented in the tables are based on the following definitions of variables:

12. The data presented in the tables are based on the following definitions of variables:

13. The data presented in the tables are based on the following definitions of variables:

14. The data presented in the tables are based on the following definitions of variables:

15. The data presented in the tables are based on the following definitions of variables:

16. The data presented in the tables are based on the following definitions of variables:

17. The data presented in the tables are based on the following definitions of variables:

18. The data presented in the tables are based on the following definitions of variables:

19. The data presented in the tables are based on the following definitions of variables:

20. The data presented in the tables are based on the following definitions of variables:

21. The data presented in the tables are based on the following definitions of variables:

## Chapter 9

# Real World Application



## PROJECT OVERVIEW

Copyright ©2016 Liferay, Inc.  
All Rights Reserved.

No material may be reproduced electronically or in print,  
distributed, copied, sold, resold, or otherwise exploited  
for any commercial purpose without express written  
consent of Liferay, Inc.

### IT'S APP TIME!

- Liferay hosts an OSGi Container where you deploy your plugins.
- Modules hold your code and are the basic unit of deployment. They're the magic carpets that carry your code to the OSGi Container.
- Components publish services.
- Portlet Components implement the Portlet API and are used to give your app a Controller and View.

## THE S.P.A.C.E. PROBLEM

- » The Space Program Academy of Continuing Education (S.P.A.C.E.) is missing a vital app for day-to-day operations.
- » Instructors lead virtual classes with students in Liferay.
- » For every class, instructors assign work for the students to complete.
- » The students submit this work to be graded by the instructor.
- » Right now, there aren't any apps that allow instructors to create assignments and grade them.

## CLASSY REQUIREMENTS

- » **Audience:** Instructors for S.P.A.C.E.
- » **Problem:** Instructors need a simple way to manage assignments.
- » **Features:** The app will need to allow instructors to:
  - » Create assignments
  - » Modify assignment details
  - » Remove unwanted assignments
  - » Check on which students have submitted an assignment
  - » Grade assignments
- » Students may also need to:
  - » Check the details of an assignment
  - » See when an assignment is due
  - » See what new assignments are posted
  - » Search for assignments in the class

---

## THE LIFE OF A PROFESSOR

- » What we know about the instructors of S.P.A.C.E. classes:
  - » Instructors create classes, which are implemented as Sites.
  - » Instructors add students to the class Site, and students become Site members of that Site.
  - » Students submit the work for the assignment as a document.

---

## APPLIED REALITY

- » We have enough information to determine what kind of basic app we need.
- » Based on the simple requirements, our app will need to deal with information about *assignments*.
- » In addition to providing an interface and displaying information, the app will also need to add data to the platform.
- » In short, the app needs to:
  - » Persist and retrieve data on assignments
  - » Provide an interface for dealing with assignment data
  - » Control access to assignment data
  - » Provide a way to search for assignment data

---

## DESIGN APPROACH

- » We'll build the app based on a basic MVC pattern.
- » We'll use the OSGi Container to separate the components of our app:
  - » **The Model** will need to deal with data related to assignments and student submissions for those assignments.
  - » **The View** will deal with displaying the assignment and submission information and provide interfaces for editing and adding assignment information.
  - » **The Controller** will handle getting the information from the View to the Model, managing page flow, and acting as the app container.
- » Implementing each of the layers will allow us to separate concerns pretty easily.

---

## WHAT'S IN A MODEL?

- » Our *Model* layer will control most of the data concerns of our app.
- » We'll need to deal with two distinct forms of data:
  - » Assignments
  - » Student Submissions
- » Our Model will also need to provide some way to allow the app to enable User searching of the data.
- » Additionally, we'll need to retrieve data from other areas of the platform (such as student names, etc.).
- » Since we want to achieve modular separation of concerns, we'll implement the model layer as a *service*.

---

## A VIEW TO ASSIGN

- » The *View* layer will be responsible for providing all of the interaction between Users and data.
- » We'll need to provide easy-to-use interfaces for the average User.
- » There needs to be a distinct View for:
  - » Displaying assignments
  - » Adding or editing assignments
  - » Viewing assignment details and student submissions
- » So the View will need to *display* the above Views and provide the needed *controls* to allow for interaction with the underlying features.
- » As is common, we'll be able to use technology such as JSPs to implement the View with the Controller.

---

## CONTROLLING THE OUTCOME

- » The *controller* of our app needs to provide the necessary glue between the Model and the View.
- » We want this app to:
  - » Display each of the Views
  - » Switch easily between the Views
  - » Provide assignment data to the Views, depending on the View
  - » Accept User input from the View and hand it to the Model to update and add new data
- » The Controller is also the main entry point for the app.
- » Liferay's basic Controller is the *portlet* — we can create a portlet component to encapsulate our Controller.

## A MODULE FOR EVERY NEED

- » We're able to separate the components of our app into distinct modules pretty easily.
- » As is common in other apps, we'll use different modules for the *Model* and *Controller* layers.
- » We'll separate the functional components of the app into distinct modules:
  - » An *API module* for our OSGi service
  - » The *Service module* for the implementation of our OSGi service
  - » The *Web module* for the Controller and View – the portlet component

## THE API MODULE

- » The API module of our application is very important.
- » It provides the interface to our Service layer that can be consumed (in the Web module, for instance).
- » It also provides the interface to our *Model layer* in the MVC pattern.

---

## THE SERVICE MODULE

- » The Service module of our application contains the implementations for the services in the API.
- » This is where most of the logic concerning our Model layer will go.
- » We'll also need to be concerned with retrieval and persistence of data in this module.

---

## THE WEB MODULE

- » The Web module of our application is where the *View* and *Controller* layers of the MVC app are implemented.
- » Here's where we'll include the *portlet component*, as well as all of the JSPs and any other UI-related features for our app.
- » We'll also need to interpret form data and retrieve model data from the service for display and formatting.

---

## PROJECT LAUNCHPAD

- » Given the basic app requirements, we've broken down the high-level structure that we're going to implement.
- » We've modularized concerns, which will also simplify development.
- » We can now put this basic plan into action, where we can:
  - » Create our app project
  - » Build a basic Controller
  - » Build out a Model layer in an OSGi service
  - » Build out Views to accommodate what we want as listed above
- » We'll also explore how to use some of Liferay's unique features and how to leverage existing tools that can help reduce our workload.

Notes:



## CREATING THE PORTLET

Copyright ©2016 Liferay, Inc.  
All Rights Reserved.

No material may be reproduced electronically or in print,  
distributed, copied, sold, resold, or otherwise exploited  
for any commercial purpose without express written  
consent of Liferay, Inc.

## APPLICATION DESIGN

- It's time to develop our multi-module application for the S.P.A.C.E. site.
- We decided to implement a *Model-View-Controller* (MVC) pattern.
- You also learned about breaking up the app into three distinct submodules:
  - The module holding the View and Controller layers, as well as additional resources: the *Web* module.
  - The module that holds the API (interfaces) for the model layer: the *API* module.
  - The module that holds the service layer and the implementation of the model: the *Service* module.

## DEVELOPMENT TOOLS

- » A wide variety of tools can be used to write apps:
  - » Standard Java EE development environment
  - » Java IDE like Eclipse or IntelliJ
  - » Bnd and Bndtools
- » Additional tools can be used to simplify development further, such as:
  - » Maven
  - » Ant + Ivy
  - » Gradle
- » We'll want to use some more substantial tools to make developing this big app easier.

## LIFERAY'S DEVELOPMENT ENVIRONMENT

- » When setting up our basic development environment, we set up:
  - » Liferay IDE with Eclipse
  - » A Liferay Workspace with a Liferay DXP Digital Enterprise bundle
- » These are part of the full development environment we'll take advantage of.
- » Liferay IDE plugins provide a number of enhancements and shortcuts for building and installing apps on Liferay easy.
- » Liferay Workspace provides a place for us to manage large projects and build easily.
- » What does the full suite of tools look like?

---

## THE BUILD TOOLS – GRADLE

- » Creating larger, more complex projects means we need to have a build management system in place.
- » We can already use Bnd for handling the build of modules easier, but it's still not streamlined for Liferay.
- » *Gradle* is a build and dependency management system for Java.
- » It is both simple and flexible, making it a powerful tool for us to use.
- » Gradle projects have a default project layout, which we can take advantage of to simplify our build files.
- » Plugins can be applied to Gradle projects to do additional processing – for instance, we can use a plugin to leverage Bnd in a Gradle build.
- » This fits in perfectly with a larger development environment.
- » You can find more information on Gradle on the project's website:
  - » <http://gradle.org/>

---

## LIFERAY BLADE

- » Developing modules involves reusing a lot of the same configuration.
- » To make it easier, as well as providing templates for different apps and modules, Liferay created a collection of skeleton projects.
- » These projects make up BLADE (Bootstrap Liferay Advanced Development Environments).
  - » BLADE can be found on Github:  
<https://github.com/liferay/liferay-blade-samples/>
- » The BLADE projects provide an easy way to get started on a new module or set of modules: copy and paste, then modify the details and add your own code.
- » BLADE projects demonstrate development on Liferay, and can be used in any IDE you wish.
- » Projects come in different flavors, whether you want to use Bnd, Bnd and Gradle, or the full set with Bnd, Gradle and the Liferay Gradle plugins.

## LIFERAY BLADE CLI

- » BLADE is incredibly useful for creating modules, but could be streamlined a bit.
- » Liferay also provides an interactive, automated interface into creating projects from BLADE.
- » *blade CLI* is a project compatible with JPM4J that provides:
  - » Access to the BLADE projects from the command line
  - » Creation commands to replace placeholder values in the templates
  - » Commands for building and installing modules on a Liferay instance
- » More information on *blade CLI*, including instructions to install it, can be found at the Github project:
  - » <https://github.com/liferay/liferay-blade-cli>

WWW.LIFERAY.COM

 LIFERAY.

## LIFERAY WORKSPACES

- » BLADE and blade CLI provide great templates for projects, and project stubs.
- » It's not a full dev solution, but fits into existing structures well.
- » A completely configured development environment needs more than just some project templates and build tools.
- » *Liferay Workspace* is a simple development environment built on BLADE, blade CLI and Gradle.
- » Liferay Workspace provides a project structure to follow that's simple, and can be used as the base of a source code repository.
- » In addition to the project structure and tools, Liferay Workspace provides automation for executing Gradle, installing and using a Liferay bundle, and build large groups of projects.
- » Liferay IDE provides a front end in Eclipse for Liferay Workspace, including blade CLI.

WWW.LIFERAY.COM

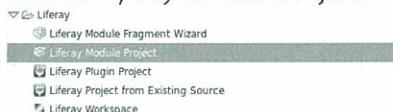
 LIFERAY.

## PROJECT TEMPLATES

- » We've been tasked with creating a Gradebook application for the academy, S.P.A.C.E.
- » We've already created a Liferay Workspace, but haven't used it yet.
- » To create our app, we know we'll need a service *and* a portlet component.
- » The available templates we can base our project on include:
  - » **Activator:** this is a simple module that contains a Bundle Activator.
  - » **Service:** a module with a service component, for publishing OSGi services.
  - » **Portlet:** a module with a portlet component — this is a basic app.
  - » **Service Builder:** an app based on *Service* and it is including additional Liferay tooling to automate code creation.
- » Since our app needs both a *service* and a *portlet* module, our best fit is the *Service Builder* project.

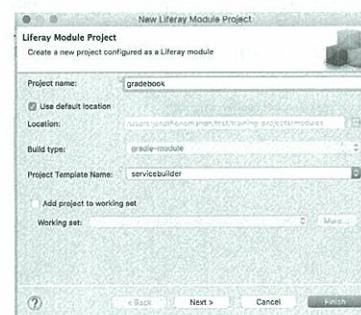
## EXERCISE: CREATING THE GRADEBOOK PROJECT

1. Right-click the Workspace's *modules* folder in your Project Explorer.
2. Click *New* → *Project*.
3. In the New Project Wizard, find the *Liferay* heading.
4. Choose *Liferay Module Project*.



## EXERCISE: PROJECT DETAILS

1. Name the project *gradebook*.
2. See that *Use default location* is checked.
3. Choose *servicebuilder* from the Project Template drop-down.
4. Click *Next*.



WWW.LIFERAY.COM

LIFERAY.

## EXERCISE: CONFIGURE A COMPONENT

» Now you're in the Configure Component Class Window.

1. Name the package `com.liferay.training.space.gradebook`.
2. Click *Finish*.

WWW.LIFERAY.COM

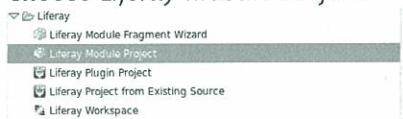
LIFERAY.

---

## EXERCISE: CREATING THE GRADEBOOK WEB MODULE

Now, we'll repeat those steps to create the web module.

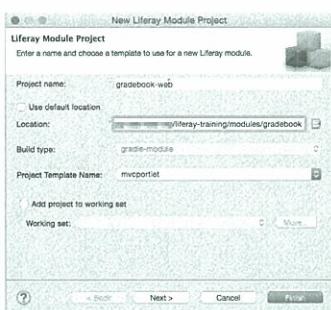
1. Right-click the Workspace's *modules* folder in your Project Explorer.
2. Click *New* → *Project*.
3. Find the *Liferay* heading in the New Project Wizard.
4. Choose *Liferay Module Project*.



---

## EXERCISE: PROJECT DETAILS

1. Name the project *gradebook-web*.
2. See that *Use default location* is unchecked.
3. Choose new location at *modules/gradebook*.
4. Choose *mvcportlet* from the Project Template drop-down.
5. Click *Next*.



---

## EXERCISE: CREATING A COMPONENT

1. Name the Component *GradebookPortlet* now that you're in the Component Creation Window.
2. Name the package `com.liferay.training.space.gradebook`.
3. Click *Finish*.
4. We didn't need to add any *Properties*; the ones we need are generated by default:

```
@Component(  
    immediate = true,  
    property = {  
        "com.liferay.portlet.display-category=category.sample",  
        "javax.portlet.display-name=gradebook Portlet",  
        "javax.portlet.security-role-ref=power-user,user",  
        "javax.portlet.init-param.template-path=/",  
        "javax.portlet.init-param.view-template=/view.jsp",  
        "javax.portlet.resource-bundle=content.Language"  
    },  
    service = Portlet.class)
```

---

## THE GRADEBOOK PROJECT STRUCTURE

- » Liferay IDE created a root folder, *gradebook*, in *liferay-training-workspace/modules* folder.
- » Underneath that, Liferay Blade Tools used templates to generate our API, Service, and Web modules.
- » Use the Package Explorer to find your way around this app via the different modules.



## WHERE'S THE CONTROLLER?



- » You'll quickly see there are three submodules to the *gradebook* root project, just as we described in the overview slides.
- » The *Web module* is where we'll find our controller.
- » Our controller is implemented as a *portlet component* using Declarative Services.
- » Take a look around the *GradebookPortlet* class source and see.

WWW.LIFERAY.COM

LIFERAY

## WHERE'S THE VIEW?



- » Since we'll use JSPs for the view layer, those are included in the *Web module*.
- » Two JSP files were generated through the magic of Liferay IDE:
  - » *init.jsp*: for collecting your imports and declaring taglibs
  - » *view.jsp*: for writing your default view

WWW.LIFERAY.COM

LIFERAY

---

## WHERE'S THE MODEL?



- So, the Controller and View layers live in the Web module.
- We're implementing the model layer using an OSGi service.
- Like the other services we've implemented, it will be split between the *API* and *Service* modules.
- At the moment, these modules are empty.
- Consider this a blank canvas on which to paint beautiful code.

---

## ON THE CUTTING EDGE

- Since we're using Gradle, we have all of our project's compile and build dependencies declared in `build.gradle` files.
- Dependencies are downloaded from various sources, such as public repositories and local files.
- Our project is being built on the cutting edge of Liferay technology.
- With the *OSGi Container*, it's easy to take advantage of the newest stable builds available.
- We'll need to update the dependencies in our modules to use the latest version of the product as we go.

---

## EXERCISE: CHECKING OUT THE LIBRARY

» We'll get started by checking the core library we use in the web module.

1. Open the project folder gradebook-web.

2. Open the file build.gradle.

3. See the following line:

```
compileOnly group: "com.liferay.portal", name: "com.liferay.portal.kernel",  
version: "2.0.0"
```

---

## EXERCISE: CHECKING THE GRADEBOOK-WEB GRADLE

1. Replace the following line:

```
compileOnly group: "org.osgi", name: "org.osgi.compendium", version: "5.0.0" \\
```

with:

```
compileOnly group: "org.osgi", name: "org.osgi.service.component.annotations",  
version : "1.3.0"
```

2. Save the file.

---

## EXERCISE: BUILDING THE GRADEBOOK PROJECT

- » The right panel in the Liferay Workspace perspective of IDE reveals the Gradle Tasks available to you for the projects in your workspace.
  1. Expand the project's drop-down menu to build the Gradebook project's JARs.
  2. Expand the build folder.
  3. Double-click *build*.
- » Each subproject's *build/libs* folder now holds its deployable JAR.

---

## EXERCISE: DEPLOYING THE MODULE

1. Start Liferay if it's not already running.
2. Deploy the Web module.
  - » **Note:** The built JAR is in *gradebook-web/build/libs/*.

```
17:52:43,509 INFO  
[com.liferay.portal.kernel.deploy.auto.AutoDeployScanner] [AutoDeployDir:213]  
Processing gradebook-web-1.0.0.jar
```

3. Sign in.
  4. Add the applications to a page.
- ✓ You can see that the Gradebook app has a default view that came as part of our project template.

## THIS IS JUST THE BEGINNING



WWW.LIFERAY.COM

LIFERAY.

Notes:



## MODEL & PERSISTENCE LAYER

Copyright ©2016 Liferay, Inc.

