

BACK-END DEVELOPMENT

Creating Real-World Applications

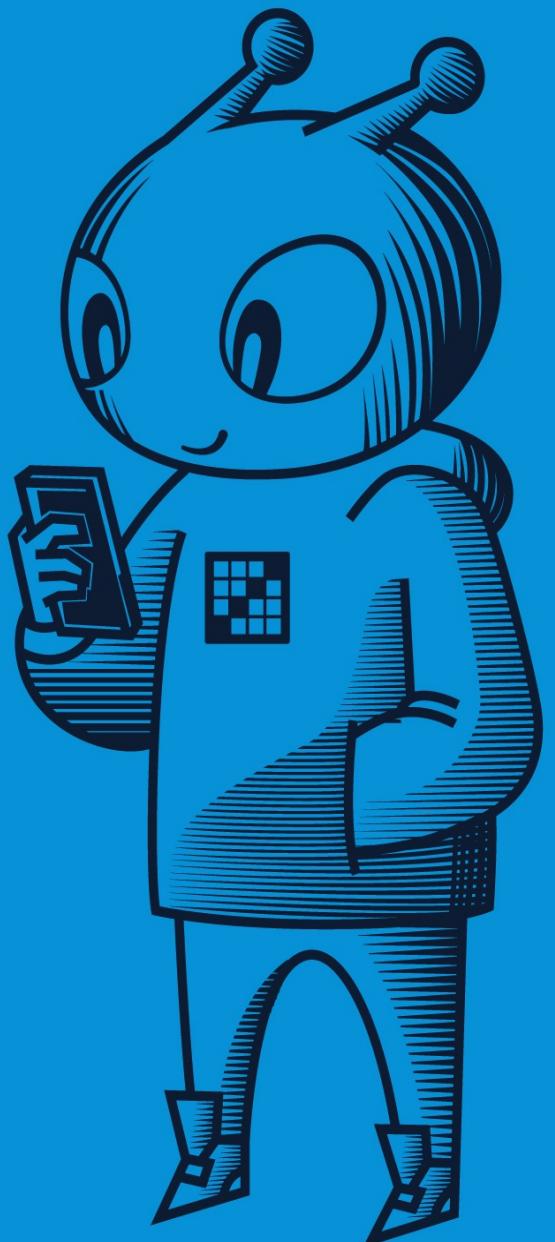


Table of Contents

| | |
|--|---------|
| Introduction | 1.1 |
| Chapter 1: Setting up the Development Environment | 1.2 |
| Set Up the Development Environment | 1.2.1 |
| Exercises: Set up the Development Environment in Linux | 1.2.1.1 |
| Exercises: Set up the Development Environment in Mac | 1.2.1.2 |
| Exercises: Set up the Development Environment in Windows | 1.2.1.3 |
| Chapter 2: OSGi Basics | 1.3 |
| Introduction | 1.3.1 |
| Basic Concepts | 1.3.2 |
| Exercises: Hello OSGi! | 1.3.2.1 |
| Bundles | 1.3.3 |
| Exercises: Change Lifecycle State of an OSGi Bundle | 1.3.3.1 |
| Components and Services | 1.3.4 |
| Exercises: Create an OSGi Service Using Declarative Services and Bndtools | 1.3.4.1 |
| Sharing Features | 1.3.5 |
| Exercises: Sharing Features with Export-Import | 1.3.5.1 |
| Framework Architecture | 1.3.6 |
| OSGi Benefits | 1.3.7 |
| Chapter 3: Liferay OSGi Container | 1.4 |
| Liferay OSGi Container | 1.4.1 |
| Working with Liferay Workspace | 1.4.2 |
| Exercises: Set up Liferay Workspace and Liferay Portal | 1.4.2.1 |
| Introducing Liferay Modules | 1.4.3 |
| Exercises: Create a Custom Form Field Using the Form Field Module Template | 1.4.3.1 |
| Manage Module Dependencies | 1.4.4 |
| Chapter 4: Managing OSGi Bundles | 1.5 |
| Manage OSGi Bundles with Gogo Shell | 1.5.1 |
| Exercises: Practice Gogo Shell Basic Commands | 1.5.1.1 |
| Exercises: Create a Custom Gogo Shell Command | 1.5.1.2 |
| Using Felix Web Console | 1.5.2 |
| Exercises: Use Felix Web Console to Find Out the Blogs Web Module Version | 1.5.2.1 |
| Exercises: Use Felix Web Console to Locate MVC Render Command Components for the Blogs Portlet | 1.5.2.2 |
| Chapter 5: Working with Portlet Modules | 1.6 |
| Java Standard Portlet | 1.6.1 |
| Exercises: JSR 286 Portlet | 1.6.1.1 |
| Working With Liferay Portlet Modules | 1.6.2 |
| Exercises: Create a Liferay MVC Portlet | 1.6.2.1 |

| | |
|--|----------|
| Chapter 6: Develop a Real-World Application | 1.7 |
| Overview | 1.7.1 |
| Create the Service Layer | 1.7.2 |
| Exercises: Using the Service Builder | 1.7.2.1 |
| Exercises: Implement AssignmentLocalServiceImpl | 1.7.2.2 |
| Exercises: Implement SubmissionLocalServiceImpl | 1.7.2.3 |
| Exercises: Implement and Test Remote Services | 1.7.2.4 |
| Implement Access Control | 1.7.3 |
| Exercises: Implement Gradebook Permissioning Support | 1.7.3.1 |
| Create the Presentation Layer | 1.7.4 |
| Exercises: Create the Gradebook Web Module | 1.7.4.1 |
| Exercises: Add Localization Resources | 1.7.4.2 |
| Exercises: Implement Portlet Actions | 1.7.4.3 |
| Exercises: Implement Portlet Permissions | 1.7.4.4 |
| Exercises: Using JSP Tag Libraries | 1.7.4.5 |
| Exercises: Implement Portlet Validation | 1.7.4.6 |
| Exercises: Add CSS Resources | 1.7.4.7 |
| Exercises: Add JavaScript Resources | 1.7.4.8 |
| Make the Application Configurable | 1.7.5 |
| Exercises: Make Gradebook Configurable | 1.7.5.1 |
| Integrate With Liferay Frameworks | 1.7.6 |
| Exercises: Integrate Gradebook with Portal Search | 1.7.6.1 |
| Exercises: Integrate Gradebook with Liferay Asset Framework | 1.7.6.2 |
| Exercises: Enable Workflows for Gradebook Assignments | 1.7.6.3 |
| Integrate With External Systems | 1.7.7 |
| Exercises: Publish a REST Service for the Gradebook Application | 1.7.7.1 |
| Logging | 1.7.8 |
| Exercises: Implement Gradebook Logging | 1.7.8.1 |
| Testing | 1.7.9 |
| Debugging | 1.7.10 |
| Exercises: Debug the Gradebook | 1.7.10.1 |
| Managing Deployment Issues | 1.7.11 |
| Exercises: Troubleshoot Failing Module Deployment | 1.7.11.1 |
| Chapter 7: Liferay Platform Architecture Overview | 1.8 |
| Core Technologies and Standards | 1.8.1 |
| Chapter 8: Customize the User Interface | 1.9 |
| Liferay User Interface Technologies Overview | 1.9.1 |
| Change the Default Application UI With Application Display Templates | 1.9.2 |
| Exercises: Create a Media Gallery Application Display Template | 1.9.2.1 |
| Customize the Application JSPs | 1.9.3 |

| | |
|--|-------------|
| Exercises: Customize the Announcements Portlet Using a JSP Fragment Module | 1.9.3.1 |
| Exercises: Customize the Blogs Portlet JSP using a Portlet Filter | 1.9.3.2 |
| Chapter 9: Extend Liferay's Schema | 1.10 |
| Introducing Custom Fields | 1.10.1 |
| Dynamically Add Custom Fields With Expando API | 1.10.2 |
| Exercises: Extend Liferay's User Profile using Expando and Lifecycle Actions | 1.10.2.1 |
| Chapter 10: Customize the Service Layer | 1.11 |
| Override and Extend Liferay Services with Service Wrappers | 1.11.1 |
| Exercises: Customize Liferay UserLocalService with a Service Wrapper | 1.11.1.1 |
| Override OSGi Service References | 1.11.2 |
| Chapter 11: Override Controller Actions | 1.12 |
| Override Struts Actions | 1.12.1 |
| Exercises: Override the Portal Logout Struts Action | 1.12.1.1 |
| Override MVC Commands | 1.12.2 |
| Exercises: Override the Documents and Media MVC Action Command | 1.12.2.1 |
| Chapter 12: Catch Portal Events | 1.13 |
| Catch Portal Lifecycle Events | 1.13.1 |
| Exercises: Creating a Post Login Event Listener | 1.13.1.1 |
| Intercept Events With Model Listeners | 1.13.2 |
| Exercises: Catching Persistence Events with Model Listeners | 1.13.2.1 |
| Chapter 13: Leverage the Liferay Message Bus | 1.14 |
| Introducing the Liferay Message Bus | 1.14.1 |
| Exercises: Create a Documents and Media Message Bus Listener | 1.14.1.1 |
| Chapter 14: Customize the Portal Search | 1.15 |
| Liferay Search Architecture | 1.15.1 |
| Basic Search Concepts | 1.15.2 |
| Customize Indexing and Search Results with processors | 1.15.3 |
| Exercises: Extend User Search using an Indexer Post Processor | 1.15.3.1 |
| Customize Index Settings and Type Mappings | 1.15.4 |
| Links and References | 1.16 |

Introduction

This is the Student's Guide for the *Liferay Backend Developer* training.

All links and references mentioned in the book can be found in the section *Links and References*.

Chapter 1: Setting Up the Development Environment

Chapter Objectives

- Understand the Liferay Development Environment Tools and Concepts
- Learn How to Set up the Development Environment

Development Environment Overview

Liferay offers developers a development platform and a comprehensive set of tools for building web applications. While the platform itself relies on Java EE and the OSGi framework for modular application development, Liferay is completely tool-agnostic when it comes to writing code for the platform. You can use any Java IDE or even a text editor to write Liferay applications.

The minimum set of tools required for developing for Liferay include:

- A code editor
- JDK 8 for compiling
- Build tool for creating the deployables
- A servlet container and a SQL database for running the portal

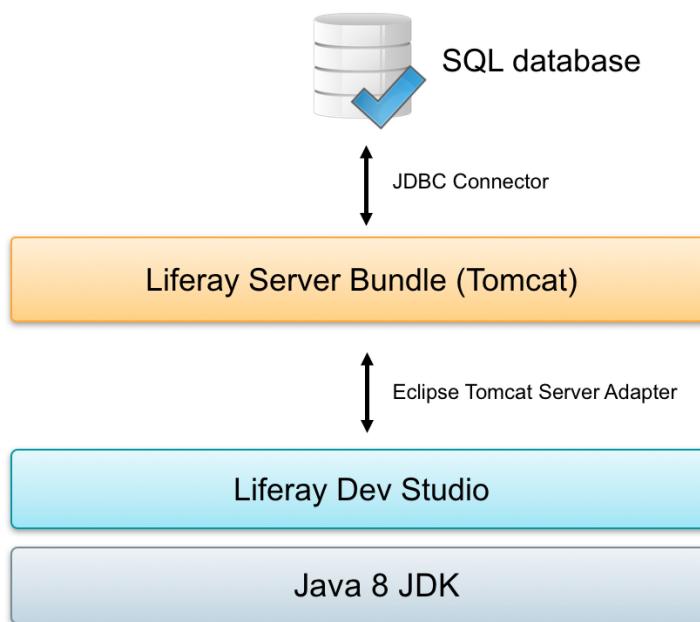


Figure: Example of a Liferay development environment

IDE support

Any Java IDE can be used for Liferay development, but support is only provided for the Eclipse-based **Liferay Dev Studio**, which is available in two versions.

Liferay Dev Studio DXP is a commercial and productized version of Eclipse available for enterprise customers. It's both the preferred and supported tool and is bundled with all tools needed for Liferay DXP development.

Liferay Dev Studio Community Edition is an open-source version of Liferay Dev Studio DXP. It can be downloaded as a standalone IDE or with Liferay Workspace. Liferay Workspace automates the download and installation of the Liferay CE bundle. Liferay Dev Studio Community Edition can be downloaded as a bundle or installed on top of an existing Eclipse installation.

With regard to other IDEs, a Liferay Workspace plugin is available for **IntelliJ**. Please see the *Links and Resources* section for more information.

Liferay Plugins SDK

Liferay Plugins SDK is deprecated but still available. It should be used only for developing Ext-plugins (deprecated) or for upgrading legacy plugins. For new projects, always use Liferay Workspace.

Java Runtime

A full JDK is required both by the development tools and for running the Liferay platform. Oracle Java 8, OpenJDK 8, and IBM J9 JDK 8 for WebSphere are supported. Using certified JDKs is recommended. Please see the *Liferay Portal Compatibility Matrix* in the *Links and References* section for more information.

Servlet Containers

Portal bundles for Apache Tomcat and WildFly are provided. But Liferay can run on any Java servlet container or EE server. When using the Liferay Workspace development environment, the Tomcat bundle can be downloaded and installed automatically from within the workspace and integrated seamlessly into the environment. Liferay Dev Studio also provides a server connector for Tomcat, which lets you control multiple Servlet containers from within the IDE.

Database

A relational database is needed for running the Liferay platform. The Liferay server bundle ships with the Hypersonic in memory database, which is **meant only for testing or evaluation purposes**. Using a supported database product also for development is recommended. Please see the *Liferay Portal Compatibility Matrix* in the *Links and Resources* section for a complete list of supported databases.

Commercial database JDBC drivers are not shipped with the bundles provided by Liferay and must be installed manually.

Tooling Scenario Examples

"Minimal"

This hypothetical scenario demonstrates the bare minimum for Liferay development.

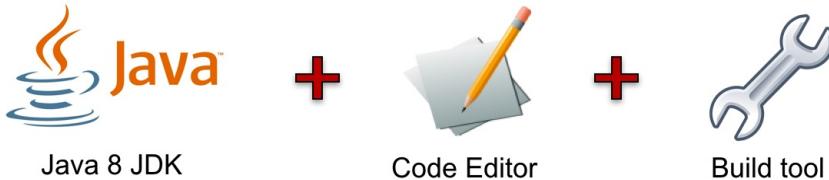


Figure: A minimal tooling scenario

"Typical"

Any Java IDE can be used for developing on the Liferay platform. But, especially when you start a new project, Liferay Dev Studio in connection with the Tomcat server bundle provides an easy and quick way to access all tools required for a productive Liferay development process.

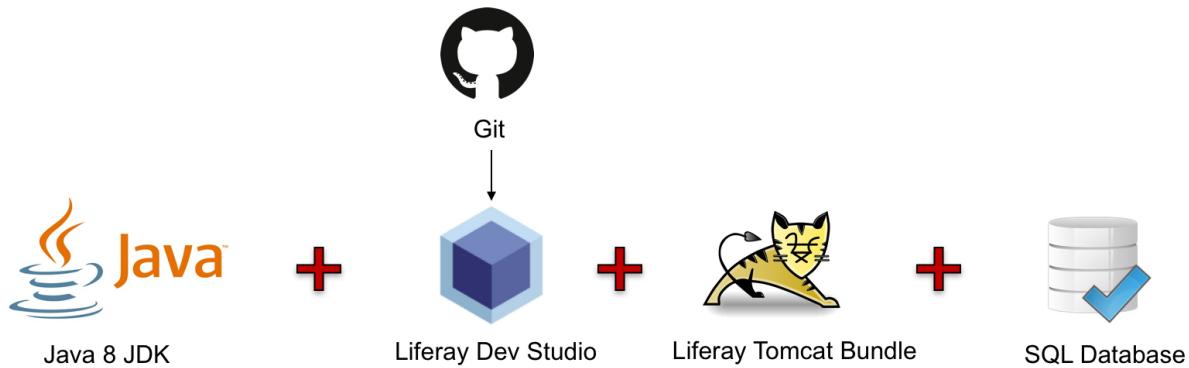


Figure: A typical tooling scenario

"Advanced"

Depending on your project's requirements, in advanced scenarios, the development environment can be integrated with additional tools like a testing and continuous integration pipeline, a Java profiler, or a graphical SQL client.

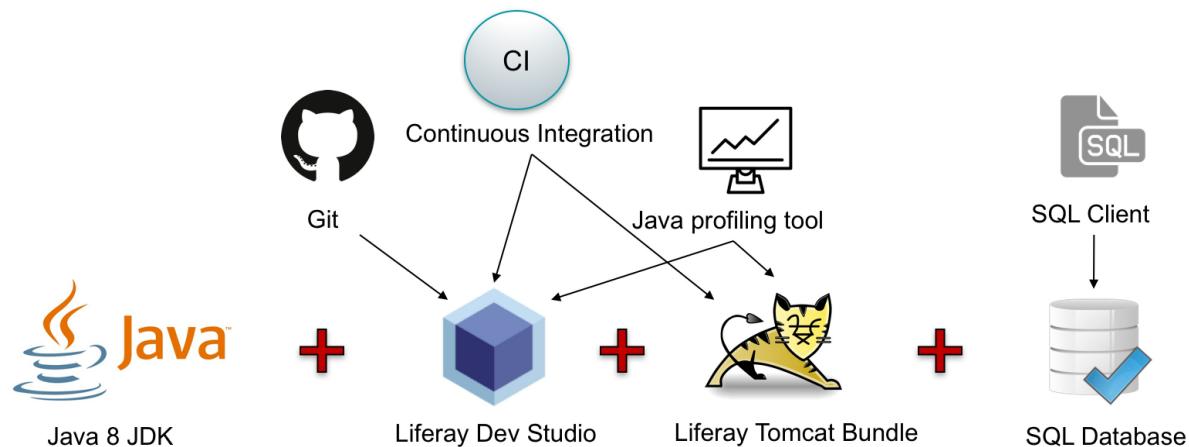


Figure: An advanced tooling scenario.

Summary

Liferay is tool-agnostic and leaves developers the freedom to choose the tools they like best and are the most familiar with. A full Eclipse-based IDE is provided under the name Liferay Dev Studio, containing all the tools needed for efficient back-end development.

The Tomcat server bundle is seamlessly integrated in the Liferay Workspace environment but can also be downloaded separately. Any servlet container or Java EE server, however, can be used for running the platform.

Links and Resources

- **Liferay Dev Studio CE Tutorials and Documentation**
https://dev.liferay.com/develop/tutorials/-/knowledge_base/7-1/liferay-ide
- **Liferay Downloads**
<https://www.liferay.com/downloads>
- **Liferay IntelliJ Plugin**
<https://plugins.jetbrains.com/plugin/10739-liferay-intellij-plugin>

- **Liferay Compatibility Matrix**

<https://web.liferay.com/services/support/compatibility-matrix>

Exercises

Set Up the Development Environment in Linux

Prerequisites

These instructions were made with Ubuntu LTS 16.04.

Before starting, you should have been provided by your trainer:

- A Developer Studio installation file
- A temporary license file

Please consult your trainer before starting if any of these are missing.

You should also have credentials to Liferay's website. If you don't have those, please register at <https://web.liferay.com/sign-in>.

Steps Overview

- ① Install Java 8 JDK
- ② Install MySQL database
- ③ Configure the database
- ④ Install Liferay Developer Studio DXP

Install Java 8 JDK

If you already have a Java 8 JDK installed, you can skip this step. Please note that a full JDK is required. You can check whether Java has been installed and its version in Command Line with:

```
java -version
```

If Java 8 is installed properly, the output should be something like:

```
java version "1.8.0_161"
Java(TM) SE Runtime Environment (build 1.8.0_161-b12)
Java HotSpot(TM) 64-Bit Server VM (build 25.161-b12, mixed mode)
```

To determine whether a full JDK is installed, type:

```
javac -version
```

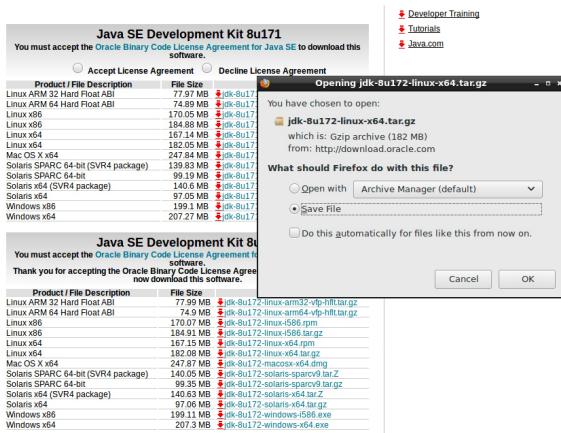
If a full JDK is installed, output should be something like:

```
java version "1.8.0_161"
```

If you need to install a JDK, you can choose to use your Linux distributions software management tools to install Java globally in your machine or to download a JDK from Oracle's website and install it only for your user profile. The following describes the steps to install Java for your user profile only:

Installing Java on Linux

1. **Open** your web browser and go to <http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html>.
2. **Find** the Java 8 JDK for your processor architecture.
3. **Choose** and download the `.tar.gz` package.



4. **Extract** the installation archive to your installation location.

- o In this example, we will be using the `/opt` folder:

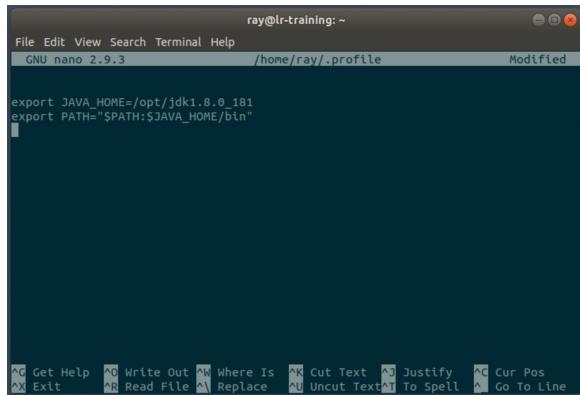
```
sudo tar -xzf jdk-8u172-linux-x64.tar.gz -C /opt/
```

5. **Open** your profile settings to add `JAVA_HOME` environment variable:

```
nano ~/.profile
```

6. **Add** the following lines to the end of the file, replacing the `/opt/jdk1.8.0_172` with your installation folder:

```
export JAVA_HOME=/opt/jdk1.8.0_172
export PATH="$PATH:$JAVA_HOME/bin"
```



7. **Run** the following to make the settings effective in the current terminal:

```
source ~.profile
```

You can check that the path is correct by running:

```
echo $JAVA_HOME
```

The output should point to your JDK installation folder.

Install MySQL Server

If you have at least the MySQL server version 5.5 installed, you can skip this step. You can check the version from the Command Line:

```
mysql -v
```

The output should be something like:

```
mysql Ver 14.14 Distrib 5.7.22, for Linux (x86_64) using EditLine wrapper
```

To continue installing the MySQL, use your Linux distributions software management tools. With Ubuntu 16.04 LTS, you can install MySQL server from the Command Line.

1. **Run** the following to install MySQL:

```
sudo apt-get install mysql-server
```

Follow the instructions on the screen and set a root user password.

Note: If you don't see prompts for setting up your root password, you can run the `sudo mysql_secure_installation` command.

Configure the Database

Next, we will install the Liferay portal database for the training.

1. **Login** to the MySQL shell from the Command Line with:

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

2. **Create** a new database by typing in the MySQL shell:

```
CREATE DATABASE backend_developer CHARACTER SET utf8;
```

```
ray@lr-training:~ ray@lr-training:~ ray@lr-training:~$ mysql -u root -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 9
Server version: 5.7.23-0ubuntu0.18.04.1 (Ubuntu)

Copyright (c) 2000, 2018, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

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

mysql> create database backend_developer character set utf8;
Query OK, 1 row affected (0.00 sec)

mysql> 
```

3. **Type** in the following to grant privileges to your database user.

- o In this training, we will be using the user 'liferay' with the password 'liferay'.

```
GRANT ALL PRIVILEGES ON backend_developer.* TO 'liferay'@'localhost' IDENTIFIED BY 'liferay';
```

4. **Type** in the following to flush privileges to become effective:

```
FLUSH PRIVILEGES;
```

5. **Type** in the following to logout from the MySQL server:

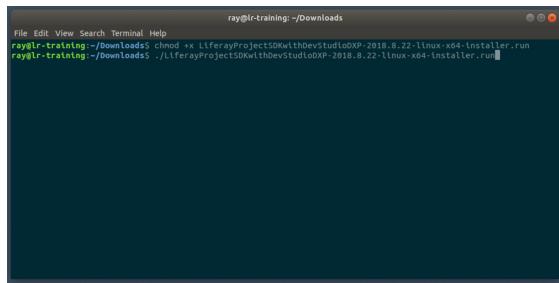
```
QUIT
```

Your database is now ready for the training.

Install the Developer Studio DXP

1. **Run** the Developer Studio installer to start the installation process.

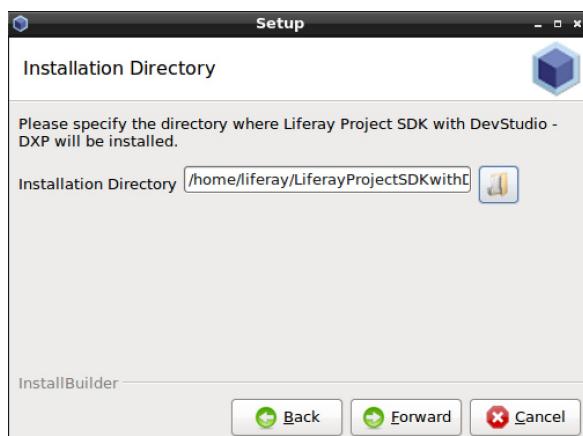
- o You can start the installer in via the command line by making it executable.



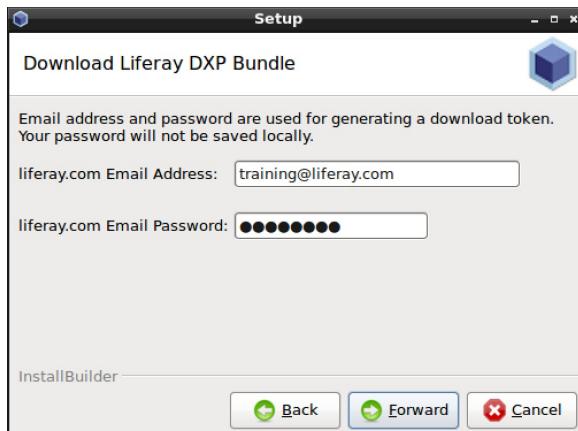
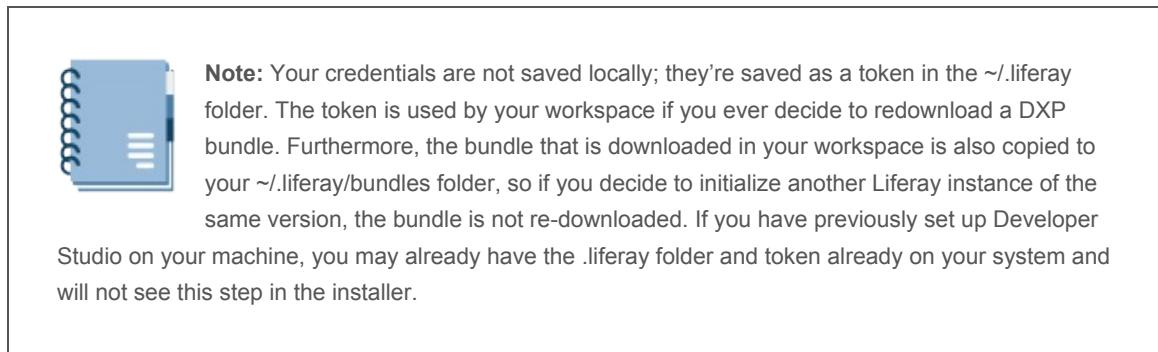
2. **Choose** the Java JDK in the first dialog pop-up:



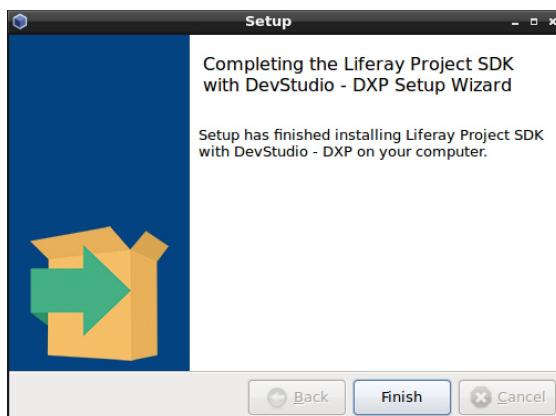
3. **Choose** the installation location for Developer Studio:



- o Choose `home/liferay` and point the installer there.
4. Type your liferay.com credentials if prompted.



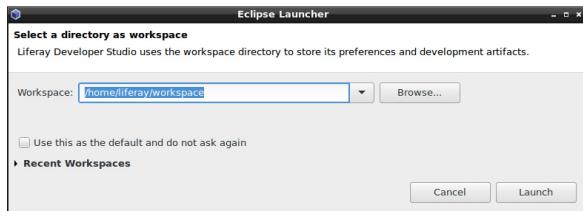
5. Click Next.
o After unpacking and installing, the setup should be ready:



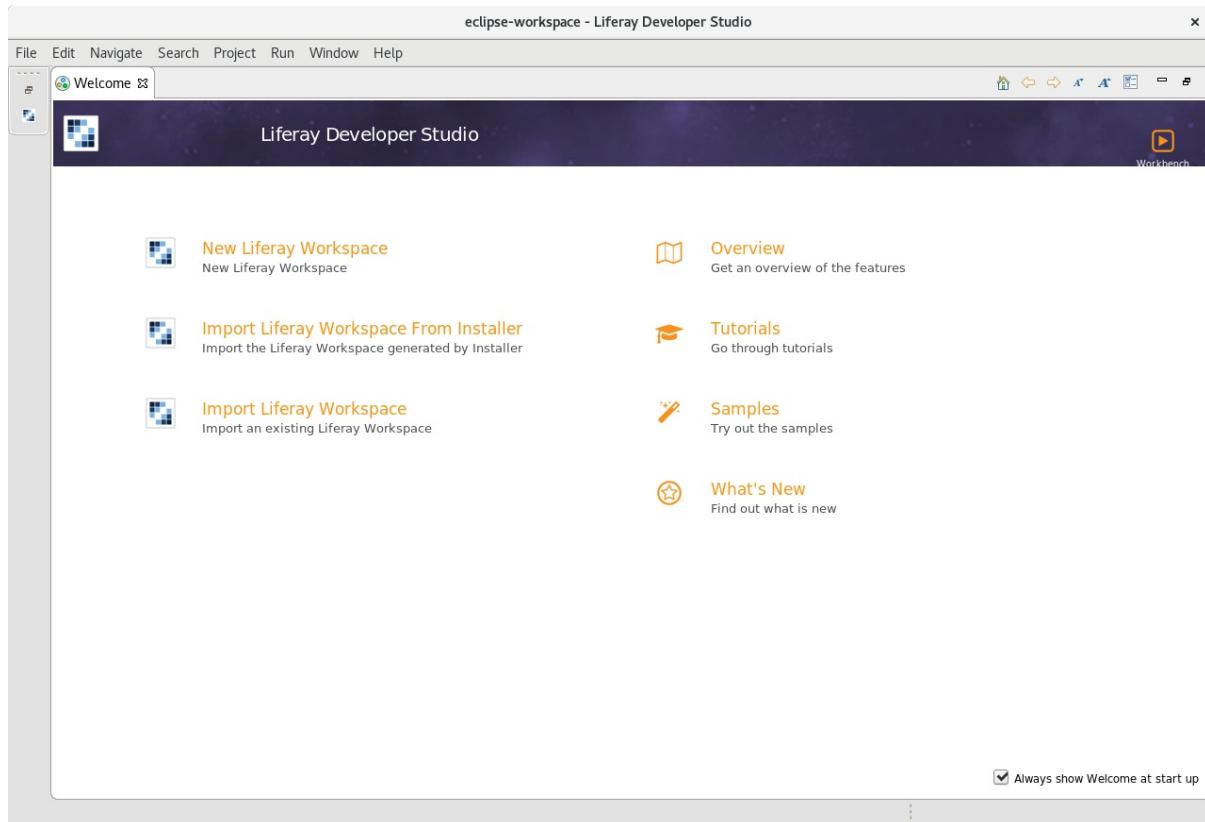
6. Start Developer Studio by replacing the path below with your installation path:

```
~/LiferayProjectSDKwithDevStudioDXP/liferay-developer-studio/DeveloperStudio
```

7. Choose an Eclipse workspace location and click Launch:



Checkpoint



Set Up the Development Environment in OSX

Prerequisites

Before starting, you should have been provided by your trainer:

- A Dev Studio installation file
- A temporary license file

Please consult your trainer before starting if any of these are missing.

You should also have credentials to Liferay website. If you don't have those, please register at <https://web.liferay.com/sign-in>.

Steps Overview

- ① Install Java 8 JDK
- ② Install MySQL database
- ③ Configure the database
- ④ Install Liferay Dev Studio DXP
- ⑤ Install training snippets and exercises

Install Java 8 JDK

If you already have a Java 8 JDK installed you can skip this step. Please note that a full JDK is required. You can check whether Java has been installed and its version in command line with:

```
java -version
```

If Java 8 is installed properly, the output should be something like:

```
java version "1.8.0_161"
Java(TM) SE Runtime Environment (build 1.8.0_161-b12)
Java HotSpot(TM) 64-Bit Server VM (build 25.161-b12, mixed mode)
```

To determine whether a full JDK is installed, type:

```
javac -version
```

If a full JDK is installed, output should be something like:

```
java version "1.8.0_161"
```

Installing Java on OSX

1. [Open http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html](http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html) in your web browser.

1. **Find** the Java 8 JDK for your processor architecture.

2. **Choose** and download `.dmg` file.
3. **Go to** where the `.dmg` file was downloaded.
4. **Double-click** the package icon to launch the installer.
5. **Open** the profile in order to add the `JAVA_HOME` environment variable to your user profile settings:

```
nano ~/.profile
```

6. **Add** the following lines to the end of the file:

```
export JAVA_HOME=$(/usr/libexec/java_home -v 1.8)"
```

7. **Type** in the following to make the settings effective in the current terminal:

```
source ~/.profile
```

You can check that path is correct by running:

```
echo $JAVA_HOME
```

The output should point to your JDK installation folder.

Install MySQL Server

1. **Go to** <https://dev.mysql.com/downloads/mysql/> in your web browser.
2. **Download** and run the MySQL installer.
 - o Make sure the version is 5.7 or above
3. **Click** to accept the terms of use.
4. **Choose** *Developer Default* for the *Setup Type*.
5. **Click** *Execute* and run through the installation leaving the default values.
6. **Set** the password.
 - o Use something easy such as *root* or *liferay*.
7. **Click** *Next* and progress through the installer leaving the default values.

Configure the Database

Next, we will set up the Liferay portal database for the training.

1. **Login** to the MySQL shell from command line with:

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

2. **Create** a new database:

```
create database backend_developer character set utf8;
```

3. **Logout** from the MySQL Command Line with:

```
quit
```

Your database is now ready for the training.

Install the Dev Studio DXP

1. **Run** the Dev Studio installer to start the installation process

2. Choose the Java JDK:

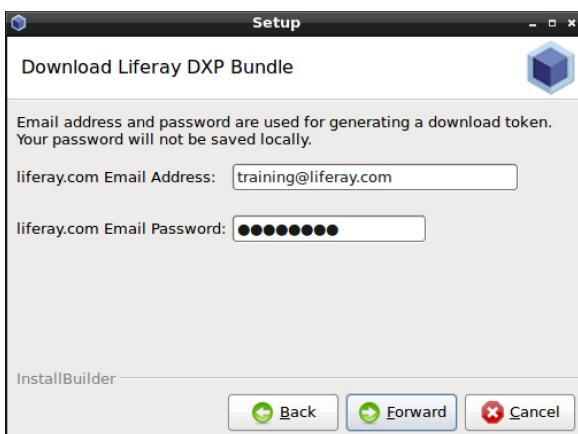
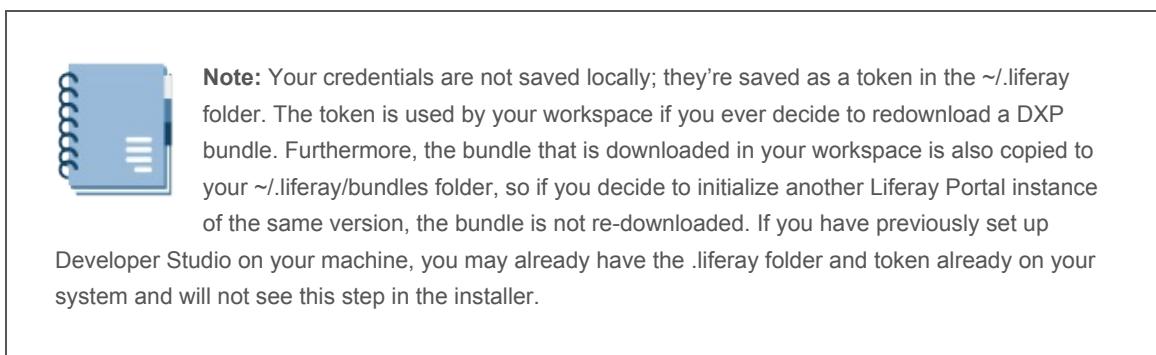


3. Choose the installation location for the Dev Studio:

- Typically we create a directory `~/liferay` or `~/lfddev` and point the installer there.