All Rights Reserved.

No material may be reproduced electronically or in print,  
distributed, copied, sold, resold, or otherwise exploited  
for any commercial purpose without express written  
consent of Liferay, Inc.

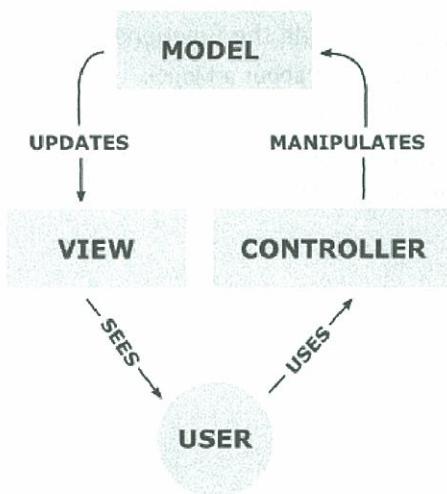
### OVERVIEW

- » A service is a module that provides some functionality.
- » Our service needs to provide the functionality of a Model layer:
  - » Store and retrieve data about a Model.
  - » Implement business logic for manipulating data.
- » Instead of just a simple service API and implementation, we'll also need to handle storing Model data.

## MODELS

- » Views and Controllers are great, but what are you going to show in your View? What will users interact with in your Controller? The Model, of course.
- » In our case, we're working with one core Model that we'll need to create.
- » The Model layer is tightly linked (in concept) with the Persistence layer.
- » We're creating Java Objects with particular attributes for the very reason that we want to store them, access them, retrieve them...
- » Let's talk about the Model's Persistence layer and its Model layer.

## THE MVC TRIUMVIRATE



---

## WHY DO WE NEED A MODEL LAYER?

- » We have our Controller, so we can pass information back and forth between the View layer and the Service layer (which we don't have yet).
- » Our application has to model the data of our core entity, so we need an object with all the necessary attributes.
- » The Model layer encapsulates the data and executes the business logic of manipulating the models.
- » It also needs to whisk the appropriately mutated Model off to be persisted in the database, or, going the other direction, get the Model from the database (to be displayed in our View, for example).
- » How do we implement this functionality? With a set of services, of course.

---

## HOW SHOULD WE MODEL THE DATA?

- » Before actually writing code for our Model, we need to design it.
- » Our basic Model represents an Assignment.
- » Let's decide what data this should have.
- » First, let's agree that an entity should have its own database table.
- » Then, let's decide what fields should go in the table.
- » We also need to decide what Java type each field's value will be.

## WHAT CONSTITUTES AN ASSIGNMENT?

| Field Name   | Field Type | Description                                 |
|--------------|------------|---------------------------------------------|
| assignmentId | long       | Unique key for each entry (pk)              |
| companyId    | long       | Platform instance of assignment             |
| groupId      | long       | Site of assignment                          |
| userId       | long       | Creator of assignment                       |
| name         | String     | Name of assignment                          |
| description  | String     | Description of the assignment               |
| createDate   | Date       | Date the assignment was created             |
| modifiedDate | Date       | Date the assignment was last modified       |
| userName     | String     | Name of the user who created the assignment |

- » Note that the Field Type refers to the Java type. Hibernate maps this value to the appropriate SQL type for us.
- » The companyId and groupId are used to make our assignments scope-able, just like other Liferay content.

## ARE WE MISSING ANYTHING?

- » This seems a plausible set of data for an Assignment.
- » One change we may want to make is to the description field: as a String, it may not be long enough.
- » As we add assignments, we'll also want to track student submissions.
- » For student submissions, we need to know:
  - » Which student submitted the Assignment (the User)
  - » Which Assignment the student submitted
  - » What date the student submitted the Assignment
  - » What grade the instructor wants to give the student
- » The easiest way to track this information is to create *another* Model – Submission.

## WHAT DO WE NEED FOR A SUBMISSION?

| Field Name   | Field Type | Description                                   |
|--------------|------------|-----------------------------------------------|
| submissionId | long       | Unique key for each entry (pk)                |
| companyId    | long       | Platform instance of submission               |
| groupId      | long       | Site of submission                            |
| userId       | long       | Creator of submission                         |
| userName     | String     | Name of the user who submitted the assignment |
| submitDate   | Date       | Date of submission                            |
| grade        | int        | Marks assigned by instructor                  |
| createDate   | Date       | Date the submission was created               |
| modifiedDate | Date       | Date the submission was last modified         |
| assignmentId | long       | Which assignment this submission is for       |

## PERSISTING THE MODEL DATA

- Since we're wanting to persist our data in a database, retrieve it, modify it, put it back, create new records using the Model object, etc., we need a bunch of code for that.
- Database access code is important, but, if possible, not something you want to spend a lot of time writing.
- You should keep it in a separate layer away from your application logic so it can be replaced more easily, if necessary.

---

## DEFINING THE MODEL

- » The first consideration to make is how to Model the data.
- » How many entities, or Model types, do we need?
- » Does any entity depend on another entity?
- » In our example, we will have two entities, Assignments and Submissions.
- » Each database record of a Submission entity will depend on a corresponding record in the Assignment entity's database table.
- » Because of this, we'll include the Primary Key for an Assignment as a column in our Submission database table.
- » Assignments will be independent of Submissions.

---

## MODELING DATA IN THE GRADEBOOK

- » The Assignment and Submission objects, and all their attributes, are part of the Model layer.
- » The Model layer of our app holds all the data and any business rules for manipulating the data.
- » A basic need in our app is to store and retrieve data. The Model layer in our MVC pattern is where the real work happens.
- » We need an Object Model, and we need a Persistence layer to insulate our app from the database transactions.

---

## SERVICE BUILDER OVERVIEW

- *Service Builder* simplifies creating a Model layer for your application.
- A Model generated using Service Builder has both a *Service* and a *Persistence* layer.
- This consists of an OSGi Service and additional Persistence integration with the Liferay platform.
- These services are called *Liferay OSGi Services*.

---

## HOW DOES SERVICE BUILDER WORK?

- Service Builder is a code generation tool provided by Liferay.
- It takes an XML file as input and generates the necessary Service and Persistence layers for you.
- It follows the code separation principles used throughout Liferay.
- The API and implementation are kept separate, in <entity-name>-api and <entity-name>-service modules respectively.
- This simplifies dependency management by allowing us to switch implementations as needed, without impacting any code using our service.

---

## SERVICE VS. PERSISTENCE LAYER

- The Service layer contains the business logic for an application.
- It acts as an intermediate layer between the Controller and the database.
- The Persistence layer handles the data access operations.
- It is responsible for talking to the database and storing your data.

---

## USING SERVICE BUILDER

- Service Builder is your best friend when developing applications for Liferay.
- You define the Model(s) your app needs in XML, and the tool generates the Model layer, the Persistence layer, and the Service layer in one fell swoop.
- You need to update the Service implementation frequently, so you simply re-run Service Builder each time you do, and your Persistence layer is updated as well.
- Once you define the Model, you can focus on the business logic and let Service Builder take care of the rest.
- What if you need to change the Model itself? Just update the XML defining your Models, and re-run Service Builder.

## LIFERAY SERVICE BUILDER

- » Liferay OSGi Services are automatically built by the code generator Service Builder.
- » Service entities are modeled in `service.xml`. Example syntax:

```
<service-builder package-path="com.liferay.training.space.gradebook">
  <namespace>SPACE</namespace>
  <author>William von Richter</author>
  <entity name="Submission" local-service="true" remote-service="false">
    <column name="submissionId" type="long" primary="true" />
    <column name="userId" type="long" />
    <column name="userName" type="String" />
    <column name="createDate" type="Date" />
    <column name="modifiedDate" type="Date" />
    <column name="assignmentId" type="long"></column>
    <column name="studentId" type="long"></column>
    <column name="submitDate" type="Date"></column>
    <column name="comment" type="String"></column>
    <column name="grade" type="int"></column>
  </entity>
</service-builder>
```

## SERVICE BUILDER AND MODULES

- » The work in generating our Model layer (including the Service and Persistence layers) begins in the `gradebook-service` module's `service.xml` file.
- » However, code will be generated in the `gradebook-api` module as well.
- » So, what goes where?
  - » *API module*: All the interfaces and wrappers
  - » *Service module*: All the implementations

---

## EXERCISE: CREATING ASSIGNMENT SERVICE

1. Open `service.xml`, which can be found in the `gradebook-service` module.
2. Replace the existing namespace value `FOO` with `SPACE`.
3. Replace the existing entity with snippet `01-service.xml-Assignment`.
  - » A snippet can be added by dragging its icon into the editor, double-clicking the icon, or right-clicking the icon and selecting *Insert*.

---

## SNIPPETS, BUILD, AND PERSPECTIVE

- » In Liferay IDE, you'll find the *Snippets* tab in the top right if you're using the *Liferay Plugins* perspective.
- » However, we'll also need to manage build options in the *Gradle Tasks* tab which is found in the *Liferay Workspace* perspective.
- » Once you've opened a perspective, it will give you a quick link in the top right, to easily switch.
- » If you can't find the options that you're looking for, make sure that you're in the right perspective.



---

## EXERCISE: CREATING SUBMISSION SERVICE

1. Add in snippet 02 – `service.xml` – Submission after this entity and before the closing `</service-builder>` tag.
2. Save.
3. Go to the *Liferay Workspace* perspective.
4. Open the *gradebook-service* → *build* folder.

---

## EXERCISE: FINISH IT UP

1. Run the *buildService* Gradle task.
2. Right-click on your root project.
3. Choose *Gradle* → *Refresh Gradle Project* to reveal the generated packages classes.

---

## BUILDING SERVICES

- » Service Builder will now create all the interfaces and classes it needs for a Service and a Persistence layer.
- » The interfaces will be created in the `gradebook-api` module.
- » The classes get placed in the `gradebook-service` module.

---

## MAKING SENSE OF SB CLASSES

- » Once you run Service Builder, you're going to get a dump truck-sized load of packages and classes in your *Service* and *API* modules.
- » These classes can be classified into 3 categories, or packages:
  - » Model Package
  - » Persistence Package
  - » Service Package
    - » Remote Services
    - » Local Services

---

## THE MODEL PACKAGE

- » There are several classes comprising the Model Package.
- » To illustrate, look at the classes for the Assignments Model Package:
  - » **AssignmentModel**: Assignment base Model interface for the Assignment Service. This interface and its implementation serve only as a container for the defaults made by Service Builder. Any helper methods and all application logic should be added to **AssignmentImpl**.
  - » **AssignmentModelImpl**: Assignment base Model implementation
  - » **Assignment**: Assignment Model interface which extends **AssignmentModel**
  - » **AssignmentImpl**: Assignment Model implementation. You can use this class to add helper methods and application logic to your Model. Whenever you add custom methods to this class, Service Builder adds corresponding methods to the **Assignment** interface the next time you run it.
  - » **AssignmentBaseImpl**: Extended Model base implementation for the Assignment Service

---

## THE PERSISTENCE PACKAGE

- » Similarly, these are the classes of our Assignment Persistence Layer.
  - » **AssignmentPersistence**: Assignment Persistence interface that defines CRUD methods for the Assignment entity such as `create`, `remove`, `countAll`, `find`, `findAll`, etc.
  - » **AssignmentPersistenceImpl**: Assignment Persistence implementation class that implements **AssignmentPersistence**
  - » **AssignmentUtil**: Assignment Persistence utility class that wraps **AssignmentPersistenceImpl** and provides direct access to the database for CRUD operations. This utility should only be used by the Service layer; in your portlet classes, use the Assignment Service (**AssignmentLocalService**) instead.

## CUSTOMIZING YOUR SERVICE

- » Most of the files Service Builder creates are overridden whenever Services are rebuilt.
- » For this reason, changes to most files will be reverted the next time Services are rebuilt.
- » However, there are a few ways to customize the generated Service to meet your needs.
- » One of the primary methods of doing so is via `portlet-model-hints.xml`.
- » This Hibernate configuration allows manual modification of field properties like type or length.
- » We will be using this file to display a large text field for:
  - » The Assignment description
  - » Comments on the submission

## EXERCISE: MODIFYING YOUR SERVICE

1. Open `portlet-model-hints.xml` found in the `src/main/resources/META-INF` folder of the `gradebook-service` module.
  2. Click on the small tab that says Source to switch to the source view.
  3. Replace the `description` field of the `Assignment` entity with snippet `03-pmh-Assignment`.
  4. Replace the `comment` field of the `Submission` entity with snippet `04-pmh-Submission`.
- ✓ Rebuild Services for your changes to take effect.

## WHAT'S UNDER THE HOOD?

- Other than the code we already talked about, generated by Service Builder, what other cool technologies are involved in our Model and Persistence layers?
- Spring and Hibernate configurations
  - Spring is used for dependency injection.
  - Spring AOP is for database transaction management.
  - Hibernate is for the Persistence framework.
- With SB, developers can take advantage of Dependency Injection (DI), Aspect-Oriented Programming (AOP), and Object-Relational Mapping (ORM) in their projects without having to manually set up a Spring or Hibernate environment or make any configurations.

### Notes:



## SERVICE LAYER

Copyright ©2016 Liferay, Inc.  
All Rights Reserved.

No material may be reproduced electronically or in print,  
distributed, copied, sold, resold, or otherwise exploited  
for any commercial purpose without express written  
consent of Liferay, Inc.

### OVERVIEW

- Service Builder allows us to quickly create a Model layer for our application.
- This Model layer consists of both a Service and a Persistence layer.
- It is split up into two modules, <entity-name>-api and <entity-name>-service.
- The API module contains interfaces, and the Service module contains classes.

---

## CUSTOMIZING YOUR SERVICE

- » Most of the files Service Builder creates are overridden whenever Services are rebuilt.
- » For this reason, changes to most files will be reverted the next time Services are rebuilt.
- » However, there are certain files that are meant to be customized and that aren't overwritten by Service Builder.
- » These allow you to add your own business logic to that provided by Service Builder.
- » We have already looked at `portlet-model-hints.xml`, which allows customization of the generated fields.
- » More extensive changes will require modifying the Service directly.

---

## ADDING BUSINESS LOGIC

- » `<entity-name>LocalServiceImpl` is where your custom business logic can be added.
- » When Service Builder is re-run, any public methods added to this class are then made available to any consumers.
- » This is done by adding them to the APIs in the `<entity-name>-api` module.

---

## FILLING IN THE GAPS

- » So what additional logic do we need to complete the Service?
- » One of the primary functions of the app is to *create* Assignments.
- » When an object is created, it needs to have a primary key.
- » Creating a primary key isn't really the right role for the Controller.
- » We'll take care of generating a primary key in the Service.
- » What other functions can we provide?

---

## FINDING YOUR WAY

- » Service Builder generates an entire Persistence package for us.
- » In our `service.xml`, we can describe not only entity columns, but *finders* as well.
- » A *finder* is a method that executes a query to find the entity.
- » This is an easy way to look up Assignments by Site (`groupId`), for instance.

---

## GETTING FOUND

- » *Finders* are really useful for executing simple queries.
- » They are generated in the Persistence package from your XML.
- » Persistence is only exposed to the Service — once we go outside the *Liferay OSGi Service*, we have no access to Persistence.
- » What if we want to access the finder from the Controller?
- » The answer is to *modify* our Service implementation.
- » We can create a Service method that calls the finder.
- » This way we can expose the finder through our Service.

---

## RENOVATION PLAN

- » So what we're looking to do is:
- » Simplify creating Assignments (and Submissions) by generating a primary key when *adding*.
- » Expose *finders* through the Service.

---

## EXERCISE: ADDING BUSINESS LOGIC

1. Open `com.liferay.training.space.gradebook.service.impl.AssignmentLocalServiceImpl.java` in the `gradebook-service` module. (Use `Ctrl+Shift+R` to simplify finding this file).
2. Replace the body of this file with snippet `01-AssignmentLocalServiceImpl`.
3. Open `SubmissionLocalServiceImpl.java` in the same module.
4. Replace the body of the class with snippet `02-SubmissionLocalServiceImpl`.
5. Type `Ctrl/Command+Shift+O` in both files to organize imports.
6. Choose `java.util.List` when prompted.

---

## EXERCISE: WATCH THE METHODS APPEAR

1. Rebuild Services.
2. See these methods appear in `AssignmentLocalService` and `SubmissionLocalService` in the `gradebook-api` module, respectively.

## WHAT HAVE WE ADDED?

- » We have added in an `addAssignment` and `addSubmission` method, as well as finders for the `Assignment` and `Submission` entities.
- » The add methods use a Service provided by Liferay to automatically generate primary keys for us.
- » This Service is the `CounterLocalService`.
- » By using the Counter Service, we can have independent primary keys per scope, and not rely on any specific database vendor's implementation of key increments.
- » We've also exposed the finders through the Service.
- » Any consumer of the Assignment Service — like the Controller — can now call those finders.

WWW.LIFERAY.COM

LIFERAY.

Notes:



## THE VIEW LAYER

Copyright ©2016 Liferay, Inc.  
All Rights Reserved.

No material may be reproduced electronically or in print,  
distributed, copied, sold, resold, or otherwise exploited  
for any commercial purpose without express written  
consent of Liferay, Inc.

## FINISHING APP

- » The basic app is nearing completion, with a basic Controller and Model layer in place.
- » We used a *portlet component* as the Controller.
- » We used a *Liferay OSGi Service* created by Service Builder to generate a Model layer.
- » We customized the Model layer to provide additional features.
- » To round out the functionality, we need to:
  - » Provide a way to view and edit Assignment data on screen.
  - » Provide Views for displaying and editing Assignments in the app.
  - » Connect the View with the Model through actions on the controller.

## A BASIC CONTROLLER

- So far, we've set up a very simple Controller.
- We have a *portlet component* that was created in the gradebook-web module.
- Instead of using a portlet based on GenericPortlet, we're going to use the Liferay MVC framework.

## LIFERAY MVC

- Why do we want to use Liferay MVC?
  - It's lightweight and doesn't take long to learn.
  - It provides a default implementation of the `doView()` method.
  - It removes the need to write a bunch of boilerplate code, which you'd need if you extended `GenericPortlet`.
  - It accomplishes this by looking for pre-defined parameters when the `init()` method is invoked.

## LIFERAY MVC INITIALIZATION PARAMETERS

- » Here are the initialization parameters Liferay MVC looks for during the portlet initialization phase.
  - » template-path
  - » view-template
  - » edit-template
  - » help-template
  - » about-template
  - » config-template
  - » edit-defaults-template
  - » edit-guest-template
  - » preview-template
  - » print-template
- » In previous versions of Liferay, these parameters were defined in `portlet.xml`.
- » Now they are declared as properties in the *portlet component*. You'll see this in action as we develop our application.

## LIFERAY PORTLET INITIALIZATION PARAMETER

- » Through its extension of `LiferayPortlet`, Liferay's MVC implementation leverages another important initialization parameter.

```
addProcessActionSuccessMessage = GetterUtil.getBoolean(
    getInitParameter("add-process-action-success-action"), true);
```
- » The `add-process-action-success-action` initialization parameter is set to `true` by default.
- » When the `processAction` method completes without error, the application displays a "success" message.

---

## LIFERAY MVC PORTLET: RENDER PHASE

- During the render phase of the portlet lifecycle, the Liferay MVC Portlet also looks for a specific render parameter named `mvcPath`.
- The `mvcPath` value should be the path of a valid JSP page.  
`mvcPath = /gradebook/edit_gradebook.jsp`
- If `mvcPath` is defined within the `renderRequest`, the portlet dispatches to the `mvcPath` directly.
- The included `com.liferay.portal.kernel.bridges.mvc.MVCPortlet` class provides all the portlet functionality you need, freeing you to spend time developing your application's View layer.
- If you need a more robust way to handle rendering multiple JSPs, there's another option.

---

## RENDER WITH CLASS

- Instead of passing `mvcPath` on the URL, you can define special *render paths*:
  - Set a parameter on the URL called `mvcRenderCommandName`.
  - Create a class with the `--RenderCommand` suffix
  - Use the following Component layout for each `RenderCommand` class:

```
@Component()  
    immediate = true,  
    property = {  
        "javax.portlet.name=" + ExportImportPortletKeys.EXPORT_IMPORT,  
        "mvc.command.name=publishLayouts"  
    },  
    service = MVCRenderCommand.class  
}
```
  - Each class must implement the `MVCRenderCommand` interface, which specifies a `render` method.

---

## ACTING WITHIN THE PORTLET

- » If your portlet requires the action phase of the portlet lifecycle (and most portlets do), the Liferay MVC Portlet provides two options.
- » The first, used for most simple cases, requires you to create your own class that extends the `MVCPortlet` and adds your action methods to that class.
- » The only requirement is that the name of the method matches the `name` attribute of the `ActionURL` you create.

---

## MULTIPLE ACTION CLASSES

- » This is used for larger, more complex portlet applications. You can break out your actions into their own classes. `MVCPortlet` invokes the appropriate action without the need for complex xml-based configuration files.
- » Name each class with the `--ActionCommand` suffix
- » Use the following Component layout for each `ActionCommand` class:

```
@Component()  
    immediate = true,  
    property = {  
        "javax.portlet.name=" + ExportImportPortletKeys.EXPORT_IMPORT,  
        "mvc.command.name=publishLayouts"  
    },  
    service = MVCActionCommand.class  
}
```
- » Each class must implement the `MVCActionCommand` interface, which specifies a `processCommand` method.

## LIFERAY MVC PORTLET FEATURES

- Liferay MVC can handle rendering different JSPs by setting a parameter, `mvcPath`, to point to the target JSP.
- The `MVCPortlet` framework can also easily handle portlets with many different actions by using the `name` attribute in the Action URL.
- Annotations are not necessary with `MVCPortlet`. When using the `name` attribute in your Action URL, just be sure that the value for the `name` attribute matches the name of your action method.
- For more complex apps, Liferay MVC can use separate classes for `render` and `action` phase commands.
- Simply set a parameter on the request called `mvcRenderCommandName` or `mvcActionCommandName`.
- Create a class that implements the `MVCRenderCommand` or `MVCActionCommand` interface.

## EXERCISE: CONNECTING GRADEBOOK-WEB TO SERVICES

- As we have already started to create Service layer implementation, let's connect `gradebook-web` to the `gradebook-api` module.
  1. Open the file `build.gradle`.
  2. Add the contents of the snippet 1 `build.gradle` to the `dependencies` block just above the closing `}` sign.
  3. Save the file.
  4. Refresh your modules by right-clicking the `gradebook` folder → `Gradle` → `Refresh`.

---

## EXERCISE: INITIALIZING THE VIEW

- » We'll set up for creating our View by organizing our taglibs and imports in one place:
  1. Open the module `gradebook-web`.
  2. Find the folder `resources/META-INF/resources`.
  3. Open the file `init.jsp`.
  4. Replace the contents of the file with the snippet *2 Init JSP*.

---

## INTENT TO DECLARE

- » In addition to some predictable imports and taglib declarations, the `init.jsp` contains a few interesting lines:

```
<portlet:defineObjects />
<liferay-theme:defineObjects />
<liferay-frontend:defineObjects />
```
- » These tags make some Java objects available on the JSP that are very useful.
- » Some objects include `request`, `renderRequest`, `user`, and `locale`.

---

## EXERCISE: GETTING THE DEFAULT DATA

- » Our default View will show us any assignments already created.
  - » We'll follow our MVC pattern here and use the Controller to retrieve the information and pass it to the View.
1. Open the class `GradebookPortlet` in your source folder.
  2. Insert the snippet `3 GradebookPortlet doView` in the body of the class.
  3. Type and hold `Ctrl/Command+Shift+O` to resolve any imports.
  4. Save the file.

---

## EXERCISE: MAKING THE DEFAULT VIEW

- » Now that we have a list of assignments, creating the default View is simple:
1. Open the `view.jsp` in `resources/META-INF/resources`.
  2. Replace the contents of the file with the snippet `4 View JSP`.
  3. Save the file.

---

## EXERCISE: GIVING OPTIONS

- » When displaying the default list of assignments, we'd like to provide a list of actions you can perform on the assignments:
  1. Go to the folder resources/META-INF/resources if you aren't there already.
  2. Create a folder named assignment.
  3. Create a file named card\_actions.jsp in the folder assignment.
  4. Insert the snippet 5 *Card Actions* into the file.
  5. Save the file.

---

## EXERCISE: ADDING THE PORTLET NAME

- » To navigate from our view.jsp to an edit View, we'll implement an MVCRenderCommand.
- » To simplify implementing command classes, we'll provide some helpful constants in a separate class.
  1. Go to the source folder of the gradebook-web module.
  2. Open the package com.liferay.training.space.gradebook.portlet.
  3. Create a Java class called GradebookPortletKeys.
  4. Replace the contents of the file with the snippet 6 *GradebookPortletKeys*.
  5. Save the file.

---

## EXERCISE: A PATH TO EDIT

- Before rendering the edit assignment JSP, we'll check to see if we need to retrieve an assignment.
  - We can retrieve the assignment data from the Model layer (Service) in the Controller before rendering the JSP.
  - We'll also set the portlet title to reflect the change in View and purpose.
1. Create a new package:  
`com.liferay.training.space.gradebook.portlet.command`.
  2. Create a new Java class named `EditAssignmentMVCRenderCommand`.
  3. Replace the contents of the file with the snippet *7 Edit Assignment Render Command*.
  4. Save the file.

---

## EXERCISE: EDITING AN ASSIGNMENT

- To implement the edit assignment View, we'll make a simple JSP with a form.
1. Open the folder `resources/META-INF/resources`.
  2. Create a file named `edit_assignment.jsp` in the folder `assignment`.
  3. Replace the contents of the file with snippet *8 Edit Assignment JSP*.
  4. Save the file.

---

## EXERCISE: PREPARING FOR ACTION

- » When the edit assignment form is submitted, it's going to perform an action.
  - » We'll implement these actions as `MVCActionCommand` classes.
  - » To make our code leaner, we'll put some helper code in a separate class.
  - » This will help us parse form data to form a valid Assignment model.
1. Open the package `com.liferay.training.space.gradebook.portlet`.
  2. Create the Java class `GradebookPortletUtil`.
  3. Replace the contents of the file with the snippet 9 `GradebookPortletUtil`.
  4. Save the file.

---

## EXERCISE: ADDING AN ASSIGNMENT

- » Let's complete the action to add an assignment.
  - » We'll implement this action as a separate class.
  - » When the action is complete, we'll be redirecting back to the default View.
1. Open the package  
`com.liferay.training.space.gradebook.portlet.command`.
  2. Create a new Java class named `AddAssignmentMVCActionCommand`.
  3. Replace the contents of the file with the snippet 10 `Add Assignment Action Command`.
  4. Save the file.

---

## EXERCISE: PREPARING SUBMISSIONS

- » Now that an assignment is added, we'd like to view the details of that assignment, such as already received submissions and grades.
  - » Though we don't have a way to add submissions, we can easily build in a way to view them.
  - » To retrieve the list of submissions for an assignment, we'll create a Controller method before we render the JSP.
1. Open the package  
*com.liferay.training.space.gradebook.portlet.command*.
  2. Create a Java class named `ViewSubmissionsMVCRenderCommand`.
  3. Replace the contents of the file with the snippet *11 View Submissions Render Command*.
  4. Save the file.

---

## EXERCISE: VIEWING SUBMISSIONS

- » When a user clicks on the title of an assignment, we'll present a new JSP.
  - » The new JSP will contain a simple list of results and some basic info about the assignment.
1. Open the folder `resources/META-INF/resources`.
  2. Create a new folder called `submission`.
  3. Create a new file named `view_submissions.jsp`.
  4. Replace the contents of the file with the snippet *12 View Submissions JSP*.
  5. Save the file.

---

## EXERCISE: ACTING ON THE UPDATE

- » To enable editing an existing assignment, we need to add this missing action class.
  1. Open the package  
*com.liferay.training.space.gradebook.portlet.command*.
  2. Create a Java class named `EditAssignmentMVCActionCommand`.
  3. Replace the contents of the file with the snippet *13 Edit Assignment Action Command*.
  4. Save the file.

---

## EXERCISE: DELETING THE ACT

- » To allow deleting an assignment, we'll create our final action class.
  1. Open the package  
*com.liferay.training.space.gradebook.portlet.command*.
  2. Create a Java class named `DeleteAssignmentMVCActionCommand`.
  3. Replace the contents of the file with the snippet *14 Delete Assignment Action Command*.
  4. Save the file.
  5. Rebuild and deploy all the gradebook modules again.

---

## WRAPPING THE APP

- » Our Controller and View layers are built on the Liferay MVC framework.
- » JSPs make up each of our Views.
- » The *render phase* is used for displaying information.
- » The *action phase* is used for processing and submitting data to the Model layer.
- » Changing the View in the Controller was accomplished through setting a render command.
- » Performing an action was done through invoking an action command.
- » Action and render commands are easily expanded and modified without interfering with other parts of the app.

Notes:



## Chapter 10

# Liferay Utilities



## COMMON UTILITIES

Copyright ©2016 Liferay, Inc.  
All Rights Reserved.

No material may be reproduced electronically or in print,  
distributed, copied, sold, resold, or otherwise exploited  
for any commercial purpose without express written  
consent of Liferay, Inc.

### UTILITY CLASSES

- A utility class is a class that provides methods that are typically common and reusable.
- Sometimes the term *helper class* is used as a synonym for utility class.
- The difference is that the methods in a utility class are **static**.
- There are a number of utility classes in Liferay. We'll take a look at them and see what they're used for.

---

## OTHER UTILITY CLASSES

- One way to find all the available utility classes is to look at Liferay's source code.
- However, as you can imagine, this can be very time-consuming.
- We will introduce you to some common utility classes provided by Liferay.
- Keep in mind that these are only the commonly-used utility classes in Liferay, and that there are many others you can use.
- For an in-depth look at some of the classes, refer to the source code for more detail.

---

## PARAMETER UTILITY

- The parameter utility class is known as `ParamUtil`.
- `ParamUtil` allows us to retrieve parameters from the request, whether it's the `HttpServletRequest`, `ServletRequest`, `PortletRequest`, or anything in between.
- The `ParamUtil` class also takes into account primitive types.
- For example, `ParamUtil.getInteger(renderRequest, "intParam")` will convert the parameter `intParam` in the `renderRequest` to an integer.

---

## PORtal UTILITY

- » The Portal Utility class is known as (you guessed it) `PortalUtil`.
- » `PortalUtil` allows us to obtain the various Liferay-specific attributes.
- » Some Liferay attributes are the Portal ID, Site ID, User ID, etc.
- » In the code, however, some attributes have a different name than what you may see in the UI.
  - » Some examples are:
  - » Portal ID = `InstanceId`
  - » Site ID = `GroupId`
  - » Web Content ID = `JournalArticleId`

---

## HTML UTILITY

- » There are always those who will want to do harm to your system by using Cross Site Scripting (XSS).
- » XSS is a means of injecting malicious code into a Site typically via scripts.
- » To protect our system from XSS, the HTML Utility known as `HtmlUtil` provides us just the method to achieve this.

---

## ESCAPE

- » Escaping converts inputs, text, etc. into a format that is safe for our system to use.
- » Suppose in a form, someone entered (Robert); DROP TABLE Students;\*\* as their name.
- » We see Little Bobby Tables over here trying to DROP our Students table.
- » To prevent this, we can escape the name field so characters such as ; and ) are converted to be safe to run in whatever environment (CSS, Javascript, etc.)
- » HtmlUtil allows us to do this very simply for any text fields.
- » All you need to do is use `HtmlUtil.escape()` for any text entered by a User.

---

## RENDERING

- » Rendering converts content into text.
- » This provides a version of content that you and I are able to read (unless you're a robot).
- » This is similar to how some email clients automatically convert HTML emails to text in their previews.

---

## REPLACING AND STRIPPING

- » Replacing in Liferay will replace new lines or returns with <br/>, the HTML tag for a single new line.
- » Stripping allows us to remove XML comments, HTML, and content outside a tag.

---

## PROPERTY UTILITY

- » The Property Utility class is known as PropsUtil.
- » PropsUtil gives us a number of methods to be able to configure properties throughout Liferay.
- » Within the PropsUtil class, you'll find a number of getters to get the property and the value of the property as well.
- » At the very bottom there is one setter method for properties.
- » For example,  
`PropsUtil.get("layout.show.portlet.access.denied")` will return the value of that property.

## Notes:



## TAG LIBRARIES

Copyright ©2016 Liferay, Inc.  
All Rights Reserved.

No material may be reproduced electronically or in print,  
distributed, copied, sold, resold, or otherwise exploited  
for any commercial purpose without express written  
consent of Liferay, Inc.

### OVERVIEW

- Tag libraries allow for a clean separation between the look of your application and its logic.
- They allow us to execute Java code from within a JSP without using scriptlets, which are harder to test and don't meet the goal of separating display code from business logic.
- Liferay provides a variety of tag libraries to simplify development and allow for easier integration.
- We'll be looking at the more commonly used ones, such as `liferay-ui`, `liferay-theme`, and `aui`.

## LIFERAY UI

- » The Liferay UI tag library contains custom tags used primarily for user-interface related code.
- » Some examples of this are error message reporting, internationalization, and drawing data tables.
- » In our application, we will primarily be using this tag library to draw the data tables.

WWW.LIFERAY.COM

LIFERAY.

## LIFERAY UI EXAMPLE

- » The most commonly used Liferay UI tag is the Search Container, used for drawing data tables.
- » The Search Container looks like this:

A screenshot of a Liferay search container displaying a list of user roles. The table has three columns: Role Name, Description, and Actions. The roles listed are Administrator, Guest, Owner, Portal Content Reviewer, Power User, and User. Each role has a corresponding icon and a detailed description below it. At the bottom of the table, there is a footer indicating 20 Entries and Showing 1 to 6 of 6 entries.

Name	Description	Actions
Administrator	Administrators are super users who can do anything	⋮
Guest	Unauthenticated users always have this role	⋮
Owner	This is an implied role with respect to the objects users create.	⋮
Portal Content Reviewer	This is an implied role with respect to the workflow definition.	⋮
Power User	Power Users have their own personal site.	⋮
User	Authenticated users should be assigned this role.	⋮

WWW.LIFERAY.COM

LIFERAY.

---

## LIFERAY THEME

- » The Liferay Theme tag library provides a number of context-related objects that can be used within your JSP.
- » This includes objects like the current User's locale, timezone, the page he or she is on, and the permission checker.
- » In our application, this library is primarily used to get access to the permission checker for the current User.
- » This will be used to verify the User's access to particular resources.

---

## ALLOY UI

- » The Alloy tag library is used when you want to use Alloy UI, Liferay's provided UI framework.
- » This is commonly used when you're working with Service Builder-based entities or even just placing things on a page.
- » As AlloyUI is integrated with Liferay, it ties very well with Service Builder-backed entities and simplifies code for you.
- » It allows for a very clean separation between the presentation logic and the business logic.
- » In our application, Alloy UI will be used for the form fields.

## ALLOY UI EXAMPLE

- » Alloy UI tags are commonly used when building forms.
- » Some useful tags for this are <aui:form>, <aui:input>, <aui:select>, and <aui:button>.
- » A form that uses all of these would look like this:

The screenshot shows a Liferay application window titled "User: New Custom Field". Inside, there's a form with two fields: "Key" and "Type". The "Type" field is currently set to "Text Field - Indexed". At the bottom of the form are two buttons: "Save" and "Cancel".