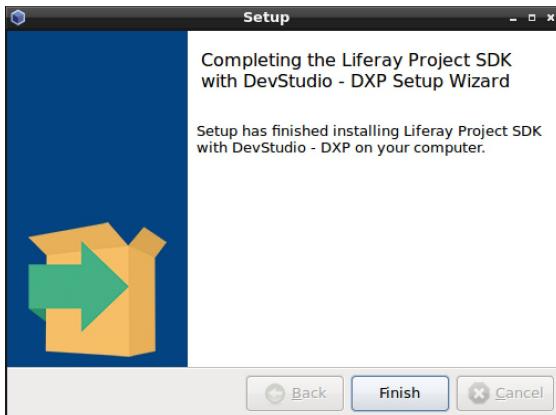


4. Type in your liferay.com credentials:

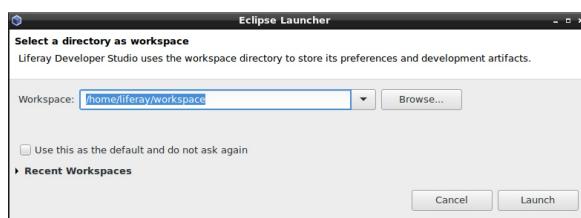


5. Click Next.

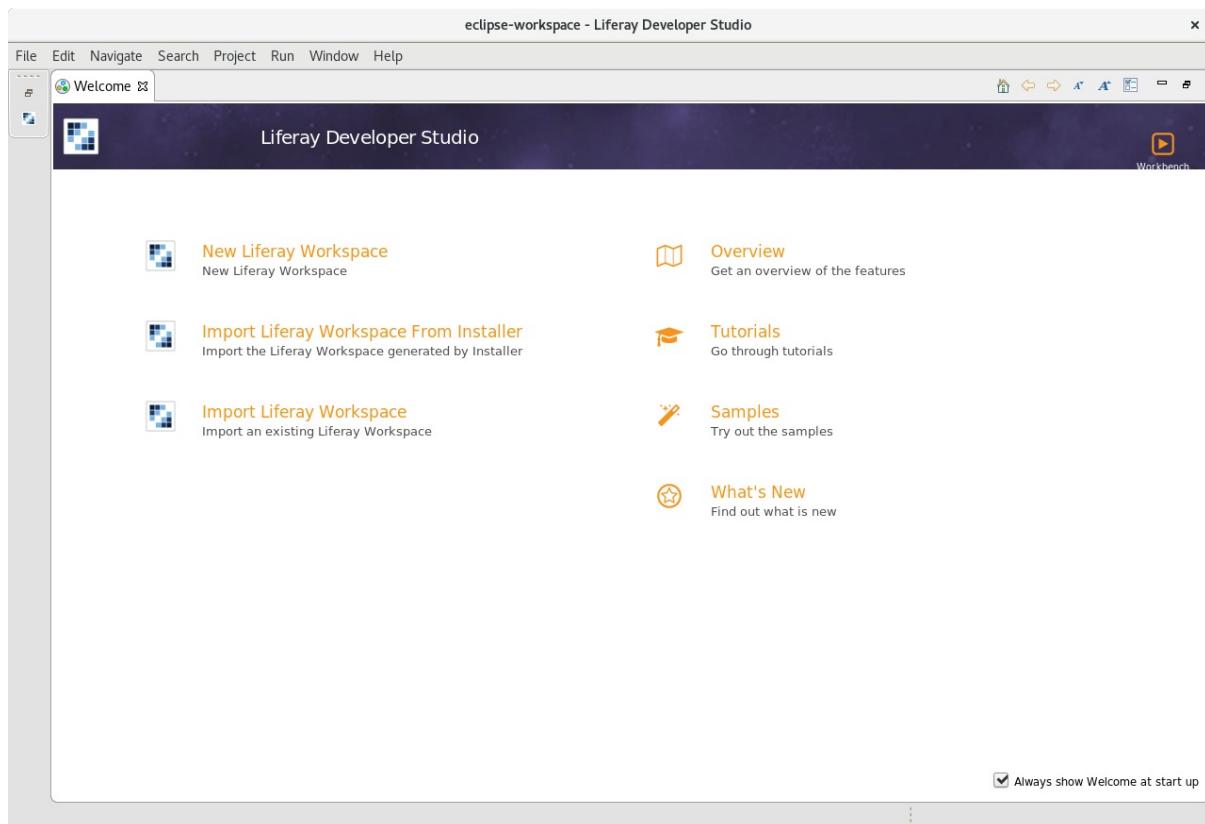
- After unpacking and installing, the setup should be ready:



6. **Start** Developer Studio.
7. **Choose** a Eclipse workspace location and click *Launch*:



Checkpoint



Set Up the Development Environment in Windows

Prerequisites

These instructions were made with Windows 10.

Before starting, your trainer should have provided you with:

- A Developer Studio installation file
- A temporary license file

Please consult your trainer before starting if any of these are missing.

You should also have credentials to Liferay website. If you don't have those, please register at <https://web.liferay.com/sign-in>.

Steps Overview

- ① Install Java 8 JDK
- ② Install MySQL database
- ③ Configure the database
- ④ Install Liferay Developer Studio DXP
- ⑤ Install training snippets and exercises

Install Java 8 JDK

If you already have a Java 8 JDK installed, you can skip this step. Please note that a full JDK is required. You can check whether Java has been installed and its version in command line with:

```
java -version
```

If Java 8 is installed properly, the output should be something like:

```
java version "1.8.0_181"
Java(TM) SE Runtime Environment (build 1.8.0_181-b13)
Java HotSpot(TM) 64-Bit Server VM (build 25.181-b13, mixed mode)
```

To determine whether a full JDK is installed, type:

```
javac -version
```

If a full JDK is installed, output should be something like:

```
javac 1.8.0_181
```

Installing Java on Windows

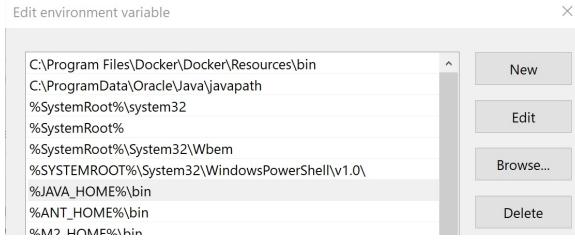
1. [Go to http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html](http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html) in your web browser.

2. **Download** the Java 8 JDK for your processor architecture (usually Windows x64).
3. **Run** the JDK installer.
4. **Click** through the installer, leaving all default options.
5. **Go to** the *System Properties* and click on the *Advanced* tab.
6. **Click** on the *Environment Variables* button.
7. **Create** a new system variable called `JAVA_HOME`.
8. **Set** the *Variable value* to the path to your JDK.

- o For example:

```
C:\Program Files\Java\jdk1.8.0_181
```

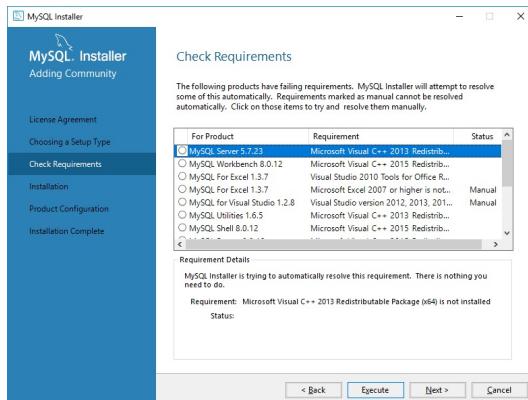
9. **Add** `%JAVA_HOME%\bin` to your `Path` system variable.



Install MySQL Server

1. **Go to** <https://dev.mysql.com/downloads/windows/installer/5.7.html> in your web browser.
2. **Download** and run the *MySQL Installer*.
 - o In Windows 10, you may need to run the installer as an Administrator in order to get it to start properly.
3. **Accept** the terms of use.
4. **Choose** *Developer Default* for the *Setup Type*.
5. **Click** *Execute* and run through the installation, leaving the default values.
6. **Set** the password.
 - o Use something easy like *root* or *liferay*
7. **Click** *Next* and progress through the installer, leaving the default values.

Note: Depending on the state of your machine, you might have to install additional software in order to run the MySQL installer. In this case the installer will prompt you for missing requirements. Press the **Execute** button to let the installer try to resolve and install the missing requirements. Confirm the license agreements you might be prompted for.



Configure the Database

1. **Click** on the *Start* menu.
2. **Find** and run the *MySQL 5.7 Command Line Client*.
3. **Type** the password set in the installer.
4. **Type** the following in the MySQL Command Line to create a new database:

```
create database backend_developer character set utf8;
```

5. **Type** in the following to grant privileges to your database user.
 - o In this training, we will be using the user 'liferay' with the password 'liferay'.

```
grant all privileges on backend_developer.* to liferay@localhost identified by 'liferay';
```

6. **Type** in the following to flush privileges to become effective:

```
flush privileges;
```

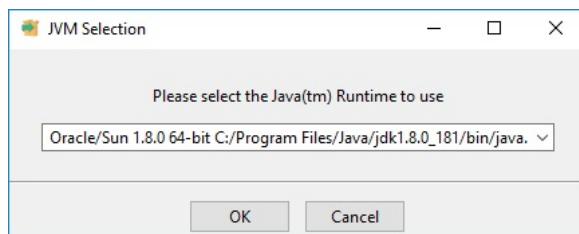
7. **Logout** of the MySQL Command Line with:

```
quit
```

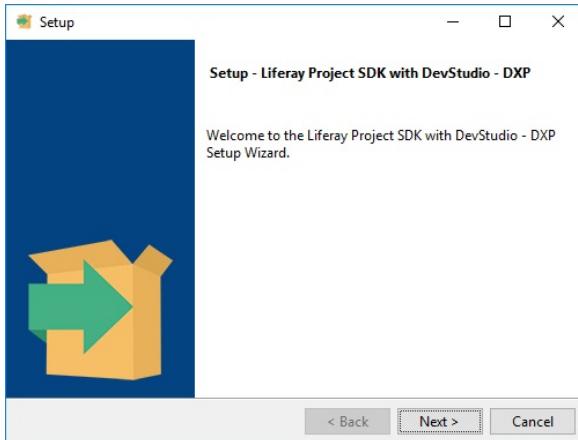
Your database is now ready for the training.

Install Developer Studio DXP

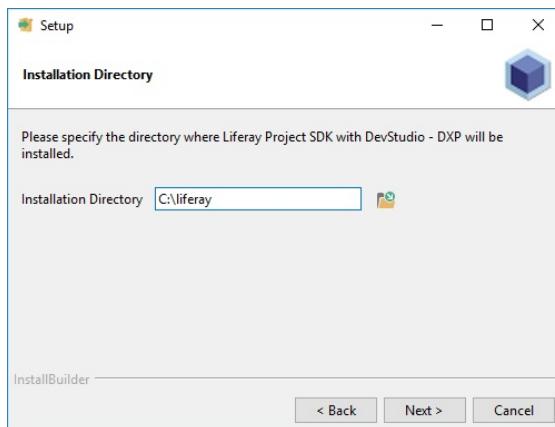
1. **Run** the Developer Studio installer to start the installation process.
 - o A notification window may appear in regards to unrecognized third party app. If it does, click *More Info* → *Run Anyway* to run the installer.
2. **Choose** the Java JDK and click *OK*:



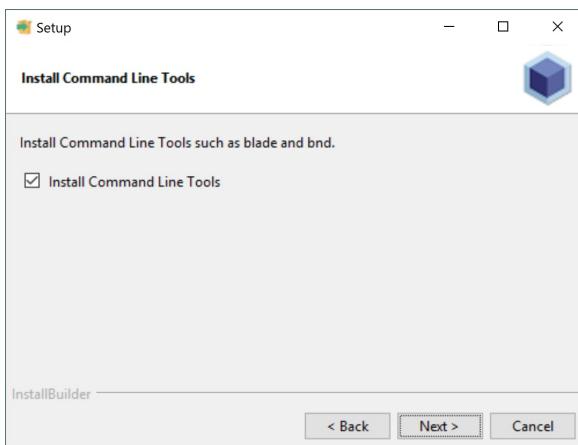
3. **Click** *Next*.



4. **Choose** the installation location for the Developer Studio:
- o Choose C:\liferay and point the installer there.



5. **Click** Next to install all Developer Studio components (Leave *Install Command Line Tools* checked).

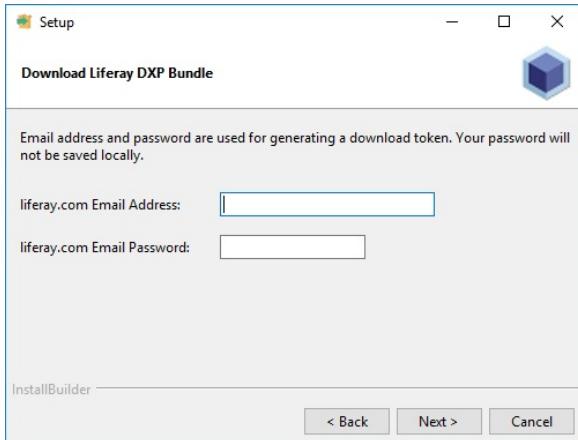


6. **Enter** your credentials for.liferay.com.



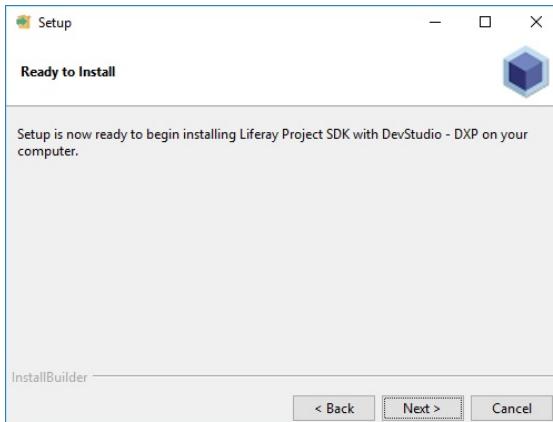
Note: Your credentials are not saved locally; they're saved as a token in the `~/.liferay` folder. The token is used by your workspace if you ever decide to redownload a DXP bundle. Furthermore, the bundle that is downloaded in your workspace is also copied to your `~/.liferay/bundles` folder, so if you decide to initialize another Liferay instance of the

same version, the bundle is not re-downloaded. If you have previously set up Developer Studio on your machine, you may already have the *.liferay* folder and token already on your system and will not see this step in the installer.

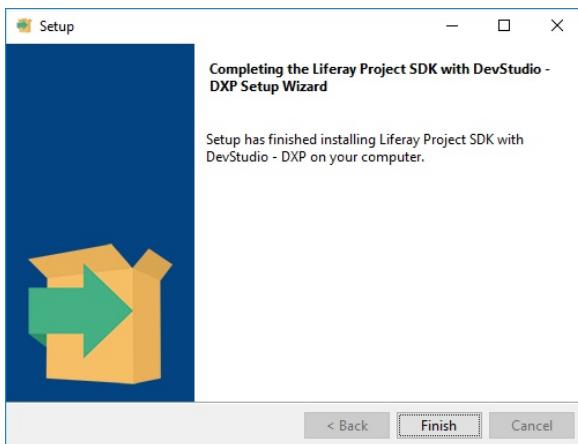


7. Click *Next* to begin the installation.

- o After unpacking and installing, the setup should be ready:



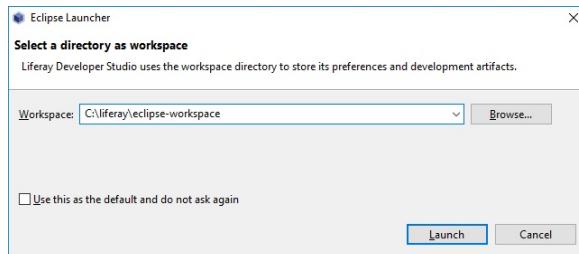
8. Click *Finish* to close the installer.



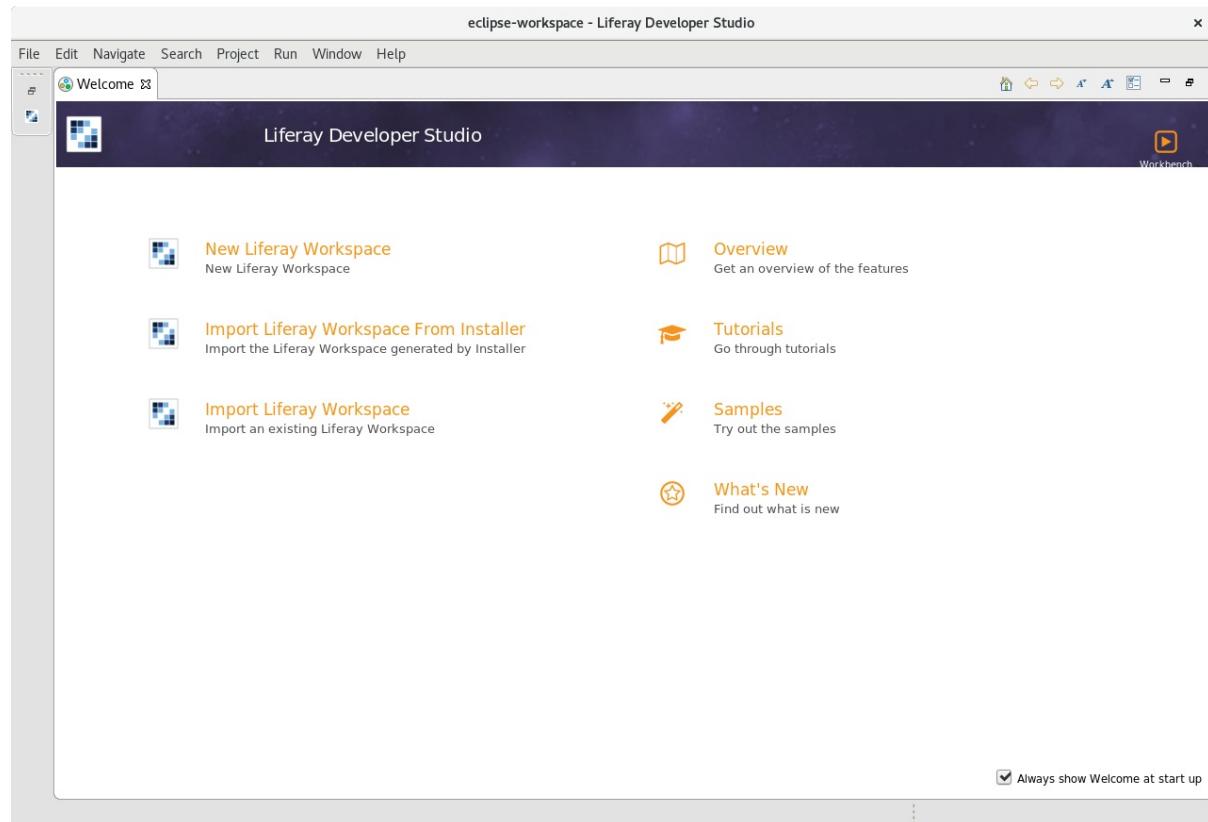
- o Feel free to delete the auto-generated *liferay-workspace* created when running the installer. We'll be

importing a custom training workspace later on.

9. Double-click the Developer Studio .exe located in `C:\liferay\liferay-developer-studio` to run the program.
10. Choose `C:\liferay\eclipse-workspace` as the Eclipse workspace location and click *Launch*:



Checkpoint



Chapter 2: OSGi Basics

Chapter Objectives

- Understand the Basic Concepts and Benefits of OSGi
- Learn How to Create OSGi Applications

Introduction

OSGi stands for Open Service Gateway Initiative. OSGi is a specification and a set of standards for modular software development with Java. The organization behind the initiative is the OSGi Alliance, which was founded in 1999.

OSGi is a stable and mature technology. OSGi Release 1 (R1) was published in May, 2000, the same year as J2SE 1.3. OSGi is widely adopted in the industry and has multiple implementations of its standard.

OSGi is at the center of Liferay's core technologies. Understanding OSGi concepts and architecture is paramount in Liferay development.



Figure: Products using OSGi technology

In this chapter, we'll introduce you to the basic development units and concepts of the OSGi framework and environment:

- OSGi Framework
- Bundle
- Component
- Service

Understanding these concepts will help you master back-end development in Liferay.

Basic Concepts

OSGi Framework

The OSGi framework is a standard for modular Java software development. Occasionally, the term *OSGi framework* is used to refer just to the OSGi application runtime, which here is called the *OSGi container*.

The basic unit of modularization is called a *Bundle*. An OSGi bundle contains an OSGi application or part of an OSGi application. Basically, an OSGi bundle is a regular JAR file made up of Java classes and resources like any other JAR archive. The only difference is that a common JAR archive has the metadata, a Bundle-SymbolicName identifier header in the JAR's `MANIFEST.MF` file:

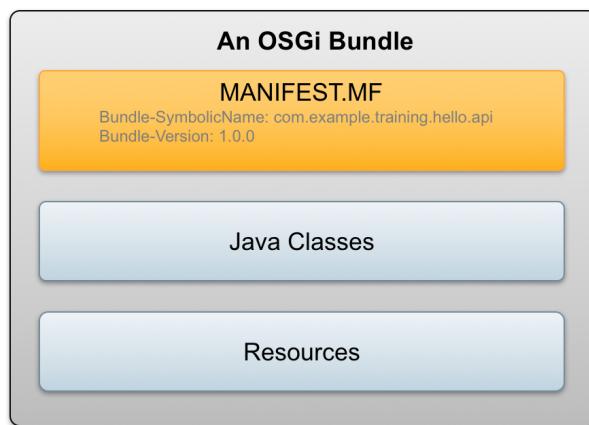


Figure: Structure of an OSGi bundle.

OSGi bundles reside in an *OSGi runtime* or an *OSGi container*, which, in turn, is running in a standard Java runtime:



Figure: OSGi container.

Exercises

Hello OSGi

Exercise Description

Liferay is built using OSGi technologies, and all the portal applications are OSGi applications. To gain a better understanding of how OSGi works, we'll be going through the fine grain detail behind the scenes. Although Liferay development tools do much of the OSGi magic for you, understanding how things work will make developing and troubleshooting much easier.

In this exercise, we will start by creating a simple Hello OSGi application and run it in the Eclipse OSGi runtime. This exercise will show the following basic ingredients of OSGi:

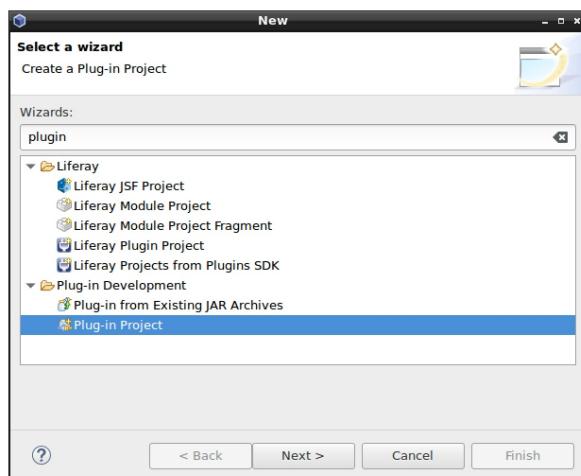
1. An OSGi **runtime**
2. An OSGi **bundle**
3. The OSGi specific **headers** in the bundle JAR's manifest file

Exercise Steps Overview

- 1 Create a New Plugin Project using the *Hello OSGi* template
- 2 Set up OSGi Framework run configuration
- 3 Run the application

Create a New Plugin Project using the *Hello OSGi* Template

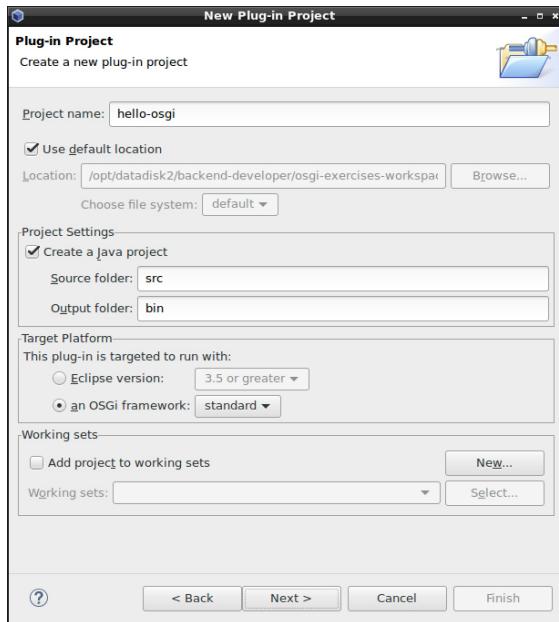
1. **Click** *File* → *New* → *Other* on the Dev Studio menu bar to open the new project wizard.
2. **Type** *plugin* in the search bar.
3. **Choose** the *Plug-in Project* project type and click *Next*:



4. **Type** *hello osgi* for the project name.
5. **Click** on the radio button next to "an OSGi framework" under the *Target Platform* section.
6. **Choose** *standard* in the dropdown next to "an OSGi framework".

7. Click Next.

- By setting the platform to standard, we guarantee that no Eclipse implementation-specific features are added:

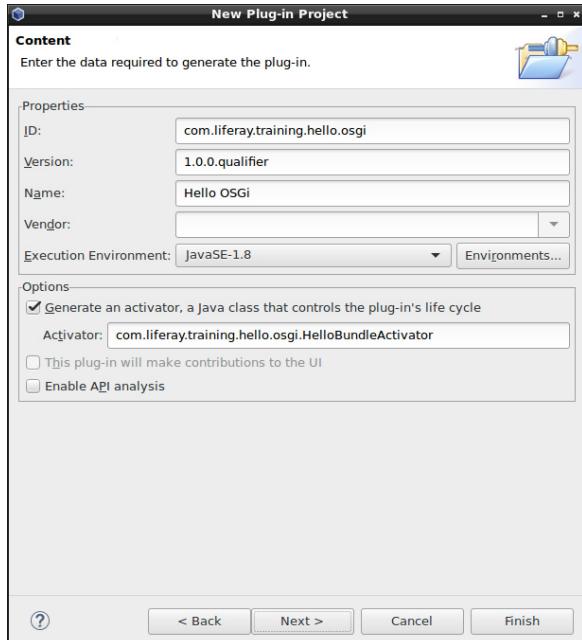


8. Type the following into the properties:

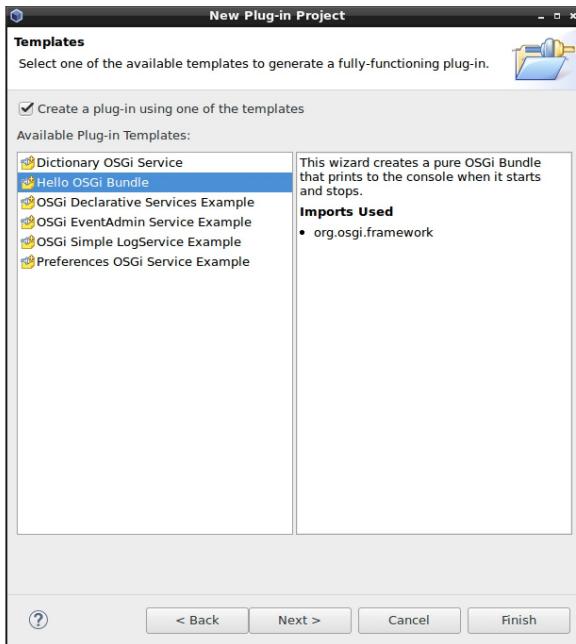
- ID:** com.liferay.training.hello.osgi
- Name:** Hello OSGi

9. Check the *Generate an activator* checkbox.10. Type `com.liferay.training.hello.osgi.HelloBundleActivator` for the activator name.

11. Click Next:

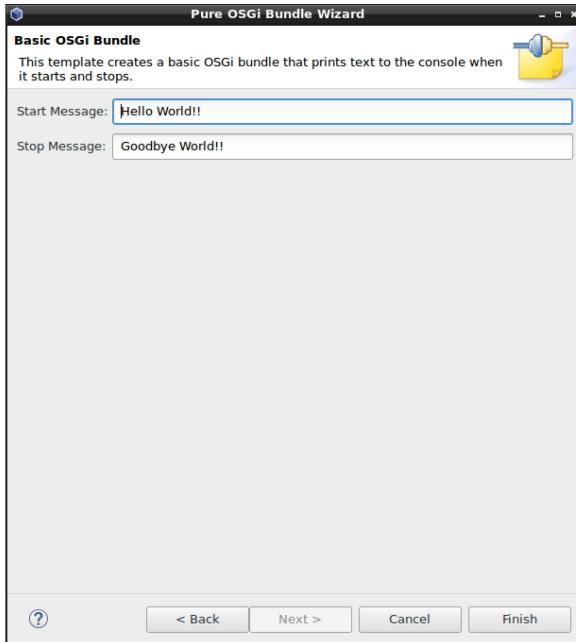
12. Choose the *Hello OSGi Bundle* project template

13. Click Next:

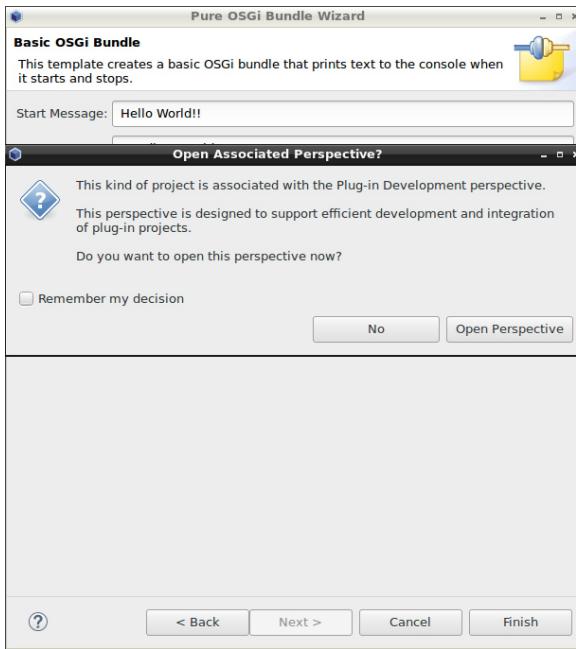


14. Enter your personal hello message or leave the defaults.

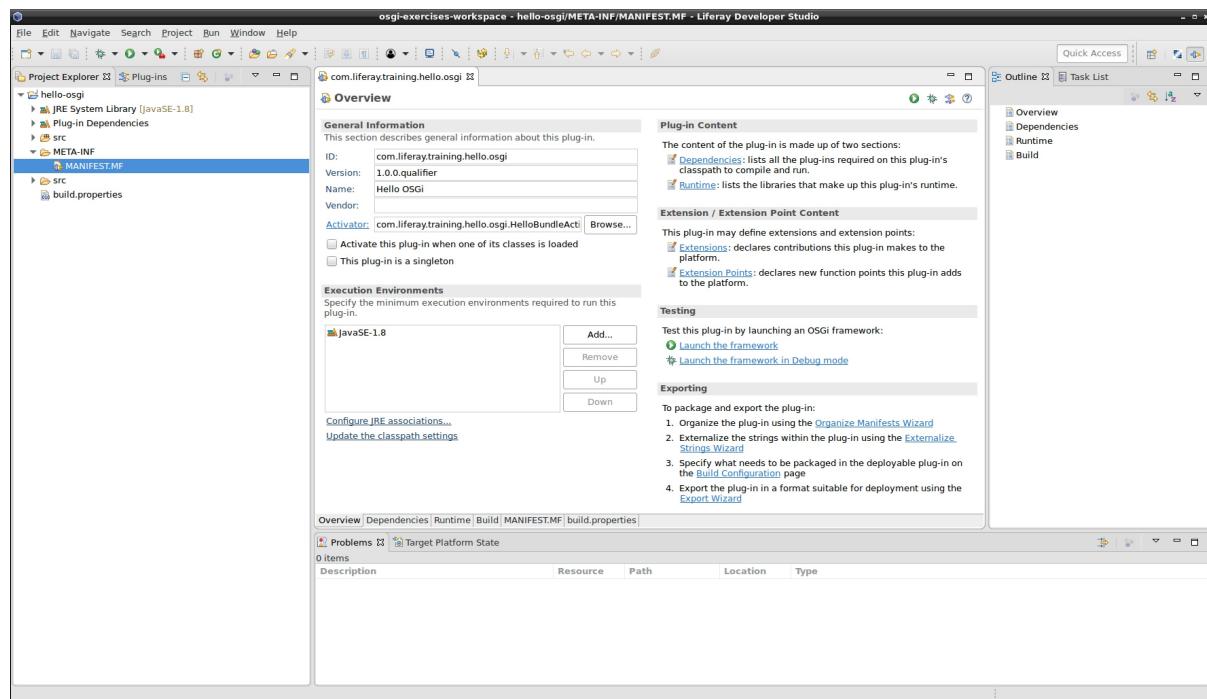
15. Click Finish to close the wizard:



16. Click Open Perspective to switch to the Plug-in project type specific view:



You have reached the first checkpoint:

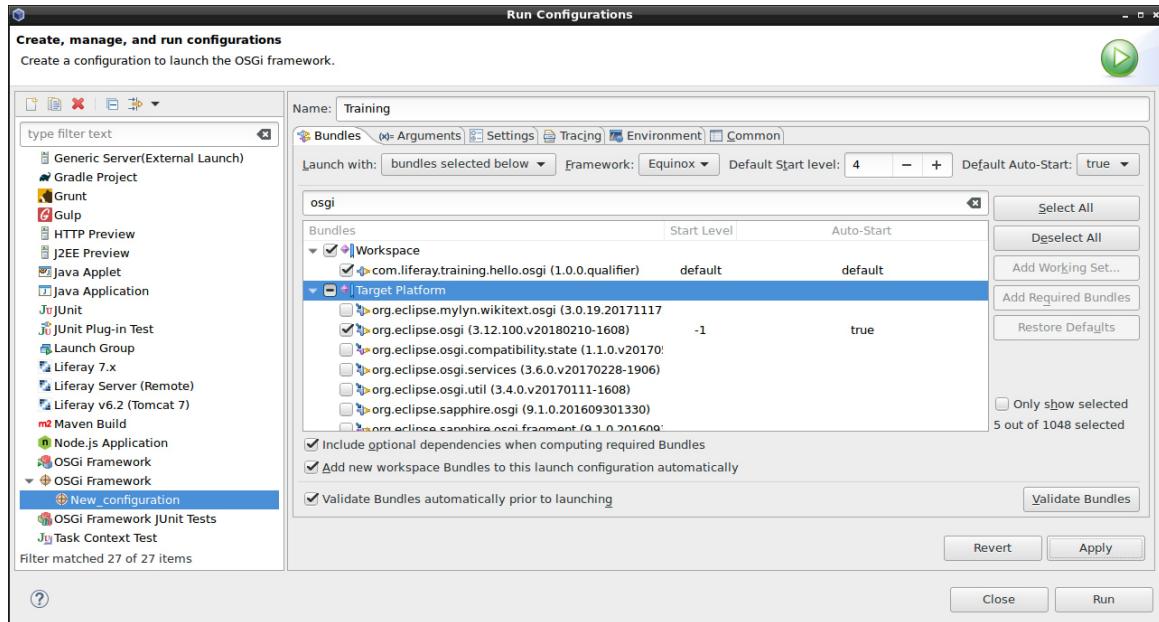


Set up OSGi Framework Run Configuration

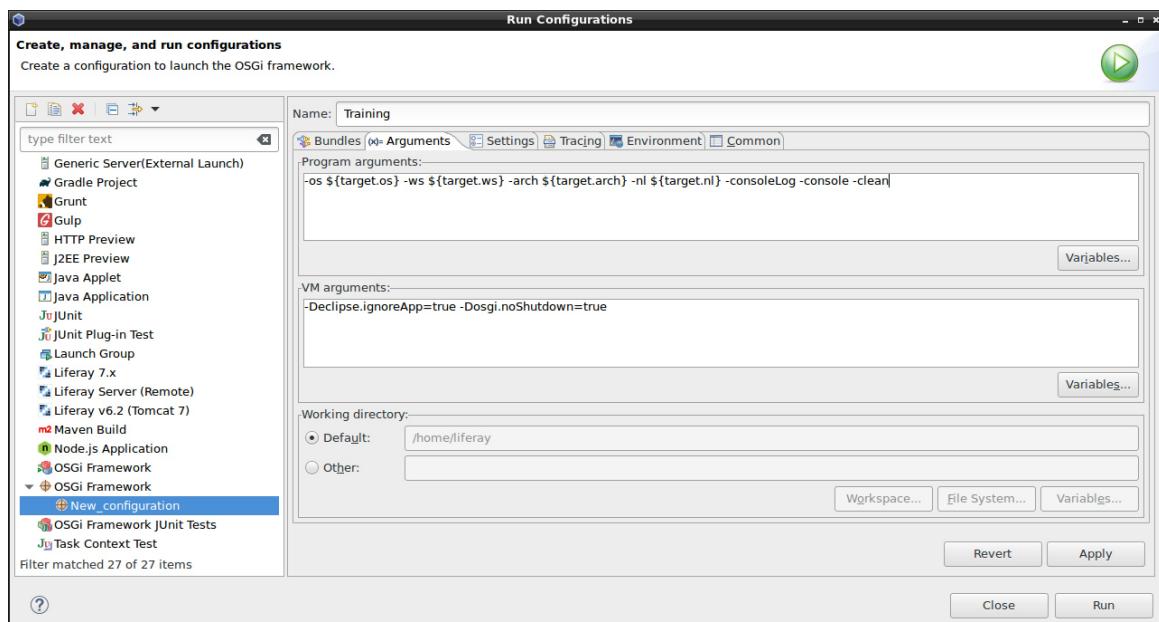
Before running the bundle, we'll enable only the bundles needed to run the OSGi container.

1. **Click** *Run → Run Configurations* on the Dev Studio menu bar to open the run configuration dialog.
2. **Double-click** *OSGi Framework* with the **target icon** to create a new launch run configuration.
3. **Expand** *OSGi Framework* and select *New_configuration* icon.
4. **Type** *Training* for the Name.
5. **Click** the *Deselect All* button to clear the bundle selections
6. **Enable** your *hello-osgi* bundle (under *Workspace*) as well as the following five(5) bundles:

- org.eclipse.osgi
- org.eclipse.equinnox.console
- org.apache.felix.gogo.command
- org.apache.felix.gogo.runtime
- org.apache.felix.gogo.shell

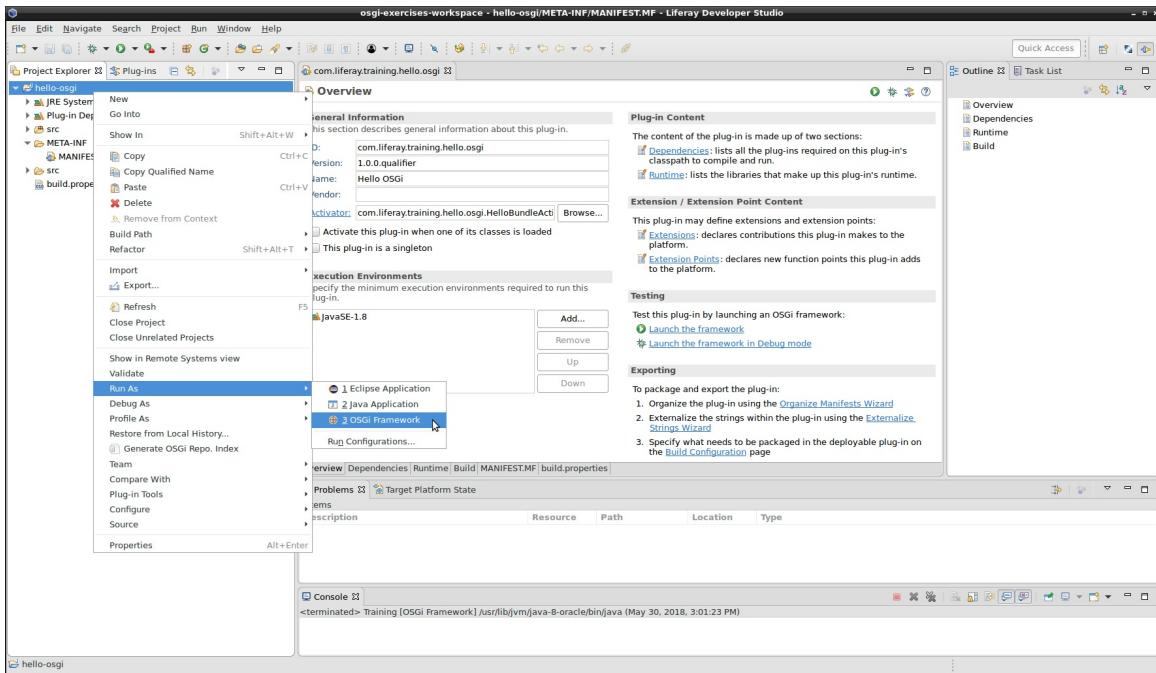


7. Click the *Arguments* tab on the dialog and add the `-clean` argument to the end of the list in the *Program Arguments* box (upper text box).
 - This avoids workbench creation-related error messages on some Eclipse distributions.
8. Click *Apply* then *Close* to close the dialog.

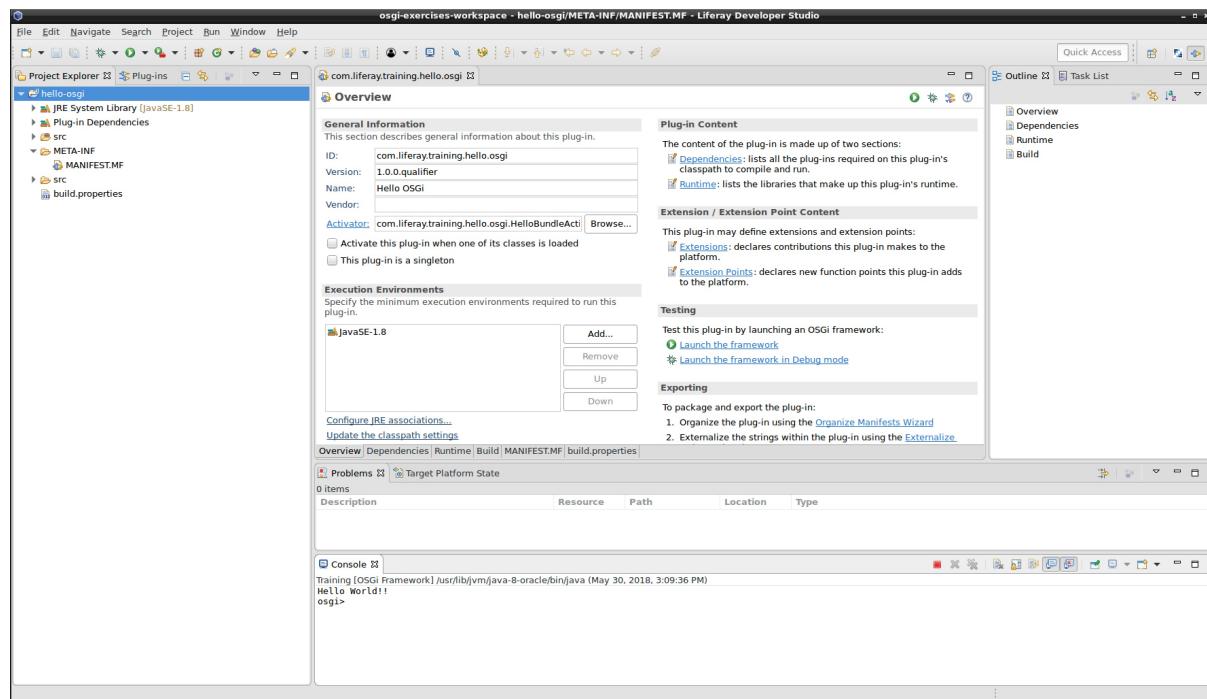


Run the Application

1. Right-click the project to open the context menu.
2. Click *Run As* → *OSGi Framework*:



You should see the following in the console:



Takeaways

Check how the following items were created:

- The bundle manifest file MANIFEST.MF
- The bundle activator class *HelloBundleActivator*

Bundles

Bundle Identifier

Every bundle inside an OSGi container has a unique identifier. The identifier has the following two headers:

- Bundle-SymbolicName
- Bundle-Version

This means that if there's a bundle with the same symbolic name but a different version, it's identified as a different bundle. Thus, multiple versions of the same bundle or library can co-exist in the same runtime. If you don't define the Bundle-Version, it will be set to 0.0.0 by the runtime.

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Osgi Exercise
Bundle-SymbolicName: com.liferay.training.osgi
Bundle-Version: 1.0.0.qualifier
Bundle-Activator: com.liferay.training.osgi.Activator
Bundle-RequiredExecutionEnvironment: JavaSE-1.8
Import-Package: org.osgi.framework;version="1.3.0"
Automatic-Module-Name: com.liferay.training.osgi
```

Figure: OSGi bundle identifier.

Bundle Semantic Versioning

Bundle version numbers provide a powerful feature called semantic versioning.

Semantic versioning means that each part of the version number has a specific meaning. The syntax is presented in the diagram below:

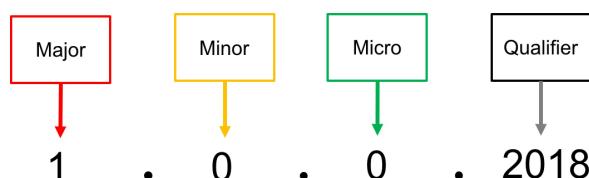


Figure: OSGi bundle semantic versioning

Every part of the version-number has a specific meaning:

- **Major:** There are breaking changes in the API
- **Minor:** API is changed but not broken
- **Micro:** No compatibility issues
- **Qualifier:** No impact

Semantic versioning doesn't allow a bundle to run with an API-incompatible dependency in an OSGi container.

The bndtools baselining tool lets you automate numbering and check the semantics. It can test the compatibility of your development bundle against the release bundle in a bundle repository and build the version number accordingly.

Using Version Ranges

Declaring dependencies to other OSGi bundles requires defining version numbers. This is done in the bundle manifest and is always done using version ranges, which have the following syntax:

- Square brackets '[' and ']' indicate **inclusiveness**
- Parentheses '(' and ')' indicate **exclusiveness**
- [1.1,2.0) means a version range from 1.1 to 2.0, including 1.1 and excluding 2.0

Below is an example of version ranges in an Export-Import bundle:

Exporting Bundle

```
Export-Package: com.liferay.training.osgi.api; version=1.2.3.20180410
```

Importing Bundle

```
Import-Package: com.liferay.training.osgi.bundle; version="[1.2,2.0)"
```

If the Import-Package doesn't specify the *version* attribute, it is added implicitly and assumed to be from 0.0.0 to infinity.

Bundle Lifecycle

OSGi bundles are state-aware. Bundle activators are classes that implement the *org.osgi.framework.BundleActivator* interface and are used, for example, to prepare and clean up resources when a bundle is started and stopped. The start method is run when the bundle starts and gets into the active state. The stop method is run when the bundle is stopped:

```
package com.liferay.training.osgi;

import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;

public class Activator implements BundleActivator {

    /*
     * (non-Javadoc)
     * @see org.osgi.framework.BundleActivator#start(org.osgi.framework.BundleContext)
     */
    public void start(BundleContext context) throws Exception {
        System.out.println("Hello World!!!");
    }

    /*
     * (non-Javadoc)
     * @see org.osgi.framework.BundleActivator#stop(org.osgi.framework.BundleContext)
     */
    public void stop(BundleContext context) throws Exception {
        System.out.println("Goodbye World!!!");
    }
}
```

Bundles can have the following **lifecycle** states in the container:

- Installed
- Resolved
- Starting
- Active
- Stopping
- Uninstalled

Possible state sequences are described in the diagram below:

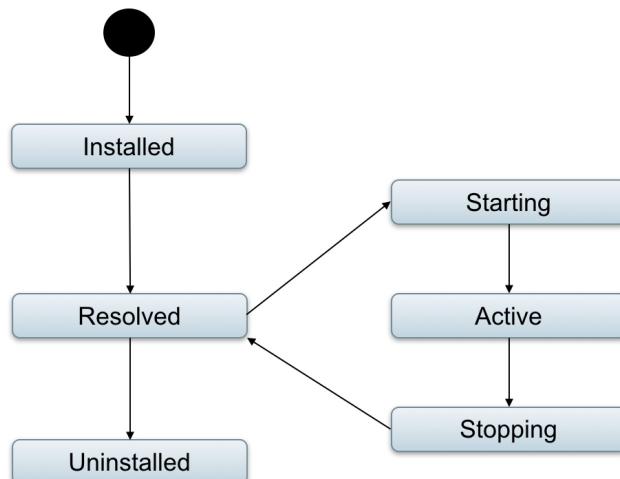


Figure: OSGi bundle lifecycle

There are command line and web-based management tools available for managing the lifecycle-states of bundles inside the container. Apache Felix Gogo Shell is a sub-project of Apache Felix and a console tool for managing the OSGi bundles and the container:



Figure: Gogo shell.

The `/b (list bundles)` command shows the installed bundles, start level priority, and their lifecycle state. In the image below, all the bundles are installed, started successfully, and are in the active state:

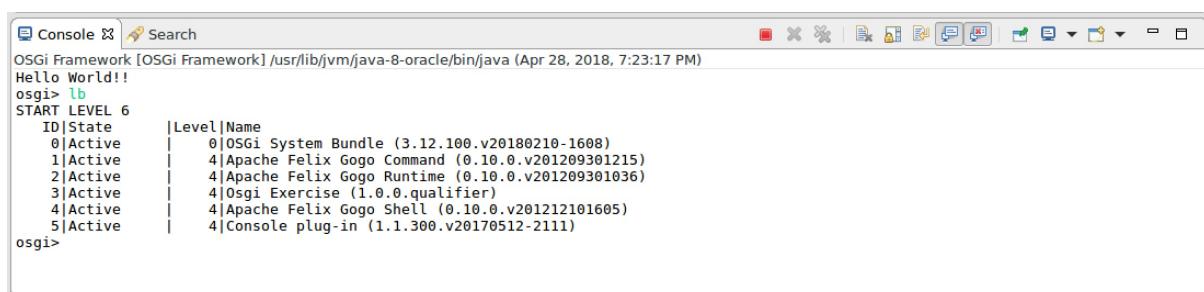


Figure: Checking the bundle state with the Gogo Shell

The bundle lifecycle can be managed with the Gogo Shell using the following commands:

- install
- uninstall
- resolve

- update
- refresh
- start
- stop

The following diagram illustrates the possible command sequences:

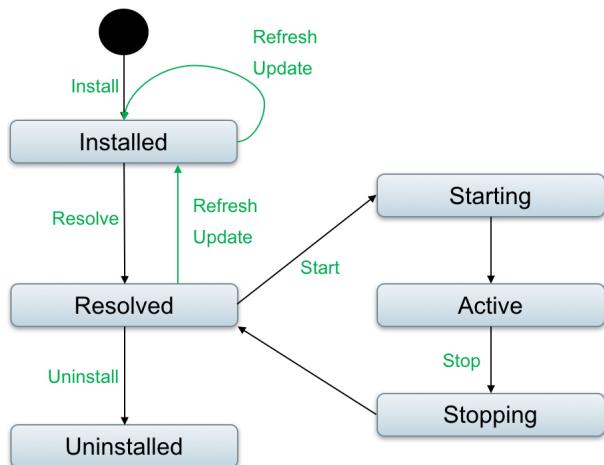


Figure: Bundle lifecycle commands

Links and Resources

- Semantic Versioning Explained: <https://www.osgi.org/wp-content/uploads/SemanticVersioning.pdf>

Exercises

Change the Lifecycle State of an OSGi Bundle

Introduction

OSGi bundles have a manageable lifecycle inside the OSGi container. In this exercise, you'll learn how to start and stop OSGi bundles.

Overview

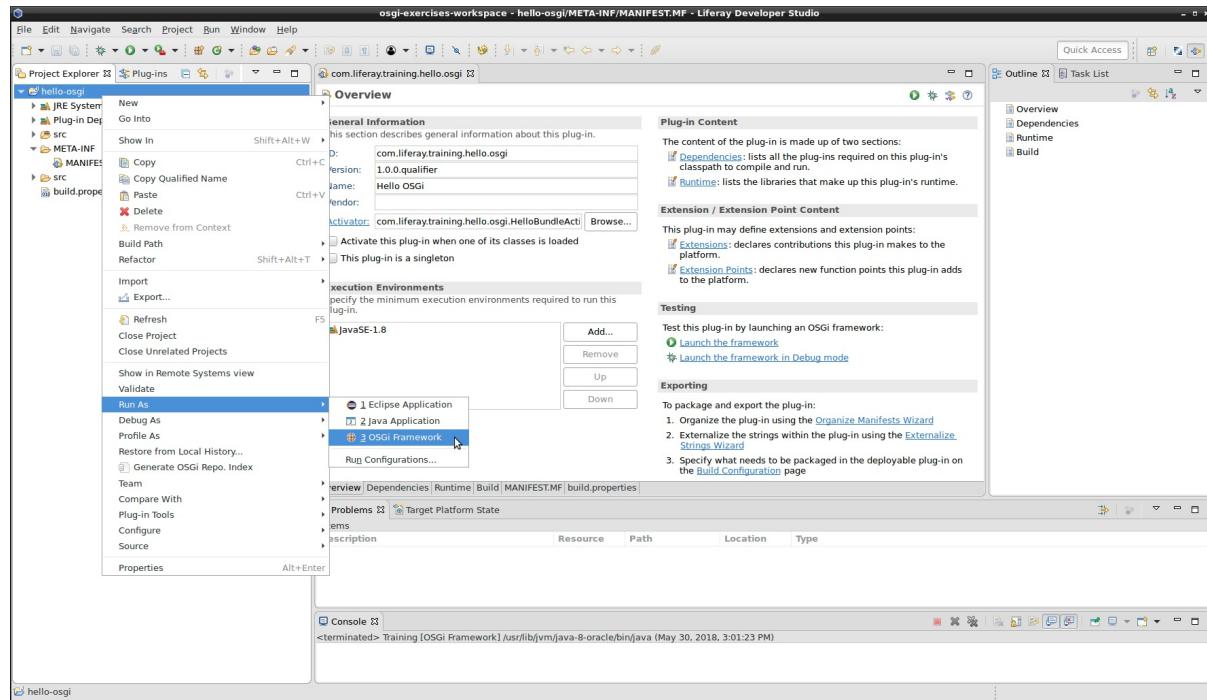
- ① Stop our previously created OSGi bundle
- ② Start the bundles again and watch the console log for "Hello World!!" message

Prerequisites

- The "Hello OSGi!" application finished

Stop Our Previously Created OSGi Bundle

If the *Hello OSGi* application is not running, right-click the *hello osgi* project to open the context menu and click *Run As* → *OSGi Framework*. You should see the *Hello World!!* message on the console.



First, we'll check the lifecycle state of the *hello osgi* bundle.

1. Type **lb** in the console and hit enter.
 - You should see a complete list of the bundles deployed to the OSGi container and their state. Hello OSGi

should be in **Active** state, which means that it has been deployed, installed, and started successfully:

```
Console >
Training [OSGi Framework] /usr/lib/jvm/java-8-oracle/bin/java (May 30, 2018, 6:21:02 PM)
Hello World!!
osgi> lb
START LEVEL 6
ID|State      |Level|Name
0|Active      |0|OSGi System Bundle (3.12.100.v20180210-1608)
1|Active      |4|Console plug-in (1.1.300.v20170512-2111)
2|Active      |4|Apache Felix Gogo Runtime (0.10.0.v201209301036)
3|Active      |4|Apache Felix Gogo Shell (0.10.0.v201212101605)
4|Active      |4|Apache Felix Gogo Command (0.10.0.v201209301215)
5|Active      |4|Hello OSGi (1.0.0.qualifier)
osgi>
```

- o Notice the ID of the bundle in the first column of the list.
2. Type **stop [bundle ID#]** to stop the *Hello OSGi* bundle.
- o You should get a *Goodbye World!!* message from the HelloBundleActivator class on the console.

```
Console >
Training [OSGi Framework] /usr/lib/jvm/java-8-oracle/bin/java (May 30, 2018, 6:37:00 PM)
Hello World!!
osgi> lb
START LEVEL 6
ID|State      |Level|Name
0|Active      |0|OSGi System Bundle (3.12.100.v20180210-1608)
1|Active      |4|Console plug-in (1.1.300.v20170512-2111)
2|Active      |4|Apache Felix Gogo Runtime (0.10.0.v201209301036)
3|Active      |4|Apache Felix Gogo Shell (0.10.0.v201212101605)
4|Active      |4|Apache Felix Gogo Command (0.10.0.v201209301215)
5|Resolved    |4|Hello OSGi (1.0.0.qualifier)
osgi> stop 5
Goodbye World!!
osgi>
```

3. Type **lb** to check the bundle state.
- o The bundle should now be in the *resolved*, state which means that all its dependencies have been satisfied but the bundle is not running. Being in an *installed* state would mean that the bundle was deployed successfully but some of its dependencies were not satisfied.

Start It Again

1. Type **start [bundle ID#]** to start the bundle again.
- o You should get the *Hello World!!* message from the HelloBundleActivator class on the console again:

```
Console >
Training [OSGi Framework] /usr/lib/jvm/java-8-oracle/bin/java (May 30, 2018, 6:47:10 PM)
Hello World!!
osgi> lb
START LEVEL 6
ID|State      |Level|Name
0|Active      |0|OSGi System Bundle (3.12.100.v20180210-1608)
1|Active      |4|Console plug-in (1.1.300.v20170512-2111)
2|Active      |4|Apache Felix Gogo Runtime (0.10.0.v201209301036)
3|Active      |4|Apache Felix Gogo Shell (0.10.0.v201212101605)
4|Active      |4|Apache Felix Gogo Command (0.10.0.v201209301215)
5|Active      |4|Hello OSGi (1.0.0.qualifier)
osgi> stop 5
Goodbye World!!
osgi> lb
START LEVEL 6
ID|State      |Level|Name
0|Active      |0|OSGi System Bundle (3.12.100.v20180210-1608)
1|Active      |4|Console plug-in (1.1.300.v20170512-2111)
2|Active      |4|Apache Felix Gogo Runtime (0.10.0.v201209301036)
3|Active      |4|Apache Felix Gogo Shell (0.10.0.v201212101605)
4|Active      |4|Apache Felix Gogo Command (0.10.0.v201209301215)
5|Resolved    |4|Hello OSGi (1.0.0.qualifier)
osgi> start 5
Hello World!!
osgi>
```

Note that restarting the bundle could also have been done with the **refresh** command.

Components and Services

A Component

An OSGi component is any Java class inside a bundle that is declared to be a component, usually by annotating it with `@Component`. When a class is declared to be a component, it becomes managed by the OSGi runtime. Like a bundle, a component has its own independent lifecycle, with two possible states: `activated` and `deactivated`. A component may contain corresponding lifecycle methods, which are identified by annotations, like `@Activate` and `@Deactivate`. As with bundle activators, those methods can be used to initialize and clean up resources on component activation and deactivation.

Why declare a class to be a Component? In addition to having a distinct lifecycle, a component can publish itself as a *service* and make itself available for other components in the OSGi container. A component may also *consume* services *published* by other components and have both *properties* and a *configuration* managed by the OSGi Configuration Admin subsystem.

Below is an example of a component having an `@Activate` method:

```
package com.liferay.training.component;

import java.util.Map;

import org.osgi.service.component.annotations.Activate;
import org.osgi.service.component.annotations.Component;

@Component
public class TrainingComponent {

    @Activate
    protected void activate(Map properties) {

        System.out.println("I'm active.")
    }
}
```

A Service

A *service* is an OSGi component registered to the OSGi container's *service registry*. Compared to a plain OSGi component, it has two additional characteristics:

- A *service component property*
- It implements an interface

All OSGi services have to implement an interface, as they are always referenced by their interface, leveraging at the same time modularity and loose coupling.

A *service client* doesn't need to know about the *service implementation*. The OSGi framework provides facilities to replace the *service reference* with its implementation at runtime. Multiple implementations of a service may co-exist in the same container. Consuming clients may have *policies* and *filters* for binding specific implementations.

Below is an example of an OSGi service with an interface (API), a service component implementing the interface and a client referencing the service by its interface.

A Service Interface

```
package com.liferay.training.service.api;

public interface EmployeeService {

    public String getName();
    public void setName(String name);
}
```

A Service Implementation

```
package com.liferay.training.service.impl;

import com.liferay.training.example.api.EmployeeService;

import org.osgi.service.component.annotations.Component;

@Component(
    service = EmployeeService.class
)
public class EmployeeServiceImpl implements EmployeeService {

    @Override
    public String getEmployeeName() {

        return _name;
    }

    @Override
    public void setEmployeeName(String name) {

        _name = name;
    }

    private String _name;
}
```

A Service Client

```
package com.liferay.training.client;

import com.liferay.training.example.api.EmployeeService;

import org.osgi.service.component.annotations.Component;
import org.osgi.service.component.annotations.Reference;

@Component
public class EmployeeClient {

    public String getEmployeeName() {
        return employeeService.getEmployeeName();
    }

    @Reference
    EmployeeService employeeService;
}
```

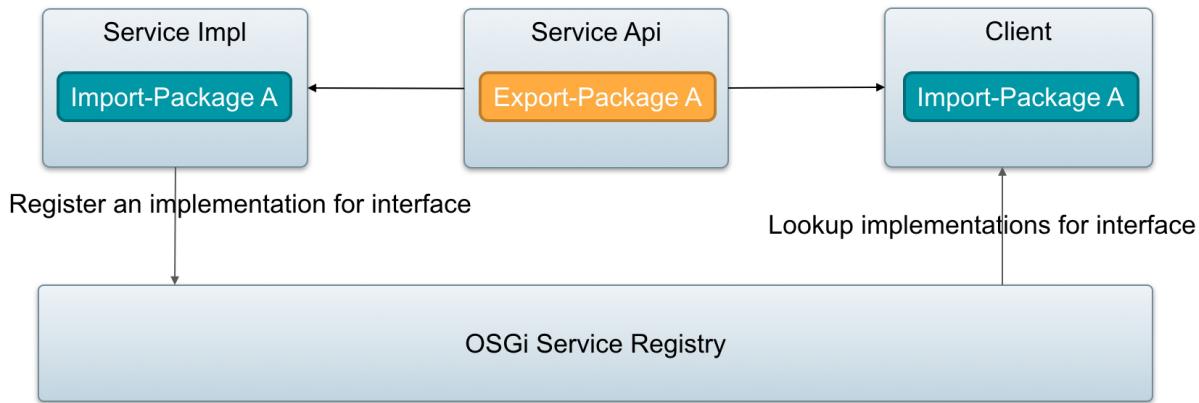
The Service Registry

The OSGi *service registry* subsystem provides mechanisms to:

- *publish* services to the OSGi container

- *look up and bind* published services dynamically
- *hide* the implementation details from the service clients
- manage *multiple* service implementations in the same runtime

The diagram below shows the OSGi service registry pattern. Interface, implementation, and client are, like in the diagram, usually in different bundles.



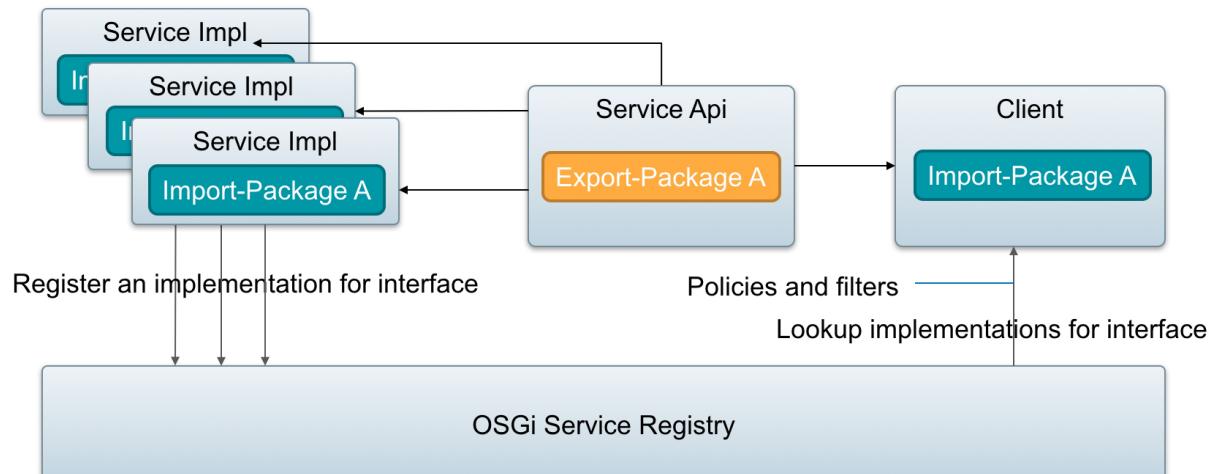
OSGi service registry

The *service API bundle* exports the package, which contains the interface.

The *service implementation bundle* (Service Impl) imports the interface from the API bundle and provides an implementation for the interface. By declaring the implementing class to be a component and defining the interface class in the *component service property*, it can register itself to the OSGi service registry. The registration is usually done with OSGi Declarative Services annotations.

The *service client bundle* contains a service consuming component. This bundle has to import the API bundle because a reference to an OSGi service is always done with the interface.

The dependency injection, a reference to the service, is usually done with a `@Reference` annotation. In case there's no implementation registered, the component's configuration may or may not be satisfied. By default, a component won't start unless all of its references are satisfied, but specific reference policies might allow starting the component even if no implementation for a referenced interface is available. The diagram below shows a scenario with multiple implementations of the same service registered in the container.



Multiple service implementations in the service registry.

If the service client doesn't specify any policies or filters for the service implementation, the first available implementation will be bound to the client component. The client may also define policies allowing multiple implementations to be bound. This is especially useful when implementing factory patterns.

Declarative Services

Declarative services (DS) is a service that handles OSGi dependency injection and allows it to *publish*, *find*, and *bind* services based on XML metadata and annotations. Declarative Services is used in Liferay by default but alternative service implementations like Apache Blueprint can also be used. It is possible to use even both in the same bundle. The specifications can be studied in the OSGi Service compendium.

The standard annotations of declarative services are:

- **@Component:** Identify the annotated class as a Service Component
- **@Activate:** Identify the annotated method as the activate method of a Service Component
- **@Deactivate:** Identify the annotated method as the deactivate method of a Service Component
- **@Modified:** Identify the annotated method as the modified method of a Service Component
- **@Reference:** Inject a service implementation to the annotated variable (or using a setter method)

Declarative Services and Bndtools

By default, Declarative Services expects the service declarations to exist as XML files in the OSGI-INF folder of the bundle. These declaration files may be created and modified manually.

Bndtools is a configuration tool that reads the annotations in the classes and allows the service declarations to be generated automatically at build time.

When bndtools is enabled in the project, it takes the responsibility of writing the manifest metadata. The instructions for the manifest file are written by default in the bnd.bnd file in the project's root directory. Combined with the information extracted from the annotations, bndtools builds the final MANIFEST.MF.

Links and Resources

- OSGi Compendium Download: <https://osgi.org/download/r6/osgi.cmpn-6.0.0.pdf>

Exercises

Create an OSGi Service Using Declarative Services and Bndtools

Introduction

The service registry is one of the most important modularity-enabling concepts of the OSGi framework. Later in this training, you will see that a lot of the development in the Liferay platform deals with OSGi services.

In this exercise, we will be using Bndtools to create the OSGi service, the service's XML metadata, and the bundle's manifest file automatically. Bndtools is also being used in Liferay module development.

We'll create an OSGi bundle containing:

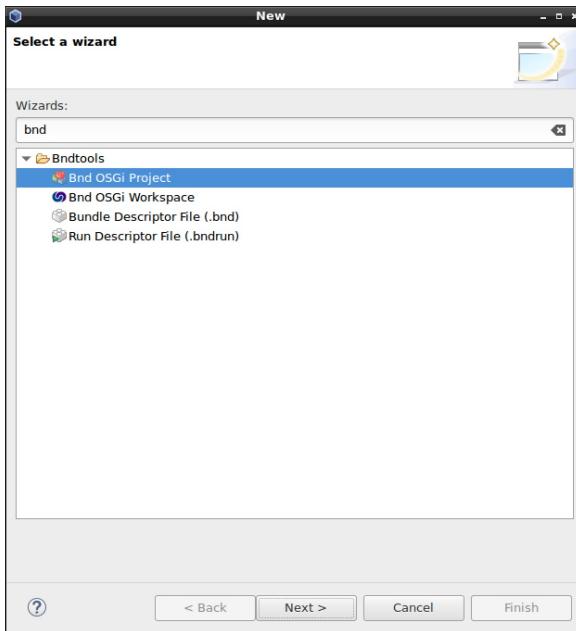
- A service interface
- A service implementation component
- A command interpreter service component for testing

Overview

- ① Create a new Bnd OSGi Project
- ② Create the service interface
- ③ Create the service implementation component
- ④ Create the command interpreter service component
- ⑤ Run and test the application in the Gogo Shell

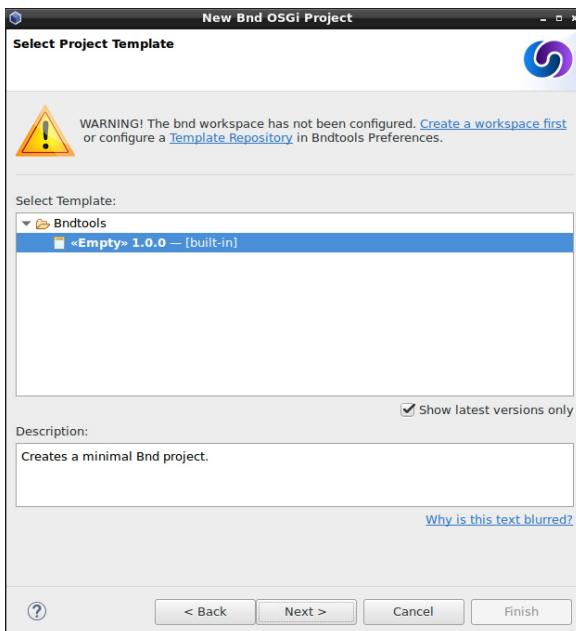
Create a New Bnd OSGi Project

1. **Click** *File → New → Other* on the Dev Studio menu bar to open the new project wizard.
2. **Type** *bnd* in the search bar.
3. **Choose** the *Bnd OSGi Project* project type from the *Bndtools*.
4. **Choose** *Next:*



- You'll get a warning that the "bnd workspace has not been configured". Bndtools uses this workspace as local repository for its various dependencies. We'll need to create the workspace before being able to create our OSGi project.

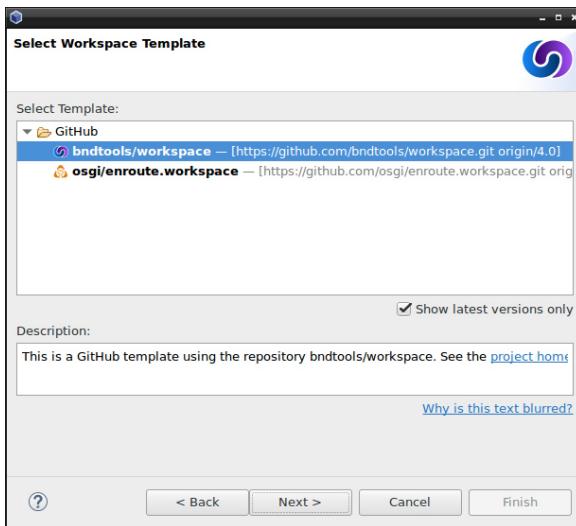
5. Click [Create a workspace first](#) link on the dialog:



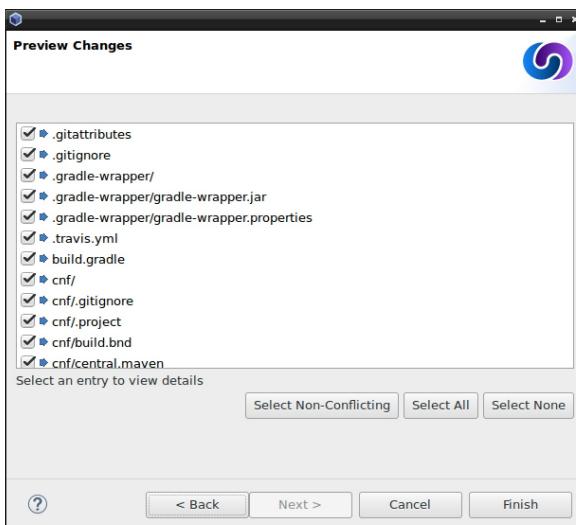
6. Click [Next](#), leaving the defaults:



7. Click **Next**, leaving the defaults as is:

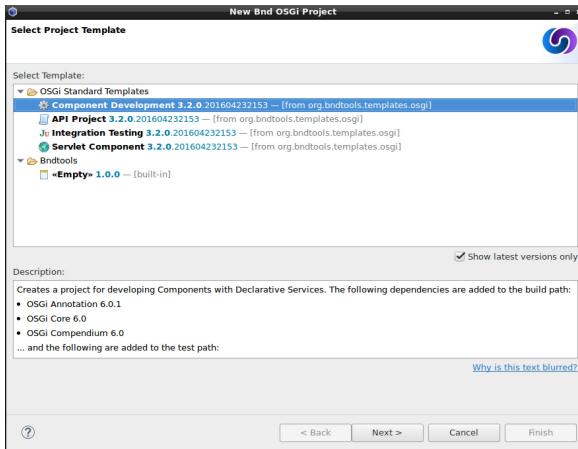


8. Click **Finish** to close the Bnd workspace wizard:



9. **Click Yes** to switch to the Bnd perspective and return to the Bnd OSGi project wizard.

10. **Click Next:**

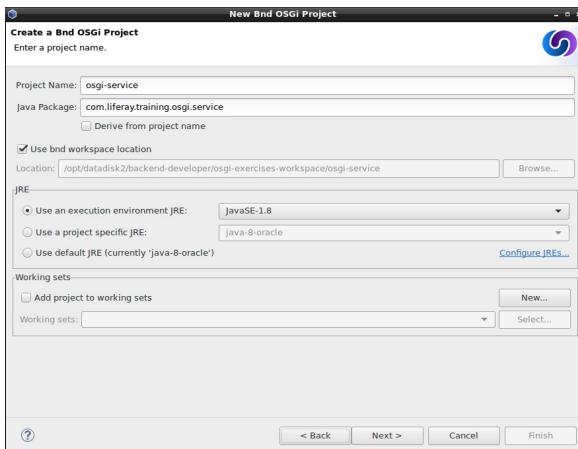


11. **Type osgi-service** for the project name.

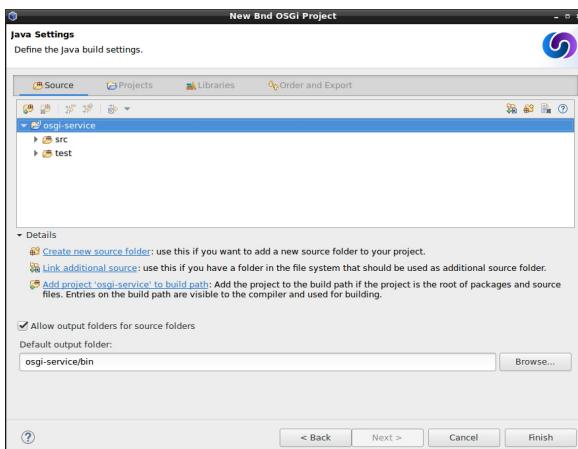
12. **Uncheck Derive from project name.**

13. **Type com.liferay.training.osgi.service** for the Java Package field.

14. **Click Next.**



15. **Click Finish** to close the wizard:



Create the Service Interface

The wizard created an example class in the package `com.liferay.training.osgi.service`, which we are going to remove. After removing the example class, we will create a new interface.

1. **Expand** `osgi-service` → `src` → `com.liferay.training.osgi.service`.
2. **Delete** `Example.java`.
3. **Right-click** on the `com.liferay.training.osgi.service` package and select `New` → `Interface`.
4. **Enter** `ClockApi` as the name of the class.
 - o The package should be prefilled with `com.liferay.training.osgi.service`.
5. **Click** `Finish`.
6. **Create** a method called `getTime()` that returns a String:

```
package com.liferay.training.osgi.service;

public interface ClockApi {

    public String getTime();

}
```

7. **Save** the file.

Create the Service Implementation Component

Next is to create a class implementing the interface. We will use the Declarative Services `@Component` annotation to declare our class component and tell the service registry, using the `service` property, which service we are publishing.

1. **Right-click** on the package `com.liferay.training.osgi.service`.
2. **Select** `New` → `Class`.
3. **Enter** `ClockImpl` in name field.
 - o The package should be prepopulated with `com.liferay.training.osgi.service`.
4. **Click** `Finish`.
5. **Implement** the class in the following way:

```
package com.liferay.training.osgi.service;

import java.util.Date;

import org.osgi.service.component.annotations.Component;

@Component(
    service = ClockApi.class
)
public class ClockImpl implements ClockApi {

    @Override
    public String getTime() {
        return new Date().toString();
    }
}
```

Create the Command Interpreter Service Component

Last, we will create a Gogo Shell command to get the current time using a command called `telltime`. For that, we will create a new class.

1. **Right-click** on the `com.liferay.training.osgi.service` package.

2. **Select New → Class.**
3. **Type** *ClockCommand* to name the field.
 - o The package should be prefilled with *com.liferay.training.osgi.service*.
4. **Click Finish.**
5. **Implement** the class in the following way:

```
package com.liferay.training.osgi.service;

import org.osgi.service.component.annotations.Component;
import org.osgi.service.component.annotations.Reference;

@Component(
    property = {
        "osgi.command.function=telltime",
        "osgi.command.scope=training"
    },
    service = ClockCommand.class
)
public class ClockCommand {

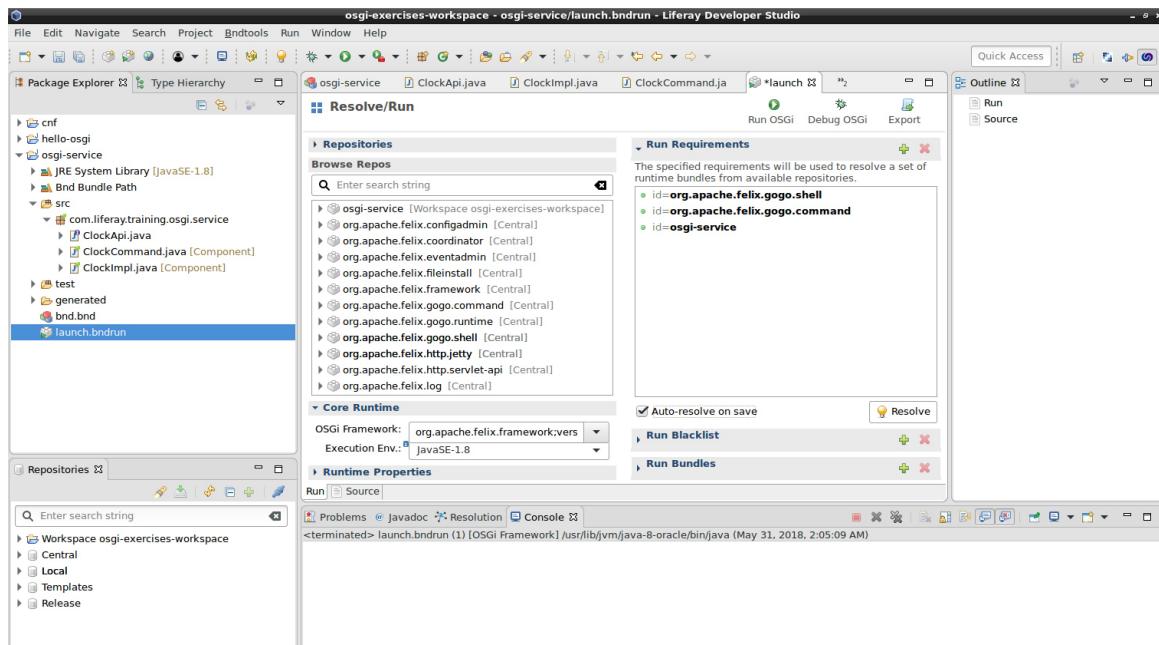
    public void telltime() {
        System.out.println(clock.getTime());
    }