## ALLOY UI FORM EXPLAINED

- » In the above image, the Key field is created using an <aui:input> tag.
- » The Type field is generated from an <aui:select> tag.
- » Finally, the Save button is drawn from an <aui:button> tag, which submits the form.

---

## OTHER TAGLIBS

- » Liferay provides a lot more tag libraries that you can use in your code.
- » You also have the ability to import tag libraries outside of those created by Liferay.
- » One very commonly used such library is JSTL (JSP Standard Tag Library).
- » This library is a collection of useful JSP tags which is meant to combine functionality common to many applications.
- » We primarily use this in our code to display things only if the User has the right permissions to access them.

---

## USING A NEW EDITOR

- » Liferay 7 provides a new type of editor we can use, the AlloyEditor.
- » AlloyEditor makes creating elegant and beautiful content very easy.
- » We will be using AlloyEditor to enter the description of an Assignment.

---

## EXERCISE: ADDING ALLOYEDITOR

1. Open the file `edit_assignment.jsp`.
2. Find the form starting with the `<aui:form />`.
3. Replace the tag `<aui:input name="description" />` in the existing form with snippet 01-alloyEditor.
  - We've hidden the existing description field and instead attached the AlloyEditor to it.
  - This will allow us to switch editors without having to rework any of our other modules.

---

## EXERCISE - ADDING JAVASCRIPT

- Since we're attaching AlloyEditor to an existing field, we need to map the two together.
  - This can be done using JavaScript.
1. Go to the bottom of `edit_assignment.jsp`.
  2. Add snippet 02-aui:script to the very bottom of the JSP.
- This contains a method which will copy the contents of the AlloyEditor to the description field when the page is saved.

---

## EXERCISE - FORM CHANGES

- » We need to make sure the JavaScript function `savePage()` gets called whenever the page is saved.
- 1. Replace the existing `<aui:form>` tag with snippet `03-aui:form`.
  - » This assigns a name to the form and ensures that the `savePage()` method is called when the page is saved.

---

## EXERCISE: DESCRIPTION

- » We also need to be able to display the description within the AlloyEditor.
- 1. Add snippet `04 - Description` underneath `String title = (String) request.getAttribute("title");`.
- 2. Save the file.
  - » This snippet will ensure that we populate the description with something, even if that something is an empty String.

---

## A NEW FEEL

- There's now a completely new input for the Assignment description.
- With a few short tags, we have a rich editing solution for a large block of text.
- Many other tags exist to assist in your app development.

Notes:



## Chapter 11

# Feedback and Validation

Feedback and validation

• Feedback is information about performance that is given to the performer.

• Validation is the process of determining whether or not a system or product meets its intended purpose.

• Feedback can be used to validate a system or product.

• Validation can be used to provide feedback to a system or product.

• Feedback and validation are both important for the success of a system or product.

• Feedback and validation are both important for the success of a system or product.

• Feedback and validation are both important for the success of a system or product.

• Feedback and validation are both important for the success of a system or product.

• Feedback and validation are both important for the success of a system or product.

• Feedback and validation are both important for the success of a system or product.

• Feedback and validation are both important for the success of a system or product.

• Feedback and validation are both important for the success of a system or product.



## VALIDATION

Copyright ©2016 Liferay, Inc.  
All Rights Reserved.

No material may be reproduced electronically or in print,  
distributed, copied, sold, resold, or otherwise exploited  
for any commercial purpose without express written  
consent of Liferay, Inc.

## BUILDING A BETTER USER EXPERIENCE

- We've now created the core of our application.
- In some cases, what we have now might be good enough to meet the requirements and be considered "complete."
- According to our requirements and from a User experience perspective, however, we're not done yet.
- To evaluate what we need to add to make this application more complete, let's do a little testing.

## EXERCISE: THE CURRENT EXPERIENCE

» Let's take a look at the current state of our application.

1. Sign in to Liferay.
2. Go to the *Assignments* application.
3. Add a new Assignment with valid data in all of the fields.
  - » Did it successfully add?
  - » How do you know?
4. Click the *Add* button.

## EXERCISE: CREATING AN ASSIGNMENT NOW

1. Create an assignment without entering any data.
2. Click the *Save* button.
  - » Are we supposed to be able to add empty assignments?

---

## MAJOR IMPROVEMENTS NEEDED

- » We've identified two major areas of improvement here.
  1. The only way we can tell if we were successful in adding anything is to check the list.
  2. Actually, we do have a way of knowing if we were successful, but it's another problem, not a solution. Right now, anything you try to add will add even if it's literally nothing.
- » First, we'll create the necessary logic to prevent invalid data from being added.
- » Once we've done that, we can fix the feedback issue so that we can provide basic feedback to a User after task completion.

---

## VALIDATION

- » Whenever an application is accepting User input, you need to provide some form of validation, for a variety of reasons.
  1. We need to check that data has actually been entered. Some fields are necessary, and we need to make sure that they're filled in.
  2. We need to ensure that the information provided is valid. If a field asks for a phone number, we should check if the text entered is a valid phone number.
  3. We need to prevent harmful data from being entered. Simple validation of fields can protect against attempts to enter malicious code through a form field.

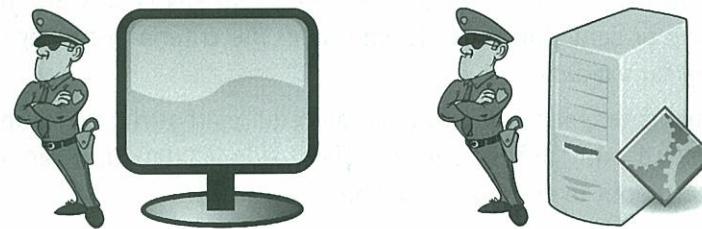


The Validator

---

## TYPES OF VALIDATION

- » Generally speaking, there are two places that we can do validation: client-side and server-side.



WWW.LIFERAY.COM

LIFERAY.

---

## CLIENT-SIDE VALIDATION

- » Client-side validation means that as the data is entered, it undergoes some level of validation.
- » In some cases, the validation might even occur before the form is submitted.
- » One example of client-side validation might be not submitting the form unless all required fields are completed.
- » Good client-side validation will improve the User experience, as it will provide faster feedback since it will all occur up front.



WWW.LIFERAY.COM

LIFERAY.

## SERVER-SIDE VALIDATION

- » Server-side validation is something that's happening on the backend before the information is added to the database.
- » This might be some more complex validation run against the contents of the fields which are entered - does it contain any characters that aren't allowed or something that doesn't meet the criteria of the logic we have set up for validation?
- » In most cases, server-side validation requires that the full form be submitted. Although slower than client-side validation, it can add greater depth to content validation.



## STATE OF THE APPLICATION

- » Our current iteration of the application will accept any User input, or even no input at all.
- » We can use Liferay's validation utilities to fix this.
- » Liferay offers both client-side and server-side validation utilities.
- » Server-side validation is provided through the `Validator` class in Liferay's kernel utilities.
- » Liferay's User interface framework provides a client-side form validation utility.
- » For our application, we'll go from the bottom up and start with server-side validation.

---

## LIFERAY'S SERVER-SIDE VALIDATION

- Liferay's server-side validation utility is implemented in `com.liferay.portal.kernel.util.Validator`.
- This class provides utility methods useful for data validation and format checking.
- We'll create simple validator classes for Assignments.
- Then, we need to import Liferay's validator class into our action command classes and into the custom validator classes we're about to create.
- While we do this, we'll also update the *add*, *update*, and *delete* methods for the gradebook portlet to use our validator features.

---

## EXERCISE: LIFERAY SERVER-SIDE VALIDATION

- First we'll create the class which will manage all of our server-side validation logic.
1. Create a new package called `com.liferay.training.space.gradebook.validator` in the `gradebook-web` project.
  2. Create a new class called `AssignmentValidator.java` inside the package.  

  3. Add the contents of the *01-Assignment Validator* snippet to the contents of the newly created `AssignmentValidator.java`.
  4. Save the file.

---

## EXERCISE: UPDATING METHODS

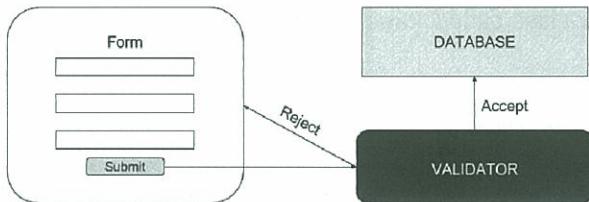
- Next, we need to update our existing methods to actually use the validation that we provided in `AssignmentValidator.java`.
  1. Replace the `doProcessAction()` method in `AddAssignmentMVCACTIONCommand.java` with the contents of the `02-addAssignment()` snippet.
  2. Replace the `doProcessAction()` method in `EditAssignmentMVCACTIONCommand.java` with the contents of the `03-editAssignment()` snippet.
  3. Add any required imports to each file, including:

```
import com.liferay.portal.kernel.util.Validator;
```
- ✓ Save the files.

---

## VALIDATION IN THE PORTLET CLASS

- Our validation uses the `isNull()` and `isBlank()` method to ensure that all of the form values contain data.
- If submitted values are valid, the data is written to the database.
- If there are any errors, we add a relevant message to the `SessionErrors` object which will be passed back to the View layer.



## THE AUI FORM VALIDATOR

- » Next, we'll implement a simple example of client-side validation.
- » As long as there is a response, we will consider it valid.
- » Client-side and server-side validation work together, so that completely empty entries never have to waste backend processing time. Any complete entry receives a more thorough check once it has passed the client-side validation.
- » Using the AUI form validator with `<aui:form>` fields is easy; you use a `<aui:validator>` tag, specifying the `name` attribute and the `errorMessage` attribute.

```
<aui:input name="title" >
    <aui:validator name="required" errorMessage="Please enter ..." />
</aui:input>
```
- » In the above example, the term *Required* will appear next to the field label, and the string specified in the `errorMessage` attribute will appear if a User leaves a field blank.

## EXERCISE: AUI FORM VALIDATOR

1. Open the `.../resources/assignment/edit_assignment.jsp`.
  - » We'll specify some fields as required:
    - » Title
    - » Due Date
2. Replace the contents of the `<aui:fieldset>` tag with `04-edit_assignment.jsp`.
3. Save the file.
4. Rebuild the gradebook modules.
5. Deploy the rebuilt modules again.

## NOTES ON THE AUI FORM VALIDATOR

- » Notice that we added `<aui:validator />` subelements to the `<aui:input />` elements.
- » We specified the specific validator rule to apply using the `name` attribute of the `<aui:validator />` element:  
`<aui:validator name="required" />`.
- » Instead of relying on the default Error Message, we specified our own using the `errorMessage` attribute:  
`<aui:validator ... errorMessage="Please enter ..." />`.
- » You can find a complete list of available AUI form validator rules in the `aui-form-validator.js` file in the AUI source:  
<https://github.com/liferay/alloy-ui/blob/master/src/aui-form-validator/js/aui-form-validator.js>

## EXERCISE: CHECKPOINT

1. Click on the *Add* button for an Assignment.
  - » The required fields are now marked on the input form as such.
2. Try to save a new Assignment while leaving the required fields blank.
  - » The AUI validator detects that a required field is blank and prevents the request from being sent to the server.



This field is required.

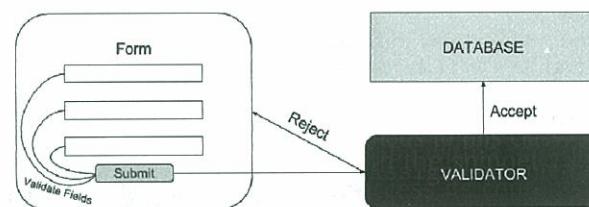
Description

Due Date \*

11/23/2016 04:03 PM

## A CLOSER LOOK AT THE FORM VALIDATOR (I)

- » This example of the AUI Validator is very straightforward.
- » We provide a field name, specify that it is required, and then provide an error message should the field be found without content in it.
- » This actually belies the true power of the AUI Validator, as there are a number of attributes that could be added for more advanced client-side validation.



## A CLOSER LOOK AT THE FORM VALIDATOR (II)

- » In this case, we specified that the field `name` was `required`, and then provided an `errorMessage`.
- » We have other options besides "required".
- » For example, we could have used the "`minLength`" attribute to specify that some minimum number of characters was required to prevent Users from simply typing one letter as a name.
- » You can also combine multiple attributes on one field to enforce formatting, character type, and length rules.

## CUSTOM AUI FORM VALIDATION (I)

- » To specify a custom validation rule that's not specified in `aui-form-validator.js`, use the `<aui:validator />` tag.
- » For example, Liferay's `edit_file_entry.jsp`, which is used for creating and editing files in Liferay's Documents and Media library, includes a custom validation rule:

```
<aui:input name="title">
    <aui:validator errorMessage="you-must-specify-a-file-or-a-title"
        name="custom">
        function(val, fieldNode, ruleValue) {
            return ((val != '') || A.one('#<portlet:namespace />file').val() != '');
        }
    </aui:validator>
</aui:input>
```

- » This rule specifies that either a file title or an actual file must be provided in order to create a Liferay DL file entry.

## CUSTOM AUI FORM VALIDATION (II)

- » Custom validation rules are created as Javascript functions.
- » The details rule ultimately comes down to a yes or no question. If it validates, return "true"; if it does not validate, return "false."
- » When determining your validation rules, you'll need to determine where and how best to implement them.
  - » Can I improve the User experience by providing this validation on the client-side?
  - » Is this available in AUI Validator's defaults?
  - » If not, can I easily implement it in Javascript?
  - » If I can implement it in Javascript, would it make more sense to include it in the JSP, or would it be better to implement it in my validator class?
- » Sometimes this isn't a matter of a right or wrong way of doing things, but more a matter of what makes the most sense for your project.

---

## BONUS EXERCISES: LIFERAY SERVER-SIDE VALIDATION

- » Now that we're verifying that data has been entered on the front end, what are some other things we could check for in our server-side validation?
- » **Bonus:** Improve the validation so that the server-side validation provides a different function than the client-side (simply validating that some data was received.)

---

## FEEDBACK, VALIDATION, AND YOUR APPLICATION

- » We now have an application that provides simple feedback for a User, validates anything added, and provides basic guidance for the contents of fields.
- » From a usability and security standpoint, this has brought our application a long way from where we were.
- » What are some other important features for applications that we haven't implemented yet?
- » What are some key features for projects that you're planning?

## Notes:



## FEEDBACK AND LOCALIZATION

Copyright ©2016 Liferay, Inc.  
All Rights Reserved.

No material may be reproduced electronically or in print,  
distributed, copied, sold, resold, or otherwise exploited  
for any commercial purpose without express written  
consent of Liferay, Inc.

### FEEDBACK

- » Now that our application features validation, we also want to provide feedback.
- » In order to provide feedback, we'll need to create and configure meaningful messages to be displayed.
- » The easiest way to do this is to display "Success" messages when a User operation completes.
- » To do this, we'll have the Controller layer add *keys* into the request objects.
- » Keys are simple strings which represent a potential message.
- » The View layer will check for the key and display the appropriate message.
- » You can compare it to a switch. When the "add-assignment-success" switch is flipped, "Assignment Was Added Successfully" displays.

---

## EXERCISE: ADD SESSION MESSAGES KEYS

- » First, we'll add the keys as `SessionMessages` and `SessionErrors` objects in the Controller (our Action Command classes are an extension of the Controller).
  1. Open `AddAssignmentMVCACTIONCommand.java`.
  2. Replace the `doProcessAction()` method with the `01-Gradebook-SessionMessages-added` snippet.
  3. Save after organizing your imports.
    - » Note: There will be errors left as we changed the `AssignmentValidator` call, but don't worry, we'll fix that later.
- » These keys provide messaging for adding an Assignment successfully or encountering an error.

---

## EXERCISE: ADDING FEEDBACK - EDIT

1. Open `EditAssignmentMVCACTIONCommand.java`.
  2. Replace the `doProcessAction()` method with the `02-Gradebook-SessionMessages-updated` snippet.
  3. Save after organizing your imports.
    - » Note: There will be errors left as we changed the `AssignmentValidator` call, but don't worry, we'll fix that later.
- » We have now provided similar messaging for the *Edit* function.

---

## EXERCISE: ADDING FEEDBACK - DELETE

1. Open DeleteAssignmentMVCActionCommand.java.
  2. Add the *o3-Gradebook-SessionMessages-deleted* underneath the  
    "*\_assignmentLocalService.deleteAssignment(assignmentId);*"  
    line in the doProcessAction() method.
  3. Save after organizing your imports.
- » Now we have Feedback for the *Delete* function.

---

## EXERCISE: UPDATING THE ASSIGNMENT VALIDATOR

1. Open AssignmentValidator.java.
  2. Replace the isAssignmentValid() method with the *o4-Update AssignmentValidator* snippet.
  3. Save after organizing your imports.
- » Now the AddAssignmentMVCActionCommand.java and  
EditAssignmentMVCActionCommand.java should compile without  
errors.

---

## EXERCISE: ADDING KEYS TO VIEW LAYER

- » Next, we'll add references to the keys to our JSPs.
  - » We'll need to add messages in `view.jsp` and `edit_assignment.jsp`.
1. Open `/resources/META-INF/resources/view.jsp`.
  2. Add the `o5-success-messages` snippet just below the `<liferay-frontend:add-menu>` block.
  3. Open `/resources/META-INF/resources/assignment/edit_assignment.jsp`.
  4. Add the `o6-error-messages` snippet right after `</c:choose>`.
  5. Save the file.
  6. Rebuild and deploy your modules again.

---

## EXERCISE: TEST FEEDBACK

1. Sign in to Liferay in your web browser.
2. Add a new Assignment in the *Gradebook Application*.
  - » Do you see an "Assignment created successfully!" message?
3. Edit the Assignment.
  - » Do you see the "Assignment updated successfully!" message?
4. Delete it.
  - » What do you see now?
5. Now try to edit an existing assignment, but leave some of the fields empty. What do you see now?