    @Reference
    private ClockApi clock;
}
```

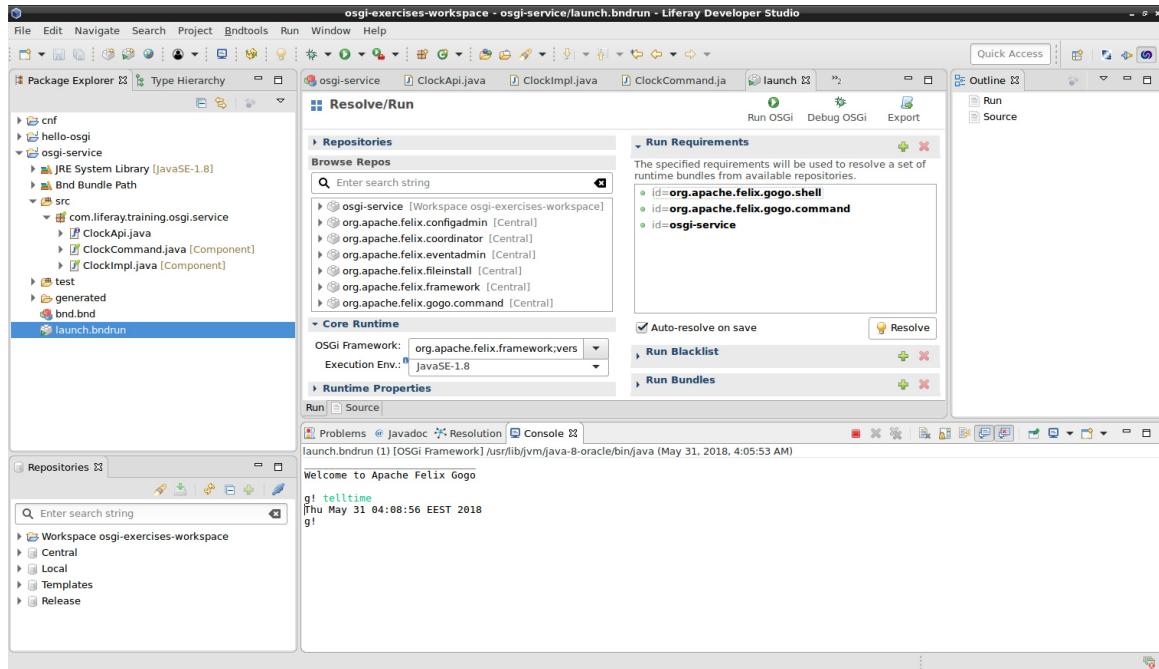
Run and Test the Application in the Gogo Shell

The Bndtools launcher will take care of compiling the MANIFEST.MF when the application is built and run. We will also enable Bndtools to automatically handle any missing requirements:

1. **Click** the *launch.bnrun* file on the Dev Studio package explorer.
2. **Check** the *Auto resolve on save* checkbox.
3. **Save** the file.
4. **Click** the *Run OSGi* button to launch the application:



- o You should see the Gogo Shell in the Console pane of the Dev Studio.
5. Type `telltime` in the console to test our application:



Sharing Features

OSGi applications typically consist of multiple bundles. Bundles by default have a *private* class and package visibility and they do not expose any features to other bundles in the container. Features in this context are:

- Classes
- Packages
- OSGi services and components

There are three primary ways to share features between bundles:

- Export Package - Import Package
- Provide Capability - Require Capability
- Require bundle

Export Package - Import Package

This method is meant to make classes available for other bundles. An exporting bundle explicitly defines which packages to export and make available to other bundles. Correspondingly, the importing bundle explicitly defines which packages to import. The OSGi *resolver* is responsible for wiring the bundles together.

Below is an example of MANIFEST.MF files exporting and importing a package:

Exporting bundle:

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: OSGi Service API
Bundle-SymbolicName: com.liferay.training.osgi.service.api
Bundle-Version: 1.0.0.qualifier
Automatic-Module-Name: com.liferay.training.osgi.service.api
Bundle-RequiredExecutionEnvironment: JavaSE-1.8
Export-Package: com.liferay.training.osgi.service.api
```

Importing bundle:

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: OSGi Service Impl
Bundle-SymbolicName: com.liferay.training.osgi.service.impl
Bundle-Version: 1.0.0.qualifier
Bundle-RequiredExecutionEnvironment: JavaSE-1.8
Import-Package: com.liferay.training.osgi.service.api,
    org.osgi.framework;version="1.3.0"
Automatic-Module-Name: com.liferay.training.osgi.service.impl
```

Provide Capability - Require Capability

A capability describes a functionality for the OSGi Container. In general terms, a capability is defined by two main attributes:

- **Namespace:** a unique identifier, like a package name
- **Attributes:** a list of properties that describe the capability

A capability is a more robust and complex mechanism than export-import and is used, for example, for sharing OSGi services. When you create an OSGi service component, during build time it will be declared as a capability in the MANIFEST.MF.

In Liferay development, bndtools automates many of the build time tasks, like creating the provide and require capability headers automatically based on the component service declarations. On the requiring end, the capability header is generated based on the dependencies defined in the build.gradle file.

Below is an example of providing and requiring headers in the MANIFEST.MF:

Provide Capability:

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

Require Capability:

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

Require Bundle

Requiring a bundle in another bundle defines an explicit dependency contract and a tight coupling between the bundles as all the packages in the imported bundle are exposed automatically to the importing bundle.

Generally speaking, this approach should not be used.

Links and Resources

- Raymond Auge's Blog About Using Requirements and Capabilities: <https://blog.osgi.org/2015/12/using-requirements-and-capabilities.html>

Exercises

Sharing Features with Export-Import

Introduction

OSGi is a framework for modular development. When creating an OSGi applications, the application typically consists of multiple bundles.

This simple exercise demonstrates feature-sharing capabilities of OSGi by using the *Import-Package - Export-Package* mechanism.

Although more complex, the *Provide-Capability - Require-Capability* used in OSGi services wiring works fundamentally the same way. Unlike *Export-Package - Import-Package*, it is also handled automatically in Liferay module development by Bndtools when using Declarative Services annotations.

In this exercise, we will again be using the OSGi Plug-in project type to create:

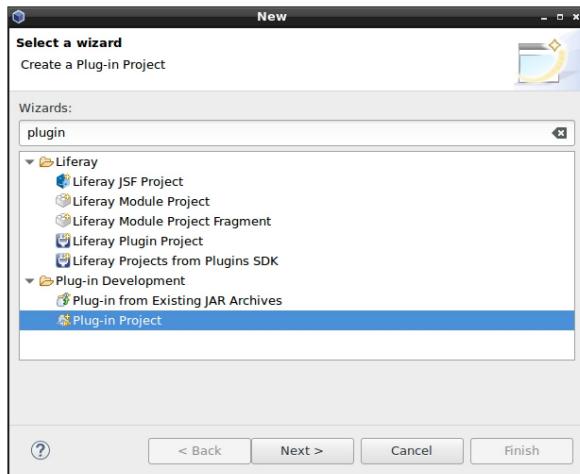
1. An API bundle with an interface exporting the API package
2. An implementation bundle importing the API package and implementing the interface

Overview

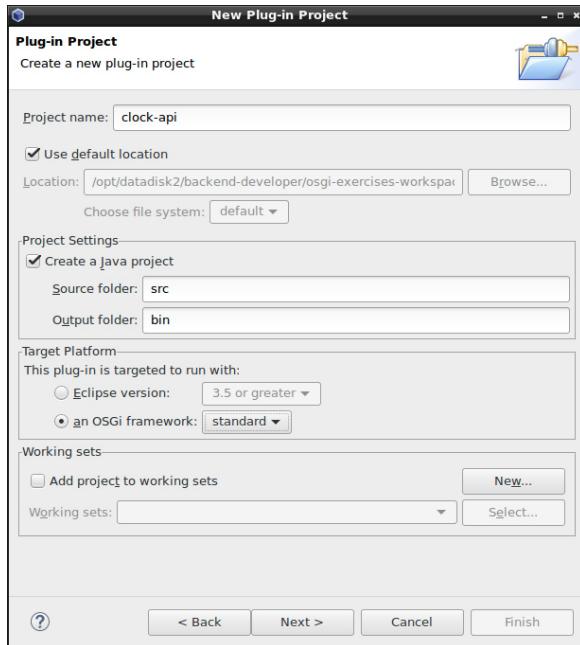
- 1 Create an API bundle
- 2 Create an interface in the API bundle
- 3 Create Export-Package headers in the MANIFEST.MF
- 4 Create an implementation bundle
- 5 Create an implementation class and Import-Package headers in the MANIFEST.MF
- 6 Create the Implementation Class
- 7 Run and test

Create an API Bundle

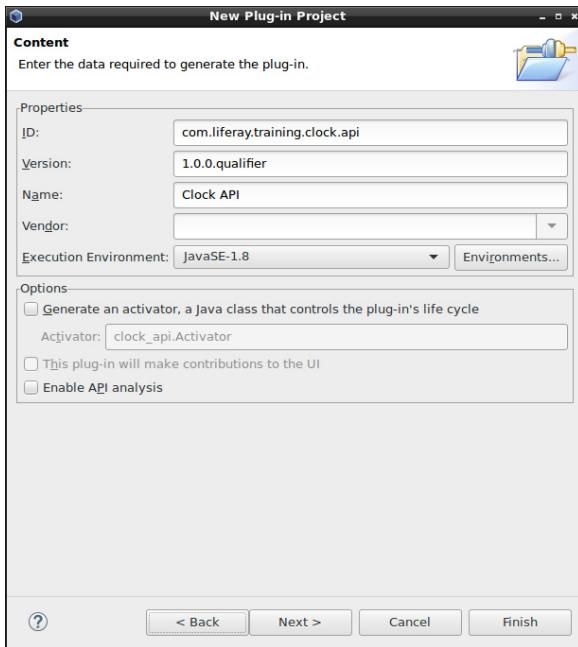
1. **Click** *File → New → Other* on the Dev Studio menu bar to open the new project wizard.
2. **Type** *plugin* in the search bar.
3. **Choose** *Plug-in Project* project type and click *Next*.



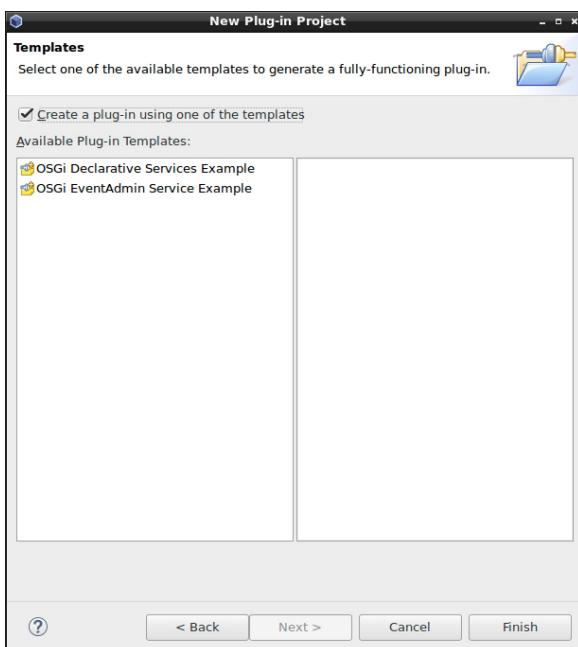
4. Type *clock-api* in the *Project Name* field.
5. Choose **standard** for the target platform.
 - o Make sure the radio button for *an OSGi framework* is selected.
6. Click **Next**.



7. Type the following into their appropriate fields:
 - o **ID:** com.liferay.training.clock.api
 - o **Name:** Clock API
8. Uncheck **Generate an activator...**



9. Click **Finish** to close the wizard.



When prompted, you can change perspectives to the Plug In perspective. You can always change it by clicking the icons at the top right.

Create an Interface in the API Bundle

1. Click the clock-api project, if it's not selected.
2. Click **File → New → Interface** on the Dev studio menu bar.
3. Enter **ClockApi** in the **Name** field.
4. Enter **com.liferay.training.clock.api** in the **Package** field.
5. Click **Finish** to close the wizard.
6. Create a `getTime()` method declaration in the interface as follows:

```
package com.liferay.training.clock.api;

public interface ClockApi {
    public String getTime();
}
```

Create Export-Package header in the MANIFEST.MF

To be able to make our interface accessible to other bundles in the OSGi container, we have to use the Export-Package manifest header.

1. **Expand** clock-api → META-INF.
2. **Double-click** the `MANIFEST.MF` file.
3. **Click** the MANIFEST.MF tab on the panel to open the source view.
4. **Add** the Export-Package header to the end of the file:
 - o The file should look as follows
 - o Note that the editor supports auto-complete using control/command + spacebar

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Clock API
Bundle-SymbolicName: com.liferay.training.clock.api
Bundle-Version: 1.0.0.qualifier
Automatic-Module-Name: com.liferay.training.clock.api
Bundle-RequiredExecutionEnvironment: JavaSE-1.8
Export-Package: com.liferay.training.clock.api
```

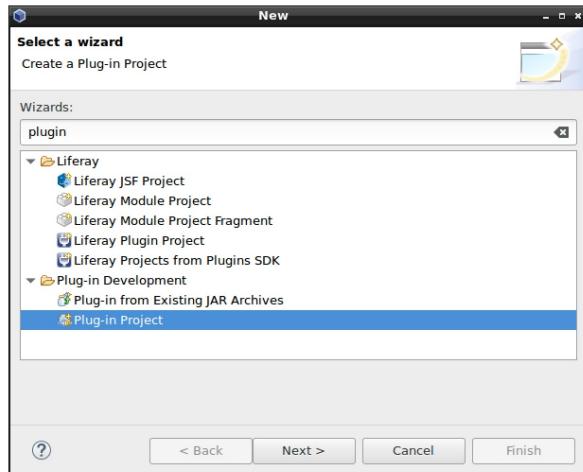
5. **Save** the file.

Our interface is now ready to be implemented.

Create an Implementation Bundle

Let's create the implementation bundle:

1. **Click** `File` → `New` → `Other` on the Dev Studio menu bar to open the new project wizard.
2. **Type** `plugin` in the search bar.
3. **Choose** `Plug-in Project` project type and click `Next`:



4. **Type** `clock-impl` in the `Project Name` field.

5. **Choose standard** for the target platform.
 - o Make sure the radio button for *an OSGi framework* is selected.
6. **Click Next.**
7. **Type** the following into their appropriate fields:
 - o **ID:** com.liferay.training.clock.impl
 - o **Name:** Clock Impl
8. **Uncheck Generate an activator** if it isn't already unchecked.
9. **Click Finish** to close the wizard.

Add the Import-Package Header in the MANIFEST.MF

If you started to create a class to implement the *ClockApi* interface, you would notice that the *ClockApi* interface is not available for the class. To make the interface available, we must first import the API package into our bundle.

1. **Expand** *clock-impl* → META-INF.
2. **Click** the *MANIFEST.MF* file on the Package Explorer.
3. **Click** the MANIFEST.MF tab on the panel to open the source view.
4. **Add** the Import-Package header at the bottom of the file:
 - o The file should look as follows
 - o Note again that the editor supports autocompletion:

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Clock Impl
Bundle-SymbolicName: com.liferay.training.impl
Bundle-Version: 1.0.0.qualifier
Automatic-Module-Name: com.liferay.training.impl
Bundle-RequiredExecutionEnvironment: JavaSE-1.8
Import-Package: com.liferay.training.clock.api
```

5. **Save** the file.

Since we're just demonstrating how OSGi shares features, we don't need to make the implementation class a component. In real life, the implementation class would be annotated with `@Component`.

Now that we have the implementation bundle importing and implementing the API bundle, both bundles are ready to be deployed.

Run and Test

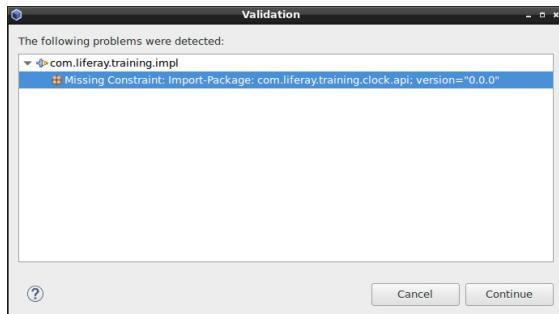
1. **Right-click** on the *clock-impl* project to open the context menu.
2. **Choose** Run As → OSGi Framework.
 - o The Gogo Shell should open in the console panel.
3. **Type** the *lb* command to check that our bundles are deployed and active:

```
Console [x]
Training [OSGi Framework] /usr/lib/jvm/java-8-oracle/bin/java (May 30, 2018, 6:21:02 PM)
Hello World!!
osgi> lb
START LEVEL 6
| ID | State   | Level | Name
| 0 | Active  | 0 | OSGi System Bundle (3.12.100.v20180210-1608)
| 1 | Active  | 4 | Console plug-in (1.1.300.v20170512-2111)
| 2 | Active  | 4 | Apache Felix Gogo Runtime (0.10.0.v201209301036)
| 3 | Active  | 4 | Apache Felix Gogo Shell (0.10.0.v201212101605)
| 4 | Active  | 4 | Apache Felix Gogo Command (0.10.0.v201209301215)
| 5 | Active  | 4 | Hello OSGi (1.0.0.qualifier)
osgi>
```

- o Let's test a little more. What would happen if the *clock-api* isn't exporting the package

```
com.liferay.training.clock.api ?
```

4. **Stop** the OSGi container by typing `exit` in the Gogo Shell or clicking the red square *Stop* icon.
5. **Remove** the header `Export-Package` from `clock-api` bundle's `MANIFEST.MF` file.
6. **Save** the file.
7. **Right-click** on the `clock-impl` project to open the context menu.
8. **Select** Run As → OSGi Framework.
 - o You will get an error message:



9. **Click** Continue and type the `/b` command again to check the bundle state:

```
Console 
Training [OSGi Framework] /usr/lib/jvm/java-8-oracle/bin/java (May 31, 2018, 7:13:58 PM)
!ENTRY org.eclipse.osgi 4 2018-05-31 19:13:59.593
!MESSAGE Bundle initial@reference:file:../../../../opt/datadisk2/backend-developer/osgi-exercises-workspace/clock-impl/ was not resolved.
osgi> /b
START LEVEL 6
ID|State      |Level|Name
 0|Active     | 0|OSGi System Bundle (3.12.100.v20180210-1608)
 1|Installed  | 4|Clock Impl (1.0.0.qualifier)
 2|Active     | 4|Apache Felix Gogo Runtime (0.10.0.v201209301036)
 3|Active     | 4|Apache Felix Gogo Command (0.10.0.v201209301215)
 4|Active     | 4|Console plug-in (1.1.300.v20170512-2111)
 5|Active     | 4|Apache Felix Gogo Shell (0.10.0.v201212101605)
 6|Active     | 4|Hello OSGi (1.0.0.qualifier)
 7|Active     | 4|Clock API (1.0.0.qualifier)
osgi>
```

10. **Stop** the OSGi container, restore the header, and try again.

OSGi Architecture



Figure: OSGi main specifications.

OSGi has two main specifications: OSGi Core and OSGi compendium. The **Core specification** specifies APIs, which are the bare minimum to run OSGi applications and which every framework implementation must implement.

There are several core implementations like:

- Apache Felix
- Eclipse Equinox
- Knopflerfish
- ProSyst

The **Compendium** specification is a collection of additional OSGi framework services and APIs, which may be used in a modular fashion, meaning that implementations may only implement selected parts of the compendium.

When creating Liferay modules, you'll see the Core implementation dependency in the project dependencies. Depending on the module type, there also might be the whole compendium (like in the example below) or just a certain subsystem dependency.

```
dependencies {
    ...
    compileOnly group: "org.osgi", name: "org.osgi.core", version: "6.0.0"
    compileOnly group: "org.osgi", name: "osgi.cmpn", version: "6.0.0"
}
```

All in all, the OSGi framework consists of five layers, also called subsystems:

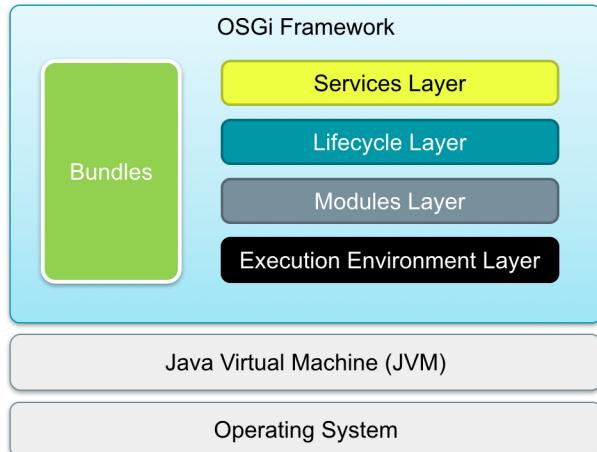


Figure: OSGi layers.

Bundles: Bundles are self-contained, manageable, and testable units of deployment. At file level, they are regular JAR files with OSGi-specific headers in their MANIFEST.MF file.

Services: The services layer contains the OSGi service registry, which allows you to publish, find, and bind services dynamically.

Lifecycle: The lifecycle layer takes care of a bundle's lifecycle transitions:

- Install
- Start
- Stop
- Update
- Uninstall

Modules: The modules layer manages bundle modularity and takes care of class loader delegation and feature-sharing like exports and imports.

Execution Environment: The execution environment layer manages the Java runtime compatibility since OSGi environments can be implemented on different Java editions. A bundle can define the Java platform compatibility with the header *Bundle-Required-Execution-Environment*

Links and Resources

- **OSGi Specifications** <https://www.osgi.org/developer/specifications>

OSGi Benefits

True Modularization

Developing applications based on the OSGi framework removes many of the restrictions of traditional web application development and allows for a truly modular application design and architecture.

In traditional Java EE web application development, because of the class-loading and contextual restrictions, web applications tended to end up being monolithic and difficult to maintain.

The adoption of the OSGi Framework removes most of those restrictions. It takes control of application class-loading, provides a dynamic application lifecycle management, and requires modules to explicitly declare what they expose of themselves. The service registry allows you to publish and bind services in a granular and controlled way across the module boundaries. Bundle identity management with semantic versioning provides a way for multiple versions of the same bundle or library to co-exist within the same container.

Solving "JAR Hell"

If you have created complex web applications before, you probably have either encountered it by yourself or have at least heard about it: "JAR hell". At the heart of the problem lies the traditional Java class loading.

Class loading is a mechanism of the Java runtime environment to dynamically load classes. Practically, it means scanning all the jars on the classpath to find a certain, requested class.

What is a **class loader**? Each object in a Java runtime environment is linked to the runtime environment via its class loader, which is just another Java class taking care of loading the classes. The class loader is a contextual boundary. Objects loaded with one class loader cannot directly access the contextual information like instance state from an object under a different class loader.

According to the Servlet Specification, each Java EE web application has its own class loader in the runtime environment. In addition to class loaders of other web applications, there are also other class loaders like the common class loader of the application server and Java Bootstrap class loader.

How Does Class Loading Work?

Class loaders are arranged in a hierarchical tree. When class scanning is executed, it is carried out in an hierarchical manner which can, depending on the runtime environment or application server, be configured. The most common class loading mode is the parent first mode.

In **parent first** class loading mode, the class loader first tries to retrieve the requested class from its parent loader. If the parent loader is unable to find the requested class, it asks its parent and so on. If the last class loader in the hierarchy is not able to find the class, the initial class loader tries to load it from its own context.

The class loading principles are:

- **Delegation:** forward a class loading request to a parent classloader
- **Visibility:** the child class loader can see the parent loader's classes but not vice versa
- **Uniqueness:** a class can be loaded only once

How does that work in practice? Below is an example of the Tomcat class loading process. When a web application asks for a class, the first one to look for it is the common class loader. From there, the lookup goes to the top and back to the web application, if the class was not found.

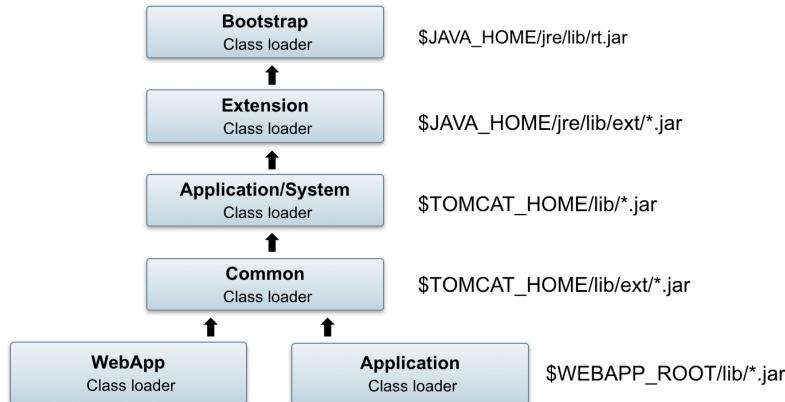


Figure: Tomcat class loading

How could this be an issue? One problematic scenario is caused by transitive dependencies. The library you are requiring requires another library, which might require another library, and so on. All the required libraries have to be available to the web application.

To illustrate this, let's consider the following scenario: a web application has a direct dependency on *Library X* and on *Library 1*. *Library 1* also has a transitive dependency on the same *Library X*. Both the web application and Library 1 require the same version, and no problem arises.

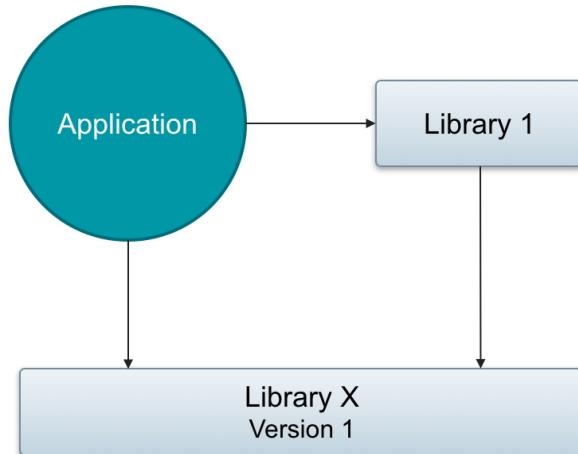


Figure: Class loading with a transitive dependency

In the second scenario, the web application requires version 1 of *Library X* but the dependent library *Library 1* requires version 2 of the same library. If the library versions are not compatible, they cannot coexist under the same class loader because either the web application or Library 1 could get into a situation, where classes from the wrong version were loaded. The Java class loader only tries to find the requested class, but doesn't care about the version.

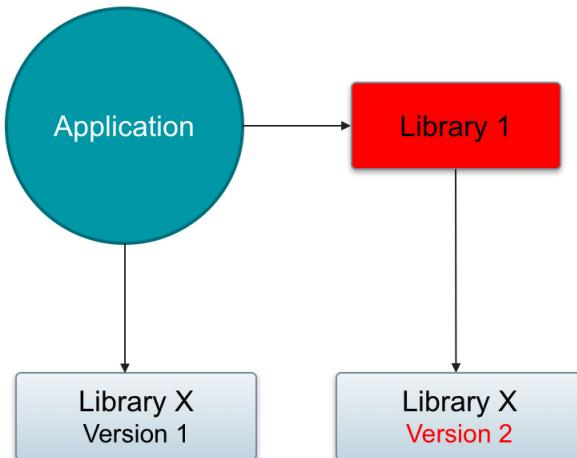


Figure: Class loading with a conflicting dependency.

In the third scenario, things gets even more complicated if Tomcat and both web applications are requiring a different version of the same library.

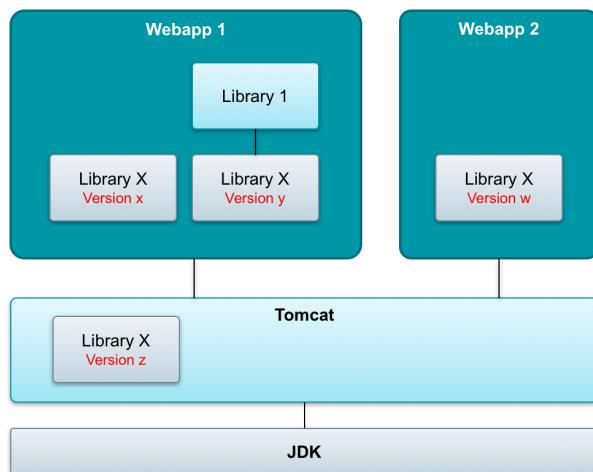


Figure: Class loading with conflicting dependencies

In this kind of scenario, dependent classes could be loaded by the web application from a conflicting library version, ending up in exceptions like:

- `ClassNotFoundException`
- `NoClassDefFoundError`
- `ClassCastException`
- ...

This brings the application to the state known as **JAR hell**.

How does OSGi Help?

In the OSGi environment, each bundle has its own class loader, but the class loader delegation is completely managed by the OSGi container.

As all bundles have a unique identifier, the OSGi container can locate the bundle directly without needing to scan the whole class path, which in turn makes OSGi class loading more efficient.

As the class and package visibility are by default private and have to be declared explicitly, no bundle ends up loading classes from another bundle by accident.

These benefits can be illustrated with the following diagrams, which display the difference of application development based on the OSGi framework compared to traditional Java EE web application development:

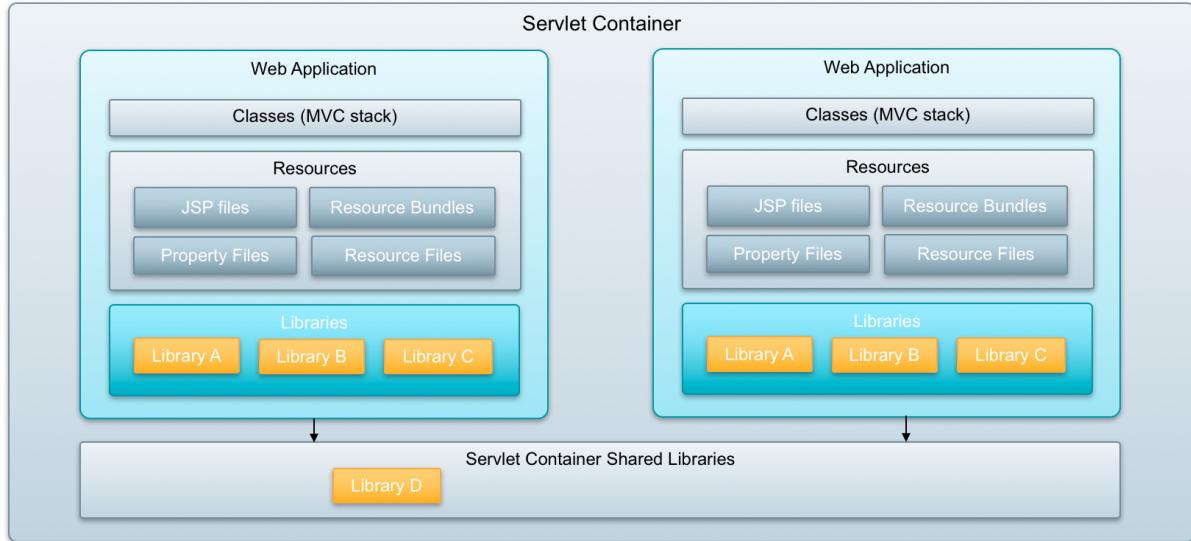


Figure: Traditional web application architecture

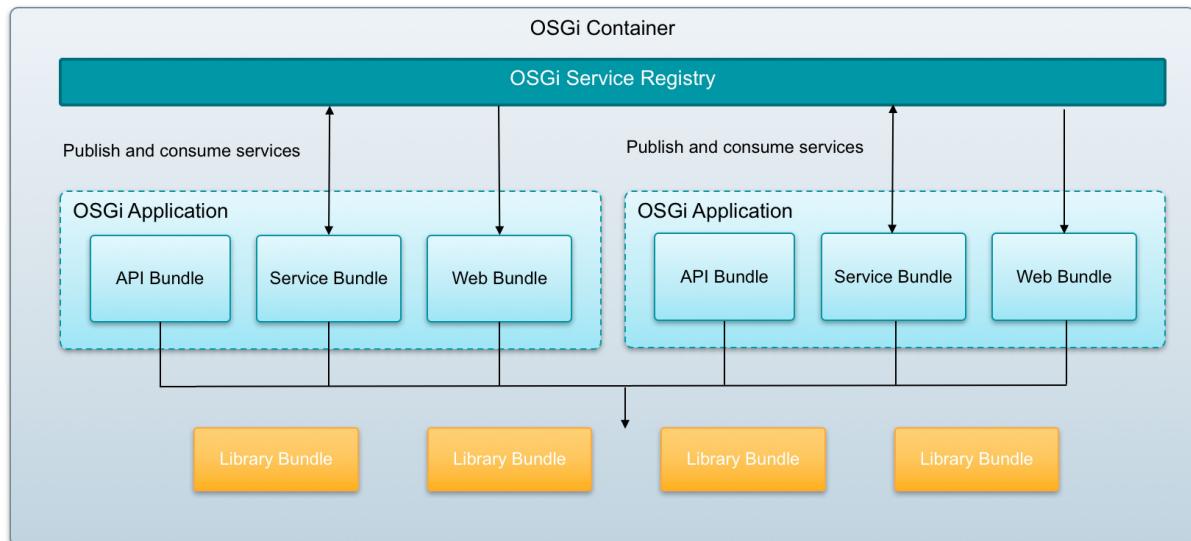


Figure: OSGi application architecture

Note on Java 9 Classloading

In Java 9, with project Jigsaw, the standard Java classloading mechanism becomes similar to OSGi. JARs can be declared modules and run with their own class loaders. This allows multiple versions of libraries to co-exist in the same runtime.

Further OSGi Benefits

The OSGi framework service component model, which requires interfacing that enforces loose coupling, making the code more modular and reusable. This also leverages consistency both in the code and in the design patterns.

Chapter 3: Liferay OSGi Container

Chapter Objectives

- Understand How the OSGi Framework is Integrated in Liferay
- Get to Know Liferay Module Project Types
- Get to Know and Work with the Liferay Workspace Environment

Liferay's OSGi Container

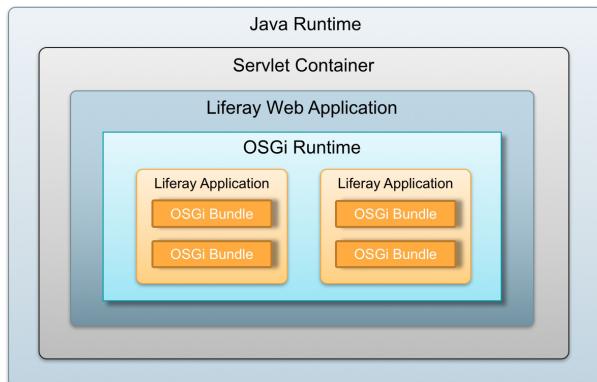
So far we have learned that OSGi applications are usually made up of multiple modules that share features with each other. The modules, called bundles, need a runtime environment called an OSGi container.

The Liferay platform is built on OSGi technology. Liferay applications are OSGi applications, and the platform has an embedded OSGi container. The container implementation is standard-compliant, which means that you can run any standard OSGi bundle inside the container.

The diagram below illustrates the structure of an OSGi container running standalone in a standard JVM:



The diagram below shows how the OSGi runtime is embedded into the Liferay platform:



Liferay DXP is still a Java EE web application running in a servlet container. But now it also has an embedded OSGi container where Liferay's OSGi applications live.

Components and Services in Liferay

OSGi components, services, and the service registry are the main ways to achieve modularity within the OSGi framework. Liferay follows these paradigms. Functionalities and services in Liferay are mostly implemented as OSGi components, and the platform's core services are exposed through the OSGi service registry.

Below are some examples of Liferay code demonstrating how Liferay leverages OSGi patterns and how Liferay functionalities are implemented as OSGi components:

The first example is a Liferay portlet. In traditional portlet development, the portlet declaration and configuration was done in XML files. Starting with Liferay DXP, the portlet is now configured with the portlet's component properties:

```

@Component(
    immediate = true,
    property = {
        "com.liferay.portlet.add-default-resource=true",
        "com.liferay.portlet.application-type=full-page-application",
        "com.liferay.portlet.application-type=widget",
        "com.liferay.portlet.css-class-wrapper=portlet-blogs",
        "com.liferay.portlet.display-category=category.collaboration",
        "com.liferay.portlet.header-portlet-css=/blogs/css/main.css",
        "com.liferay.portlet.icon=/blogs/icons/blogs.png",
        "com.liferay.portlet.preferences-owned-by-group=true",
        "com.liferay.portlet.private-request-attributes=false",
        "com.liferay.portlet.private-session-attributes=false",
        "com.liferay.portlet.render-weight=50",
        "com.liferay.portlet.scopeable=true",
        "com.liferay.portlet.struts-path=blogs",
        "com.liferay.portlet.use-default-template=true",
        "javax.portlet.display-name=Blogs", "javax.portlet.expiration-cache=0",
        "javax.portlet.init-param.always-display-default-configuration-icons=true",
        "javax.portlet.init-param.template-path=/",
        "javax.portlet.name=" + BlogsPortletKeys.BLOGS,
        "javax.portlet.resource-bundle=content.Language",
        "javax.portlet.security-role-ref=guest,power-user,user",
        "javax.portlet.supported-public-render-parameter=categoryId",
        "javax.portlet.supported-public-render-parameter=resetCur",
        "javax.portlet.supported-public-render-parameter=tag",
        "javax.portlet.supports.mime-type=text/html"
    },
    service = Portlet.class
)
public class BlogsPortlet extends BaseBlogsPortlet {

    ...
}

```

The second example is a form text field:

```

@Component(
    immediate = true,
    property = {
        "ddm.form.field.type.description=text-field-type-description",
        "ddm.form.field.type.display.order=Integer=2",
        "ddm.form.field.type.group=basic", "ddm.form.field.type.icon=text",
        "ddm.form.field.type.js.class.name=Liferay.DDM.Field.Text",
        "ddm.form.field.type.js.module=liferay-ddm-form-field-text",
        "ddm.form.field.type.label=text-field-type-label",
        "ddm.form.field.type.name=text"
    },
    service = DDMFormFieldType.class
)
public class TextDDMFormFieldType extends BaseDDMFormFieldType {

    @Override
    public Class<? extends DDMFormFieldTypeSettings>
        getDDMFormFieldTypeSettings() {

        return TextDDMFormFieldTypeSettings.class;
    }

    @Override
    public String getName() {
        return "text";
    }
}

```

The third example is an Audience Targeting application rule:

```
@Component(
    immediate = true,
    service = Rule.class
)
public class LanguageRule extends BaseJSPRule {

    @Activate
    @Override
    public void activate() {
        super.activate();
    }

    @Deactivate
    @Override
    public void deActivate() {
        super.deActivate();
    }

    @Override
    public boolean evaluate(
        HttpServletRequest request, RuleInstance ruleInstance,
        AnonymousUser anonymousUser)
    throws Exception {
        ...
    }
}
```

Wrapping It Up

Liferay has an embedded OSGi container. All of the Liferay platform's core applications are OSGi applications. Services and functionalities within the Liferay platform leverage the OSGi component and service model.

A typical Liferay development flow has the following steps:

1. If creating a service: find or create an interface to implement.
2. Find a superclass to extend (if creating a Liferay service component).
3. Create a component.
4. If creating a service, define the `service` component property.
5. Define component properties.
6. Implement methods.
7. Reference the component or service from other, consuming components.

Working With Liferay Workspace

Liferay Workspace is a generated wrapper environment for Liferay projects. It provides you with all the tools needed to create and build Liferay modules and can be used from within a Java IDE or independently.

Liferay Workspace is not mandatory for Liferay development, but in most cases it simplifies and speeds up the development process significantly.

Liferay Workspace provides:

- Blade CLI tools
- Gradle wrapper when Gradle is chosen as the build tool

Blade CLI

Blade is an acronym for *Bootstrap Liferay Advanced Developer Environment*, which is a set of backbone command line tools for creating and managing Liferay module projects and the Liferay Workspace environment. Blade CLI is an independent set of tools and can also be used outside of the Liferay Workspace.

The most relevant Blade CLI commands are listed below:

- **convert**: Converts a Plugins SDK plugin project to a Gradle Workspace project
- **create**: Creates a new Liferay module project from available templates
- **deploy**: Builds and deploys bundles to the Liferay module framework
- **gw**: Executes a Gradle command using the Gradle Wrapper, if detected
- **help**: Gives help on a specific command
- **init**: Initializes a new Liferay Workspace
- **install**: Installs a bundle into Liferay's module framework
- **open**: Opens or imports a file or project in Liferay IDE
- **samples**: Generates a sample project
- **server**: Starts or stops a server defined by your Liferay project
- **sh**: Connects to Liferay, executes a Gogo command, and returns output
- **update**: Updates Blade CLI to latest version
- **version**: Displays version information about Blade CLI

Gradle Wrapper

Gradle is the default build and dependency management tool for Liferay plugin development. Liferay Workspace ships with Gradle Wrapper scripts taking care of downloading and making a compatible version of Gradle available. That way, you don't have to install Gradle manually in your development environment and also ensures that everybody who uses the project builds it with exactly the same version of Gradle.

Available Gradle Tasks are listed in the Gradle Tasks pane of the Liferay Workspace perspective of the IDE. The perspective can be selected from Dev Studio's *Window* menu. By default, the Gradle Tasks pane is displayed in the perspective's upper right corner:

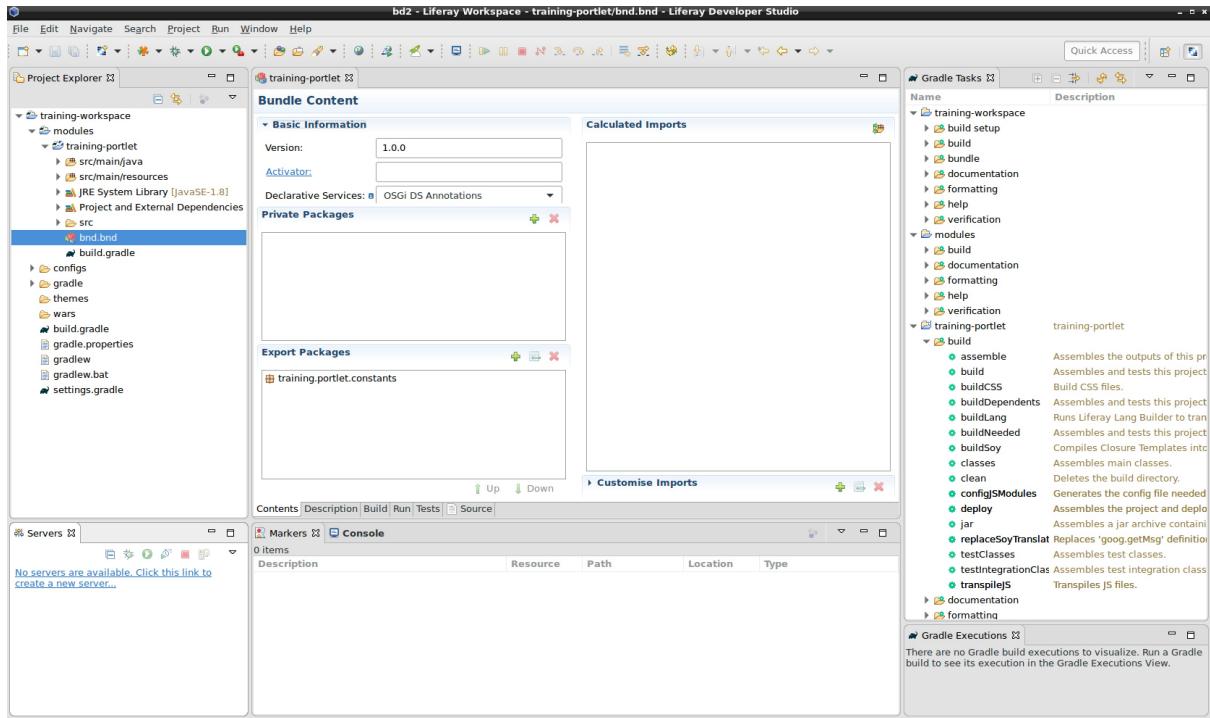


Figure: Liferay Workspace Gradle tasks

From the command line, the available tasks can be listed by running the wrapper script in the root folder of the Liferay Workspace:

```

liferay@liferay-VirtualBox:/opt/liferay-workspace$ ./gradlew tasks
:tasks

-----
All tasks runnable from root project
-----

Build tasks
-----
assemble - Assembles the outputs of this project.
build - Assembles and tests this project.
clean - Deletes the build directory.