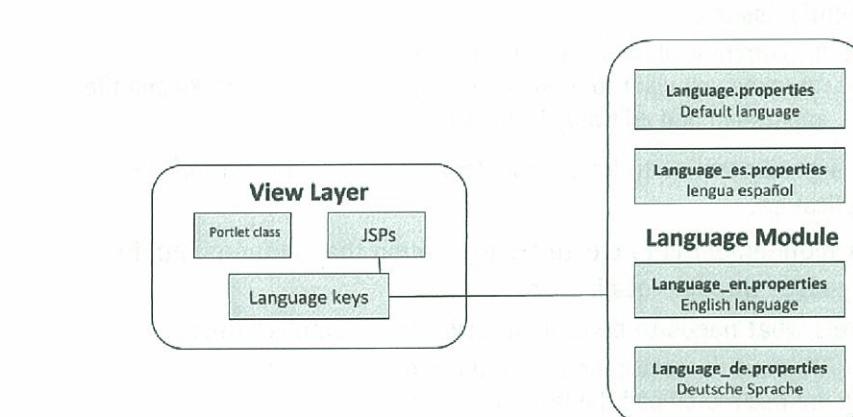
## RESOURCE BUNDLES (I)

- » Now we have basic feedback for User actions, but there are a few potential issues.
  1. It's currently all hard-coded in English.
  2. Any time you want to make a change, you'll need to crack open the application and edit the View layer directly.
- » It's a good practice to keep basic text outside of the core of the application.
- » Our requirement is to create an application that supports multiple languages through localization.
- » Here's what needs to be done in order to accomplish this:
  - » We need to replace the text with customizable keys.
  - » We need to offload the text into a resource bundle.
  - » We need to store that language bundle in its own module.

## RESOURCE BUNDLES (II)

- » We'll need to use a resource bundle to provide language keys in our application.
- » In the interest of modularity, we'll create a separate resource bundle module to handle localizations for labels, errors, alerts, table headers, or descriptions.
- » Essentially, on the backend, all of the text should be handled as "keys," and all of the keys should have their full text - and any localized versions - stored separately in a resource bundle.
- » This helps us reinforce "separation of concerns," where each module represents one discrete function.
- » This will also enable us to easily manage content for text in the application without having to touch any of the modules which contain core functionality.

## RESOURCE BUNDLES (III)



## EXERCISE: REPLACE TEXT WITH KEYS

1. Replace the group of keys with snippet *o7-success-messages-keys* back in *view.jsp*.
  2. Replace the keys with *o8-error-messages-keys* in *edit\_assignment.jsp*.
  3. Find the *<aui:validator>errorMessage* in *edit\_assignment.jsp* that we created before.
  4. Replace it with "assignment-title-format-error".
  5. Save the file.
- » This will change the format of your key from hardcoded text like, "Assignment created successfully!", to a language key like, "assignment-added-successfully".

---

## EXERCISE: CREATING THE MODULE

» Now we'll create a new module for our resource bundle that will translate those keys into readable strings.

1. Right-click in Liferay IDE on your workspace.
2. Choose *New* → *Liferay Module Project*.
3. Type *gradebook-lang* for the *Project name*.

---

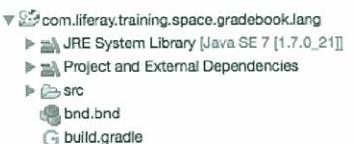
## EXERCISE: CHOOSING A LOCATION

1. Uncheck the *Default location*.
2. Choose *modules/gradebook* for the new location.
3. Choose *activator* for the *Project Template Name*.
4. Click *Finish*, leaving the rest of the defaults.

---

## EXERCISE: MODULE CLEANUP

- » We just used a default template, but we don't really need everything in here.
  - » For the sake of keeping our project clean, we'll remove the Java classes, and we'll edit `bnd.bnd` to provide the necessary basic information about our module.
1. Delete the `/src/main/java` folder in your new project.
  2. Open `bnd.bnd`.
  3. Click on the *Source* tab.
  4. Delete the `Bundle-Activator` element.
  5. Save the `bnd.bnd` file.



WWW.LIFERAY.COM

LIFERAY.

---

## EXERCISE: CREATING A -LANG BUNDLE

- » Your keys and their values need to go in a `Language.properties` file.
1. Create a `resources` folder in the `src/main` directory.
  2. Create a `content` folder in the `resource` folder.
  3. Create the `Language.properties` file in the `content` folder.
  4. Add snippet `o9-language.properties` to the `Language.properties` file.
  5. Save the file.
- » We've added English language keys for all of our keys.
  - » In addition, we've added specific keys for some of Liferay's default keys.

WWW.LIFERAY.COM

LIFERAY.

## LIFERAY'S DEFAULT KEYS

- » If you've noticed, throughout the creation of our app, there are some places where we've used a "key" in a JSP, and it's been changed into more readable text at runtime without us having to specify language keys.
- » There are a number of fields like "title" or "description" which are used frequently in Liferay and are picked up by default.
- » This means that when you create a "title" in your JSP, Liferay will automatically translate that to "Title" or maybe "Título" if you're viewing the page in Spanish.
- » In many cases, you can leave these alone, and Liferay handles the translation for you, but in other cases, you may wish to change the text or have a specific translation that differs from the default.
- » In those cases, you can override Liferay's default by simply adding the key to your own language bundle, like we've done here.

WWW.LIFERAY.COM



## EXERCISE: REFERENCING THE RESOURCE BUNDLE - LANGUAGE

- » First, we'll add a reference to the language module inside the web module's build.gradle.
  1. Open the build.gradle file for the gradebook-web project.
  2. Add the following line under dependencies:

```
compile project(':modules:gradebook:gradebook-lang')
```
  3. Add this line to the top of the file:

```
apply plugin: "com.liferay.lang merger"
```
- » We've now referenced the language bundle module in the workspace and sets it as a dependency for the Gradebook project to build.
- » We're also now using Liferay's plugin to merge -lang bundles with the web module.

WWW.LIFERAY.COM



---

## EXERCISE: REFERENCING THE RESOURCE BUNDLE - ROOT

- » In order for the Liferay Language Bundle Merger plugin to work, we need to add the correct reference in the root module.
  1. Open the `build.gradle` from parent `gradebook` module.
  2. Add snippet `10-lang.merger` to the bottom of the file.
  3. Save the `build.gradle` file.
  4. Rebuild and deploy the `gradebook` modules again.
- » This will include the plugin in the build for all of the modules and enable our language module to smoothly integrate without having to write additional code.

---

## EXERCISE: WHAT HAVE WE DONE SO FAR?

- » With these new changes, let's see the fruits of our labor.
  1. Click the *Add* button for a new Assignment.
  2. Save the form after completing it.
    - » Did you see the right message?
  3. Run as many actions as you can think of to test the feedback messages provided using the `Language.properties` as your guide.

---

## SUPPORTING MULTIPLE LANGUAGES (I)

- » We've used our `lang` module to replace the hard-coded labels in our form with messages stored in a `Language.properties` file, but we haven't implemented the mechanism for supporting multiple languages.
- » You can do this by providing companion files to our original `Language.properties` file.
- » These companion files will have a two-letter language code appended to them.

---

WWW.LIFERAY.COM



---

## SUPPORTING MULTIPLE LANGUAGES (II)

- » For example, `Language_es.properties` or `Language_fr.properties` are for Spanish or French, respectively.
- » Our application will use the values in these files if the User's locale is set to `es` or `fr`.
- » The translation process is a lot of work, requiring someone who is knowledgeable in the language to translate the file for each language you want to support.

---

WWW.LIFERAY.COM



---

## BONUS EXERCISE: CREATING ALTERNATE LANGUAGE KEYS

1. Create an alternate language properties file for a language supported by Liferay. (You can look at this blog post for reference - <https://www.liferay.com/web/manish.gupta/blog/-/blogs/liferay-now-have-34-languages-ootb>).
2. Create alternate language text for each key.
3. Choose the language that you created properties for from the *Site Settings* menu.

---

## APPLICATION EXPERIENCE

- » Providing appropriate feedback to Users in a localized fashion is a big step in providing an excellent User experience.

Notes:



## Chapter 12

# Liferay Services and Permissions



## HOW TO USE LIFERAY SERVICES

Copyright ©2016 Liferay, Inc.  
All Rights Reserved.

No material may be reproduced electronically or in print,  
distributed, copied, sold, resold, or otherwise exploited  
for any commercial purpose without express written  
consent of Liferay, Inc.

### SERVICES AND THE PLATFORM

- Liferay is built on individual components, each of which provide some functionality.
- That functionality is made available through *OSGi Services*.
- Once a service is published as an OSGi Service, any other module can use it to gain additional functionality.
- We've already created our own services and used them in other modules.
- Now we're going to look at the wide range of functionality available in *Liferay OSGi Services*.

## OVERVIEW

- » Liferay is built on the idea of small components providing bits of functionality.
- » Applications can then tie these components together to achieve their goals.
- » This functionality is made available through *OSGi services*.
- » Any module can make use of these published services to gain additional functionality.
- » Liferay provides a lot of its functionality through these services.

WWW.LIFERAY.COM



## WHY USE SERVICES?

- » These services serve a few important purposes in your application.
- » They allow you to use a pre-existing Liferay service, rather than having to reinvent the wheel to perform certain tasks.
- » They also provide for uniformity between all of the applications in Liferay and seamless integration with Liferay.

WWW.LIFERAY.COM



## WHAT DO SERVICES DO?

- » Liferay services can be used to do a number of things.
- » Some examples are:
  - » Getting Site memberships of *Users*
  - » Adding new *Tags* and *Categories*
  - » Searching for *Blogs* that have the tag *editorial-picks*
- » In short, anything you can do through the web interface, you can do using a service.

## A SERVICE FOR EVERY OCCASION

- » Liferay services are built for an underlying model, some of which are:
  - » Users
  - » Blogs
  - » Web Content Articles
  - » Message Board Posts
  - » Pages

---

## WHICH SERVICE DO I USE?

- » When considering which service you should use, the following questions will prove useful:
  - » What *Model* or *entity* do I need?
  - » Do I need to *get* or *set* something?
  - » Do I need to *change* anything about the Model?

---

WWW.LIFERAY.COM

LIFERAY.

---

## TYPES OF SERVICES

- » Liferay provides services in two varieties:
  - » Local
  - » Remote
- » *Local* services are meant to be consumed within the same *Liferay Instance*.
- » *Remote* services are designed to be triggered from a web service call.

---

WWW.LIFERAY.COM

LIFERAY.

---

## LOCAL V. REMOTE: DAWN OF SERVICES

- » If you have both local and remote Services, which one should you use?
- » In general, it's better to use a *local Service* than a *remote Service*, unless:
  - » You need to use a method only offered in the remote Service.
  - » You need the Service to provide some permission checking.
  - » As a general rule, Liferay's remote Services act as wrappers around its local Services and add permission checking for security.

---

WWW.LIFERAY.COM

LIFERAY.

---

## RTLM: READ THE LIFERAY MANUAL

- » Once you've decided on a service that fits your needs, how would you find more information?
- » There are three main places to find information relating to a Liferay service.

---

WWW.LIFERAY.COM

LIFERAY.

---

## LIFERAY DEVELOPER NETWORK

- ▶ The Liferay Developer Network is now the official source of all documentation.
- ▶ It can be found at <http://dev.liferay.com>.

---

WWW.LIFERAY.COM



---

## LIFERAY JAVADOCS

- ▶ The JavaDocs are the definitive source of information for the Liferay OSGi service APIs.
- ▶ They can be found at <http://docs.liferay.com/portal/7.0/javadocs/portal-service/>.

---

WWW.LIFERAY.COM



## LIFERAY MODULE SOURCE

- » The source code of the modules is a great way to see the services being used.
- » The source code for the Liferay platform is freely downloadable here:  
<https://github.com/liferay/liferay-portal>

WWW.LIFERAY.COM

 LIFERAY.

Notes:



## LIFERAY CORE SERVICES

Copyright ©2016 Liferay, Inc.

All Rights Reserved.

No material may be reproduced electronically or in print,  
distributed, copied, sold, resold, or otherwise exploited  
for any commercial purpose without express written  
consent of Liferay, Inc.

### USING LIFERAY SERVICES

- So we now have an understanding of Liferay's services and what they do.
- We know where to go to access more information and get a full list of services.
- Now let's dig a little deeper and talk about how to practically use Liferay services.

---

## ACCESSING LIFERAY SERVICES

- » There are two main ways that we can use Liferay's services:
  - » We can import portal-kernel.jar and reference services in our classes.
  - » Or we can access them via JSON web services and Javascript.

---

## SERVICE REFERENCES (I)

- » When we need to use a service in one of our Java classes, we don't call the service directly, but instead use the @Reference annotation.
- » The @Reference annotation identifies the annotated method as a bind method of a Service Component.
- » It enables the service locator without having to manually create a configuration - the metadata provided with the service reference replaces the configuration.

---

## SERVICE REFERENCES (II)

- » For example, if you wanted to instantiate an instance of a service class, your code would look something like this:

```
@Reference(  
    name = "some.service",  
    service = SomeService.class,  
)  
protected void setSomeService(SomeService someService) {  
    this.someService = someService;  
}
```

- » Also many times is just enough to annotate member variable of the class:

```
@Reference  
protected someService;
```

---

## LIFERAY WEB SERVICE API

- » We can access many of Liferay's services and functions through the JSON web service API.
- » You can access the API through languages other than Java, like Groovy, Javascript, Python, or Ruby.
- » Liferay provides a page which will allow you to browse and invoke various service methods for reference and testing.

## JSON WEB SERVICES

- » To access the JSON web services API, you can navigate in your browser to `localhost:8080/api/jsonws`, if you have Liferay running locally.
- » To find a relevant service to your needs, you can browse by *Context Name*, use the search, or simply scroll down to find what you're looking for.
- » When you click on a service method, you will be provided additional information about that method.



WWW.LIFERAY.COM

LIFERAY.

## PRACTICAL USES

- » JSON Web Services essentially allow you to break down any activity like creating, updating, retrieving, or deleting an object on Liferay into a URL.
- » You use Javascript or another scripting language to add this into a JSP for a variety of uses.
- » Ultimately it adds power and flexibility to your application and your development options.

WWW.LIFERAY.COM

LIFERAY.

## Notes:



## PERMISSIONS

Copyright ©2016 Liferay, Inc.  
All Rights Reserved.

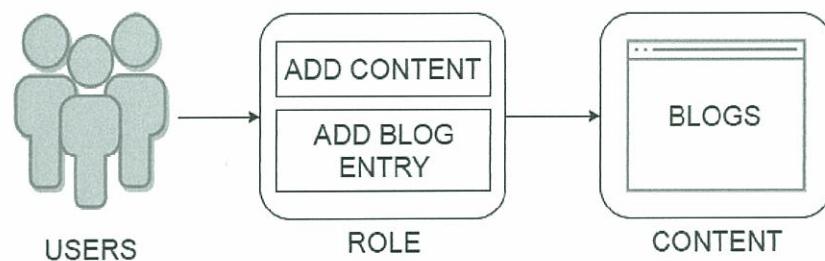
No material may be reproduced electronically or in print,  
distributed, copied, sold, resold, or otherwise exploited  
for any commercial purpose without express written  
consent of Liferay, Inc.

### OVERVIEW

- Users in your Organization have different jobs and belong to different groups.
- Users in one group need to be able to work on things that Users in another group don't know about.
- Users throughout your Organization should also be able to view certain content that they cannot necessarily modify.

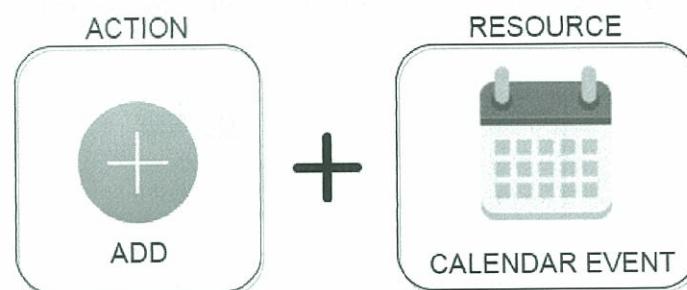
## ACCESS CONTROL IN LIFERAY

- » In Liferay, Users are given access to content via *Roles*.
- » *Roles* are simply groupings of *Permissions*, meant to simplify access control.



## WHAT ARE PERMISSIONS?

- » Permissions in Liferay are a combination of two things:
  1. A resource you want your Users to interact with
  2. An action a User should be able to perform upon this resource



## RESOURCES

- » A resource is anything you want the User to interact with.
- » Almost everything in Liferay is considered a resource.

WWW.LIFERAY.COM

LIFERAY.

## TYPES OF RESOURCES

- » There are two types of resources in Liferay.
- » *Portlet resources* are used to define actions for each portlet window.
- » *Model resources* are used to define actions on specific objects and are normally used by services.
- » *Model resources* are used to control the information portlets display.



PORLET  
RESOURCES



MODEL  
RESOURCES

WWW.LIFERAY.COM

LIFERAY.

## ACTIONS

- » Actions are tasks you want the User to be able to perform.
- » Common examples of actions are ADD, DELETE, VIEW, and EDIT.

## TYPES OF ACTIONS

- » There are two types of actions in Liferay.
- » *Top-level actions* are actions that are not applied to an existing resource.
- » The most common example of a *top-level action* is *ADD*.
- » *Resource actions* are actions that are applied to an existing resource.
- » Most actions fall under the umbrella of *resource actions*.



TOP-LEVEL  
ACTIONS



RESOURCE  
ACTIONS

---

## RESOURCES + ACTIONS

- » Permissions in Liferay are a combination of resources and actions.
- » ADD WEB CONTENT is a combination of a resource and an action. WEB CONTENT is the resource; ADD is the action.

---

## ROLES

- » Liferay uses *Roles* to control User access.
- » Roles are a list of permissions (resource actions) in a scope, which can be global, within an Organization, or within a Site.

## TYING IT ALL TOGETHER - OVERVIEW

- » To see how this works, let's take a look at an example of access control in Liferay.
- » Let's take the case of a guest User (not logged in) who wants to add a new blog entry.
- » Liferay doesn't give the guest User an option to do so, as he or she doesn't have the necessary permissions.
- » Why does this happen?

The screenshot shows a Liferay page with a 'Blogs' portlet. The portlet title is 'Blogs' and it includes an 'RSS' link. A message below the portlet states: 'Guest users are not allowed to add entries. Please log in to add a new entry.' At the bottom of the page, there is a footer with the URL 'WWW.LIFERAY.COM' and the Liferay logo.

## TYING IT ALL TOGETHER - PERMISSION CHECKING

- » When Liferay is loading the page, it checks the Roles the current User belongs to.
- » In this example, there is only one Role - Guest.
- » Liferay then checks the permissions for the *Blogs* portlet, which looks like this:

The screenshot shows the 'Entries Permissions' configuration screen. It displays a table with two rows: 'Guest' and 'Owner'. The 'Guest' row has three empty checkboxes under 'Subscribe', 'Permissions', and 'Add Entry'. The 'Owner' row has three checked checkboxes under 'Subscribe', 'Permissions', and 'Add Entry'. The table has columns for 'Role', 'Subscribe', 'Permissions', and 'Add Entry'.

WWW.LIFERAY.COM LIFERAY.

---

## TYING IT ALL TOGETHER - ACCESS CONTROL

- » Liferay sees that the Guest Role doesn't have *Add Entry* permissions and, as a result, doesn't display the button.

---

## HOW DO I ENABLE PERMISSIONS?

- » We're glad you're interested in securing your application!
- » Adding support for permissions to your application has three parts:
  1. Creating resources for your data
  2. Defining actions for these resources
  3. Checking a User's access levels

## CREATING RESOURCES

- » To enable permissions, your application needs *resources*.
- » *Resources* are objects used for permission-checking.
- » They are typically added to the database at the same time your entity is added.
- » When your entity is deleted, its corresponding resource gets deleted as well.

## WHAT DO RESOURCES CONTAIN?

- » Resources are meant to be an abstraction of your model.
- » They act as a placeholder so that your model can hold permissions.
- » They only contain permission-related information, not your business data, so they are not touched when you update your entity.
- » Liferay uses each resource to store a list of *actions* each *Role* can perform upon it.

## HOW DO I ADD RESOURCES?

- » Resources are an abstraction of your model.
- » They need to be added at the same time as your model.
- » This is traditionally done in the Service layer of your application.
- » The best place to add or delete a resource is in the Service layer, when you do the same to your model.

## EXERCISE: SERVICE LAYER RESOURCE CODE

1. Open AssignmentLocalServiceImpl in the gradebook-service module.
2. Replace the addAssignment method with snippet 01-AssignmentLocalServiceImpl.
  - » Note that we now have a deleteAssignment method as well.
3. See that your imports are organized.
4. Rebuild services.

## RESOURCELOCALSERVICE

- » Adding a resource looks like this:

```
resourceLocalService.addModelResources(assignment, serviceContext);
```

- » We create the resource using the service context, and the Assignment model object itself.
- » The object contains all of the necessary data about the Assignment to create the resource.

## DEFINING ACTIONS

- » Now that we've created resources, their corresponding actions need to be declared.
- » These actions also need to be tied to their respective resources.
- » This is done in a file called default.xml.

## ACTION CREATION

- » Remember, we have two types of resources we are creating actions for.
  - » **Portlet resources** and **model resources**.
- » Actions for these will be created in different modules:
  - » gradebook-web will contain the actions for the portlet resources.
  - » gradebook-service will contain the actions for the model resources.

## EXERCISE: ACTION CREATION - PORTLET RESOURCES

1. Create a file called `portlet.properties` in the `gradebook-web` module under the `src/main/resources` folder.
2. Add snippet `02-gradebook-web portlet.properties` to this file. This points the way to `default.xml`, but we haven't created the file yet or the folder in which it resides.
3. Create a new folder called `resource-actions` in `src/main/resources`.
4. Create a file called `default.xml` in the `resource-actions` folder.
5. Add contents of snippet `03-gradebook-web default.xml` to the file.

---

## EXERCISE: ACTION CREATION - MODEL RESOURCES

- » Now we're going to repeat those steps to create the same files for the service module.
- 1. Create a file called `portlet.properties` in the `gradebook-service` module under the `src/main/resources` folder.
- 2. Add snippet `04-gradebook-service portlet.properties` to this file.
- 3. Create a new folder called `resource-actions` in `src/main/resources`.
- 4. Create a file called `default.xml` in the `resource-actions` folder.
- 5. Add the contents of snippet `05-gradebook-service default.xml` to the file.

---

## TROUBLESHOOTING

- » If you have any red Xs or issues building, before you get to debugging code issues or misplaced snippet, remember:
  - » Clean up your editor by closing any tabs that you aren't currently using.
  - » Run build services on the service module.
  - » Right click on the project and run "Refresh" or "Gradle → Refresh Gradle Projects."

---

## CHECKING PERMISSIONS

- » We have created our resources and their actions.
- » We now need to make sure that a User's access levels are checked at the right time.
- » This will allow us to grant or deny access as needed.

---

## PERMISSION HELPER CLASSES

- » We will be creating helper classes to handle permission-checking for us.
- » This is a pattern followed by Liferay and is considered a best practice.
- » It decouples the permission-checks from the rest of your application.
- » It also simplifies changing the permission-checks if needed.

---

## EXERCISE: PERMISSION CHECKER HELPER CLASS

- » Let's create a helper class to simplify our permission checking.
  - » Liferay modules follow this pattern, and it eases the process of changing any permission checks.
1. Create a new package called  
com.liferay.training.space.gradebook.service.permission  
in the gradebook-service project.
  2. Create a new class in our newly created package called  
AssignmentPermissionChecker.java.
  3. Replace the contents of this file with snippet *o6-AssignmentPermission*.
  4. Open the modules bnd.bnd.
  5. Add the line Export-Package:  
com.liferay.training.space.gradebook.service.permission.
  6. Save all files.

WWW.LIFERAY.COM

LIFERAY.

---

## EXERCISE: PERMISSION CHECKING - CARD ACTIONS

- » Let's head back to the gradebook-web module, where we'll add support for permissions into the actions menu for each assignment.
1. Open card\_actions.jsp.
  2. Add snippet *o7-card-actions permissions* under the line `<%@ include file="/init.jsp"%>` at the very top of the file.
  3. Replace the contents of the `<liferay-ui:icon-menu>` tag with snippet *o8-liferay-ui:icon-menu*.

WWW.LIFERAY.COM

LIFERAY.

---

## EXERCISE: PERMISSION CHECKING - VIEW

1. Open `view.jsp`.
2. Replace the `<liferay-frontend:add-menu>` tag at the top of the file with snippet `09-view.jsp`.
3. Add snippet `10-init.jsp` to `init.jsp` as well for imports that will be used across multiple files.
4. See if any orphaned elements got left behind as we replaced content in the JSPs.

---

## TEST PERMISSIONS

- » If you now test permissions, you will see that they work as expected.
- » However, we are only checking permissions at the View level.
- » If an attacker were to get the URL for an action they wouldn't otherwise have access to, they would still be able to perform the action.
- » To fix this, we will add permission checks to the Controller as well.

---

## EXERCISE: PERMISSION CHECKING - CONTROLLER

1. Open GradebookPortlet.java.
2. Add snippet 11-GradebookPortlet Component to the attribute property inside @Component annotation at the top of the file.
3. Replace the doView method with snippet 12-GradebookPortlet doView.
4. See that imports are organized.
5. Save the file.

---

## EXERCISE: PERMISSION CHECKING - REMOTE SERVICE LAYER

1. Open AssignmentServiceImpl.java in the gradebook-service module.
2. Add the methods from the snippet 13-Remote Service methods to body of the file.
3. See that the imports are organized.
4. Rebuild services.
5. Refresh the gradebook modules.

---

## EXERCISE: INLINE PERMISSIONS

1. Add methods from the snippet 14-Inline permissions methods, still in `AssignmentServiceImpl`.
2. See that the import errors are resolved.
3. Rebuild services.
4. Refresh all of your modules.

---

## USING REMOTE SERVICES

- » So far, we've been directly referencing `AssignmentLocalServiceImpl`, but generally, for permission checking, we want to route all requests through the remote service class, which will then call the methods in the local service class.
- » This will enable us to consistently check permissions across all cases, rather than potentially have different cases for the same resources in the Local and Remote classes.
- » In order to do this, we'll need to redirect the traffic we're already generating from our `MVCActionCommand` and `MVCRenderCommand` classes to `AssignmentLocalServiceImpl` into `AssignmentServiceImpl` in our web module.

---

## EXERCISE: CHANGE THE CONTROLLERS TO USE REMOTE SERVICES

1. Replace the body of the `AddAssignmentMVCActionCommand.java` with snippet `15-AddAssignment`.
  - » Next, we'll replace the references of `AssignmentLocalServiceImpl` with `AssignmentLocalService` in the remaining classes.
2. Open `DeleteAssignmentMVCActionCommand.java`.
3. Edit all the references of `AssignmentLocalService` to `AssignmentService`, and use [CTRL+SHIFT+O] to resolve imports.

---

## COULD YOU REPEAT THAT PLEASE?

1. Repeat these steps for `EditAssignmentMVCActionCommand.java`.
2. Repeat the steps again for `EditAssignmentMVCRenderCommand.java`.
3. Repeat these steps for `ViewSubmissionsMVCRenderCommand.java`.

**Notes:**

1. The following notes apply to the entire section.

2. The following notes apply to the first two columns of the table.

3. The following notes apply to the last two columns of the table.

4. The following notes apply to the first three columns of the table.

5. The following notes apply to the last three columns of the table.

6. The following notes apply to the first four columns of the table.

7. The following notes apply to the last four columns of the table.

8. The following notes apply to the first five columns of the table.

9. The following notes apply to the last five columns of the table.

10. The following notes apply to the first six columns of the table.

11. The following notes apply to the last six columns of the table.

12. The following notes apply to the first seven columns of the table.

13. The following notes apply to the last seven columns of the table.

14. The following notes apply to the first eight columns of the table.

15. The following notes apply to the last eight columns of the table.

16. The following notes apply to the first nine columns of the table.

17. The following notes apply to the last nine columns of the table.

18. The following notes apply to the first ten columns of the table.

19. The following notes apply to the last ten columns of the table.



## GETTING USER DATA

Copyright ©2016 Liferay, Inc.

All Rights Reserved.

No material may be reproduced electronically or in print,  
distributed, copied, sold, resold, or otherwise exploited  
for any commercial purpose without express written  
consent of Liferay, Inc.

### OVERVIEW

- » Liferay, being a Digital Experience Platform, is very User-oriented.
- » The vast majority of Liferay's data has ties to specific Users.
- » Liferay's functionality is primarily exposed using services.
- » This means that a lot of services require some User information.
- » This also means that services exist to allow you to retrieve User information.

---

## WHY DO WE NEED TO KNOW THIS? (I)

- » Let's look at an example of why we would need to access information about a User.
- » A large company using Liferay would have an environment consisting of:
  - » Departments modeled as organizations
  - » Permissions assigned through Roles
  - » Project groups created using User Groups

---

## WHY DO WE NEED TO KNOW THIS? (II)

- » A given User may be able to view certain content based on their memberships.
- » The pages a User can access will be determined based on the Sites they have access to.
- » These Site Memberships can be controlled based on the Organizations or User Groups a User belongs to.
- » All of these checks can be done through Liferay's services.

---

## HOW CAN I ACCESS THIS INFO?

- » Liferay provides a number of APIs for accessing information in different areas.
- » The APIs most commonly used for working with User information are:
  - » User
  - » Organization
  - » Role
  - » User Group

---

## HOW DO I USE THESE APIs?

- » A very common use case for these APIs is when you want to add custom attributes to a User.
- » This also involves using the Expando API which is how Liferay exposes its custom fields.
- » The Expando Bridge is a useful gateway to this API, and simplifies a lot of its use.
- » More information on this can be found here: <https://www.liferay.com/community/wiki/-/wiki/Main/Developing+with+Expando>

---

## HOW DOES THIS WORK?

- » The basic steps for adding custom attributes for a User are as follows:
  - » Get the ExpandoBridge associated with the User model.
  - » Create an ExpandoColumn for a custom attribute (if you're creating a new one; not needed if the attribute already exists).
  - » Add a value for this column using the ExpandoValue service.

---

## EXAMPLE

- » Currently viewing an assignment doesn't allow you to easily view any associated submissions.
- » Let's go ahead and fix that.

## CREATING SUBMISSIONS

- » To simplify the training, we'll be adding a helper function.
- » This helper function will generate submissions for the assignment.
- » This gives us data that we can view easily.

## EXERCISE: CREATING SUBMISSIONS

1. Open `AssignmentLocalServiceImpl.java` in the `gradebook-service` module.
  2. Add snippet 01-addSubmissions after the `addAssignment` method.
  3. Add call to the `addBlankSubmissions()` method just before return statement.
  4. See that any import errors are resolved.
  5. Save the file.
  6. Rebuild services.
- ✓ Once this is done, create a new assignment. This new assignment will have blank submissions generated for it.

---

## EXERCISE: VIEWING SUBMISSIONS

1. Open `view_submissions.jsp` in the `gradebook-web` module.
2. Replace the contents of `<liferay-ui:search-container-row>` with snippet `02-view_submissions.jsp`.
3. Save the file.

---

## WIRING EVERYTHING TOGETHER

- » We've made changes in our View and in our Model.
- » To connect the two, we also need to make changes in our Controller.

---

## EXERCISE: CONTROLLER

1. Open `ViewSubmissionsMVCRenderCommand.java` in the `gradebook-web` module.
2. Replace the `render` method with snippet 03-render.
3. Add snippet 04-UserService, at the bottom of this file, below  
`private SubmissionLocalService _submissionLocalService;.`
4. See that any import errors are resolved.
5. Save the file.

Notes:



## Chapter 13

# Integrating Liferay Frameworks



## THE CONTENT FRAMEWORK

Copyright ©2016 Liferay, Inc.

All Rights Reserved.

No material may be reproduced electronically or in print,  
distributed, copied, sold, resold, or otherwise exploited  
for any commercial purpose without express written  
consent of Liferay, Inc.

### CONTENT IN LIFERAY

- » Liferay provides a wide variety of content:
  - » Blogs
  - » Wiki Articles
  - » Web Content Articles
  - » Documents
  - » Message Board Posts
- » These content types allow users to:
  - » Add comments
  - » Apply a business process using Workflow
  - » Search for content
  - » Add metadata like tags

## A WORTHY ASSET

- » Liferay provides a wide range of features for apps to use, including:
  - » User information
  - » Localization
- » Additional features are provided by the content framework.
- » Blogs, Documents, and other apps use features by integrating with the content framework
- » Your app can easily integrate with the content framework, allowing you to add support for:
  - » Search
  - » Workflow
  - » Tags
  - » Categories
  - » Staging

WWW.LIFERAY.COM



## SHOW ME THE ASSET

- » Support for different content types is provided by the *Asset Framework*.
- » Documents integrate with the *Asset Framework* to add new content types like *Documents*, *Document Types*, and more.
- » These content types are called *Assets* in the *Asset Framework*.
- » A *Document* in Liferay is represented by an *Asset* object in the *Asset Framework*.

WWW.LIFERAY.COM



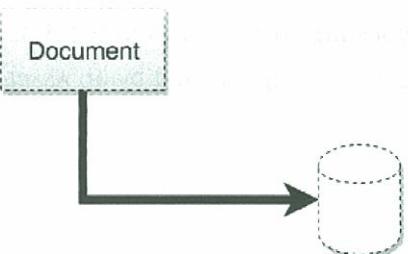
## FROM HERE TO SEARCH

- » Assets are abstract representations of your data.
- » Documents are represented by an Asset object that points back to the original document.
- » How does the *Asset Framework* know about them?



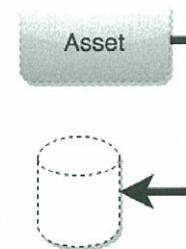
## GIVE ME A NEW NAME

- » When you upload a document in the *Documents and Media* app, a new Document model is created.



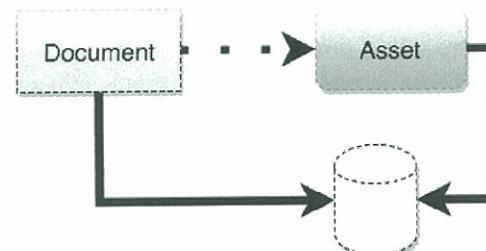
## ASSET CHILD

- When the Document is added, the app also creates an Asset object.



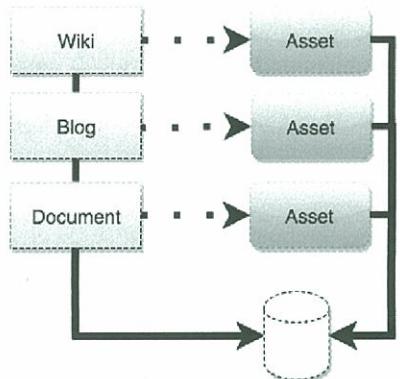
## JUST AN ALIAS

- The app takes this new Asset and points it back to the original document.
- Now the Asset Framework will know *what type* the Asset is and *where* the Asset comes from.



## AN ASSET AMONG MANY

- » This Asset lives in a long line of Assets in the database.
- » Each Asset represents a different concrete model.
- » When the *Asset Framework* needs to show the content available in Liferay, it has a ready list of Assets to pull from.