Build Setup tasks
-----
init - Initializes a new Gradle build. [incubating]
wrapper - Generates Gradle wrapper files. [incubating]

Bundle tasks
-----
createToken - Creates a.liferay.com download token.
distBundleTar - Assembles the Liferay bundle and zips it up.
distBundleZip - Assembles the Liferay bundle and zips it up.
initBundle - Downloads and unzips the bundle.

...

```

Liferay Workspace Structure

By default, a Liferay Workspace is generated with the following files and folders:

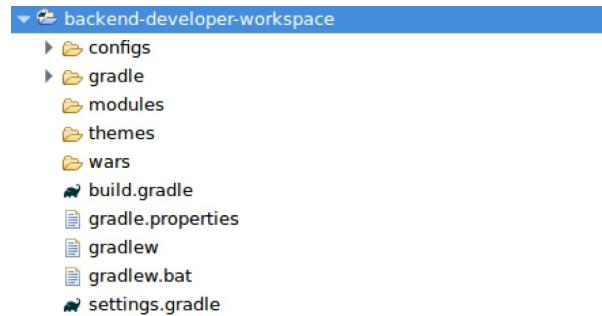


Figure: Liferay Workspace folder structure

- **bundles (generated)**: the default folder for Liferay Portal bundles
- **configs**: holds the configuration files for different environments
- **gradle**: holds the Gradle Wrapper used by your workspace
- **modules**: holds your custom modules
- **plugins-sdk (generated)**: holds plugins to migrate from previous releases
- **themes**: holds your custom themes, which are built using the Theme Generator
- **wars**: holds traditional WAR-style web application projects
- **build.gradle**: the common Gradle build file
- **gradle-local.properties**: sets user-specific properties for your workspace
- **gradle.properties**: specifies the Workspace's project locations and Liferay's server configuration globally
- **gradlew**: executes the Gradle command wrapper
- **settings.gradle**: applies plugins to the workspace and configures its dependencies

Maven Support

You can choose between Gradle and Maven as build system when you create a new Liferay workspace:

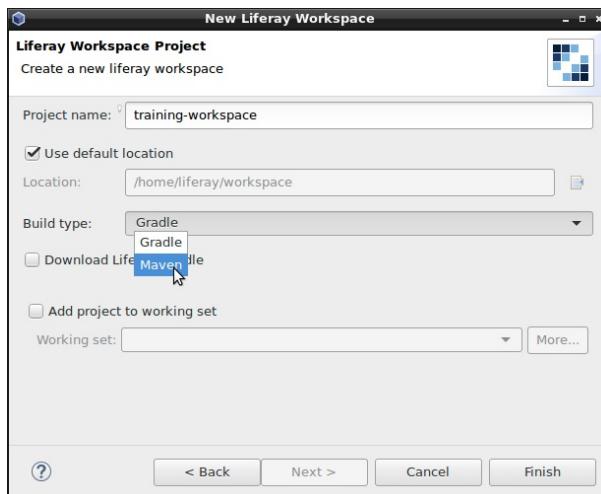


Figure: Selecting Maven for Liferay new Liferay Workspace.

Liferay Maven Workspace can also be created from the command line with:

```
mvn archetype:generate \
-DarchetypeGroupId=com.liferay \
-DarchetypeArtifactId=com.liferay.project.templates.workspace \
-DgroupId=[GROUP_ID] \
-DartifactId=[WORKSPACE_NAME] \
-Dversion=[VERSION]
```

For more information on how to use Liferay Workspace with Maven, see the developer documentation at:
https://dev.liferay.com/develop/tutorials/-/knowledge_base/7-1/maven-workspace

Using Liferay Workspace without IDE

Liferay Workspace can be used without an IDE as well. Below is an example of the steps required to build a Workspace project which, for example, has been downloaded from GitHub:

1. Have a Java 8 JDK installed on your computer
2. Clone a Liferay workspace project containing Liferay modules from GitHub
3. Run the gradle build tool script (example for Linux):

```
PATH_TO_LIFERAY_WORKSPACE/gradlew clean build
```

After your build has completed successfully, the deployables can be found in the *build/libs* directories of the respective project folders.

Setting Up the Liferay Portal for Liferay Workspace

If you use Tomcat as your development server, you can choose to download a Liferay Tomcat bundle and create a server in the Dev Studio workspace using the Liferay 7.x server adapter, or to let Liferay Workspace do the work for you.

When you start a new Liferay Workspace project, you can choose to download a Liferay bundle. If the bundle has not already been downloaded and cached, the workspace launcher will first download the bundle and then set up the Tomcat server automatically:

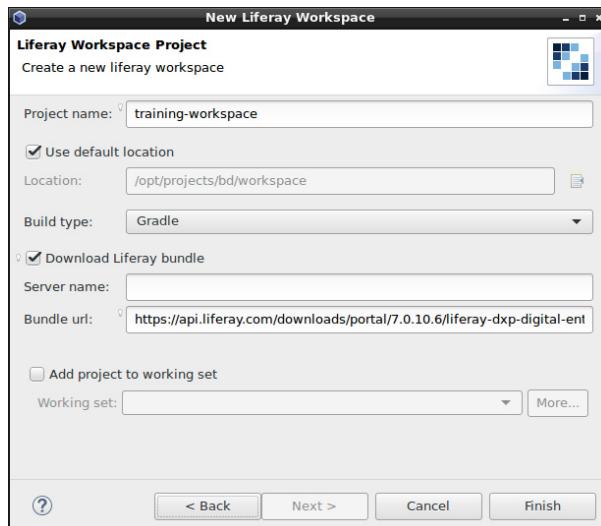


Figure: Installing server bundle automatically

If you didn't choose to download the server when you created the Liferay Workspace, you can use the *initBundle* Gradle task at any time to download and extract the bundle to the bundles subdirectory of the Liferay Workspace. Afterward, you can create a new server just as if you were creating a workspace server manually. You must only point the *Liferay Portal Bundle Directory* to the directory where the bundle is downloaded and extracted.

The download address of the bundle can be defined in the *gradle.properties* file or in the Liferay Workspace create dialog. For more information, see: https://dev.liferay.com/develop/tutorials/-/knowledge_base/7-1/configuring-a-liferay-workspace.

Using Liferay Workspace server bundles allows you to generate and configure multiple server environments in your development environment simultaneously. For more information about this scenario, please see:

https://dev.liferay.com/develop/tutorials/-/knowledge_base/7-1/development-lifecycle-for-a-liferay-workspace#testing-projects.

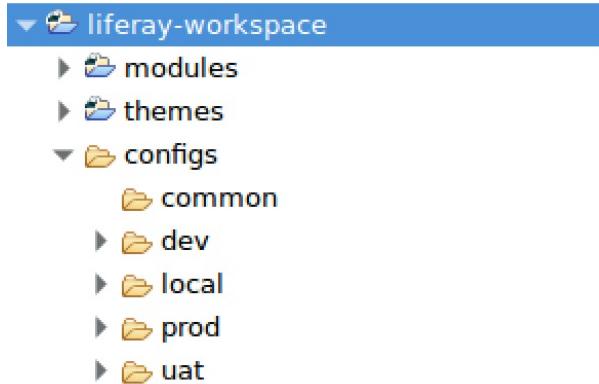


Figure: Liferay Workspace environment configurations folders

Portal Configuration Files

The main portal configuration file, *portal.properties*, defines every aspect of the platform configuration. Amongst others, it defines default settings for:

- Liferay home folder
- Database connection of Database pool reference
- Clustering settings

The default settings can be inspected in the source file, which can be extracted from the portal-impl.jar package in your Liferay installation or downloaded from GitHub at <https://github.com/liferay/liferay-portal/blob/7.1.x/portal-impl/src/portal.properties>.

The platform configuration can be customized by overriding the *portal.properties* settings with one or both of the following files:

- *portal-ext.properties*
- *portal-setup-wizard.properties* (generated only if the setup wizard is run)

Configuration files are processed at startup in the following locations:

- `LIFERAY_HOME_FOLDER`
- `LIFERAY_WEB_APPLICATION_ROOT/WEB-INF/classes`



Figure: Configuration files location

Some of the settings found in *portal.properties* can be managed through the portal instance's Control Panel, too. In case of overlapping settings, the order of precedence is the following (last one remains):

1. *portal.properties*
2. *portal-ext.properties*
3. *portal-setup-wizard.properties*
4. settings persisted through the Control Panel

Good to Know

Many of the settings in *portal.properties* are in the process of becoming Control-Panel managed. When updating or patching the portal, it's always a good practice to check the release notes for possible changes.

Links and Resources

- **Blade CLI Tutorials**
https://dev.liferay.com/develop/tutorials/-/knowledge_base/7-1/blade-cli
- **Blade Samples**
<https://github.com/liferay/liferay-blade-samples>
- **Configuring Liferay Workspace**
https://dev.liferay.com/develop/tutorials/-/knowledge_base/7-1/configuring-a-liferay-workspace
- **Setting up Multiple Environments on Liferay Workspace**
https://dev.liferay.com/develop/tutorials/-/knowledge_base/7-1/development-lifecycle-for-a-liferay-workspace#testing-projects
- **Using Liferay Workspace with Maven** https://dev.liferay.com/develop/tutorials/-/knowledge_base/7-1/maven-workspace

Exercises

Set Up Liferay Workspace and Liferay Portal

Introduction

In this exercise, we will create a Liferay Workspace environment for the Liferay modules we are going to create during the training. We will import the provided Liferay Workspace, which contains code samples and solutions for the exercises. Liferay Portal will be set up using the initBundle Gradle task of Liferay Workspace and the Tomcat adapter provided by Liferay Developer Studio.

Overview

- ① Create a new Developer Studio (Eclipse) workspace
- ② Import the provided Liferay Workspace
- ③ Run the initBundle Gradle task on the Liferay workspace
- ④ Create a new Liferay server using the Tomcat server adapter
- ⑤ Start the server and run the portal setup wizard
- ⑥ Activate the portal by copying the license file to the deploy folder
- ⑦ Login to your portal instance

Prerequisites

For this exercise, you should have been provided:

- A Liferay license file
- A compressed Liferay Workspace file

Please consult your trainer if any of the prerequisites are not met.

Before starting the exercise, extract the compressed Liferay Workspace archive to your preferred location (e.g. `C:\liferay\training-workspace` or `~/liferay/training-workspace`). Liferay Workspace doesn't have to be located inside the Developer Studio (Eclipse) workspace, but can be placed anywhere. When extracted there will be two folders. The first is the `downloads` folder, which contains a cached copy of the portal bundle so that we don't have to download it. The second is the actual Liferay Workspace itself. The Training Workspace will also contain solutions to the exercises if needed.

Create a New Developer Studio (Eclipse) Workspace

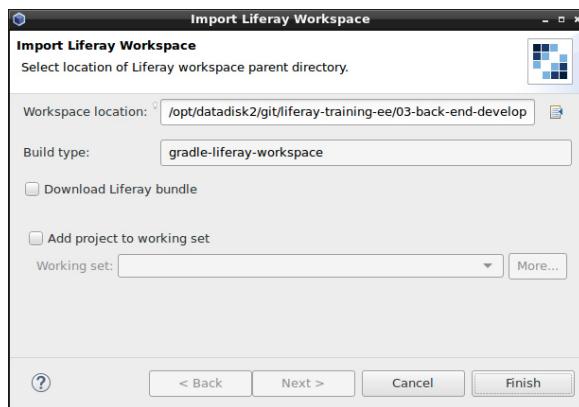
1. **Find** your `training-workspace` folder.
 - Make sure it has been extracted/unzipped.
2. **Click** `File → Launch Workspace → New Workspace` on the Developer Studio menu bar.
3. **Type** the location for a new Eclipse workspace to be created.
4. **Click** `Launch`.

Import the Provided Liferay Workspace

1. Click **Import Liferay Workspace** link on the Welcome Screen:



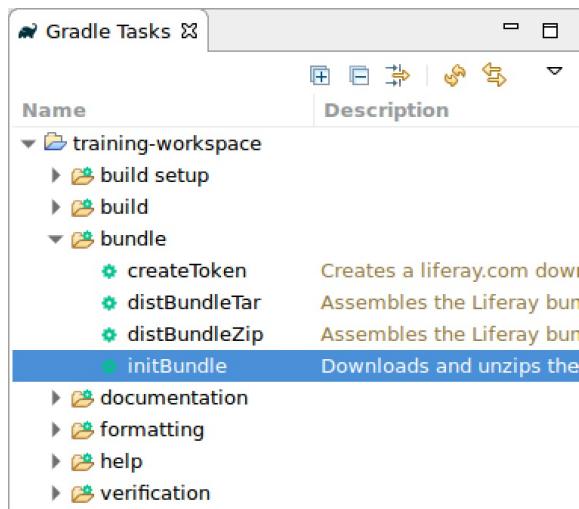
2. Click on the browse button to navigate to the location of the training workspace.
 - This is the folder where you extracted the `training-workspace` that the trainer provided.
3. Click **Finish** to close the wizard:



Run the initBundle Gradle Task

Next, we will prepare the portal bundle for Liferay Workspace. We'll run the `initBundle` Gradle task, which downloads or uses a local copy of portal Tomcat bundle defined in `gradle.properties`.

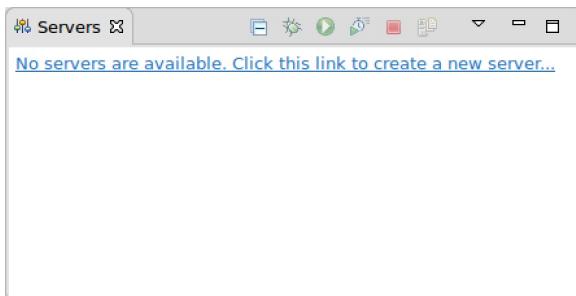
1. Double-click to run the `initBundle` task located in the `training-workspace/bundle` folder in the *Gradle Tasks* panel on the right hand side of Developer Studio.



2. Press **F5** to refresh the Project Explorer panel and see that the portal bundle has been extracted to the `bundles` directory, after you see a *BUILD SUCCESSFUL* message on the console.

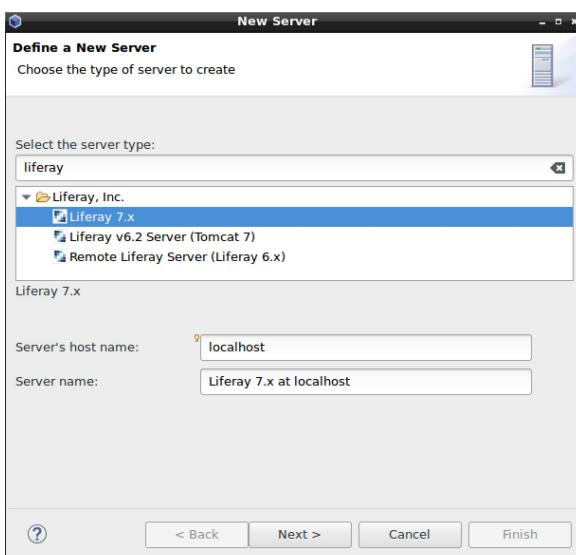
Create a New Liferay Server Using the Tomcat Server Adapter

1. Click the link saying that there's no server available on the Servers panel.



2. Choose Liferay 7.x as the server type.

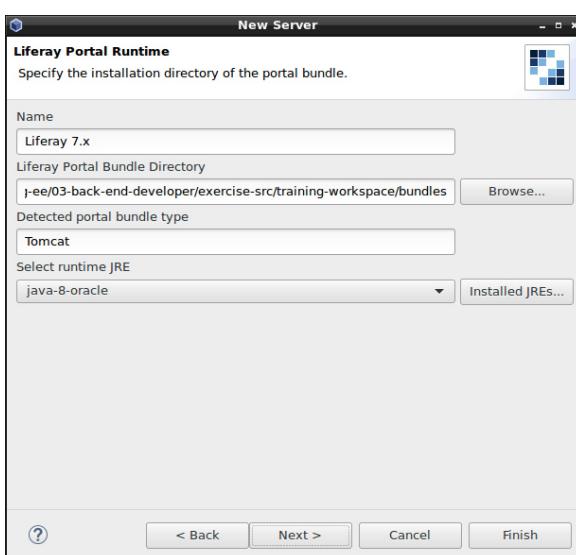
3. Click Next:



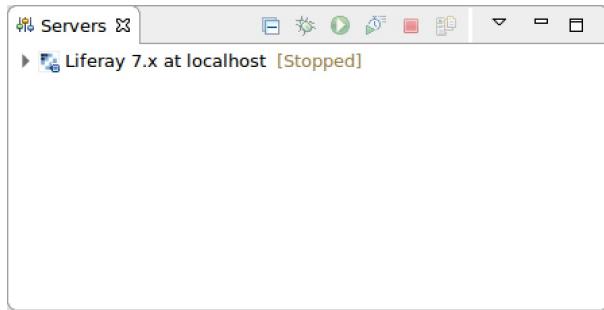
4. Click Browse and locate the bundles directory of our Liferay Workspace (`training-workspace/bundles`).

- o The wizard should detect the portal bundle type automatically.
- o Make sure the *JDK* is selected under the runtime JRE field.

5. Click Finish:



The new server should appear in the *Servers* panel:



Start the Server and Run the Portal Setup Wizard

1. **Click** on the *Liferay 7.x* server at the bottom left.
2. **Click** the green *Start the Server* icon on the *Servers* panel to start our newly-created Liferay server.
 - o Watch the console log for the server startup message:

```
16-May-2018 15:07:20.054 INFO [main]
org.apache.catalina.startup.Catalina.start Server startup in 164332 ms
```

- o A web browser should start and go to address <http://localhost:8080> automatically, opening the setup wizard when the portal is started up. If it didn't, open your browser and enter <http://localhost:8080> manually. Once on the setup page:
3. **Type** *Training Portal* for the Portal Name.
 4. **Uncheck** the Add Sample Data box.
 5. **Click** *Change* in the Database section.
 6. **Choose** MySQL in *Database Type*.
 7. **Change** the database name in the *JDBC URL* (replace *lportal* with *backend_developer* if you followed the setup instructions when setting up MySQL).
 8. **Enter** the *User Name* for the *backend_developer* database (*liferay* if you followed the setup instructions when setting up MySQL).
 9. **Type** the password for the user *liferay* under *Password* (*liferay* if you followed the setup instructions when setting up MySQL).
 10. **Click** *Finish Configuration*.
 11. **Click** the "Play" button in the Developer Studio *Servers* panel again to restart.

Liferay

Portal

Portal Name
Training Portal
For example, Liferay.

Default Language English (United States) Change

Add Sample Data

Administrator User

First Name Test

Last Name Test

Email * test@liferay.com

Database

« Use Default Database

Database Type MySQL

JDBC URL * jdbc:mysql://localhost/lportal?characterEncoding=UTF-8&dontTrackOpenResources=true&holdResultsOpenOverStat

JDBC Driver Class Name * com.mysql.jdbc.Driver

User Name liferay

Password *****

Finish Configuration

Powered By Liferay

After the portal has been restarted, you will be redirected to an error page telling you about the invalid license.

Activate the Portal by Copying the License File to the Deploy Folder

When Liferay starts for the first time, it creates a subfolder called `deploy` under the `LIFERAY_HOME` directory. This is the folder where applications and the license are deployed.

1. **Copy** the license file provided for the training.
2. **Paste** the file into the `deploy` folder and watch the console log in Developer Studio.

After the license has been deployed, refresh the portal from the page in your browser.

Liferay DXP

Search... Sign In

Welcome

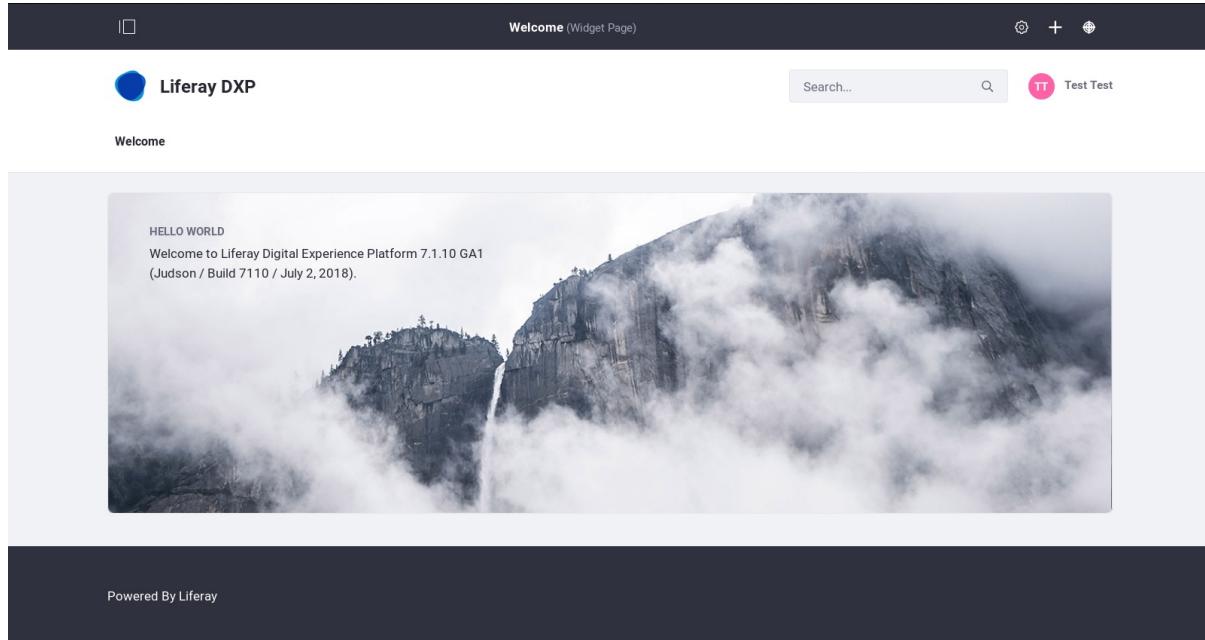
HELLO WORLD
Welcome to Liferay Digital Experience Platform 7.1.10 GA1
(Judson / Build 7110 / July 2, 2018).

Powered By Liferay

Login

1. **Click** the *Sign in* link in the top right corner of the page.
2. **Sign in** using the login info below.
 - o **Email Address:** test@liferay.com
 - o **Password:** test
3. **Click** *I Agree* on the Terms of Use page.
4. **Type** a new password if prompted.
5. **Choose** a password reminder query.
6. **Click** Save.

You are now logged in to Liferay!



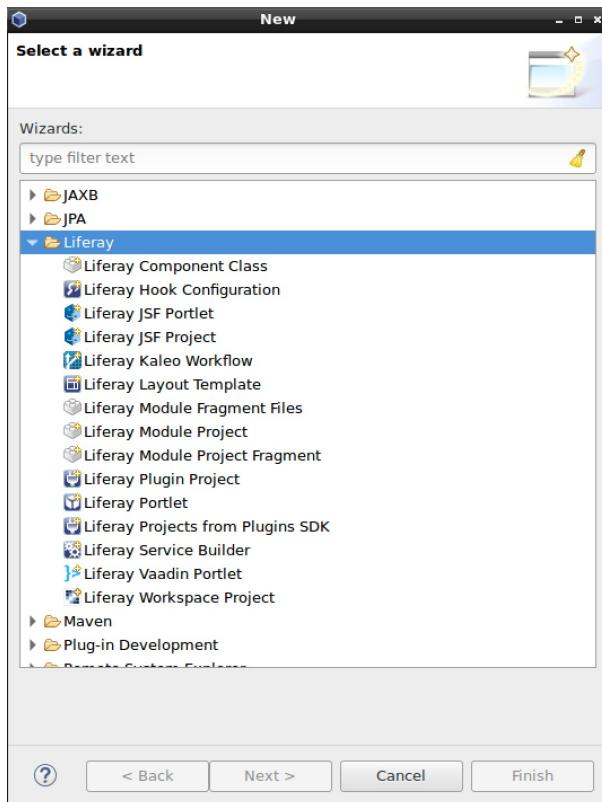
Introducing Liferay Modules

Since Liferay 7.0 / DXP, all Liferay plugins are called *modules*. Basically, a module is equivalent to an OSGi bundle. The only difference lies in the project lifecycle state. A module is a project type, but after being built and deployed, it becomes an OSGi bundle running in the OSGi container. In the Liferay documentation resources, the term *module* is used to refer to both.

It should be noted that although the Liferay platform still offers the option to deploy legacy style WAR plugins, they are automatically converted to OSGi bundles on deploy time by the WAB plugin.

Liferay provides a comprehensive set of project, module, and component templates you can use to generate project stubs and classes.

Liferay Project Templates



Liferay project templates

For more information on the different templates, please see: https://dev.liferay.com/develop/reference/-/knowledge_base/7-1/project-templates

Module Templates

When you start a new Liferay module project, you can choose from a selection of templates. Generally speaking, if you are interested only in a blank module project, the API template is a good choice, since only a minimal structure is generated.

```
activator
api
content-targeting-report
content-targeting-rule
content-targeting-tracking-action
control-menu-entry
form-field
freemarker-portlet
layout-template
mvc-portlet
panel-app
portlet
portlet-configuration-icon
portlet-provider
portlet-toolbar-contributor
rest
service
service-builder
service-wrapper
simulation-panel-entry
soy-portlet
spring-mvc-portlet
template-context-contributor
theme
theme-contributor
```

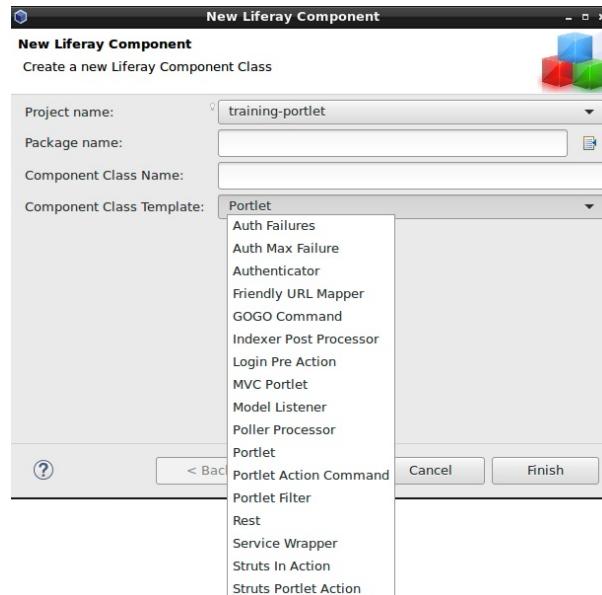
Liferay module templates

A module template creates the basic project skeleton with default settings for the deployable components. The project skeleton comprises:

- The folder structure
- bnd.bnd
- build.gradle
- The component class stub
- The default component properties for the template component(s)

Component Templates

The Liferay component wizard provides a selection of common component templates. You can use this wizard to add components to your existing project:



Component templates

Creating Modules Using the Blade CLI

All module templates are also available from the command line and can be created with the Blade CLI tool:

List available templates from Blade CLI:

```
blade create -l
```

Create a MVC portlet from Blade CLI:

```
blade create -t mvc-portlet training-portlet
```

Listing Project Templates with Maven

If you are using Maven, you can list the project template archetypes with:

```
mvn archetype:generate -Dfilter=liferay
```

Liferay project templates are prefixed with **com.liferay.project.templates**

Links and Resources

- **Liferay Project Templates** https://dev.liferay.com/develop/reference/-/knowledge_base/7-1/project-templates

Exercises

Create a Custom Form Field Using the Form Field Module Template

Introduction

In this exercise, you'll create a new form field for the forms portlet using the *form field* Liferay module template. The template creates the folder structure and component stubs and demonstrates the benefits of using Liferay module templates.

Overview

- ① Create a new Liferay module project using the form field template.
- ② Deploy and check that the field appears in the fields selection of the Forms portlet.

Create a New Liferay Module Project Using the Form Field Template

1. **Click** *New* → *Liferay Module Project* in Developer Studio.
2. **Type** *training-form-field* in the *Project Name* field.
3. **Choose** 7.1 for the Liferay Version.
4. **Choose** *form-field* in the Project Template Name field.
5. **Click** *Next*.
6. **Type** *TrainingFormField* for the *Component Class Name* field.
7. **Type** *com.liferay.training.form.field* for the *Package Name* field.
8. **Click** *Finish* to close the wizard.
9. **Open** the *training-form-field.soy* file located in the *src/main/resources/META-INF/resources* folder in the *training-form-field* project.
10. **Delete** the following lines near the top of the file:

```
/***
 * Defines the delegated template for the form field.
 */
{deltemplate ddm.field variant="'training-form-field'"}
    {call .render data="all" /}
{/deltemplate}
```

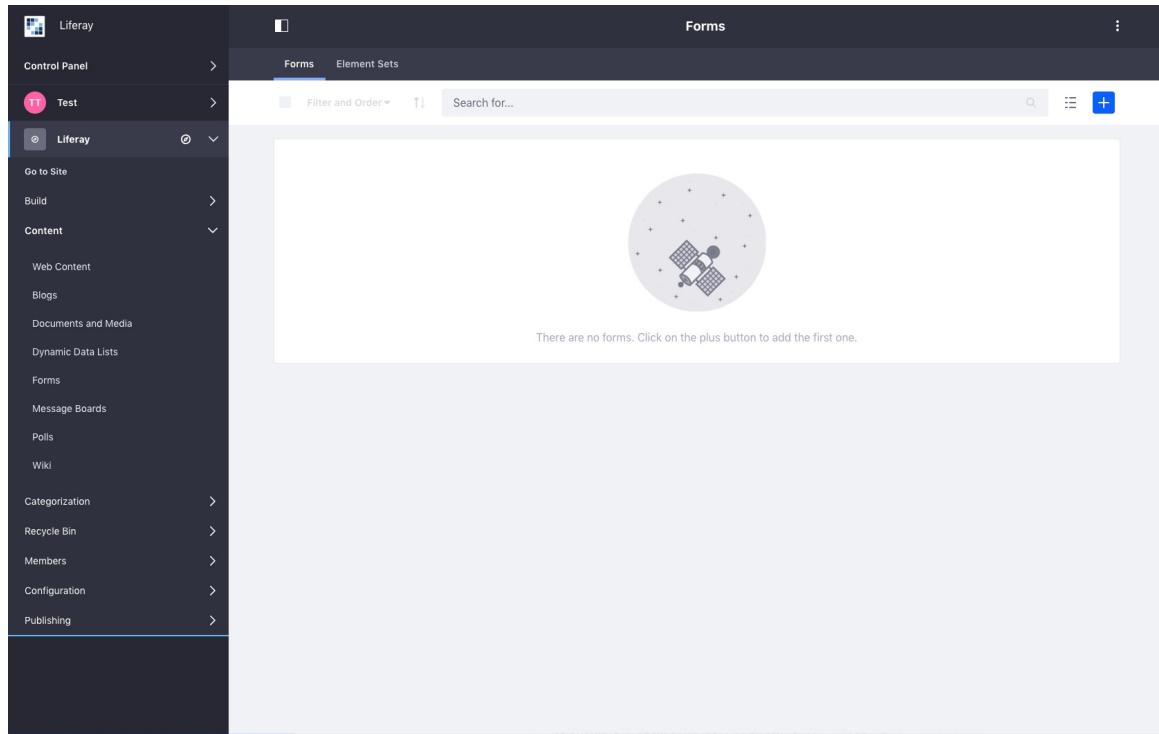
11. **Save** the file. (The previous step is only required because of a yet unresolved issue with the form-field module template and can be omitted as soon as this issue has been fixed.)

Deploy and Check That the Field Appears in the Fields Selection of the Forms Portlet

1. **Drag** the *training-form-field* project from the *modules* folder onto the Liferay server in the *Servers* panel to deploy the module.
 - Watch the console. The module is successfully deployed when you see a message like:

```
2018-05-31 22:34:35.075 INFO [Thread-48][BundleStartStopLogger:35]
STARTED com.liferay.training.form.field_1.0.0 [774]
```

- Whenever you edit your project and save your modifications, your module will automatically be redeployed to the server and restarted by Liferay's OSGI container.
- Open** your browser and login to the platform.
 - Click** the *Menu* icon on the top left corner to open the Product Menu.
 - Click** *Site Administration* → *Content* → *Forms*.



- Click** the *Add* button on the right top corner to add a new form.

You should see our custom form field (`training-form-field Label`) available under the *Customized Elements* panel:

CUSTOMIZED ELEMENTS

| | |
|---|--|
|  | Numeric It accepts only numbers. |
|  | training-form-field Label |
|  | Upload Send files via upload. |

Manage Module Dependencies

Introduction

Liferay applications are modular. Typically, they consist of multiple dependent modules. In addition to inter-module dependencies within the project workspace, often external dependencies have to be managed in your Liferay development projects as well.

Gradle and Maven are supported as dependency management tools. Liferay doesn't restrict you from using any other tool, however.

Since Gradle is the preferred build and dependency management tool which is also used in the core Liferay development, this training material and its examples will focus on the use of Gradle. For information about Maven usage, please see the Liferay Developer Network resources.

Class Visibility in the OSGi container

Every OSGi bundle in the container has its own class loader. Class loader delegation is managed by the OSGi container. The OSGi framework requires bundles to explicitly define what they expose from themselves to the container. This means that by default, none of the bundle's own classes nor any embedded libraries are exposed to other bundles in the container.

Although classes from Liferay web application's WEB-INF/classes or lib folders **are not available** to the OSGi container, classes from the application server's global class loaders **are available**. Liferay provides an additional, configurable control mechanism to manage availability, however. This mechanism will be discussed later in this section.

OSGi Dependencies vs Gradle Dependencies

OSGi and Gradle dependencies are not the same and must not be confused. While Gradle fetches and makes dependencies available, Bndtools wires them together and writes the headers to the bundle manifest. The same applies to versions. Gradle lets you define the versions of the dependencies to be fetched, but what gets wired to the bundle depends only on the bundle's headers in MANIFEST.MF.

In Liferay projects, Gradle dependencies are defined in the *build.gradle* file. Since Bndtools is taking care of creating the final bundle manifest, the OSGi dependencies are defined in the projects *bnd.bnd* file.

About OSGi Compliance

An OSGi-compliant library bundle can be deployed to the OSGi container and made available to other bundles. But what do you do if your project depends on a library that is not OSGi-compliant?

Eclipse Orbit and ServiceMix Bundles are projects that try to solve this problem by providing OSGi variants for many common libraries. If these projects don't provide an OSGi version of the library required by your project, you have to consider other options. First, if you have the source code for the respective library available, you can try to make it OSGi-compliant yourself. Second, you can include and embed the required libraries in your bundle directly. This option will be discussed later in this section.

It should also be noted that even if there was an OSGi-compliant version of your dependent library available, it cannot always be deployed to the container, since in some cases, it would require deploying its transitive dependencies, too, which might not always be desired and might cause complications with other bundles.

Common Dependency Management Tasks

Below is a list of common tasks you might encounter when dealing with module dependencies:

1. Reference another module project in the project environment
2. Find a library bundle containing a missing Liferay class (for example JournalArticle)
3. Determine the right dependency scope for a library (compile, provided...)
4. Define the version range for the dependent library
5. Embed non-OSGi-compliant libraries into your module
6. Use a class from the global classloader (f.ex. TOMCAT/lib/ext)

1 - Referencing Another Module Project With Gradle

Referencing another Liferay module project that is going to be deployed as part of the same application is preferably done in the following way:

build.gradle

```
...
compileOnly project(":modules:training-module:training-module-api")
...
```

2 - Finding a Library Bundle Containing a Liferay Class

When you work on a Liferay customization project, you often depend on certain Liferay core or module-classes. But how do you find the right dependency declaration for the required class? Liferay provides users access to its bundle repository. Below is an example of locating the right Liferay library bundle for the BlogsEntry class:

Step 1 - Find the Bundle for BlogsEntry

Go to <https://repository.liferay.com/nexus/index.html> and find the bundle by the missing class name. Try to use the lowest possible compatible version:

The screenshot shows the Nexus Repository Manager interface. On the left, there's a sidebar with 'Sonatype™' branding, 'Artifact Search' (selected), 'Advanced Search', 'Views.Repositories', 'Repositories', and 'Help'. The main area has a 'Welcome' banner and a 'Search' bar with 'BlogsEntry' typed in. Below the search bar is a table with columns: Group, Artifact, Version, Age, Popularity, and Download. The table lists various Liferay artifacts, including com.liferay.blogs.api versions 3.2.0-SNAPSHOT, 3.1.0-SNAPSHOT, 2.0.0, 1.0.1, and 1.0.0. To the right of the table is a 'Log In' link and a note about the license. At the bottom of the search results, it says 'Displaying Top 159 records' and 'Clear Results'. Below the search results is a tree view under 'Liferay Public Snapshots' for the 'com.liferay' group, showing 'com.liferay.blogs.api' with versions 3.0.1-SNAPSHOT, 3.1.0-SNAPSHOT, and 3.2.0-SNAPSHOT. The 3.2.0-SNAPSHOT node is expanded, showing sub-artifacts: com.liferay.blogs.api-3.2.0-SNAPSHOT-javadoc.jar, com.liferay.blogs.api-3.2.0-SNAPSHOT-sources-commercial.jar, com.liferay.blogs.api-3.2.0-SNAPSHOT-sources.jar, and com.liferay.blogs.api-3.2.0-SNAPSHOT.jar. On the right side, there's a 'Maven' tab selected, showing the Maven dependency XML for the com.liferay.blogs.api artifact at version 3.2.0-SNAPSHOT.

Using repository.liferay.com

Step 2 - Find the Bundle in Maven Central

Copy the Maven dependency. If you are using Gradle, write the dependency information manually or search Maven Central for the respective bundle:

The screenshot shows the Maven Central search results for the artifact `com.liferay.com.liferay.blogs.api` version `3.1.1`. The page includes a chart showing the number of indexed artifacts from 2004 to 2018, a sidebar with popular categories, and a detailed view of the artifact's metadata including license (LGPL 2.1), date (Apr 06, 2018), files (pom, jar), repositories (Central), and dependencies (7 artifacts). Below the details is a code block for Maven dependencies.