WWW.LIFERAY.COM

LIFERAY.

## DOCUMENT HOARDER

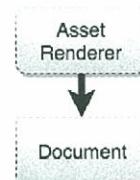
- » An *Asset Publisher* app was added to a page, showing recent uploads.
- » A User will want to see some basic information about the document:
  - » Name or title
  - » What's inside
  - » Date created or uploaded
  - » Document size
- » For the app to show this information, it needs to know more about the Asset.

WWW.LIFERAY.COM

LIFERAY.

## WHAT'S INSIDE

- » The Asset Framework doesn't know anything about Document objects.
- » There needs to be some way to get information about the Asset without knowing its type.
- » An interpreter is provided by the *Documents and Media* app to help us out.
- » This interpreter is called an *Asset Renderer*.

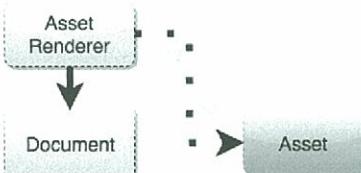


## SHOW THE ASSET

- » When the Asset Publisher asks for information about the Document Asset, the Asset Framework looks around for an Asset Renderer.
- » Documents and Media tells Liferay about its Asset Renderer, and the Asset Framework gets ahold of it.

## OPEN THE PACKAGE

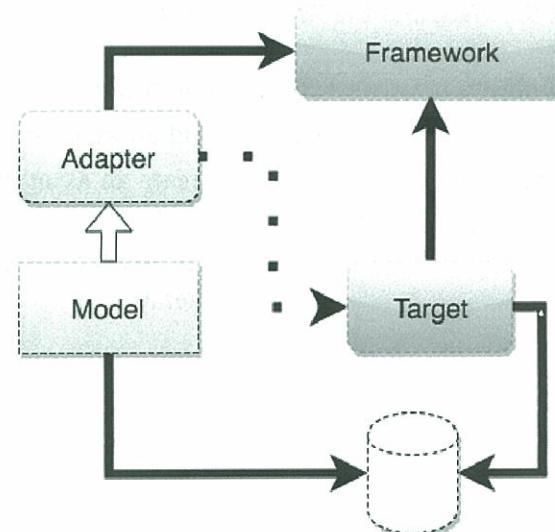
- » The document is wrapped up in a package, and given to the *Asset Framework*.
- » The package can only be opened by the *Asset Renderer*.
- » In the Documents and Media app, the *Asset Renderer* can open the package and get any information about the document.
- » The *Asset Renderer* returns all the information the Asset Publisher wants, which it shows to the User.



## THE INTEGRATION PATTERN

- » You may have noticed a pattern in the app's integration with the *Asset Framework*:
  - » A **model** was created – Document.
  - » A **target** object was created that links to the model – an Asset.
  - » An **adapter** was created that links the *model* to the *target* – the *Asset Renderer*.
  - » The *adapter* was **registered** with Liferay to make it available to the framework – the *Asset Framework*.
- » This is based on a combination of design patterns, like the *Adapter Pattern*.
- » This is a repeatable pattern that can be used to integrate with many frameworks in Liferay.

## A FITTING DIAGRAM



WWW.LIFERAY.COM

LIFERAY.

## ASSETS APPLIED

- » Let's apply this simple pattern to a new model.
- » We've created an app that generates Warg objects.
- » We'd like to have Warg be a new type of Asset.

WWW.LIFERAY.COM

LIFERAY.

---

## ROAD TO SUCCESS

- » To integrate the Warg into the *Asset Framework*:
  1. Add a **model** object.
  2. Add a **target** object for the *model*.
  3. Create an **adapter** for the *target*.
  4. **Register** the *adapter* with Liferay.
  1. Add a **Warg**.
  2. Add an **Asset** for the Warg.
  3. Create an **Asset Renderer** for the Warg.
  4. **Register** the *Asset Renderer* with Liferay.

---

## GETTING WARGY WITH IT

- » This is a simple example with a fictitious entity, but the same rules apply in your own apps.
- » You can easily add new content types in the *Asset Framework*.
- » Integrating with the *Asset Framework* means a lot of additional features can be used.
- » Once you integrate with *Assets*, integrating with other frameworks is similar.

## APPLYING THE PATTERN

- ▶ Liferay frameworks provide a way to integrate new features into your app.
- ▶ These features use the same *integration pattern*.
- ▶ Although the details will vary, the basic mechanism and goal is the same.

WWW.LIFERAY.COM



Notes:



## ASSETS

Copyright ©2016 Liferay, Inc.  
All Rights Reserved.

No material may be reproduced electronically or in print,  
distributed, copied, sold, resold, or otherwise exploited  
for any commercial purpose without express written  
consent of Liferay, Inc.

## THE RIGHT STUFF

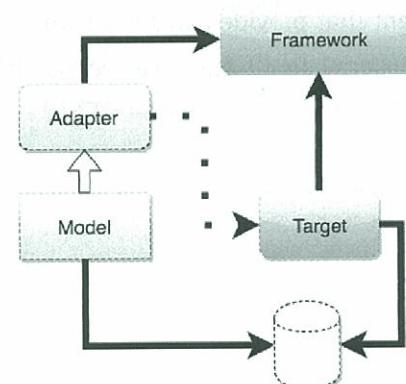
- » Assets are the cornerstone of the *Asset Framework*.
- » Using the *integration pattern*, we can abstract any model as an *Asset*.
- » Once we have *Assets*, we can integrate easily with:
  1. Search
  2. Tags
  3. Categories
  4. Workflow
  5. Comments

## ASSET NOT FOUND

- » We have an app that would benefit from integrating with the *Asset Framework*.
- » The Gradebook app provides a new model that could be added as a content type (*Asset*) in the framework:
  - » Assignments
- » With Assignments added as a new *Asset*, we could display them on a landing page, showing a student new assignments as they come up.

## FULLY INTEGRATED

- » To integrate our Assignment model with the *Asset Framework*, we'll need to follow the *integration pattern*.



## THE SETUP

- » Let's apply the pattern to our Gradebook app:
  - 1. Add a **model** object.
  - 2. Add a **target** object for the *model*.
  - 3. Create an **adapter** for the *target*.
  - 4. **Register** the *adapter* with Liferay.
  - 1. Add an **Assignment**.
  - 2. Add an **AssetEntry**.
  - 3. Create an **AssetRenderer**.
  - 4. **Register** the *Asset Renderer* with Liferay using the *GradebookPortlet* component.

## THE FOLLOW-THROUGH

- » To add an **Asset** when we create **Assignments**, we'll need to modify our Model layer – the service.
- » To create an **Asset Renderer** (our adapter or interpreter), we'll need to create a component that provides an **AssetRenderer**.
- » We'll need to tell Liferay about our **Asset Renderer** by setting a property on our portlet component – *GradebookPortlet*.

---

## CHECKING ASSET READINESS

- » To integrate properly into the *Asset Framework*, we need to make sure we provide the required fields to identify and track our *Assets*:
  - » **UUID**: a unique identifier that's designed to be independent of the object's type or even Liferay instance.
  - » **Status**: a series of status fields need to be available to let the *Asset Framework* know if an *Asset* is published or not.
  - » **Owner**: information about the User that made the *Asset*, for permissioning.
- » Let's check and see if we've included the necessary fields in our model.

---

## EXERCISE: EXAMINING MODEL FIELDS

- » All of our model's fields are outlined in `service.xml`:
  1. Go to the project `modules/gradebook/gradebook-service` in your Liferay Workspace.
  2. Open the file `service.xml`.

## EXERCISE: FIND MISSING FIELDS

1. See and compare our columns in Assignment with our additional Asset columns in our service.xml:

- UUID
- userId
- userName
- status
- statusByUserId
- statusByUserName
- statusDate
- createDate
- modifiedDate

2. Find the fields we already have.
3. Type the fields we are missing.

## RECON: WHAT WE HAVE

- Let's take stock of what's already in the model:
  - **UUID:** this is already provided by setting the `uuid` attribute of the `entity` node to true:

```
<entity name="Assignment" uuid="true" local-service="true"  
       remote-service="false">
```
  - **Owner:** we've already included the `userId` and `userName` fields:

```
<column name="userId" type="long" />  
<column name="userName" type="String" />
```
  - **Dates:** we're already tracking create and modified dates:

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

---

## DATA THAT I USED TO KNOW

- » To complete the information the *Asset Framework* needs, we'll include:
  - » status
  - » statusByUserId
  - » statusByUserName
  - » statusDate
- » We'll need to also add a finder for status and make sure it's available to the *Asset Framework*.
- » To avoid any bad data, we'll also verify that all fields are being set up properly.

---

## EXERCISE: EXPOSING THE STATUS FINDER

- » Along with our new status fields, we added a new finder.
  - » We'll need to make sure we have access to this through the service.
1. Open `AssignmentLocalServiceImpl`.
  2. Insert the snippet `o3-getAssignmentsByStatus` at the bottom of the class.
  3. See that imports are resolved if they need to be.
  4. Build services to add the new method to your service.
  5. Refresh your Gradle projects.
- » Now the *Asset Framework* can check, modify, and list assignments by status.

---

## CHARTING PROGRESS

- » Let's see where we are on the road to integration:
  - ✓ Modify Assignment to contain the proper fields.
  - » Add an Assignment.
  - » Add an AssetEntry.
  - » Create an AssetRenderer.
  - » Register the Asset Renderer with Liferay using the GradebookPortlet component.

---

## ADDING THE MODEL

- » Since we've already completed the base app, we don't need additional code to add our Assignment model.
- » We've added some new fields, though, so we should take care to set the relevant data in our add and edit actions.

## EXERCISE: CHECKING THE SERVICE

1. Open GradebookPortletUtil.
  2. See if our assembleAssignment() method sets important fields like:
    - userId
    - createDate
    - groupId
- Everything looks okay here, so let's continue on to adding the Asset.

WWW.LIFERAY.COM

LIFERAY.

## BLUEPRINT CHECK

- Let's see where we are in fulfilling the blueprint:
- ✓ Modify Assignment to contain the proper fields.
  - ✓ Add an Assignment.
  - Add an AssetEntry.
  - Create an AssetRenderer.
  - Register the Asset Renderer with Liferay using the GradebookPortlet component.

WWW.LIFERAY.COM

LIFERAY.

## EXERCISE: ADDING AN ASSET

- » We'll need to make sure we add and update an Asset when we *add*, *update*, or *delete* an assignment.
1. Open `AssignmentLocalServiceImpl` in the `gradebook-service` module.
  2. Insert the snippet *o4-Update Asset Method* to create a helper method to add and update Assets.
  3. Insert the method call `updateAsset(assignment, serviceContext)` in the `addAssignment()` method. (Just before `addBlankSubmissions(groupId, assignmentId);` method call.)

WWW.LIFERAY.COM



## EXERCISE: UPDATING AND DELETING AN ASSET

- » We've created an asset; let's also implement our methods for updating and deleting an asset.
1. Add the `updateAssignment()` method with the snippet *o5-Update Assignment Method*.
  2. Add the snippet *o6-Delete AssetEntry call* after `deleteResource()` call at `deleteAssignment()` method.
  3. See if you need to resolve any import errors.
  4. Refresh your gradle projects using *Gradle → Refresh*.
  5. Save the service.

WWW.LIFERAY.COM



## ON THE ROAD AGAIN

- » Let's see where we are in the great journey:
  - ✓ Modify Assignment to contain the proper fields.
  - ✓ Add an Assignment.
  - ✓ Add an AssetEntry.
    - » Create an AssetRenderer.
    - » Register the Asset Renderer with Liferay using the GradebookPortlet component.

## A FACTORY FOR CONTENT

- » To implement our adapter – the Asset Renderer – we'll need to implement another pattern: the factory.
- » The Asset Framework asks for Asset Renderers to be created by an Asset Renderer Factory.

---

## EXERCISE: CREATING THE ASSET RENDERER FACTORY

» To create an *Asset Renderer*, we'll need to first implement an *Asset Renderer Factory*, followed by an *Asset Renderer*.

1. Find the *gradebook-web* project in your Project Explorer.
2. Right-click on the the *source folder*.
3. Create a new *Java Class*:
  - » Package path: `com.liferay.training.space.gradebook.asset`
  - » Class name: `AssignmentAssetRendererFactory`
  - » Superclass: `BaseAssetRendererFactory`

---

## EXERCISE: FINISHING THE FACTORY

1. Click *Finish*.
2. Replace content of the file with the snippet *07-Assignment Asset Renderer Factory*.
3. Save the file.

## PRODUCING ASSET RENDERERS

- » The *Asset Renderer Factory* provides some useful information to the *Asset Framework*:
  - » Type information
  - » Retrieving the Asset's target model
  - » Instantiating the *Asset Renderer*
- » Once we have an *Asset Renderer*, we can provide content information to the *Asset Framework*.

## EXERCISE: CREATING AN ASSET RENDERER

1. Right-click on the *source folder* in the gradebook-web project.
2. Create a new *Java Class*:
  - » **Package path:** com.liferay.training.space.gradebook.asset
  - » **Class name:** AssignmentAssetRenderer
  - » **Superclass:** BaseAssetRenderer
3. Click *Finish*.
4. Replace content of the file with the snippet *o8-Assignment Asset Renderer*.

## THE RENDER FARM

- » An Asset Renderer performs the important function of adapting a model to the Asset Framework.
- » It provides information about its *type*.
- » It provides *summary* information to the framework for display.
- » It controls access to the Asset using permission checks:
  - » `hasEditPermission()`
  - » `hasViewPermission()`
- » It can also provide a way to display more information, such as an *abstract* or *full content* display of the Asset.

## BEYOND THE SURFACE

- » One easy way to provide access to templates for *abstract* and *full content* views is to use a JSP Asset Renderer.
- » We simply provide basic information about the Asset and offload the rest to a JSP.
- » By setting an attribute on the request, we have easy access in our JSPs to the model.

---

## EXERCISE: RENDERING ABSTRACT JSPS

1. Open the resources/META-INF/resources folder in the gradebook-web project.
2. Create a new folder called asset.
3. Create a file called abstract.jsp.
4. Insert the contents of the snippet *9 Abstract JSP* into the file.
5. Save the file.

---

## EXERCISE: RENDERING FULL CONTENT JSPS

1. Create a file called full\_content.jsp.
  2. Insert the contents of the snippet *10 Full Content JSP* into the file.
  3. Save the file.
- » These JSPs provide a view into our assignments in the Asset Publisher app.
- » When the Asset Publisher asks for an abstract or full content, the *Asset Renderer* returns the appropriate JSP.

## LAND, AHOY!

» Let's see where we are on The Quest:

- ✓ Modify `Assignment` to contain the proper fields.
- ✓ Add an `Assignment`.
- ✓ Add an `AssetEntry`.
- ✓ Create an `AssetRenderer`.
- » Register the `Asset Renderer` with Liferay using the `GradebookPortlet` component.

## EXERCISE: REGISTERING THE ASSET RENDERER

» To finish the integration, we need to let Liferay know about our `Asset Renderer`.

» The Module Framework provides an easy way to register *components* that implement specific functionality.

1. Open the `AssignmentAssetRendererFactory` class.
2. Insert the snippet `11 Asset Renderer Component` at the top of the class.
3. Save the file.

## EXERCISE: EXPORTING THE PACKAGE

- » In order for other modules to *import* classes from our asset package, we need to add the package to our *export packages*.

1. Open the `bnd.bnd` file.
2. Click the green *plus* icon in the *Export Packages* area of the form.
3. Choose the `com.liferay.training.space.gradebook.asset` package.

✓ Save the file.

## NEARING SHORE

- » Let's see where we are in our voyage:
  - ✓ Modify `Assignment` to contain the proper fields.
  - ✓ Add an `Assignment`.
  - ✓ Add an `AssetEntry`.
  - ✓ Create an `AssetRenderer`.
  - ✓ Register the `Asset Renderer` with Liferay using the `GradebookPortlet` component.

---

## TESTING THE WATERS

- » All we need to do now is *install* the new version of our app!
- » **Note:** Due to the model and service changes, you may want to delete prior Assignments before installing the new version of the app.
- » Once installed, add an *Asset Publisher* app to the page.
- » Add or update an Assignment.
- » Does the Assignment show up in the *Asset Publisher*?
- » The *Asset Publisher* displays Assets, but only Assets that have been *indexed*.
- » To have your model appear in the *Asset Publisher*, you'll need to integrate with the Search Framework.

Notes:



## SEARCH AND INDEXING

Copyright ©2016 Liferay, Inc.  
All Rights Reserved.

No material may be reproduced electronically or in print,  
distributed, copied, sold, resold, or otherwise exploited  
for any commercial purpose without express written  
consent of Liferay, Inc.

### OVERVIEW

- » Liferay stores most of its information in its database.
- » It makes this information available to be searched using a search engine.
- » This is done because directly querying the database can be a very expensive operation.

---

## HOW DOES LIFERAY'S SEARCH ENGINE WORK?

- » Liferay indexes this information using a search engine.
- » The search engine's job is to make searching through information fast and efficient.
- » It stores this information in the form of *documents*.

---

## WHAT IS A DOCUMENT?

- » *Documents* in a search index are objects that represent an entity that is stored in Liferay's database.
- » They often contain searchable fields from many different tables in a database.
- » To be useful, *documents* should be created *before* a search is performed, so that they can reduce the time taken to provide results.