```
// https://mvnrepository.com/artifact/com.liferay/com.liferay.blogs.api
provided group: 'com.liferay', name: 'com.liferay.blogs.api', version: '3.1.1'
```

Using Maven Central

Step 3 - Add Dependency to build.gradle

Next, paste the dependency to `build.gradle` or correspondingly, to `pom.xml` (if you are using Maven).

```
dependencies {
    compileOnly group: "com.liferay.portal", name: "com.liferay.portal.kernel", version: "2.0.0"
    compileOnly group: "com.liferay.portal", name: "com.liferay.util.taglib", version: "2.0.0"
    compileOnly group: "javax.portlet", name: "portlet-api", version: "2.0"
    compileOnly group: "javax.servlet", name: "javax.servlet-api", version: "3.0.1"
    compileOnly group: "jstl", name: "jstl", version: "1.2"
    compileOnly group: "org.osgi", name: "osgi.cmpn", version: "6.0.0"

    // https://mvnrepository.com/artifact/com.liferay/com.liferay.blogs.api
    provided group: 'com.liferay', name: 'com.liferay.blogs.api', version: '3.1.1'
}
```

Step 4 - Check the Dependency Scope

Configure the right dependency scope. Usually, `compileOnly` scope should be used:

```
dependencies {
    compileOnly group: "com.liferay.portal", name: "com.liferay.portal.kernel", version: "2.0.0"
    compileOnly group: "com.liferay.portal", name: "com.liferay.util.taglib", version: "2.0.0"
    compileOnly group: "javax.portlet", name: "portlet-api", version: "2.0"
    compileOnly group: "javax.servlet", name: "javax.servlet-api", version: "3.0.1"
    compileOnly group: "jstl", name: "jstl", version: "1.2"
    compileOnly group: "org.osgi", name: "osgi.cmpn", version: "6.0.0"

    // https://mvnrepository.com/artifact/com.liferay/com.liferay.blogs.api
    compileOnly group: 'com.liferay', name: 'com.liferay.blogs.api', version: '3.1.1'
}
```

3 - Determining Gradle Dependency Scope

If a dependent library is not available in the OSGi container at runtime, you usually should try to use the `compileOnly` scope, which includes the classes only at compile time. Since the `compileOnly` scope is not transitive, dependencies of the dependent library are not included automatically. This means that you have to take care of making the internal dependencies of your dependent library available by yourself by also declaring them as dependencies in the `build.gradle` file. That way you can, even on a class level, retain complete control over what's being included in the compiled bundle.

Below is a list of Gradle dependency scopes commonly used in Liferay plugin development. In addition to the standard Gradle scopes, Liferay provides some custom scopes. For more information, please visit the Liferay Developer Network website.

- **compile**: classes are included both at compile and at build time, transitive
- **compileOnly**: classes are included only at compile time, non-transitive
- **compileInclude**: like compileOnly but takes care of including the resources too
- **provided**: classes only needed at runtime and provided by the container
- **runtime**: classes needed at runtime
- **testCompile**: like compileOnly but for tests

4 - Defining Version Range for the Dependency

Imported packages in the bnd.bnd file are always declared using version ranges:

```
Import-Package = com.liferay.training.module;version="[1.3,2)"
```

Although Gradle dependencies declared in the build.gradle file only make the libraries available for the build, the actual OSGi dependencies are declared in the bnd.bnd file. The effective version range of the compiled module is a combination of both the gradle dependencies and the Import-Package configuration in the bnd.bnd file. Thus, the version made available by build.gradle becomes effectively the lowest accepted version. Take a look at a few examples illustrating this behavior:

In the first example, the compiled module accepts any version between 1.3 and 2, including the 2, runtime:

bnd.bnd

```
Import-Package = com.liferay.training.module;version="[1.3,2)"
```

build.gradle

```
compileOnly group: "com.liferay", name: "com.liferay.training.module", version: "1.3.0"
```

In the second example, the resulting OSGi bundle only accepts versions from 1.3.2 to 2:

bnd.bnd

```
Import-Package = com.liferay.module;version="[1.3.0,2)"
```

build.gradle

```
compileOnly group: "com.liferay", name: "com.liferay.module", version:"1.3.2"
```

Using, for example, version 1.3.1 would result in:

```
Unresolved requirement: Import-Package: com.liferay.module; version="[1.3.2,4.0.0)"
```

5 - Embed non-OSGi-compliant libraries into your module

The Liferay-provided custom Gradle scope *compileInclude* provides a convenient way to embed a dependent artifact into the lib folder of a module's JAR. It also adds the required Bundle-ClassPath header to the bundle manifest automatically. Transitive optional dependencies are not included and have to be defined manually:

```
dependencies {
    compileInclude group: 'org.apache.shiro', name: 'shiro-core', version: '1.1.0'
}
```

For more information on this approach, please refer to: https://dev.liferay.com/develop/tutorials/-/knowledge_base/7-1/adding-third-party-libraries-to-a-module

External libraries can be included in the module also by using standard Gradle scopes. Below is an example of a bundle that includes the Google Guava library:

build.gradle

```
dependencies {
    compileOnly group: 'com.google.guava', name: 'guava', version: '21.0'
}
```

bnd.bnd (Option 1)

```
Include-Resource: @guava-21.0.jar
```

bnd.bnd (Option 2)

```
Bundle-ClassPath:\n  .,\n  lib/guava.jar\n-includerelource:\n  lib/guava.jar=guava-21.0.jar
```

6 - Importing a Library From Global Class Loader

Generally speaking, placing libraries under the global class loader should be avoided. In case this approach is needed, the library is copied, in case of Tomcat, to Tomcat's lib/ext folder and declared in Liferay's portal properties as an extra package:

portal-ext.properties

```
module.framework.system.packages.extra=my.dependent.library.package
```

Wrapping it Up

- A Gradle dependency is not an OSGi dependency.
 - **build.gradle** provides the dependencies
 - **bnd.bnd** wires the dependencies
- Generally speaking, try to use *compileOnly* Gradle scope for not provided dependencies.
- Try to use the lowest compatible version for dependencies.
- bnd.bnd and build gradle version information is merged. In case of build.gradle providing a higher minor number than is accepted in bnd.bnd, the version from build.gradle becomes the minimum in effect.

Practical Tips

- Run *Gradle refresh* whenever you modify the dependency configuration.
- In case of problems, clear the Gradle cache.
 - Located by default in a user's home folder

Links and Resources

- **Liferay Gradle Tutorials**
https://dev.liferay.com/develop/reference/-/knowledge_base/7-1/gradle
- **Liferay Maven Tutorials**
https://dev.liferay.com/develop/tutorials/-/knowledge_base/7-1/maven

Chapter 4: Managing OSGi Bundles

Chapter Objectives

- Learn How to Manage OSGi Bundles from the Gogo Shell and Felix Web Console

Manage OSGi Bundles with Gogo Shell

OSGi framework implementations provide tools for managing the bundles inside the container. Some of the common bundle management tasks are:

- Check which bundles are installed and what their versions are
- Bundle Lifecycle management: start, stop, install, uninstall
- Refresh bundle service bindings with update, refresh
- Find bundles, components, or services in the container
- Checking bundle headers
- Checking component properties
- Check which service implementation your service reference is bound to
- Troubleshoot why a bundle or component didn't activate

Apache Felix project provides Gogo Shell and Felix Web Console tools for bundle management. The Gogo Shell is an OSGi application itself and a command shell interpreter for the OSGi container. The Gogo Shell provides full access to the OSGi container, making all the registered services and components available from within the Shell environment. It's based on the Tiny Shell Language (TSL) and supports features like:

- Completion
- Pipes
- Closures
- Variable setting
- Referencing container services
- Scripting

Notice that since version 7.1 Gogo shell access from the command line is **disabled** by the portal default configuration (portal.properties). You can however, access the Gogo shell also from Control panel UI directly.

Command Namespaces

Gogo Shell commands are divided in namespaces:

- **dependencymanager**: Dependency management commands
- **ds**: Declarative Services specific
- **equinox**: Equinox specific
- **felix**: Felix specific
- **formNavigator**: Liferay form navigator
- **gogo**: Gogo shell utility commands
- **obr**: OSGi bundle repository commands
- **scr**: service component runtime namespace for component management
- **thumbnails**: Liferay thumbnails management
- **verify**: Liferay module upgrade commands

Using namespace prefixes is not mandatory, but you should be aware that some of the commands exist in multiple namespaces, and without using the namespace, there's no guarantee which one will be executed. These commands are, for example, *list* or *refresh*.

Basic Commands

- **help**: show list of all commands
- **help [command]**: show command-specific help

- **disconnect**: disconnect from the shell (*exit* shuts down the container)
- **lb**: list all bundles
- **services**: list all registered services
- **start / stop [bundleid]**: start / stop a bundle
- **update [bundleid]**: reinstall a bundle
- **enable [componentid]**: enable a component
- **b [bundleid]**: display information about a bundle
- **diag [bundleid]**: diagnose a bundle
- **headers [bundleid]**: show Manifest headers of a bundle
- **inspect capability service [bundleid]**: list all services provided by a bundle
- **dm wtf**: show any missing dependencides
- **getProperties**: show system properties

Command Examples

lb

Find bundles by their symbolic name, containing a word *blogs*

```
g! lb -s | grep "blogs"
1548|Active    | 10|com.liferay.adaptive.media.blogs.editor.configuration (1.0.1)
1549|Active    | 10|com.liferay.adaptive.media.blogs.item.selector.web (1.0.1)
1550|Active    | 10|com.liferay.adaptive.media.blogs.web (1.0.1)
1551|Resolved   | 10|com.liferay.adaptive.media.blogs.web.fragment (1.0.1)
1725|Active    | 10|com.liferay.blogs.api (3.1.0)
1726|Active    | 10|com.liferay.blogs.editor.configuration (1.0.10)
1727|Active    | 10|com.liferay.blogs.item.selector.api (1.1.0)
1728|Active    | 10|com.liferay.blogs.item.selector.web (2.0.0)
1729|Active    | 10|com.liferay.blogs.layout.prototype (2.0.13)
1730|Active    | 10|com.liferay.blogs.recent.bloggers.api (1.0.1)
1731|Active    | 10|com.liferay.blogs.recent.bloggers.web (2.0.0)
1732|Active    | 10|com.liferay.blogs.service (1.1.11)
1733|Active    | 10|com.liferay.blogs.web (2.0.3)
1779|Active    | 10|com.liferay.microblogs.api (2.1.3)
1780|Active    | 10|com.liferay.microblogs.service (2.1.15)
1781|Active    | 10|com.liferay.microblogs.web (2.0.25)
true
g!
```

Show locations of bundles containing a word *com.liferay.blogs.web*

```
g! lb -l | grep "com.liferay.blogs.web"
1733|Active    | 10|/com.liferay.blogs.web-2.0.3.jar?lpkgPath=/opt/liferay-dxp-digital-enterprise-7.0-sp
7/osgi/marketplace/Liferay Collaboration.lpkg
true
```

inspect

Show services provided by a bundle

```
g! inspect capability service 1780
com.liferay.microblogs.service_2.1.15 [1780] provides:
-----
service; com.liferay.portal.upgrade.registry.UpgradeStepRegistrar with properties:
```

```

component.name = com.liferay.microblogs.internal.upgrade.MicroblogsServiceUpgrade
component.id = 2752
service.id = 1295
service.bundleid = 1780
service.scope = bundle
Used by:
    com.liferay.portal.upgrade.impl_1.0.1 [1521]
service; com.liferay.portal.kernel.security.permission.ResourcePermissionChecker with properties:
    resource.name = com.liferay.microblogs
component.name = com.liferay.microblogs.service.permission.MicroblogsPermission
component.id = 2754
service.id = 1296
service.bundleid = 1780
service.scope = bundle
...

```

scr:info

Show component info

```

g! scr:info com.liferay.portal.target.platform.indexer.IndexerFactory

*** Bundle: com.liferay.portal.target.platform.indexer (2120)
Component Description:
    Name: com.liferay.portal.target.platform.indexer.IndexerFactory
    Implementation Class: com.liferay.portal.target.platform.indexer.IndexerFactory
    Default State: enabled
    Activation: immediate
    Configuration Policy: optional
    Activate Method: activate
    Deactivate Method: deactivate
    Modified Method: -
    Configuration Pid: [com.liferay.portal.target.platform.indexer.IndexerFactory]
Services:
    com.liferay.portal.target.platform.indexer.IndexerFactory
    Service Scope: singleton
    Component Description Properties:
    Component Configuration:
        ComponentId: 28
        State: active
        Component Configuration Properties:
            component.id = 28
            component.name = com.liferay.portal.target.platform.indexer.IndexerFactory

```

Scripting Support

All component and service objects in the OSGi runtime are available and accessible in the Gogo Shell. Below is an example script of how to list all portal users from within the Shell:

```

portalUtilReference=(serviceReference "com.liferay.portal.kernel.util.PortalUtil")

portalUtil=(service $portalUtilReference)

defaultCompanyId=($portalUtil getDefaultCompanyId)

userServiceReference=(serviceReference "com.liferay.portal.kernel.service.UserService")

userService=(service $userServiceReference)

users=$userService getCompanyUsers $defaultCompanyId 0 100

each $users { $it getFullName }

```

Using Gogo Shell With Blade CLI

You can also execute Gogo commands from Blade CLI. This allows you, for example, to pipe the Gogo command output to your Bash shell. The syntax for the commands is:

```
blade sh [gogo_shell_command]
```

The following Bash script example finds all bundles that have *blogs* in their name and finds references to *com.liferay.blogs.kernel.model* in their headers:

```
#!/bin/bash

while read -r bundle
do
    printf "\n$bundle\n"
    for i in $(echo $bundle | awk -F\| '{print $1}')
    do
        blade sh "headers $i"
    done | grep com.liferay.blogs.kernel.model

done <<< "$(blade sh lb -s | grep -i blogs | awk '{print}')"
```

Creating Custom Shell Commands

Gogo commands are regular OSGi service components that define the command name and scope in its properties. To create your own Gogo commands, you just have to create an OSGi command component and deploy it to the OSGi container. Below is an example of a Gogo Shell command component:

```
@Component(
    property = {
        "osgi.command.function=portalStatistics",
        "osgi.command.scope=blade"
    },
    service = Object.class
)
public class PortalStatisticsCommands {

    ...
}
```

Shell Configuration

In production environments, you might need to disable the Gogo Shell completely or to change its host and port. Gogo Shell configuration can be done through portal-ext.properties:

```
module.framework.properties.osgi.console=localhost:11311
```

See the properties here <https://github.com/liferay/liferay-portal/blob/7.1.x/portal-impl/src/portal.properties>

Links and Resources

- [Apache Felix Gogo Project](#)

<http://felix.apache.org/documentation/subprojects/apache-felix-gogo.html>

Exercises

Practice Gogo Shell Basic Commands

Introduction

When troubleshooting Liferay applications, the Gogo Shell is often the first tool that's used. With the Gogo Shell, you can check the lifecycle of a bundle, whether its components have been activated, and if service references are injected properly. In this exercise, you'll walk through some of the most common bundle management commands.

Overview

- ① Telnet to the Liferay OSGi container
- ② Use `lb` to check the bundle state
- ③ Use `dm wtf` to show information about missing dependencies
- ④ Use `headers` to check a bundle's manifest headers
- ⑤ Use `scr:list` and `scr:info` to show information about a component
- ⑥ Use `services` to find information about a service
- ⑦ Use `inspect` to inspect bundle capabilities and requirements

Prerequisites

Telnet client installed or enabled.

Telnet to the Liferay OSGi Container

We use Telnet to connect to Liferay's embedded OSGi container.

1. [Open](#) a Terminal/Command prompt window.
2. [Type](#) in the following to connect:

```
telnet localhost 11311
```

- In Windows, you have to first enable the telnet client to use telnet in the command line or you can use an application with telnet, such as Putty.

```

liferay@liferay-VirtualBox:/$ telnet localhost 11311
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.

Welcome to Apache Felix Gogo
g! 

```

Use **lb** to Check the Bundle State

The **lb** (list bundles) command shows a list of all bundles installed in the container. We can use the **grep** command, familiar for Linux users, to filter the list. The following example lists bundles with their symbolic name and filters only the bundles that have string *blogs* in their name.

1. Type `lb -s | grep blogs` and hit *Enter*.
 - o Alternatively, you can omit `| grep` and simply type `lb -s blogs`

```

liferay@liferay-VirtualBox:/$ telnet localhost 11311
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.

Welcome to Apache Felix Gogo
g! lb -s | grep "blogs"
228|Active   | 10|com.liferay.adaptive.media.blogs.editor.configuration (2.0.0)
229|Active   | 10|com.liferay.adaptive.media.blogs.item.selector.web (2.0.0)
230|Resolved | 10|com.liferay.adaptive.media.blogs.web.fragment (2.0.0)
231|Active   | 10|com.liferay.adaptive.media.blogs.web (2.0.0)
318|Active   | 10|com.liferay.blogs.api (4.0.0)
319|Active   | 10|com.liferay.blogs.editor.configuration (2.0.0)
320|Active   | 10|com.liferay.blogs.item.selector.api (2.0.0)
321|Active   | 10|com.liferay.blogs.item.selector.web (3.0.0)
322|Active   | 10|com.liferay.blogs.layout.prototype (3.0.0)
323|Active   | 10|com.liferay.blogs.reading.time (1.0.0)
324|Active   | 10|com.liferay.blogs.recent.bloggers.api (2.0.0)
325|Active   | 10|com.liferay.blogs.recent.bloggers.web (3.0.0)
326|Active   | 10|com.liferay.blogs.service (2.0.0)
327|Active   | 10|com.liferay.blogs.uad (1.0.0)
328|Active   | 10|com.liferay.blogs.web (3.0.0)
true
g! 

```

Use **dm wtf** to Show Information About Missing Dependencies

dm wtf (dependency manager where is the failure) shows if there are any dependency resolution problems in the bundles.

1. Type `dm wtf` and hit *Enter*.
 - o If there are no problems, the output will show:

```

liferay@liferay-VirtualBox:/$ telnet localhost 11311
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.

Welcome to Apache Felix Gogo

g! lb -s | grep blogs
228|Active   | 10|com.liferay.adaptive.media.blogs.editor.configuration (2.0.0)
229|Active   | 10|com.liferay.adaptive.media.blogs.item.selector.web (2.0.0)
230|Resolved | 10|com.liferay.adaptive.media.blogs.web.fragment (2.0.0)
231|Active   | 10|com.liferay.adaptive.media.blogs.web (2.0.0)
318|Active   | 10|com.liferay.blogs.api (4.0.0)
319|Active   | 10|com.liferay.blogs.editor.configuration (2.0.0)
320|Active   | 10|com.liferay.blogs.item.selector.api (2.0.0)
321|Active   | 10|com.liferay.blogs.item.selector.web (3.0.0)
322|Active   | 10|com.liferay.blogs.layout.prototype (3.0.0)
323|Active   | 10|com.liferay.blogs.reading.time (1.0.0)
324|Active   | 10|com.liferay.blogs.recent.bloggers.api (2.0.0)
325|Active   | 10|com.liferay.blogs.recent.bloggers.web (3.0.0)
326|Active   | 10|com.liferay.blogs.service (2.0.0)
327|Active   | 10|com.liferay.blogs.uad (1.0.0)
328|Active   | 10|com.liferay.blogs.web (3.0.0)
true
g! dm wtf
No missing dependencies found.
g!

```

- o Let's stop the blogs-api bundle and test again
 - o Check the bundle id for the `com.liferay.blogs.api` (Try using the `lb -s | grep blogs` command to find the blogs.api module).
2. Type `stop [BLOGS_API_BUNDLE_ID]` and hit *Enter*.
3. Type `dm wtf` and hit *Enter*.

The output will show:

```

liferay@liferay-VirtualBox:/$ telnet localhost 11311
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.

Welcome to Apache Felix Gogo

g! lb -s | grep blogs
228|Active   | 10|com.liferay.adaptive.media.blogs.editor.configuration (2.0.0)
229|Active   | 10|com.liferay.adaptive.media.blogs.item.selector.web (2.0.0)
230|Resolved | 10|com.liferay.adaptive.media.blogs.web.fragment (2.0.0)
231|Active   | 10|com.liferay.adaptive.media.blogs.web (2.0.0)
318|Active   | 10|com.liferay.blogs.api (4.0.0)
319|Active   | 10|com.liferay.blogs.editor.configuration (2.0.0)
320|Active   | 10|com.liferay.blogs.item.selector.api (2.0.0)
321|Active   | 10|com.liferay.blogs.item.selector.web (3.0.0)
322|Active   | 10|com.liferay.blogs.layout.prototype (3.0.0)
323|Active   | 10|com.liferay.blogs.reading.time (1.0.0)
324|Active   | 10|com.liferay.blogs.recent.bloggers.api (2.0.0)
325|Active   | 10|com.liferay.blogs.recent.bloggers.web (3.0.0)
326|Active   | 10|com.liferay.blogs.service (2.0.0)
327|Active   | 10|com.liferay.blogs.uad (1.0.0)
328|Active   | 10|com.liferay.blogs.web (3.0.0)
true
g! dm wtf
No missing dependencies found.
g! stop 318
g! dm wtf
No missing dependencies found.
Please note that the following bundles are in the RESOLVED state:
* [318] com.liferay.blogs.api
g!

```

Use `headers` to Investigate Bundle Manifest Headers

The `headers` command shows the bundle's manifest headers. With this command, you can, for example, check an Import-Package header and see if there's a problem with a dependent bundle.

1. Choose a bundle to investigate.
 - o For example, try finding the `training-form-field` exercise bundle.
 - o You can run `lb` to see a list of all deployed modules.
2. Type `headers [bundle_id]` and hit *Enter*.

```

File Edit View Search Terminal Help
g! headers 774

solution-form-field (774)
-----
Manifest-Version = 1.0
Bnd-LastModified = 1527806073735
Bundle-ManifestVersion = 2
Bundle-Name = solution-form-field
Bundle-SymbolicName = com.liferay.training.form.field
Bundle-Version = 1.0.0
Created-By = 1.8.0_161 (Oracle Corporation)
Import-Package = com.liferay.dynamic.data.mapping.form.field.type;version="[1.0,2)",com.liferay.portal.kernel.template;version="[7.0,8)"
Java-Debug = on
Java-Derepracation = off
Java-Encoding = UTF-8
Liferay-JS-Config = /META-INF/resources/config.js
Private-Package = com.liferay.training.form.field.form.field.content
Provide-Capability = osgi.service;objectClass>List<String>="com.liferay.dynamic.data.mapping.form.field.type.DDMFormFieldRenderer",osgi.service;objectClass>List<String>="com.liferay.dynamic.data.mapping.form.field.type.DDMFormFieldType",liferay.resource.bundle;bundle.symbolic.name="com.liferay.training.form.field";resource.bundle.base.name="content.Language"
Require-Capability = osgi.extender;filter="(&(osgi.extender= osgi.component)(version>=1.3.0)(>(version>=2.0.0)))",osgi.ee;filter="(&(osgi.ee=JavaSE)(version>=1.8))"
Service-Component = OSGI-INF/com.liferay.training.form.field.form.field.TrainingFormFieldDDMFormFieldRenderer.xml,OSGI-INF/com.liferay.training.form.field.TrainingFormFieldDDMFormFieldRenderer.xml
Tool = Bnd-3.5.0.201709291849
Web-ContextPath = /solution-form-field-form-field
g!

```

Use `scr:list` and `scr:info` to Show Information About a Component

1. Type `scr:list` and hit *Enter* to list all the components in the OSGi container.
2. Choose one component id to investigate.
3. Type `scr:info [COMPONENT_ID]` to show info about the component.

The output should look like:

```

File Edit View Search Terminal Help
g! scr:info 774
*** Bundle: con.liferay.adaptive.media.image.impl (243)
Component Description:
  Name: com.liferay.adaptive.media.internal.configuration.AMImageConfigurationEntryParser
  Implementation Class: com.liferay.adaptive.media.internal.configuration.AMImageConfigurationEntryParser
  Default State: enabled
  Activation: immediate
  Configuration Policy: optional
  Activate Method: activate
  Deactivate Method: deactivate
  Modified Method: -
  Configuration Pid: [com.liferay.adaptive.media.image.internal.configuration.AMImageConfigurationEntryParser]
Services:
  com.liferay.adaptive.media.image.internal.configuration.AMImageConfigurationEntryParser
  Service Scope: singleton
  Reference: _http
    Interface Name: com.liferay.portal.kernel.util.Http
    Cardinality: 1..1
    Policy: static
    Policy option: reluctant
    Reference Scope: bundle
Component Description Properties:
Component Configuration:
  ComponentId: 774
  State: active
  SatisfiedReference: _http
    Target: null
    Bound to: 1886
    Reference Properties:
      bean.id = com.liferay.portal.util.HttpImpl
      objectClass = [com.liferay.portal.kernel.util.Http, com.liferay.portal.util.HttpImpl]
      original.bean = true
      service.bundleid = 0

```

Use `services` to Find Information About a Service

The `services` command lists all the registered (published) services in the OSGi container. You can use `grep` to filter the list, which is usually a long one, but LDAP-style filters can be used to do more exact filtering.

1. Type `services (objectClass=com.liferay.blogs.web*)` and hit *Enter* to list all the services matching the filter:

```

liferay@liferay-VirtualBox: /
File Edit View Search Terminal Help
g! services (objectClass=com.liferay.blogs.web*)
[com.liferay.blogs.web.internal.BlogsItemSelectorHelper]={component.name=com.liferay.blogs.web.internal.BlogsItemSelectorHelper, component.id=1122, service.id=666, service.bundleId=328, service.scope=bundle}
    "Registered by bundle:" com.liferay.blogs.web_3.0.0 [328]
    "Bundles using service"
        com.liferay.blogs.web_3.0.0 [328]
[com.liferay.blogs.web.internal.util.BlogsEntryUtil]={component.name=com.liferay.blogs.web.internal.util.BlogsEntryUtil, component.id=1181, service.id=690, service.bundleId=328, service.scope=bundle}
    "Registered by bundle:" com.liferay.blogs.web_3.0.0 [328]
    "Bundles using service"
        com.liferay.blogs.web_3.0.0 [328]
[com.liferay.blogs.web.internal.upload.ImageBlogsUploadResponseHandler]={component.name=com.liferay.blogs.web.internal.upload.ImageBlogsUploadResponseHandler, component.id=1179, service.id=1443, service.bundleId=328, service.scope=bundle}
    "Registered by bundle:" com.liferay.blogs.web_3.0.0 [328]
    "Bundles using service"
        com.liferay.blogs.web_3.0.0 [328]
[com.liferay.blogs.web.internal.configuration.BlogsPortletInstanceConfiguration]={service.id=3133, service.bundleId=72, service.scope=singleton}
    "Registered by bundle:" com.liferay.portal.configuration.settings_3.0.0 [72]
    "No bundles using service."
[com.liferay.blogs.web.internal.messaging.LinkbackMessageListener]={ModuleServiceLifecycle.target=(module.service.lifecycle=portal.initialized), component.name=com.liferay.blogs.web.internal.messaging.LinkbackMessageListener, component.id=1138, service.id=5720, service.bundleId=328, service.scope=bundle}
    "Registered by bundle:" com.liferay.blogs.web_3.0.0 [328]
    "Bundles using service"
        com.liferay.blogs.web_3.0.0 [328]
[com.liferay.blogs.web.internal.trackback.Trackback]={component.name=com.liferay.blogs.web.internal.trackback.Trackback, component.id=1176, service.id=5723, service.bundleId=328, service.scope=bundle}
    "Registered by bundle:" com.liferay.blogs.web_3.0.0 [328]
    "Bundles using service"
        com.liferay.blogs.web_3.0.0 [328]
[com.liferay.blogs.web.internal.security.permission.resource.BlogsEntryPermission]={component.name=com.liferay.blogs.web.internal.security.permission.resource.BlogsEntryPermission, component.id=1171, EntryModelPermission.target=(model.class.name=com.liferay.blogs.model.BlogsEntry), service.id=5860, service.bundleId=328, service.scope=bundle}
    "Registered by bundle:" com.liferay.blogs.web_3.0.0 [328]

```

Use *inspect* to Inspect Bundle Capabilities and Requirements

The *inspect* command can be used to show information about a service *inspect* also shows information about bundle's capabilities and requirements.

1. **Select** a bundle id to investigate
2. **Type** `inspect cap service [BUNDLE_ID]`.

The output should look like:

```

liferay@liferay-VirtualBox: /
File Edit View Search Terminal Help
g! inspect cap service 774
com.liferay.training.form.field_1.0.0 [774] provides:
-----
service; org.osgi.service.http.context.ServletContextHelper with properties:
    osgi.http.whiteboard.context.name = solution-form-field-form-field
    osgi.http.whiteboard.context.path = /solution-form-field-form-field
    rtl.required = true
    service.id = 9120
    service.bundleId = 774
    service.scope = singleton
Used by:
    com.liferay.frontend.css.rtl.servlet_2.0.0 [443]
    com.liferay.training.form.field_1.0.0 [774]
    com.liferay.portal.language.servlet.filter_3.0.0 [89]
service; javax.servlet.ServletContextListener with properties:
    osgi.http.whiteboard.listener = true
    osgi.http.whiteboard.context.select = (osgi.http.whiteboard.context.name=solution-form-field-form-field)
    service.id = 9122
    service.bundleId = 774
    service.scope = singleton
Used by:
    org.eclipse.equinox.http.servlet_1.2.2.v20180212-2026 [47]
service; javax.servlet.ServletContext with properties:
    osgi.web.contextpath = /o/solution-form-field-form-field
    osgi.web.symbolicname = com.liferay.training.form.field
    osgi.web.version = 1.0.0
    osgi.web.contextname = solution-form-field-form-field
    service.id = 9123
    service.bundleId = 774
    service.scope = singleton
Used by:
    com.liferay.portal.template.velocity_3.0.0 [191]
    com.liferay.portal.template.freemarker_4.0.0 [187]
    com.liferay.frontend.js.bundle.config.extender_2.0.0 [463]

```

Create a Custom Gogo Shell Command

Introduction

Working in a shell environment, creating custom commands or scripts for common complex or multi-step tasks often saves a lot of time. As a developer or an administrator, for example, it would be convenient to get basic statistics of the system with a single shell command. In this task, you'll create a new Gogo Shell command to get the number of users and groups in the system.

Overview

- ① Create a new module project of type API
- ② Create an OSGi command component
- ③ Implement the command logic
- ④ Deploy and test

Create a New Module Project of Type API

1. **Click** *File → New → Liferay Module Project* in Developer Studio.
2. **Type** *training-gogo-command* in the *Project Name* field.
3. **Choose** *api* in Project Template Name field.
4. **Click** *Next*.
5. **Type** *Dummy* to the *Component Class Name* field.
 - We won't need this class.
6. **Type** *com.liferay.training.gogo.command* in the *Package Name* field.
7. **Click** *Finish* to close the wizard.
8. **Expand** the *training-gogo-command* module project folder.
9. **Delete** the *Dummy* class created by the wizard located in the *com.liferay.training.gogo.command.api* package under *src/main/java*.
10. **Right-click** on the *com.liferay.training.gogo.command.api* package.
11. **Choose** *Refactor → Rename*.
12. **Remove** *.api* from the package name. (You may safely ignore the warning, that the package *com.liferay.training.gogo.command* already exists.)
13. **Click** *Ok → Continue*
14. **Click** on the *bnd.bnd* file in the Project Explorer.
15. **Click** on the *Source* tab.
16. **Delete** the *com.liferay.training.gogo.command.api* from the exported packages in *bnd.bnd*.
17. **Save** the file.

Create an OSGi Command Component

1. **Right-click** on the *training-gogo-command* project to open the context menu.
2. **Select** *New → Liferay Component Class*.
3. **Type** *com.liferay.training.gogo.command* in the *Package Name* field.
4. **Type** *TrainingCommand* in the *Component Class Name*.
5. **Choose** *GOGO Command* component class template.
6. **Click** *Finish* to close the wizard.

Implement the Command Logic

The command component template includes the logic for getting the user count already. We'll change the command name and add logic to display the site count to the results.

1. **Edit** the `osgi.command.function` property value to `portalstats` to rename the command.

```
...
@Component(
    immediate = true,
    property = {
        "osgi.command.scope=blade",
        "osgi.command.function=portalstats"
    },
    service = Object.class
)
...
```

2. **Edit** the `usercount()` method's name to match the `osgi.command.function` property's value:

```
...
public void portalstats() {
    System.out.println(
    ...
}
```

3. **Add** a reference to `GroupLocalService` to the end of the class.

```
...
@Reference
GroupLocalService _groupLocalService;
...
```

4. **Type** `CTRL+SHIFT+O` to use Eclipse's *Organize Imports* feature and add the required import.

5. **Add** the groupcount to the output by modifying the `portalstats()` method.

- o The function should look like:

```
public void portalstats() {
    System.out.println(
        "# of users: " + getUserLocalService().getUsersCount());
    System.out.println(
        "# of groups: " + _groupLocalService.getGroupsCount());
}
```

6. **Save** the file.

Deploy and Test

1. **Drag** the `training-gogo-command` project onto the Liferay server on the `Servers` panel to deploy the module.
- o Watch the console. The module is successfully deployed when you see a message like:

```
2018-06-01 02:13:23.421 INFO [Thread-109][BundleStartStopLogger:35]
STARTED com.liferay.training.gogo.command_1.0.0 [775]
```

2. **Run** `telnet localhost 11311` in your Terminal/Command prompt to start the Gogo Shell if it isn't already running.
3. **Type** `portalstats`, and hit *Enter*.

The output should look like:

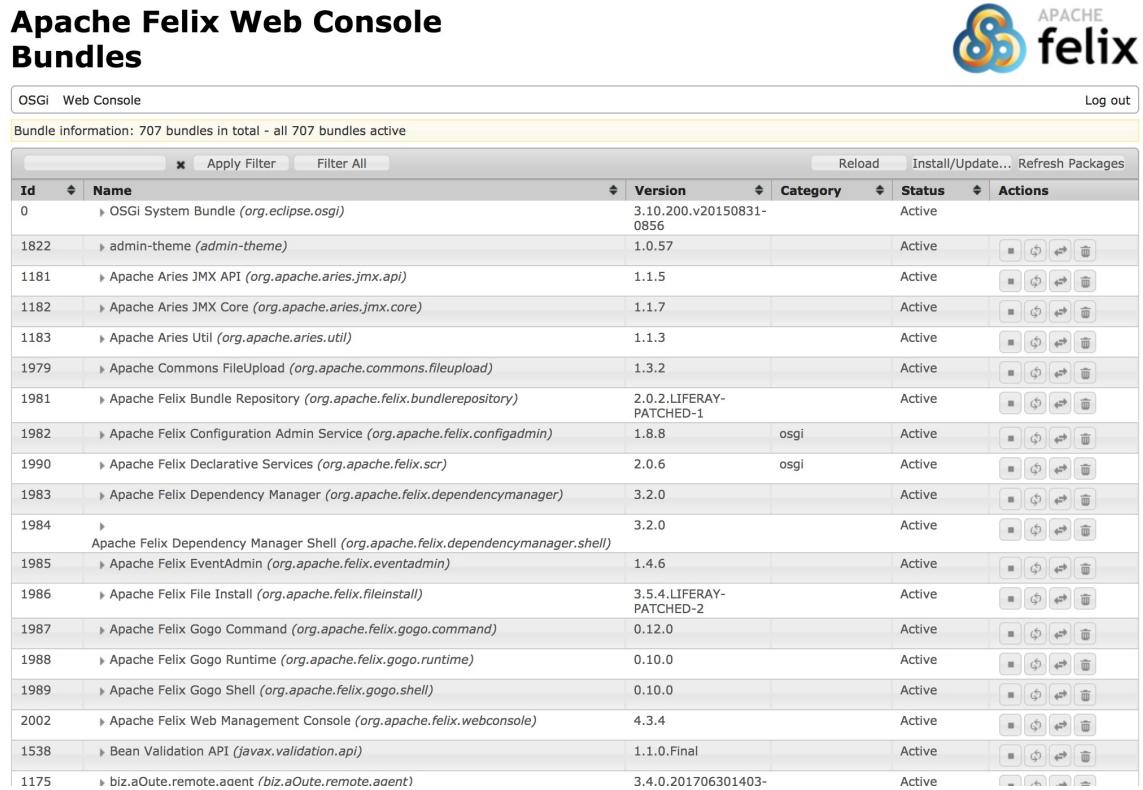


A screenshot of a terminal window titled "liferay@liferay-VirtualBox: /". The window has a menu bar with "File", "Edit", "View", "Search", "Terminal", and "Help". The main pane displays the command "g! blade:portalstats" followed by its output: "# of users: 2" and "# of groups: 11". The window has standard window controls (minimize, maximize, close) at the top right.

```
liferay@liferay-VirtualBox: /  
File Edit View Search Terminal Help  
g! blade:portalstats  
# of users: 2  
# of groups: 11  
g!
```

Using Felix Web Console

Apache Felix Web Console is a web-based OSGi container management tool. It provides a graphical user interface for the most common bundle management tasks, like installing, uninstalling, starting, and stopping the bundles and checking the bundle configuration but lacks advanced features like scripting. Like the Gogo Shell, it's extensible.



The screenshot shows the Apache Felix Web Console interface. At the top, there is a header with the Apache Felix logo and navigation links for 'OSGI' and 'Web Console'. On the right, there is a 'Log out' button. Below the header, a message states 'Bundle information: 707 bundles in total - all 707 bundles active'. A search bar with 'Apply Filter' and 'Filter All' buttons is followed by a 'Reload' button and links for 'Install/Update...' and 'Refresh Packages'. The main area is a table titled 'Bundles' with columns: Id, Name, Version, Category, Status, and Actions. The table lists 707 bundles, including the OSGi System Bundle, various Apache Aries components, and several Apache Felix utilities and management services. Each row in the table includes a set of icons for managing the bundle: a square, a circular arrow, a double-headed arrow, and a trash can.

| Id | Name | Version | Category | Status | Actions |
|------|---|-------------------------|----------|--------|---------|
| 0 | ↳ OSGi System Bundle (<i>org.eclipse.osgi</i>) | 3.10.200.v20150831-0856 | | Active | [icons] |
| 1822 | ↳ admin-theme (<i>admin-theme</i>) | 1.0.57 | | Active | [icons] |
| 1181 | ↳ Apache Aries JMX API (<i>org.apache.aries.jmx.api</i>) | 1.1.5 | | Active | [icons] |
| 1182 | ↳ Apache Aries JMX Core (<i>org.apache.aries.jmx.core</i>) | 1.1.7 | | Active | [icons] |
| 1183 | ↳ Apache Aries Util (<i>org.apache.aries.util</i>) | 1.1.3 | | Active | [icons] |
| 1979 | ↳ Apache Commons FileUpload (<i>org.apache.commons.fileupload</i>) | 1.3.2 | | Active | [icons] |
| 1981 | ↳ Apache Felix Bundle Repository (<i>org.apache.felix.bundlerepository</i>) | 2.0.2.LIFERAY-PATCHED-1 | | Active | [icons] |
| 1982 | ↳ Apache Felix Configuration Admin Service (<i>org.apache.felix.configadmin</i>) | 1.8.8 | osgi | Active | [icons] |
| 1990 | ↳ Apache Felix Declarative Services (<i>org.apache.felix.scr</i>) | 2.0.6 | osgi | Active | [icons] |
| 1983 | ↳ Apache Felix Dependency Manager (<i>org.apache.felix.dependencymanager</i>) | 3.2.0 | | Active | [icons] |
| 1984 | ↳ Apache Felix Dependency Manager Shell (<i>org.apache.felix.dependencymanager.shell</i>) | 3.2.0 | | Active | [icons] |
| 1985 | ↳ Apache Felix EventAdmin (<i>org.apache.felix.eventadmin</i>) | 1.4.6 | | Active | [icons] |
| 1986 | ↳ Apache Felix File Install (<i>org.apache.felix.fileinstall</i>) | 3.5.4.LIFERAY-PATCHED-2 | | Active | [icons] |
| 1987 | ↳ Apache Felix Gogo Command (<i>org.apache.felix.gogo.command</i>) | 0.12.0 | | Active | [icons] |
| 1988 | ↳ Apache Felix Gogo Runtime (<i>org.apache.felix.gogo.runtime</i>) | 0.10.0 | | Active | [icons] |
| 1989 | ↳ Apache Felix Gogo Shell (<i>org.apache.felix.gogo.shell</i>) | 0.10.0 | | Active | [icons] |
| 2002 | ↳ Apache Felix Web Management Console (<i>org.apache.felix.webconsole</i>) | 4.3.4 | | Active | [icons] |
| 1538 | ↳ Bean Validation API (<i>javax.validation.api</i>) | 1.1.0.Final | | Active | [icons] |
| 1175 | ↳ biz.aQute.remote.agent (<i>biz.aQute.remote.agent</i>) | 3.4.0.201706301403- | | Active | [icons] |

Figure: Felix web Console

Links and Resources

- **Apache Felix Web Console Project Page** <http://felix.apache.org/documentation/subprojects/apache-felix-web-console.html>

Exercises

Using the Felix Web Console to Find the Blogs Web Module Version

Introduction

Whether you want a graphical user interface or lack shell access, the Apache Felix Web Console is a good alternative for common bundle management and troubleshooting tasks. In troubleshooting deployments, one of the common tasks is to find out the version of certain modules. In this exercise, you'll use the Felix Web Console to find out the current version of the Blogs Web Module.

Overview

- ① Install Felix Web Console
- ② Find the Blogs web module version

Prerequisites

Telnet client installed or enabled.

Install Felix Web Console

Felix Web Console is an OSGi bundle that we have to install into the OSGi container.

1. **Open** the connection to the Gogo Shell using your telnet client (`telnet localhost 11311`)
2. **Type** `install http://www.apache.org/dist//felix/org.apache.felix.webconsole-4.3.8.jar` and hit *Enter*.
 - Note the bundle id in the result output.
3. **Type** `start [BUNDLE_ID]` to start the Felix Web Console.
4. **Go to** `http://localhost:8080/o/system/console` in your web browser.
5. **Login** with the following credentials:
 - Username: `admin`
 - Password: `admin`

Find the Blogs Web Module Version

1. **Type** `com.liferay.blogs.web` to the filter field and hit *Enter*.
2. **Click** the arrow on the left side of the bundle name to open the information.

Here you can see the version number for the bundle.

Apache Felix Web Console Bundles



OSGi Web Console

Bundle Information: 777 bundles in total, 769 bundles active, 7 active fragments, 1 bundles resolved

| com.liferay.blogs.web | | Apply Filter | Filter All | Reload | Install/Update... | Refresh Packages | |
|-----------------------|---|---|------------|----------------|-------------------|------------------|----------------|
| Id | Name | | | Version | Category | Status | Actions |
| 328 | Liferay Blogs Web (com.liferay.blogs.web) | | | 3.0.0 | | Active | |
| | Symbolic Name | com.liferay.blogs.web | | | | | |
| | Version | 3.0.0 | | | | | |
| | Bundle Location | file:/opt/datalink2/git/liferay-training-ee/03-back-end-developer/exercise-src/training-workspace/bundles/osgi/modules/com.liferay.blogs.web.jar | | | | | |
| | Last Modification | Thu May 31 22:26:13 GMT 2018 | | | | | |
| | Vendor | Liferay, Inc. | | | | | |
| | Start Level | 10 | | | | | |
| | Fragments Attached | com.liferay.adaptive.media.blogs.web.fragment (230) | | | | | |
| | Exported Packages | --- | | | | | |
| | Imported Packages | aQute.bnd.annotation.metatype.version=1.45.0 from org.eclipse.osgi (0) com.liferay.adaptive.media.content.transformer.version=2.0.0 from com.liferay.adaptive.media.content.transformer.api (232) com.liferay.adaptive.media.content.transformer.constants.version=1.0.0 from com.liferay.adaptive.media.content.transformer.api (232) com.liferay.application.list.version=1.0.0 from com.liferay.application.list.api (277) com.liferay.asset.kernel.exception.version=1.0.0 from org.eclipse.osgi (281) com.liferay.asset.kernel.model.version=1.0.0 from org.eclipse.osgi (0) com.liferay.asset.kernel.service.version=1.7.0 from org.eclipse.osgi (0) com.liferay.asset.kernel.service.persistence.version=1.2.0 from org.eclipse.osgi (0) com.liferay.asset.kernel.service.persistence.constants.version=1.0.0 from org.eclipse.osgi (281) com.liferay.blogs.configuration.version=1.0.4 from com.liferay.blogs.api (318) com.liferay.blogs.configuration.definition.version=2.0.0 from com.liferay.blogs.api (318) com.liferay.blogs.constants.version=1.1.0 from com.liferay.blogs.api (318) com.liferay.blogs.exception.version=1.0.0 from com.liferay.blogs.api (318) com.liferay.blogs.item.selector.criterion.version=1.0.0 from com.liferay.blogs.item.selector.api (320) com.liferay.blogs.item.selector.criteria.version=1.0.0 from com.liferay.blogs.item.selector.api (320) com.liferay.blogs.model.version=1.0.0 from com.liferay.blogs.api (318) com.liferay.blogs.service.version=1.0.0 from com.liferay.blogs.api (318) com.liferay.blogs.settings.version=1.0.0 from com.liferay.blogs.api (318) com.liferay.blogs.social.version=1.0.0 from com.liferay.blogs.api (318) com.liferay.document.library.accessor.version=1.0.0 from com.liferay.document.library.accessor.api (318) com.liferay.document.library.display.convert.version=2.3.0 from com.liferay.document.library.api (354) com.liferay.document.library.kernel.exception.version=1.3.0 from org.eclipse.osgi (0) com.liferay.document.library.kernel.model.version=1.0.0 from org.eclipse.osgi (0) com.liferay.document.library.kernel.service.version=1.2.0 from org.eclipse.osgi (0) com.liferay.document.library.kernel.service.persistence.version=1.3.0 from org.eclipse.osgi (0) com.liferay.document.library.kernel.url.version=1.3.0 from org.eclipse.osgi (0) com.liferay.exportimport.kernel.lar.version=1.8.0 from org.eclipse.osgi (0) com.liferay.exportimport.portlet.preferences.processor.version=1.1.0 from com.liferay.exportimport.api (414) com.liferay.friendly.url.exception.version=1.0.0 from com.liferay.friendly.url.api (439) com.liferay.friendly.url.model.version=1.0.0 from com.liferay.friendly.url.api (439) com.liferay.friendly.url.service.version=1.0.0 from com.liferay.friendly.url.api (439) com.liferay.frontend.taglib.clay.service.taglib.util.version=0.0.0 from com.liferay.frontend.taglib.clay (475) com.liferay.item.selector.version=2.0.0 from com.liferay.item.selector.api (499) com.liferay.item.selector.criteria.version=1.0.0 from com.liferay.item.selector.criteria.api (500) com.liferay.item.selector.criteria.image.criterion.version=1.0.0 from com.liferay.item.selector.criteria.api (500) com.liferay.item.selector.criteria.upload.criterion.version=1.2.0 from com.liferay.item.selector.criteria.api (500) | | | | | |

Use Felix Web Console to Locate MVCRender Command Components for the Blogs Portlet

Introduction

MVC render commands are controller components that handle the render requests from the user interface. Sometimes you want to override these components but first need to find out the original component. Both the Gogo Shell and Felix Web Console let you use LDAP-style filters to look for OSGi component properties. In this exercise, you'll use the Apache Felix Web Console tool to locate all the MVC command components listening to the action path starting with /blogs.

Overview

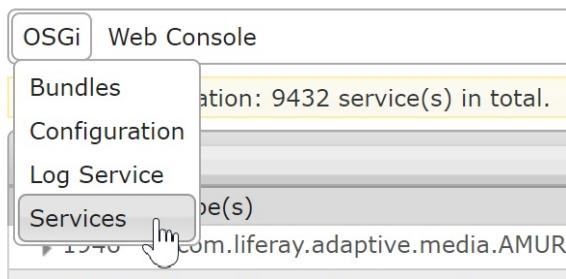
- ① Open Felix Web Console
- ② Find MVC command components listening to the action path starting with /blogs

Open the Felix Web Console

1. Go to <http://localhost:8080/o/system/console> in your web browser if you aren't already there.
 - The username and password are both **admin**.

Find MVC Command Components Listening to the Action Path Starting with /blogs

1. Click on **OSGi → Services** in the main menu.



2. Type `(mvc.command.name=/blogs*)` in the *Filter* field and hit enter:

Apache Felix Web Console Services



| OSGi Web Console | | Log out | |
|---|--|-----------------------------|---------|
| Services information: 13 service(s) in total. | | | |
| Id | Type(s) | Bundle | Ranking |
| 5724 | [com.liferay.portal.kernel.portlet.bridges.mvc.MVCActionCommand] | com.liferay.blogs.web (328) | |
| 5875 | [com.liferay.portal.kernel.portlet.bridges.mvc.MVCActionCommand] | com.liferay.blogs.web (328) | |
| 5878 | [com.liferay.portal.kernel.portlet.bridges.mvc.MVCActionCommand] | com.liferay.blogs.web (328) | |
| 5880 | [com.liferay.portal.kernel.portlet.bridges.mvc.MVCActionCommand] | com.liferay.blogs.web (328) | |
| 5881 | [com.liferay.portal.kernel.portlet.bridges.mvc.MVCActionCommand] | com.liferay.blogs.web (328) | |
| 5882 | [com.liferay.portal.kernel.portlet.bridges.mvc.MVCActionCommand] | com.liferay.blogs.web (328) | |
| 5887 | [com.liferay.portal.kernel.portlet.bridges.mvc.MVCActionCommand] | com.liferay.blogs.web (328) | |
| 683 | [com.liferay.portal.kernel.portlet.bridges.mvc.MVCRenderCommand] | com.liferay.blogs.web (328) | |
| 684 | [com.liferay.portal.kernel.portlet.bridges.mvc.MVCRenderCommand] | com.liferay.blogs.web (328) | |
| 685 | [com.liferay.portal.kernel.portlet.bridges.mvc.MVCRenderCommand] | com.liferay.blogs.web (328) | |
| 686 | [com.liferay.portal.kernel.portlet.bridges.mvc.MVCRenderCommand] | com.liferay.blogs.web (328) | |
| 1982 | [com.liferay.portal.kernel.portlet.bridges.mvc.MVCRenderCommand] | com.liferay.blogs.web (328) | |
| 5244 | [com.liferay.portal.kernel.portlet.bridges.mvc.MVCRenderCommand] | com.liferay.blogs.web (328) | |

Services information: 13 service(s) in total.

Chapter 5: Working with Portlet Modules

Chapter Objectives

- Understand Portlet Concepts and the Portlet Lifecycle
- Learn How to Create Portlet Modules
- Learn How to Set Portlet Properties

Java Standard Portlet

Liferay does not strictly rely on the portlet specification to build out the platform's user interface. Although there are many ways of building out the UI, using the MVC pattern along with portlets as the implementation of the Controller and View Layer is still the essence of Liferay development.

Portlets are not only used in building your own applications but also in customizing Liferay's controller layer (see *Chapter 11: Override Controller Actions*).

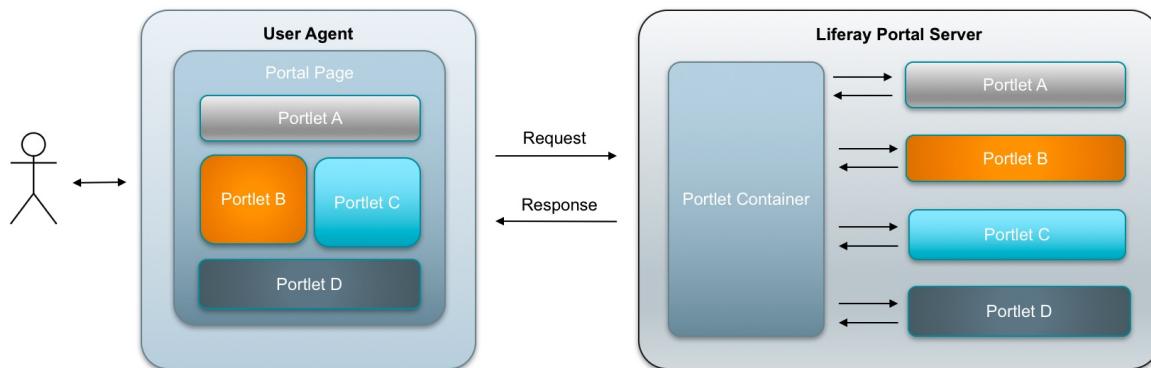
Understanding key portlet concepts and the portlet lifecycle is fundamental to learning back-end development in Liferay.

A portlet is a web component or application that produces an HTML fragment of a page. Generally, any application that has a user interface in Liferay uses a portlet. There are other ways of creating the user interface, but the main way for us as back-end developers is with portlets.

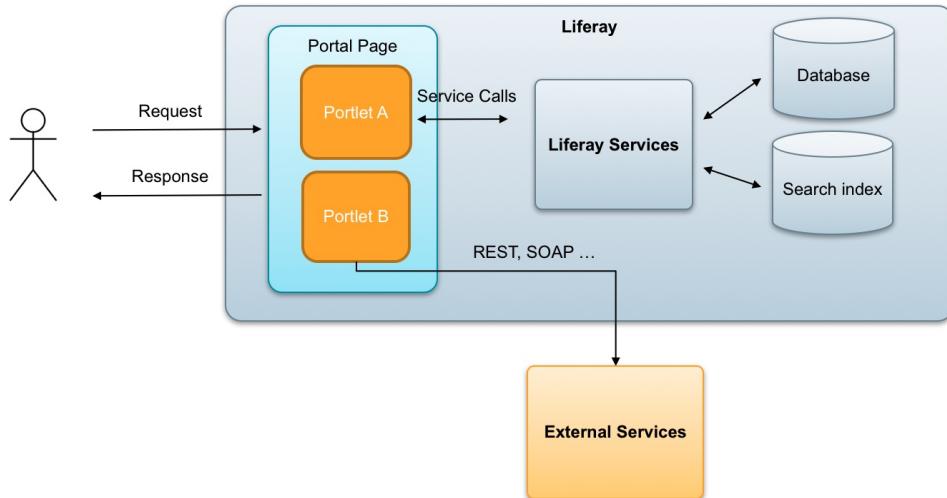
To summarize, a portlet is:

- An application running in a portlet runtime environment called a **portlet container**
- An application in Liferay that has a **user interface**. (Not all user interfaces are portlets.)
- An application that follows the standards governed by the Portlet Specification **JSR-168**, **JSR-286**, or **JSR-362**

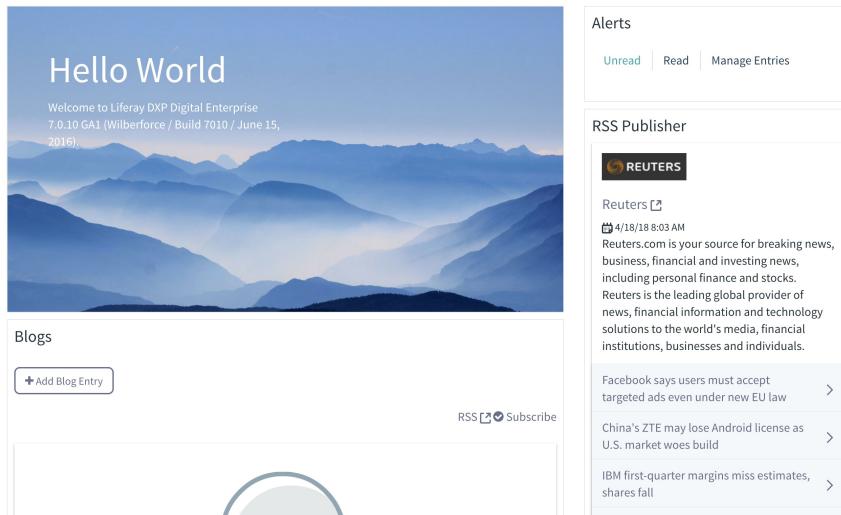
The diagram below shows how when a User comes to a page in Liferay, they can see and interact with the portlets on the page. As the User interacts with a portlet, a request may be sent to Liferay, which the portlet container handles and then has the portlet complete a task. A response is sent back up the chain, and this affects how the User's interaction may be seen. Each one of the portlets on the page is contained and managed by the portlet container in Liferay.



Let's say a User does something to Portlet A. As developers, we can make Portlet A make a service call in Liferay to complete a task. Whether that task is manipulating something in the database or search index, we as developers have control. We can also create a portlet that makes external or web service calls to services that exist outside of Liferay.

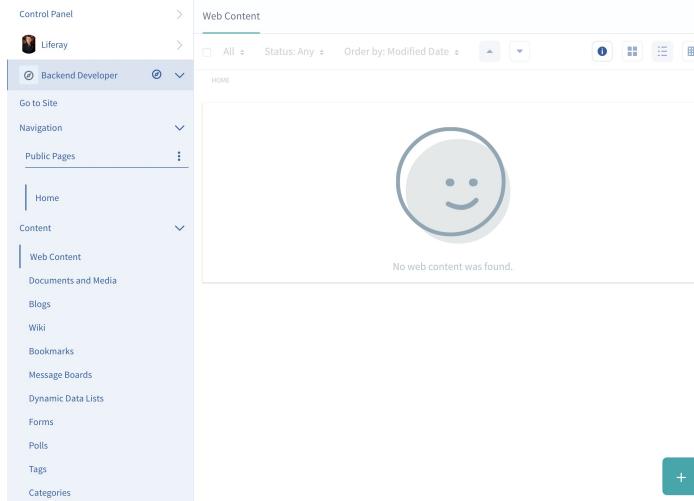


On a Liferay page, the different UI elements that we see are implemented as portlets. The Blogs, Alerts, Hello World, and RSS Publisher are all different portlets on the page.



What are Portlets in the Control Panel?

Even elements within the various parts of the menu are implemented as portlets. Creating Web Content in Liferay happens through the Web Content portlet.



Building Blocks of a Java Standard Portlet

Java Standard Portlet creation requires several key elements. The first are deployment descriptors. Two deployment descriptors are necessary when you create a standard portlet.

- Descriptors:
 - `web.xml` - Deployment descriptor used to declare various things such as what servlets filters to use, what tablibs to use and the URI for them, and configuration parameters
 - `portlet.xml` - Portlet Deployment Descriptor used to declare various properties of a portlet

Next is the portlet class or Java class that extends a specific implementation of the Portlet Specification. The code implementation of a Java Standard Portlet (a portlet completely governed by the Portlet Specification) is called `GenericPortlet`. This is the default implementation of the `Portlet` interface. When creating a Java Standard Portlet, you extend `GenericPortlet`.

Most of the time, when we want to see something displayed in a portlet, we use JSPs. Portlets by default come with different modes, and with each mode there is a JSP that can be displayed based off the mode that is selected.

Key Concepts of the Portlet Standards

Now that we have the building blocks of a portlet, we're going to see how portlets work by taking a look at the following:

- Portlet Lifecycle
- Portlet Modes
- Window States
- Inter-Portlet Communication
- Events

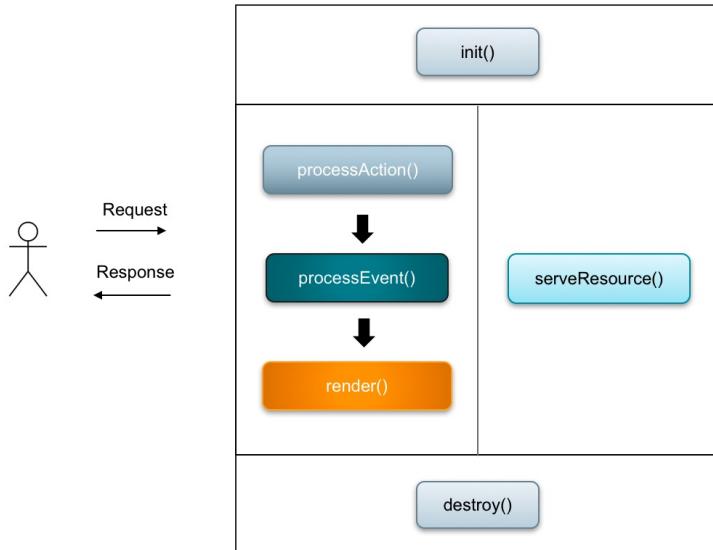
Portlet Lifecycle

The Portlet Lifecycle is what defines how a portlet works and how portlets are to handle specific types of requests. The portlet container is the one that will handle the requests that come in and determine what phase of the portlet lifecycle should be invoked. There are six phases of the portlet lifecycle. The phases are:

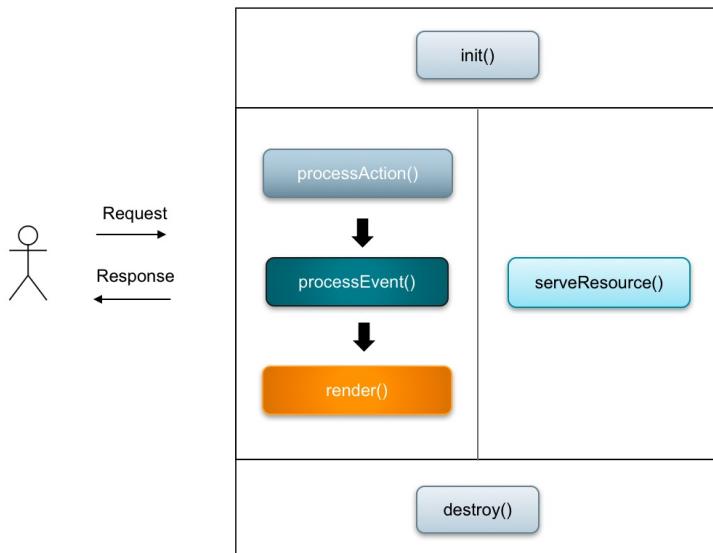
- Init Phase
- Action Phase
- Event Phase
- Render Phase

- Resource Serving Phase
- Destroy Phase

Each phase has a corresponding method that goes with it and most of the default implementation.



Init Phase



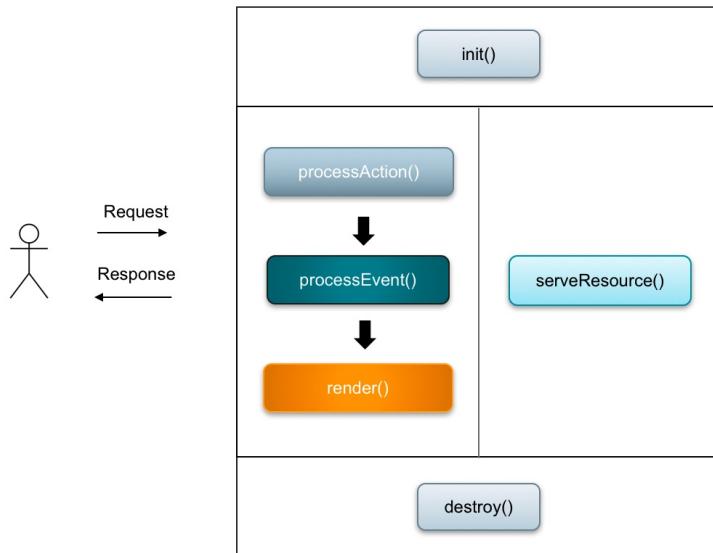
The Init phase of the portlet is called when portlet class initializes or in other words when the portlet is deployed. During the Init phase is when initialization parameters are read. This phase only occurs once, when the portlet is deployed. Each phase of the Portlet Lifecycle has a corresponding method in `GenericPortlet`. For the init phase, it is `init()`.

Render Phase

The render phase is used to generate an **HTML fragment**. In order for our JSPs to be displayed or rendered, this phase has to be invoked. Every portlet that is seen on a page had to have gone through the render phase at one point in time. If ever the page changes or is refreshed, the portlet or portlets on a page will have to go through the render phase again.

This phase along with the action phase is part of what's known as the **request response lifecycle**. When interacting with this phase, there are wrapper objects, **RenderRequest** and **RenderResponse**, that are used to store and retrieve various attributes that help with our development and control over what's happening throughout the render phase and other phases.

The method in `GenericPortlet` that corresponds with the render phase of the Portlet Lifecycle is `render()`.



Calling Render Phase Example

There are a number of different ways the render phase is called, like when a portlet is added to a page or when the page is loaded or reloaded or after the action phase is finished. The most notable and important way is when we as developers make it possible for the User to (unbeknownst to the User) call the Render phase. This allows the User to display different JSPs that are relevant to what they are doing.

When developing a way for the User to call the Render phase, there are two parts. In the JSP, the first part involves a render URL that's used to invoke the Render phase. As seen below, we assign a variable name to the render URL. We reference the variable name or `var` in the anchor tag so that when the link is clicked, the Render phase is invoked.

JSP

```

<portlet:renderURL var="viewEntryUrl">
    <portlet:param name="entryId" value="<%=" String.valueOf(entry.getEntryId()) %>" />
</portlet:renderURL>

<a href="<%=" viewEntryUrl %>">Click here to view the entry</a>
  
```

When the render URL is invoked, it creates an URL. The URL contains parameters such as which portlet is invoking the phase, what lifecycle phase is being invoked, and other information about the portlet.

Generated Portlet URL

```

http://localhost:8080/web/guest/home?p_p_id=com_liferay_blogs_web_portlet_BlogsPortlet&p_p_lifecycle=0&p_p_state=normal&p_p_mode=view&_com_liferay_blogs_web_portlet_BlogsPortlet_entryId=34403
  
```

Render in the Portlet Class

The second part of developing a way for the User to call the Render phase is the Java class, or portlet class. In the portlet class is `render()` with the **renderRequest** and **renderResponse** objects. As seen below, we are able to retrieve parameters that were set in the JSP and stored in the **renderRequest** and then call services using those parameters.

For example, we may want to have a specific entry displayed to the end User. To display the entry on a JSP, we may first need to retrieve that entry from the database. As the JSP is being rendered, the render method is called. In the **renderRequest** object is the **entryId**. So we get the **entryId** by calling a service and then sending the entry object over to the JSP to be displayed. Below is an example of how that might happen.

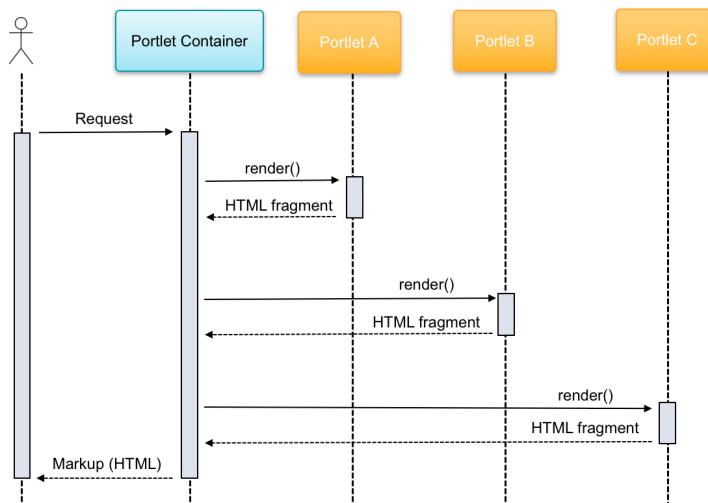
```
@Override
public void render(RenderRequest renderRequest, RenderResponse renderResponse) throws IOException, PortletException {

    String entryId = ParamUtil.getString(renderRequest, "entryId");

    // Fetch entry code here

    super.render(renderRequest, renderResponse);
}
```

Render Phase Flow

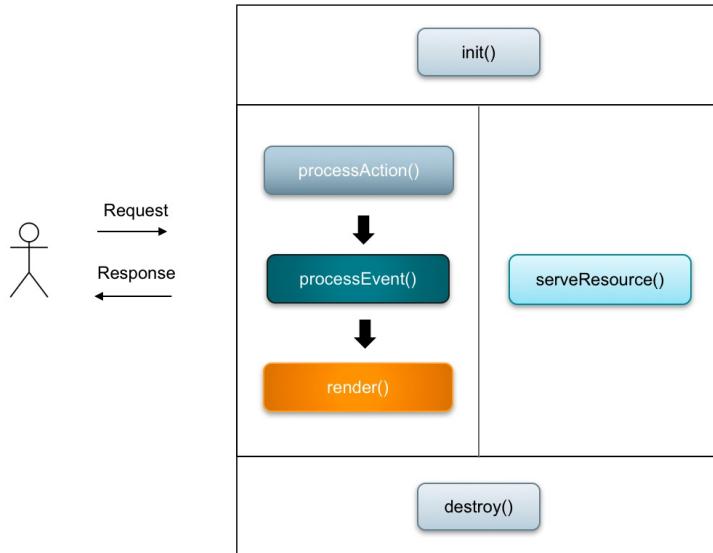


The render request is handled by the Portlet Container and will determine which portlet or portlets need to render. Depending on the situation, not all portlets may go through the render phase at the same time. If all the portlets do go through the render phase, the collection of the HTML fragments are then assembled together and the render response will contain the Markup for the page.

Action Phase

The Action Phase of a portlet is used to respond to any specific actions the User performs. A vast majority of the time we as back-end developers are the ones who expose to the User the ability to call the Action Phase. The most common way the action phase is triggered is through an HTML form submit.

The corresponding method that goes with the Action Phase is **processAction()**. Just like the Render Phase, there is also an **ActionRequest** and **ActionResponse** object. During the Action Phase, **events** can be triggered that invoke another phase called the Event Phase. Assuming the Event Phase isn't called, after the Action Phase, the Render Phase is called and all the portlets on the page will render.



Calling Action Phase Example

Just as with the Render Phase, there are two parts involved: the JSPs and the portlet class. On the JSP side, there is an Action URL used to call the Action Phase. In the example that follows, we see the variable name tag or `var=editEntryURL` referenced in the `aui:form action`. So when an action is performed on the form (e.g., clicking on a Submit button), the actionURL will be invoked, calling the Action Phase. One difference between the Render and Action URL is the `name` tag. This name tag is used to handle multiple actions, specifically targeting a specific `processAction()`.

JSP

```

<portlet:actionURL name="/blogs/edit_entry" var="editEntryURL" />

<aui:form action="<% editEntryURL %>" method="post" name="fm">
    <aui:input name="entryId" type="hidden" value="<% entryId %>" />
    <aui:input name="title" type="text" />
</aui:form>

```

As with the Render URL, the actionURL will generate a literal URL, as you can see below. It contains information such as which portlet is invoking the Action Phase and other parameters that are used to control what happens over in the Java side, mainly which `processAction()` is called.

Generated Portlet URL

```

http://localhost:8080/web/guest/home?p_p_id=com_liferay_blogs_web_portlet_BlogsPortlet&p_p_lifecycle=1&p_p_state=maximized&p_p_mode=view&.com_liferay_blogs_web_portlet_BlogsPortlet_javascript.portlet.action=%2Fblogs%2Fedit_entry&.com_liferay_blogs_web_portlet_BlogsPortlet_entryId=34403&.com_liferay_blogs_web_portlet_BlogsPortlet_cmd=update&.com_liferay_blogs_web_portlet_BlogsPortlet_title=Working+Example&p_auth=7u0mzvxB

```

Process Action in the Portlet Class

Once the actionURL is invoked in the portlet class, `processAction()` is called. Inside `actionRequest` will contain a number of parameters that are set over on the JSP side. In the example below from `actionRequest`, we retrieve a parameter called "title". From there, we could call a service to do something such as updating a blog post's title field.

```

@Override
public void processAction(ActionRequest actionRequest, ActionResponse actionResponse) throws IOException, PortletException {

```

```

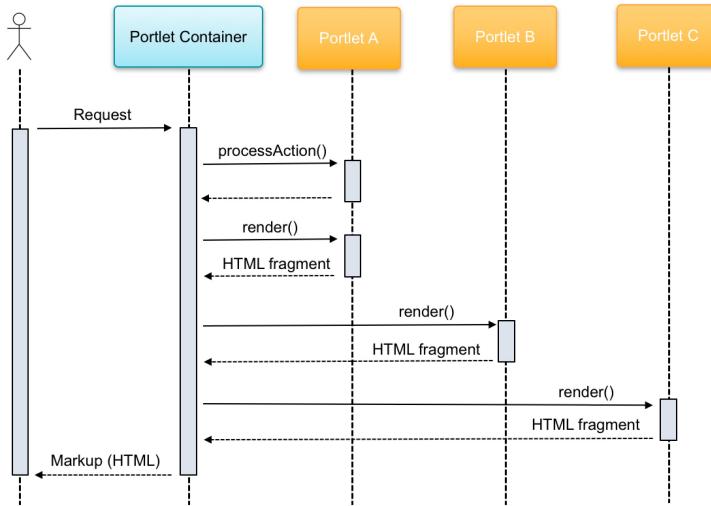
String title = ParamUtil.getString(actionRequest,"title");

// Get the rest of parameters and call model layer

super.processAction(actionRequest, actionResponse);
}

```

Action Phase Flow



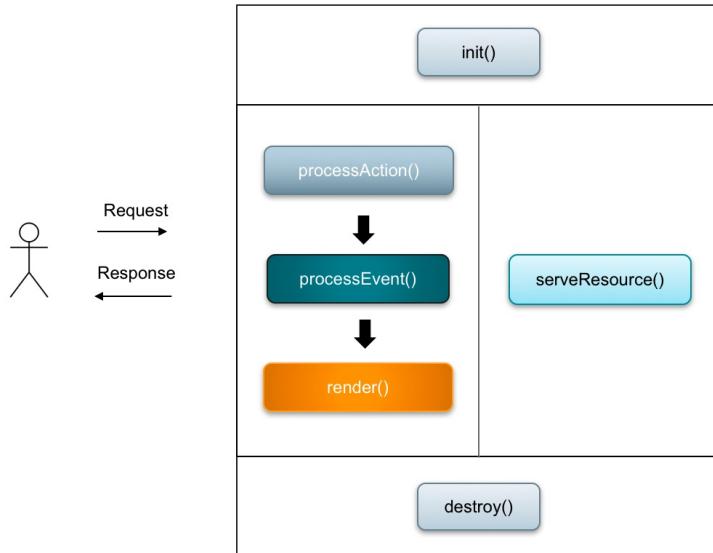
Once a User makes an action request, the portlet container will invoke the action phase for the appropriate portlet. Once the action phase is finished, the render phase is invoked. If there are 3 portlets on a page, Portlet A, B, and C, all the portlets will enter the render phase.

Event Phase

The Event Phase was introduced as a way for portlets to communicate with each other. The idea of portlet talking with each other is known as **Interportlet Communication (IPC)**.

For events to work, we have to register which portlets are going to be setting/publishing events and which portlets are going to be listening/processing events. The portlet that is publishing an event will do so during its Action Phase and the portlet(s) processing an event will do so during the Event Phase. All portlets configured to process an event will process the event once the event is set.

Once the event phase is finished, the Render Phase will be called. The corresponding method for the Event Phase is `processEvent()`.



Calling Event Phase Example

Process Action Method in the Portlet A

```

@Component(
    immediate = true,
    property = {
        "com.liferay.portlet.display-category=category.sample",
        "com.liferay.portlet.instanceable=true",
        "javax.portlet.display-name=Event Publisher Portlet",
        "javax.portlet.init-param.template-path=/",
        "javax.portlet.init-param.view-template=/view.jsp",
        "javax.portlet.name=" + LifecyclePortletKeys.EVENT_PUBLISHER,
        "javax.portlet.resource-bundle=content.Language",
        "javax.portlet.security-role-ref=power-user,user",
        "javax.portlet.supported-publishing-event=message;http://www.liferay.com",
    },
    service = Portlet.class
)
public class EventPublisherPortlet extends MVCPortlet {

    @Override
    public void processAction(ActionRequest actionRequest,
        ActionResponse actionResponse)
        throws IOException, PortletException {

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

        QName qName = new QName("http://www.liferay.com", "message");
        actionResponse.setEvent(qName, message);

        super.processAction(actionRequest, actionResponse);
    }
}

```

Portlet A is the publisher of event `message;http://www.liferay.com` __ Process Event Method in the Portlet B __

```

@Component(
    immediate = true,
    property = {
        "com.liferay.portlet.display-category=category.sample",
        "com.liferay.portlet.instanceable=true",
        "javax.portlet.display-name=Event Receiver Portlet",
    }
)
public class EventReceiverPortlet extends MVCPortlet {

```

```

"javax.portlet.init-param.template-path=/",
"javax.portlet.init-param.view-template=/view.jsp",
"javax.portlet.name=" + LifecyclePortletKeys.EVENT_RECEIVER,
"javax.portlet.resource-bundle=content.Language",
"javax.portlet.security-role-ref=power-user,user",
"javax.portlet.supported-processing-event=message;http://www.liferay.com",
},
service = Portlet.class
)
public class EventReceiverPortlet extends MVCPortlet {

@ProcessEvent(qname = "{http://www.liferay.com}message")
public void handleProcesseuserEmailAddressEvent(EventRequest request,
EventResponse response)
throws javax.portlet.PortletException, java.io.IOException {

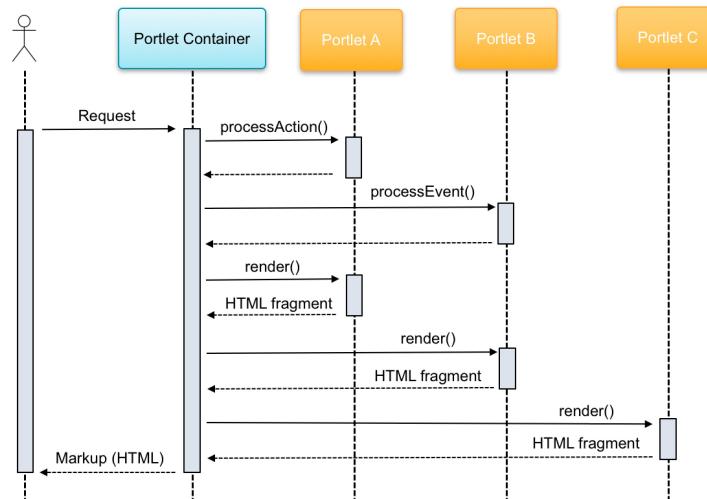
Event event = request.getEvent();
String message = (String) event.getValue();

response.setRenderParameter("messageReceived", message);
}
}

```

Portlet B is the processor of event `message;http://www.liferay.com`

Event Phase Flow

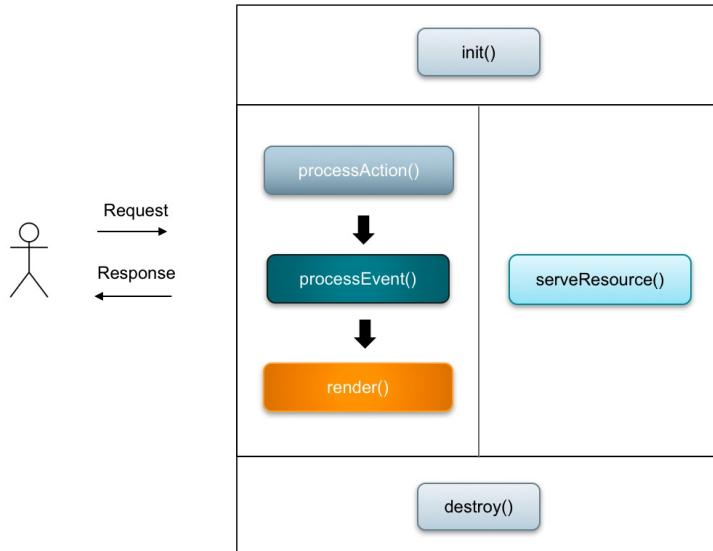


A User invokes the Action Phase of Portlet A. In `processAction()` method, the event is set and Portlet B's Event Phase will be invoked. Once the Event Phase of Portlet B is finished, all of the portlets on the page will enter the render phase.

Resource Serving Phase

As the name implies, the Resource Serving Phase will provide a way to serve resources or in other words provide a way to send resources to the client. We send things such as html, json, an image, and so on. What makes this Lifecycle phase unique is that it bypasses all the other phases. No render or action phase is called when the resource serving phase is invoked.

One of the biggest features of the Resource Serving Phase is the ability to use Ajax.



Resource Serving Phase Use Cases

- Auto completion of a search field
- Refreshing news content area without page refresh
- Doing any background operation without page refresh

Calling Serve Resource Phase Example

JSP

```

<liferay-portlet:resourceURL var="resourceRequestURL" >
    <liferay-portlet:param name="entryId" value="" />
</liferay-portlet:resourceURL>

<a href="#" onclick="fetchData()">Fetch Data</a>

<div id="data-element"></div>

<aui:script>
    function fetchData() {
        AUI().use('aui-io-request', function(A) {
            A.io.request('', {
                method: 'post',
                on: {
                    success: function() {

                        var data = JSON.parse(response.responseText);
                        A.one('#data-element').html(data);
                    }
                }
            });
        });
    }
</aui:script>

```

When the anchor tag is clicked, the function `fetchData()` is called and will invoke the `resourceURL`. The `resourceURL` will pass the parameter `"entryId"` with the value of `<%=entryId%>` (let's assume that value entryId is set else not pictured). Once the `serveResource` method is executed without error, we'll parse the JSON from the response of `serveResource()` and display it.

serveResource Method in the Portlet Class

```

@Component(
    immediate = true,
    property = {
        "com.liferay.portlet.display-category=category.sample",
        "com.liferay.portlet.instanceable=true",
        "javax.portlet.display-name=Lifecycle Portlet",
        "javax.portlet.init-param.template-path=/",
        "javax.portlet.init-param.view-template=/view.jsp",
        "javax.portlet.name=" + LifecyclePortletKeys.LIFECYCLE,
        "javax.portlet.resource-bundle=content.Language",
        "javax.portlet.security-role-ref=power-user,user",
    },
    service = Portlet.class
)
public class LifecyclePortlet extends GenericPortlet {

    @Override
    public void serveResource(ResourceRequest resourceRequest,
        ResourceResponse resourceResponse) throws IOException,
        PortletException {

        String entryId = ParamUtil.getString(resourceRequest, "entryId");

        String entryData = getEntryData(entryId);

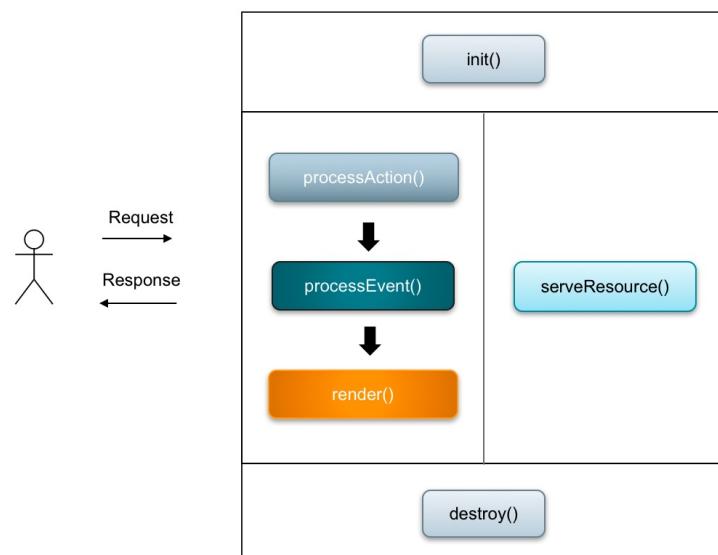
        JSONObject json = JSONFactoryUtil.createJSONObject();
        json.put("entryData", entryData);

        JSONPortletResponseUtil.writeJSON(
            resourceRequest, resourceResponse, json);
    }
    ...
}

```

When the anchor tag is clicked and the resourceURL is invoked, `serveResource()` will be called and the parameter name `entryId` will be set to the String `entryId`. The value `entryId` will be set to the String `entryData`. We'll then create a JSON object to store the value `entryData` to the parameter "entryData" and finally store the JSON object in the resourceRequest object, where it will be retrieved back over at the JSP.

Destroy Phase



When a portlet is undeployed, the Destroy phase is invoked to do any clean-up before the portlet is removed. Resources are cleaned up and the portlet itself is released from the portlet container to be eligible for garbage collection. The method that is called when the Destroy Phase is invoked is `destroy()`.

Portlet Modes

The portlet specification gives the ability for the portlet to have different perspectives or modes. Each mode, when selected, will render a JSP that corresponds with that mode. The typical naming convention is `view.jsp` for the VIEW mode, `edit.jsp` for the EDIT mode, and `help.jsp` for the HELP mode.

There are three modes, each with their respective uses. If you really wanted to, you could use the mode for something it's not designed for, but the three modes are the following:

```
* __VIEW__: Standard mode used as a general point of view
* __EDIT__: Configuration mode used to customize the behavior of the portlet
* __HELP__: Displays portlet's help information
```

The only mode that is required is the VIEW mode.

Liferay Custom Modes

Liferay also provides the following modes that can be leveraged. The same idea is in place with these Liferay Modes, where another perspective or JSP can be enabled to display relevant information based off that specific mode.

- About
- Config
- Edit default
- Edit guest
- Print
- Preview

Portlet Modes

In order for the JSP of each mode to render, `GenericPortlet` has a method that is called for each mode. For the VIEW mode, `GenericPortlet` will call `doView()`, EDIT mode will call `doEdit()`, and the HELP mode will call `doHelp`.

All of this is done from `render()` in `GenericPortlet`. In `render()`, `doDispatch` is called and figures out which portlet mode is selected. Once the portlet mode is selected, the corresponding method is called, but it's up to us as the developers to implement `doView`, `doEdit`, and `doHelp` if we are creating a Java standard portlet.

There are two different ways you can set portlet modes. It can be done either in the JSP or in the Portlet Class.

Setting Portlet Mode

Setting Mode in JSP

```
<portlet:renderURL portletMode="VIEW" var="renderURL" />
```

Setting Mode in the Portlet Class

```
actionResponse.setPortletMode(PortletMode.EDIT);
```

javax.portlet.GenericPortlet.render()

In `render()` we're looking to see how to render the portlet, most of the time `doDispatch()` is called.

```
public void render(RenderRequest request, RenderResponse response) throws PortletException, java.io.IOException {
    Object renderPartAttrValue = request.getAttribute(RenderRequest.RENDER_PART);
    if (renderPartAttrValue != null) {
        // streaming portal calling
        if (renderPartAttrValue.equals(RenderRequest.RENDER_HEADERS)) {
            doHeaders(request, response);
            Collection<PortletMode> nextModes = getNextPossiblePortletModes(request);
            if (nextModes != null)
                response.setNextPossiblePortletModes(nextModes);
            response.setTitle(getTitle(request));
        } else if (renderPartAttrValue.equals(RenderRequest.RENDER_MARKUP)) {
            doDispatch(request, response);
        } else {
            throw new PortletException("Unknown value of the 'javax.portlet.render_part' request attribute");
        }
    } else {
        // buffered portal calling
        doHeaders(request, response);
        Collection<PortletMode> nextModes = getNextPossiblePortletModes(request);
        if (nextModes != null)
            response.setNextPossiblePortletModes(nextModes);
        response.setTitle(getTitle(request));
        doDispatch(request, response);
    }
}
```

javax.portlet.GenericPortlet.doDispatch()

In `doDispatch`, we're looking to see first which window state the portlet is in (we'll discuss window states soon).

Assuming that the window state is not **MINIMIZED**, we'll then render the portlet based on which mode the portlet is in.

First, by checking to see if the portlet mode is cached, then checking to see what portlet was set.

```
protected void doDispatch(RenderRequest request, RenderResponse response) throws PortletException,
    java.io.IOException {
    WindowState state = request.getWindowState();

    if (!state.equals(WindowState.MINIMIZED)) {
        PortletMode mode = request.getPortletMode();
        // first look if there are methods annotated for
        // handling the rendering of this mode
        try {
            // check if mode is cached
            Method renderMethod = renderModeHandlingMethodsMap.get(mode.toString());
            if (renderMethod != null) {
                renderMethod.invoke(this, request, response);
                return;
            }
        } catch (Exception e) {
            throw new PortletException(e);
        }

        // if not, try the default doXYZ methods
        if (mode.equals(PortletMode.VIEW)) {
            doView(request, response);
        } else if (mode.equals(PortletMode.EDIT)) {
            doEdit(request, response);
        } else if (mode.equals(PortletMode.HELP)) {
            doHelp(request, response);
        } else {
            throw new PortletException("unknown portlet mode: " + mode);
        }
    }
}
```

Window States

Window States are indicators of how much space the portlet is going to take up on a page once it renders. There are three windows states that the Portlet Specification defines, **NORMAL**, **MAXIMIZED**, and **MINIMIZED**.

The **NORMAL** window state is the default window state where this portlet may share the page with other portlets.

The **MAXIMIZED** window state will have the portlet be the only portlet rendered on the page and typically take up the whole page.

The **MINIMIZED** window state is where little to nothing is rendered. In Liferay, **MINIMIZED** will only display the title bar of a portlet.

Setting Window State

Window states, just like portlet modes, can be set in either the JSP or in the Portlet class.

Setting Window State in JSP

```
<portlet:renderURL windowState="" var="renderURL">
```

Setting Window State in the Portlet Class

```
actionResponse.setWindowState(WindowState.NORMAL);
```

Inter-Portlet Communication

One of the limitations of the first Portlet Specification, JSR-168, was that there wasn't a standard way for portlets to communicate with each other. In the second Portlet Specification, JSR-268, two methods of Inter-Portlet Communication (IPC) were established, Events and Public Render Parameters. We've already discussed Events, let's now take a look at Public Render Parameters.

Public Render Parameters

Public Render Parameters, when we break down the name, can derive what Public Render Parameters are. A Public Render Parameter is a parameter that is read during the render phase that is public. Let's go into a little bit more detail of Public Render Parameters.

First off, we as developers determine which portlets support which public render parameters. Our Public Render Parameters are usually uniquely identified by a namespace.

Public Render Parameters can be read and manipulated in all the lifecycle phases of a portlet, `processAction()`, `processEvent()`, `render()`, `serveResource`.

Declaring Public Render Parameters

When declaring a Public Render Parameter in a Java Standard Portlet, this happens in **portlet.xml**. We define the Public Render Parameter itself and which other Public Render Parameters we want to use/support. We see in the example below that `trainingPortlet` is both declaring the public render parameter and which public render parameter it supports or is able to use.

portlet.xml

```
<?xml version="1.0"?>

<portlet-app>
  ...
  <portlet>
    <portlet-name>trainingPortlet
    <display-name>Training Portlet
    ...
    <supported-public-render-parameter>tag
  </portlet>

  <public-render-parameter>
    <identifier>tag
    <qname xmlns:x="http://www.liferay.com/public-render-parameters">x:tag
  </public-render-parameter>
</portlet-app>
```

Using Public Render Parameters

We are able to use Public Render Parameter like any other portlet parameter.

JSP

```
<portlet:actionURL var="editEntryURL" />

<aui:form action="<% editEntryURL %>" method="post" name="fm">
    <aui:input name="tag" type="text" />
</aui:form>
```

Portlet Class

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

Non-Standard Communication Methods

There are other ways of IPC that have been used for portlet communication. Though these methods will work, they are technically non-standard methods of IPC. When using IPC, a best practice is to use either Events or Public Render Parameters.

Portlet Sessions

Portlet can share the otherwise private session data by declaring the following in liferay-portlet.xml:

```
<private-session-attributes>false</private-session-attributes>
```

Client-Side IPC

Client-Side IPC can be achieved by using JavaScript or Ajax to communicate with other portlets on the same page. Legacy methods are provided in Liferay's JavaScript library. As with all IPC, the portlets that are communicating with each other have to be on the same page.

Client-Side IPC Example

Sender

```
<script type="text/javascript">

    Liferay.fire('eventName', {
        parameterName1:parameterValue1,
        parameterName2:parameterValue2
    });
</script>
```

Receiver (on the same portal page)

```
<script type="text/javascript">
```

```
Liferay.on('eventName',function(event) {  
  
    console.log(event.parameterName1)  
    console.log(event.parameterName2)  
  
});  
  
</script>
```

Things to Remember / Common Pitfalls

- ActionRequest and RenderRequest force all the portlets on a page to rerender.
- The render phase is for producing the HTML fragment. It cannot do redirection.
- A window state can only be set in the action phase.
- By default, ActionRequest parameters are not available in the render phase but must be set programmatically through the respective methods of the ActionResponse object.
- Liferay's MVCPortlet simplifies this process, by copying all ActionRequest parameters to the render phase. This behaviour can be disabled in portlet.xml.

Exercises

Implement a Basic JSR-286-Compliant Portlet

Introduction

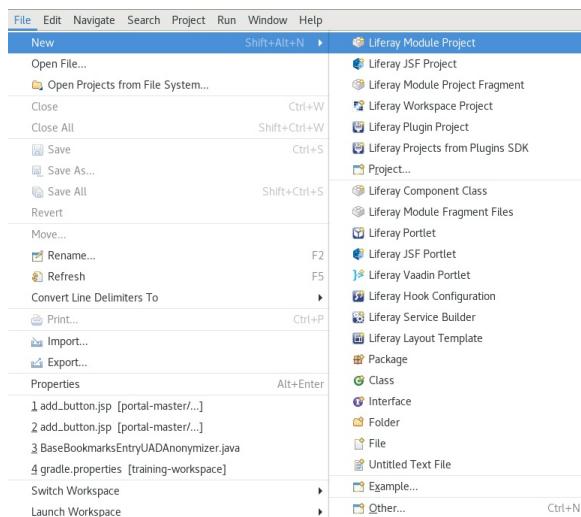
In this exercise, we will implement a very basic JSR-286-compliant portlet that illustrates the most basic concepts of the Portlet Specification 2.0, like the portlet lifecycle, the portlet modes, and window states.

Overview

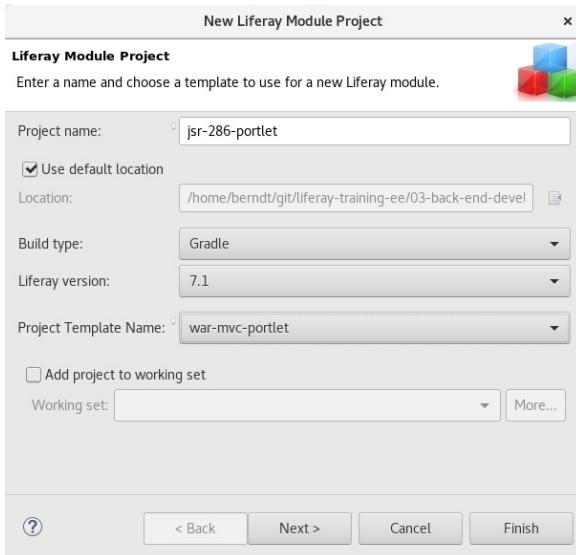
- 1 Create a portlet WAR project
- 2 Implement the portlet class
- 3 Configure the portlet class in portlet.xml
- 4 Build and deploy your JSR-286 portlet to Liferay
- 5 Deploy the portlet and monitor the output in the server log
- 6 Include a JSP for your response output

Create a Portlet WAR Project

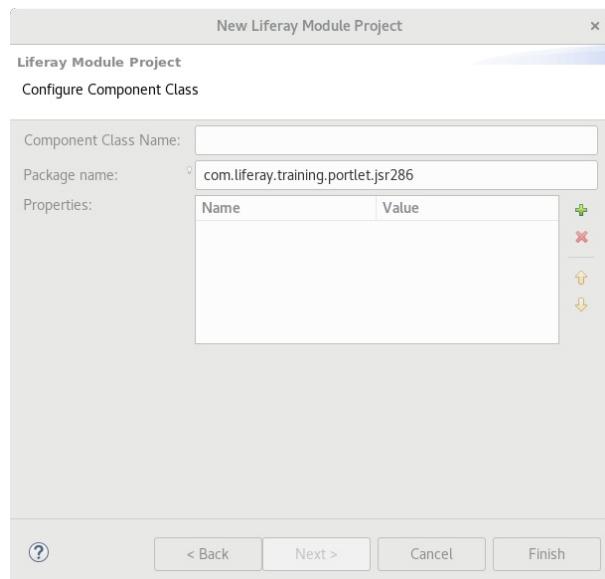
1. Click on *File* → *New* → *Liferay Module Project* in the Developer Studio menu.



2. Type `jsr-286-portlet` for the Project name.
3. Select `war-mvc-portlet` as the Project Template Name.
 - o Make sure that Liferay Version 7.1 is selected.
4. Click **Next**.

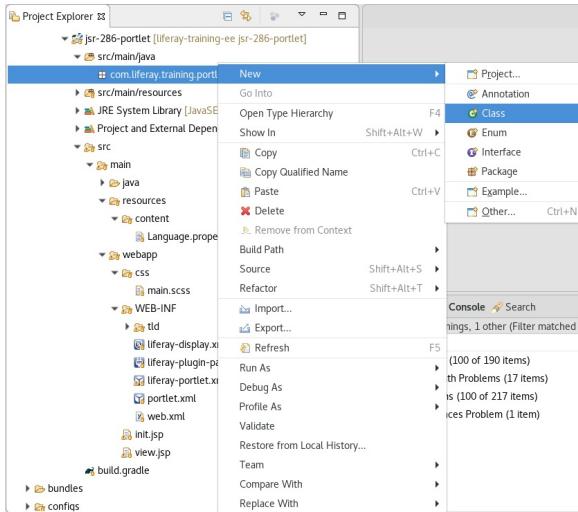


5. Type `com.liferay.training.portlet.jsr286` for the Package name.
 - o We'll leave the Component Class Name field empty, since we won't use a component class in this project.
6. Click **Finish**.

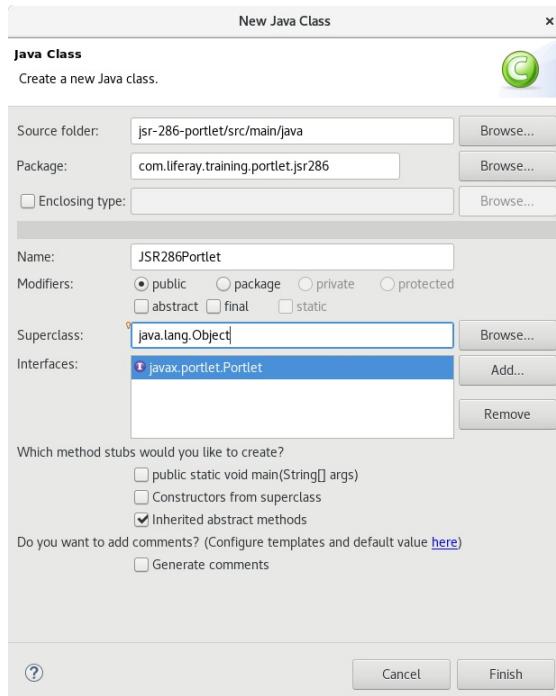


Implement the Portlet Class

1. Find the `jsr-286-portlet` folder in the Project Explorer under the `wars` folder.
2. Expand the project folder node.
3. Expand the `src/main/java` node.
4. Right-click the empty `com.liferay.training.portlet.jsr286` package.
5. Choose **New → Class**.



6. Type `JSR286Portlet` for the Name in the New Java Class dialogue.
7. Click Add next to the Interfaces field.
8. Type `javax.portlet.Portlet` in the Choose interfaces field.
9. Choose the `Portlet` interface.
10. Click Ok.
 - o Make sure that the "Inherited abstract methods" is checked in the "Which method stubs would you like to create?" section.



11. Click `Finish`.
12. Add status messages to the lifecycle methods defined by the `javax.portlet.Interface` and output a message in the `render()` method.

The method stubs are generated for you automatically when you implement the 'javax.portlet.Portlet' interface.

```
@Override
```

```

public void init(PortletConfig config) throws PortletException {
    System.out.println("JSR286Portlet.init()");
}

@Override
public void processAction(ActionRequest actionRequest, ActionResponse actionResponse)
    throws PortletException, IOException {

    System.out.println("JSR286Portlet.processAction()");
}

@Override
public void render(RenderRequest renderRequest, RenderResponse renderResponse)
    throws PortletException, IOException {

    System.out.println("JSR286Portlet.render()");

    PrintWriter printWriter = renderResponse.getWriter();
    printWriter.write("Output from the JSR286Portlet's render() method.");
}

@Override
public void destroy() {
    System.out.println("JSR286Portlet.destroy()");
}

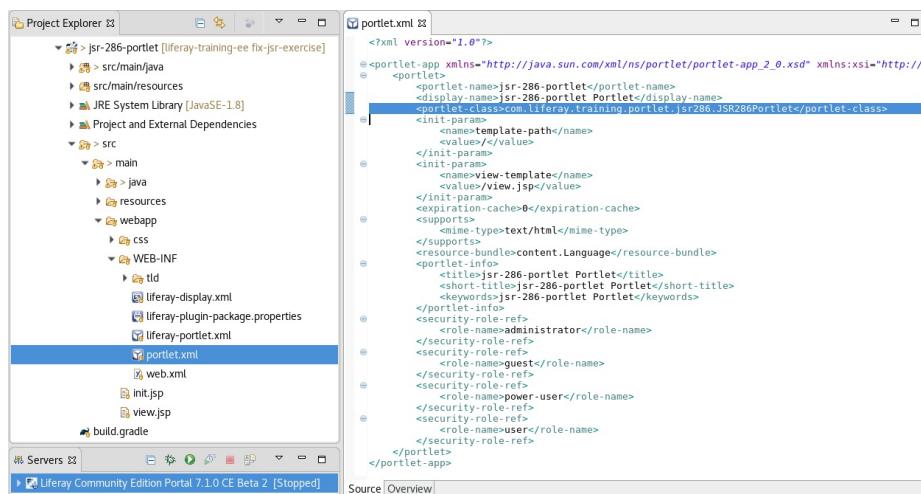
```

- You can use `CTRL+SHIFT+O` to resolve imports.

13. Save the file.

Configure the Portlet Class in portlet.xml

1. Find the `portlet.xml` file in `src → main → webapp → WEB-INF`.
2. Double-click the `portlet.xml` to open it in the XML editor.
3. Click on the Source tab if it isn't already selected.
4. Replace the content between the `<portlet-class></portlet-class>` tags with `com.liferay.training.portlet.jsr286.JSR286Portlet` to configure it as the portlet-class.



- Make sure that the `<portlet-class></portlet-class>` node is inserted between the `<display-name></display-name>` and `<init-param></init-param>` nodes since the sequence of elements is defined in the DTD of the `portlet.xml` document:

```

...
<display-name>jsr-286-portlet Portlet</display-name>

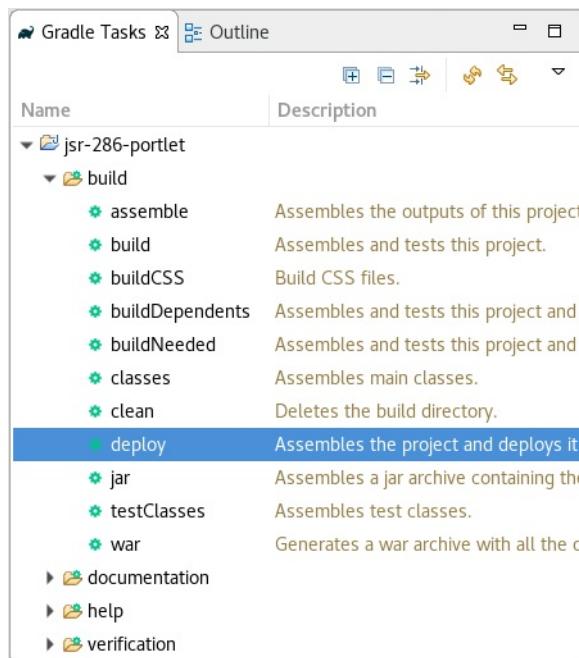
```

```
<portlet-class>com.liferay.training.portlet.jsr286.JSR286Portlet</portlet-class>
<init-param>
    <name>template-path</name>
    <value>/</value>
</init-param>
...
```

5. **Save** the file.

Build and Deploy Your JSR-286-Portlet to Liferay

1. **Find** the *jsr-286-portlet* project in the *wars* file in the Gradle Tasks view of Developer Studio.
2. **Expand** the project's `build` node in the tasks browser.
3. **Double-Click** the `deploy` task to deploy your portlet to Liferay.
4. **Monitor** the console output of your development server.



- The `deploy` task will first call the `compileJava` and `war` tasks to compile and package your project. It will then copy the generated `jsr-286-portlet.war` file from the project's `build/libs` directory to the server's `deploy` directory where Liferay's `AutoDeployListener` will pick it up and deploy it to the server. The server will report the installation progress with several log messages.

You should see the `JSR286Portlet.init()` message from your `init()` method implementation:

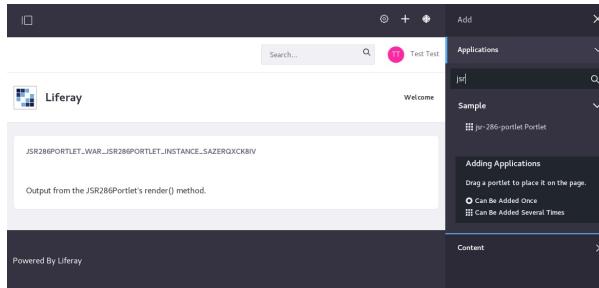
```
2018-05-04 12:55:55.845 INFO [com.liferay.portal.kernel.deploy.auto.AutoDeployScanner][AutoDeployDir:263] Processing jsr-286-portlet.war
2018-05-04 12:56:01.792 INFO [fileinstall-/home/berndt/training-workspace/bundles/osgi/war][BaseAutoDeployListener:43] Copying portlets for /home/berndt/training-workspace/bundles/tomcat-9.0.6/temp/20180504125601791KEAVXNHT/jsr-286-portlet.war
2018-05-04 12:56:01.822 INFO [fileinstall-/home/berndt/training-workspace/bundles/osgi/war][BaseDeployer:876] Deploying jsr-286-portlet.war
2018-05-04 12:56:01.867 INFO [fileinstall-/home/berndt/training-workspace/bundles/osgi/war][BaseAutoDeployListener:50] Portlets for /home/berndt/training-workspace/bundles/tomcat-9.0.6/temp/20180504125601791KEAVXNHT/jsr-286-portlet.war copied successfully
2018-05-04 12:56:02.261 INFO [fileinstall-/home/berndt/training-workspace/bundles/osgi/war][HotDeployImpl:226] Deploying jsr-286-portlet from queue
2018-05-04 12:56:02.262 INFO [fileinstall-/home/berndt/training-workspace/bundles/osgi/war][PluginPackageUtil:1003] Reading plugin package for jsr-286-portlet
```

```
04-May-2018 12:56:02.263 INFO [fileinstall-/home/berndt/training-workspace/bundles/osgi/war] org.apache.catalin
a.core.ApplicationContext.log Initializing Spring root WebApplicationContext
2018-05-04 12:56:02.268 INFO [fileinstall-/home/berndt/training-workspace/bundles/osgi/war][PortletHotDeployLi
stener:186] Registering portlets for jsr-286-portlet
JSR286Portlet.init()
2018-05-04 12:56:02.373 INFO [fileinstall-/home/berndt/training-workspace/bundles/osgi/war][PortletHotDeployLi
stener:298] 1 portlet for jsr-286-portlet is available for use
2018-05-04 12:56:02.481 INFO [fileinstall-/home/berndt/training-workspace/bundles/osgi/war][BundleStartStopLog
ger:35] STARTED jsr-286-portlet_7.1.0.1 [700]
```

Deploy the Portlet and Monitor the Output in the Server Log

After your portlet application has been installed on the server, you can deploy to a portal page.

1. [Login](#) to the portal.
2. [Click](#) on the *Add* button in the upper-right corner.
3. [Expand](#) the *Widget* menu.
4. [Expand](#) the *Sample* section.
5. [Drag](#) the *jsr-286-portlet* Portlet onto a portal page.
 - o You can also use the Add button, which appears when you hover over an application entry.



After you have deployed the portlet onto a portal page, you can see the area occupied by your portlet on the page. By default, the portlet's name is displayed in the portlet-header section. The output from the render method is displayed in the portlet-body section of the portlet. Whenever you reload the page, you will see a `JSR286Portlet.render()` message in your server's log, indicating that your `render()` method has been called. When you navigate to another page of your platform or remove the portlet from the page, these messages are no longer displayed.

Congratulations! You have implemented and deployed a fully JSR-286 standard-compliant portlet!

Include a JSP for Your Response Output

If you want to format your `render()` method's output, you can add HTML markup to the string passed to the `PrintWriter`'s `write()` method. But because this approach quickly becomes cumbersome, the Portlet API provides the `PortletRequestDispatcher` interface, which defines an object that receives requests from the client and sends them to the specified resource, e.g., a JSP file on the server. The `PortletRequestDispatcher`'s `include()` method allows you to include the contents of a resource (= your JSP file) in the response. In essence, this method enables programmatic server-side includes.

The `PortletRequestDispatcher` object can be obtained from the `PortletContext`, which is obtained from the `PortletSession`. The `PortletSession` is obtained from the `PortletRequest`. In the case of the `render()` method, it is the `RenderRequest`.

1. [Open](#) the `JSR286Portlet` class in the `com.liferay.training.portlet.jsr286` package.
2. [Define](#) the path to the `view.jsp` in `src/main/resources`.
3. [Acquire](#) the `PortletSession` from the `RenderRequest`.

4. **Acquire** the PortletContext from the PortletSession.
5. **Acquire** the PortletRequestDispatcher from the PortletContext.
6. **Check** whether your include is valid.
 - o The include-path must start with a slash (""/"), which refers to the WAR's root directory.
7. **Include** your JSP with the PortletRequestDispatcher.
8. **Disable** the PrintWriter output.
 - o Resolve any missing imports with `CTRL+SHIFT+O`.
 - o After you have completed these steps, your render() method should look like this:

```

@Override
public void render(RenderRequest renderRequest, RenderResponse renderResponse)
    throws PortletException, IOException {

    System.out.println("JSR286Portlet.render()");

    // PrintWriter printWriter = renderResponse.getWriter();
    // printWriter.write("Output from the JSR286Portlet's render() method.");

    String path = "/view.jsp";

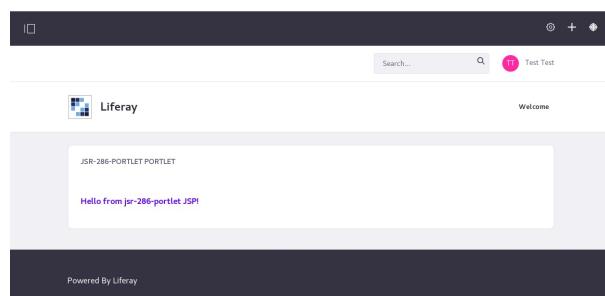
    PortletSession portletSession = renderRequest.getPortletSession();
    PortletContext portletContext = portletSession.getPortletContext();
    PortletRequestDispatcher portletRequestDispatcher = portletContext.getRequestDispatcher(path);

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

```

- o Make sure to resolve any missing imports by using `CTRL+SHIFT+O`
9. **Run** the Gradle `deploy` task to redeploy the portlet.
 10. **Refresh** the page where you have installed your JSR-286-portlet Portlet.

The render method now includes and outputs the `view.jsp` generated by the wizard.



Bonus Exercise: Set a Render Parameter

With the `view.jsp` included by the `PortletRequestDispatcher`, you can make use of the tag-libraries available, simplifying the implementation of the portlet view.

The taglib includes are defined in the `init.jsp`, which is included by the `view.jsp`. Both files have been created by the wizard when you created the project. The Tag Library Descriptor files can be found in the `src/main/webapp/WEB-INF/tld` folder of your project.

In the `view.jsp`, we will use the standard portlet tag library to create a render url, set a render parameter, and explore the behavior of our portlet in the render phase. The exercise has the following steps:

1. **Define** two render urls with the default portlet tag library and set a render parameter with different values:

```
<portlet:renderURL var="viewRedURL">
    <portlet:param name="backgroundColor" value="red"/>
</portlet:renderURL>
<portlet:renderURL var="viewYellowURL">
    <portlet:param name="backgroundColor" value="yellow"/>
</portlet:renderURL>
```

2. **Use** the render urls as parameters for different hyperlink's href attributes.

```
<div class="btn-group">
    <a class="btn btn-default" href="<%= viewRedURL %>">Set red</a>
    <a class="btn btn-default" href="<%= viewYellowURL %>">Set yellow</a>
</div>
```

3. **Process** the parameter value and set the background color style accordingly.

```
```
<% String cssStyle = "";

String backgroundColor = renderRequest.getParameter("backgroundColor");

if (backgroundColor != null && !backgroundColor.isEmpty()) {
 cssStyle = "background-color: " + backgroundColor + ";";
}

%>
```

1. `<span class="action">Wrap</span>` the caption paragraph into a DIV and set its background color with the prepared style attribute.

At the end, your `view.jsp` should look like:

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

<%
    String cssStyle = "";

    String backgroundColor = renderRequest.getParameter("backgroundColor");

    if (backgroundColor != null && !backgroundColor.isEmpty()) {
        cssStyle = "background-color: " + backgroundColor + ";";
    }
%>
<div style="<%= cssStyle %>">

    <p class="caption">
        <liferay-ui:message key="jsr-286-portlet.caption" />
    </p>

</div>

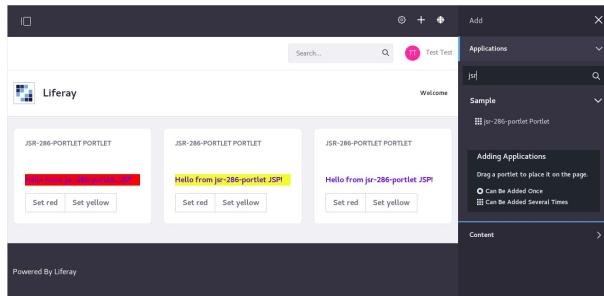
<portlet:renderURL var="viewRedURL">
    <portlet:param name="backgroundColor" value="red"/>
</portlet:renderURL>
<portlet:renderURL var="viewYellowURL">
    <portlet:param name="backgroundColor" value="yellow"/>
</portlet:renderURL>

<div class="btn-group">
```

```
<a class="btn btn-default" href="<%=\ viewRedURL %>">Set red</a>
<a class="btn btn-default" href="<%=\ viewYellowURL %>">Set yellow</a>
</div>
```

1. **Explore** the portlet's behavior when it is deployed multiple times on the same page.

Deploy the refactored portlet to Liferay and add it two or more times to your page. When you click the button in one instance, the background color of the respective portlet changes to the selected color, while all other instances remain the same. Change the color in another instance of the JSR-286-portlet. The background color of the portlet changes, but the background color of the first portlet remains