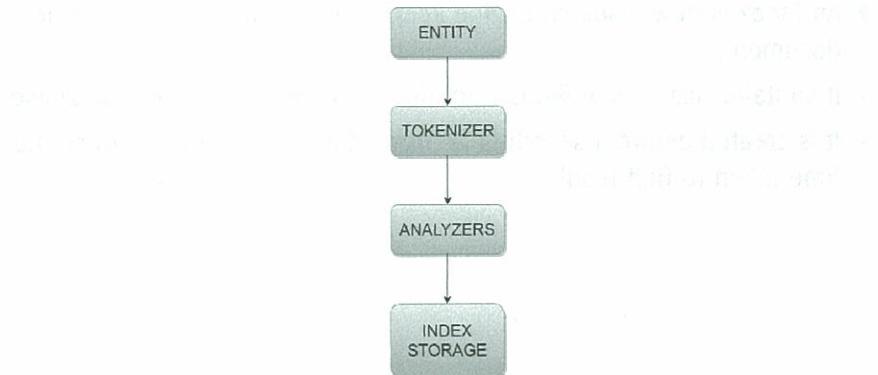
## WHAT IS AN INDEX?

- » An index is how a search engine keeps track of the content of your document.
- » It contains searchable fields from many different tables in a database.
- » It is created before a search is performed and is meant to reduce the time taken to find results.

## HOW DOES INDEXING WORK?

- » During indexing, a document is sent to the search engine.
- » This document contains fields of various types (string, integer, date, etc.).
- » The search engine processes each field within the document.

## HOW INDEXING WORKS



## DOCUMENT PROCESSING

- ▶ For each field, the search engine determines whether it should store the field as-is, or if further processing is needed.
- ▶ For fields requiring further processing, the search engine first tokenizes the values.
- ▶ Following tokenization, the search engine passes each token through a series of analyzers.

---

## ANALYZERS

- » Analyzers perform different functions.
- » Some remove common words or stop words (e.g., "the", "and", "or", etc.), while others perform operations like lowercasing all characters.
- » Once the search engine passes each token through analyzers, it then adds it to its index.

---

## ANALYZER EXAMPLE - CODE

- » `index-settings.json` in the Elasticsearch module contains the following:

```
{  
  "analysis": {  
    "analyzer": {  
      "keyword_lowercase": {  
        "tokenizer": "keyword",  
        "filter": "lowercase"  
      }  
    },  
    ...  
  }  
}
```

---

## ANALYZER EXAMPLE - EXPLAINED

- » The intent of the analyzer `keyword_lowercase` above is:
  - » To lowercase all the tokens (`"filter": "lowercase"`)
  - » Instead of tokenizing individual words, it treats the entire string being sent in as a single token (`"tokenizer": "keyword"`)

---

## SEARCHING

- » When a User searches for something, he or she is sending a search query and obtaining results from the search engine.
- » These results are known as *hits*.
- » A search query can have both *queries* and *filters*.

---

## FILTERS

- » A *filter* asks a question that can be answered with either a "Yes" or "No" for every document.
- » *Filters* are used for fields that contain exact values.
- » An example of a filter would be "I want all content created in 2016."

---

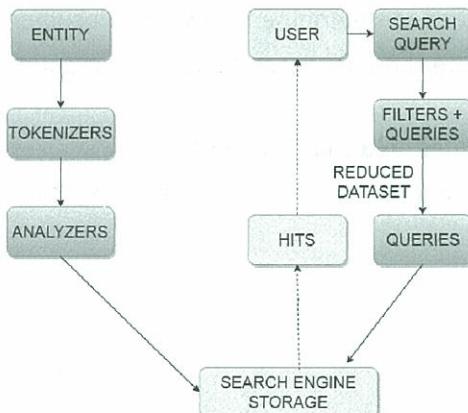
## QUERIES

- » A *query* also tries to determine how well a document matches.
- » *Queries* calculate the relevance of each document and assign each one a score.
- » This is best suited to a full-text search.
- » An example of a query would be "I want all articles about rockets."

## MAPPING DEFINITIONS

- Mappings control how a search engine processes a given field.
- They describe the fields that a particular type of document should have, and how they should be processed.
- For example, field names ending in "es\_ES" should be processed as Spanish.
- The Elasticsearch mapping is in JSON and can be found in the `portal-search-elasticsearch` module.

## HOW IT ALL FITS TOGETHER



## INDEXING EXAMPLE: BLOGS ENTRY - OVERVIEW

- » Consider Liferay blog posts. When a User creates a blog post in Liferay, how is that blog post indexed?
- » When a new blog post is published, Liferay checks if there's a registered indexer class that knows how to index blog entries.
- » For blog entries, the Liferay Blogs Service module contains a `BlogsEntryIndexer` class that fits the bill.
- » This class registers itself as an indexer in Liferay's OSGi runtime by specifying `service = Indexer.class` in a `@Component` annotation on the class declaration.
- » This class extends `BaseIndexer<BlogsEntry>`, which is one of the classes generated by Service Builder.

## INDEXING EXAMPLE: BLOGS ENTRY - CODE

- » The `doGetDocument` method is very important, as it's where developers specify which fields should be indexed.
- » Here's the `doGetDocument` method of `BlogsEntryIndexer`:

```
@Override protected Document doGetDocument(BlogsEntry blogsEntry)
throws Exception \{ Document document =
getBaseModelDocument(CLASS\_NAME, blogsEntry);

document.addText(Field.CAPTION, blogsEntry.getCoverImageCaption());
document.addText(
    Field.CONTENT, HtmlUtil.extractText(blogsEntry.getContent()));
document.addText(Field.DESCRIPTION, blogsEntry.getDescription());
document.addDate(Field.MODIFIED_DATE, blogsEntry.getModifiedDate());
document.addText(Field.SUBTITLE, blogsEntry.getSubtitle());
document.addText(Field.TITLE, blogsEntry.getTitle());

return document;
}
```

## INDEXING EXAMPLE: BLOGS ENTRY - CODE REVIEW

- » The `doGetDocument` method of `BlogsEntryIndexer` first invokes `getBaseModelDocument`, which creates and returns a base document.
- » The base document includes fields which are common to all Liferay entities.
- » After getting the base document, the `doGetDocument` adds these additional fields to the document before returning the document: `caption`, `content`, `description`, `modifiedDate`, `subtitle`, and `title`.
- » Remember that the configured document mappings (e.g., in `liferay-type-mappings.json` for Elasticsearch) affect the values that are stored in a document's fields.

## SEARCH EXAMPLE: BLOGS ENTRY (I)

- » Liferay indexers play an important role both in defining the structure of documents that are written to the index and in searching for documents in an index.
- » Each registered indexer class has a chance to add search terms to a search query or filter before the search runs.
- » The `postProcess*` methods, such as `postProcessSearchQuery` and `postProcessBooleanFilter`, in `BlogsEntryIndexer` do this.

## SEARCH EXAMPLE: BLOGS ENTRY (II)

- » The `postProcessSearchQuery` method in `BlogsEntryIndexer`, for example, checks to see if any keywords have been specified.
- » A keyword is the name of a field to search against.
- » If no keywords have been specified (this happens often since Users typically specify only values but not keywords), then the `description`, `title`, and `userName` fields are added to the search query.
- » The presence of similar methods in many different indexer classes is why a long list of fields is searched against when you search for a term in Liferay.

## SEARCH ENGINE FEATURES

- » Search engines make your search results more relevant and useful.
- » They can assign scores to search results and sort results by relevance.
- » Search engines can also provide algorithms such as "More Like This", geolocation, and faceting of results.

## ELASTICSEARCH

- » Elasticsearch is a popular and efficient open-source search engine.
- » Although Liferay includes an embedded Elasticsearch server, it can also be configured to use a remote search server instead.
- » This is the recommended approach for a production environment.
- » Liferay can also be configured to use Solr as its remote search server.

## LIFERAY SEARCH INFRASTRUCTURE

- » Liferay wraps Elasticsearch's native API with an additional layer called its *search infrastructure*.
- » The *search infrastructure* allows for:
  - » Ensuring that documents are indexed with fields required by Liferay (eg: companyId and groupId)
  - » Ensuring that search queries are automatically filtered (eg: by groupId, staging status, etc.)
  - » Adding capabilities such as permission checking and hit summaries to be displayed
  - » This is done through Liferay's search API.

## LIFERAY'S SEARCH API

- » Liferay's Search API allows Users to build a search query or filter, execute it, and obtain hits that match.
- » Most search engines do not distinguish between queries and filters at the API level.
- » Liferay's Search API provides distinct APIs for each of these.

WWW.LIFERAY.COM



## QUERIES V. FILTERS: DAWN OF SEARCH

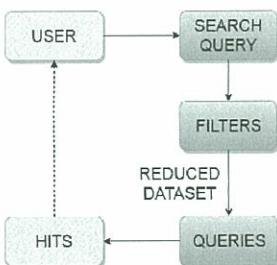
- » A filter may ask if the status field is set to "staging" or "live".
- » A query may ask if the document contains the words "Liferay", "Content", and/or "Management".
- » A query also calculates the relevancy of each matching document's content based on the search terms.

WWW.LIFERAY.COM



## CAN QUERIES AND FILTERS WORK TOGETHER?

- » Filters are faster than queries since their results can be cached.
- » Liferay uses a combination of both filters and queries.
- » Filters are used to minimize the number of documents a query must search through.



WWW.LIFERAY.COM

LIFERAY.

## USING LIFERAY'S API TO SEARCH

- » The `doSearch` methods in Liferay's `ElasticSearchIndexSearcher` is a good example of how to run a search and obtain results.
- » This method leverages Elasticsearch's API and returns a `SearchResponse`.
- » It is invoked transparently by Liferay's built-in search.
- » As a developer, understanding how it works can greatly aid in writing a custom search.

WWW.LIFERAY.COM

LIFERAY.

## ELASTICSEARCHINDEXSEARCHER - DOSEARCH

```
public class ElasticsearchIndexSearcher extends BaseIndexSearcher {  
  
    protected SearchResponse doSearch(  
        SearchContext searchContext, Query query) {  
  
        Client client = _elasticsearchConnectionManager.getClient();  
        QueryConfig queryConfig = query.getQueryConfig();  
        SearchRequestBuilder searchRequestBuilder = client.prepareSearch()  
            .getSelectedIndexNames(queryConfig, searchContext));  
        ...  
        queryBuilder QueryBuilder = _queryTranslator.translate(  
            query, searchContext);  
        ...  
        searchRequestBuilder.setQuery(QueryBuilder);  
        SearchResponse SearchResponse = searchRequestBuilder.get();  
        ...  
        return SearchResponse;  
    }  
  
}
```

WWW.LIFERAY.COM

LIFERAY.

## INTEGRATING SEARCH - OVERVIEW

- » We would like to integrate search capabilities into our application.
- » This will allow us to search through all received submissions.
- » To do this, we will be following the *integration* pattern.

WWW.LIFERAY.COM

LIFERAY.

## EXERCISE: THE INTEGRATION PATTERN

» Let's apply the pattern to our Gradebook app:

1. Add a *model* object.
2. Add a *target* object for the *model*.
3. Create an *adapter* for the *target*.
4. Add the *adapter* to Liferay.
1. Add an Assignment.
2. Add annotations to mark the *model* as Indexable.
3. Create an AssignmentIndexer.
4. Add the AssignmentIndexer to Liferay.

## EXERCISE: INTEGRATING SEARCH - ADAPTER

1. Create a new package in the `src/main/java` folder of the gradebook-web module called `com.liferay.training.space.gradebook.search`.
2. Create a new class in this package called `AssignmentIndexer.java` that extends `BaseIndexer`.
3. Replace content of the file with snippet 02-AssignmentIndexer name.

---

## EXERCISE: INTEGRATING SEARCH - TARGET

1. Open AssignmentLocalServiceImpl.java in the gradebook-service module.
2. Add snippet 03-reindex above addAssignment and updateAssignment.
3. Add snippet 04-delete above deleteAssignment.
4. See that you resolve any import errors.
5. Rebuild Services.

---

## EXERCISE: WRAPPING IT ALL UP

1. Open bnd.bnd.
2. Add the search package com.liferay.training.space.gradebook.search into the Export-Package directive.

---

## CHECKPOINT (I)

- » A *search index* is a collection of documents.
- » A *document* is a Java object that represents an entity that has been saved to Liferay's database.
- » Documents often contain fields from multiple tables in Liferay's database.
- » A *field* is a specific attribute of a document.
- » For example, a user document would have Screen Name, Email Address, First Name, and Last Name (and many other) fields.

---

## CHECKPOINT (II)

- » A *search query* asks a yes or no question for each document in a search index AND calculates a relevancy score for each matching document.
- » A *search filter* simply asks a yes or no question for each document in a search index.
- » The term *hits* refers to a list of search results.
- » In Liferay, a *Hits* object contains a list of documents that match a particular search.

---

## CHECKPOINT (III)

- Now you should be able to also use Liferay faceted search to search your entities.

Notes:

---

## WHEN TO CONVERT?

- » If you have a large application with hundreds or thousands of lines of code, and many developers working on it concurrently
- » Splitting into modules will help provide agility by allowing for more frequent releases.
- » If your application has parts that you want to consume from elsewhere
- » Because modules make dependency management easier than traditional WAR-style applications, this will make sharing just a portion of your application very easy, and allow you to reuse existing code.

---

## WHEN NOT TO CONVERT?

- » If you have a portlet that's JSR-286 compatible and you want to retain the ability to deploy it into another portlet container.
- » If you're using a framework heavily tied to the more traditional Java EE programming model

## WHY SHOULD I CONVERT?

- » Going modular allows you to split up your application into smaller sections that are a lot easier to manage.
- » This model also allows for incremental release cycles, as modules can be updated independently of each other.
- » For example, if a JSP needs to be changed due to a security issue, the client module can be updated without needing to touch the persistence module.
- » Dependency management is simplified, as the OSGi Container explicitly lists all of its dependencies and will refuse to run unless they are satisfied, eliminating obscure run time errors that may otherwise occur.
- » Going modular also allows for better integration into the Liferay platform, and makes it easier to communicate with other modules installed in the OSGi Container.

## CAN I DEVELOP USING WARS?

- » WARs, by definition, are not modular, since they don't dynamically manage their dependencies and imports/exports.
- » In Liferay DXP, we're moving towards greater modularity with the new OSGi Container.
- » To achieve this, all plugins are now deployed as modules.

## LIFERAY'S COMPATIBILITY LAYER

- » Liferay has a compatibility layer that still allows WARs to be deployed and automatically converts them to modules.
- » This compatibility layer is extensive and will work out-of-the-box for most portlets.
- » Although extensive, the compatibility layer does not entirely replicate the legacy behavior of deploying plugins to your appserver.

WWW.LIFERAY.COM

 LIFERAY

## HOW DOES THE COMPATIBILITY LAYER DIFFER?

- » The goal of the compatibility layer is to support application server independence and ultimately allow for greater modularity.
- » To achieve this, some of the configuration common to traditional webapps is abstracted away.
- » This means that the compatibility layer won't provide the same level of control as a module will.

WWW.LIFERAY.COM

 LIFERAY

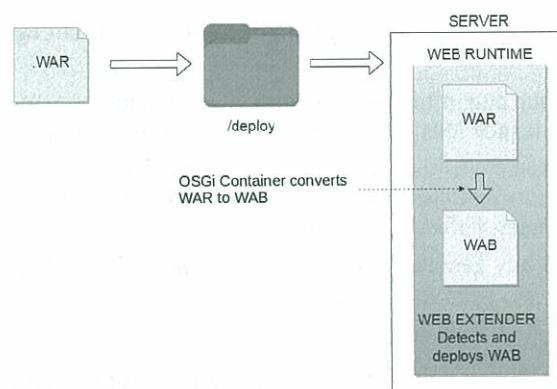
## TRADITIONAL WAR DEPLOYMENT



WWW.LIFERAY.COM

LIFERAY.

## WAR DEPLOYMENT IN THE OSGI CONTAINER



WWW.LIFERAY.COM

LIFERAY.

## WAR DEPLOYMENT IN THE OSGI CONTAINER EXPLAINED

- » In the OSGi Container's compatibility layer, all web applications are deployed as Web Application Bundles (WABs).
- » A WAB is simply a WAR file with a manifest added, telling the OSGi Container how to deploy the application. This manifest also allows for headers to be added, which allow support for JSPs, tag library definitions, etc.
- » The compatibility layer converts a WAR file to a WAB upon deployment.
- » The web extender module of the OSGi Container detects and deploys this WAB file.

## HOW CAN MY PLUGINS RUN ON LIFERAY 7?

- » To get traditional WAR-style applications to run on Liferay 7, all you need to do is:
  - » Copy your plugin to the new version of the Plugins SDK.
  - » Update legacy APIs that have been converted to modules.
    - » A comprehensive list of these can be found here:  
[https://dev.liferay.com/develop/reference/-/knowledge\\_base/7-0/classes-moved-from-portal-service-jar](https://dev.liferay.com/develop/reference/-/knowledge_base/7-0/classes-moved-from-portal-service-jar)
    - » Remove any generated JAR files from your project's /lib folder. These will already exist in Liferay.
    - » Deploy your WAR file.
  - » This process is strongly recommended even if you plan on converting your application to modules, as it helps you isolate and fix any breaking API changes to make the transition to modules go more smoothly.
  - » Other breaking changes can also be found at [https://github.com/liferay/liferay-portal/blob/master/readme/7.0/BREAKING\\_CHANGES.markdown](https://github.com/liferay/liferay-portal/blob/master/readme/7.0/BREAKING_CHANGES.markdown)

## HOW DO I CONVERT MY PLUGINS TO MODULES?

- » We're glad you've decided to convert your plugins to modules!
- » Here's the best way to do so:
  - » Upgrade your existing application using the steps previously mentioned.
  - » Determine what modules to create (web, api, service).
  - » Define manifest attributes and dependencies.
  - » Update any Liferay-specific configuration files.
  - » Run Service Builder to generate code for your application's service and API modules.
- » More information on this process can be found at  
*[https://dev.liferay.com/develop/tutorials/-/knowledge\\_base/7-0/modularizing-an-existing-portlet](https://dev.liferay.com/develop/tutorials/-/knowledge_base/7-0/modularizing-an-existing-portlet)*

Notes:

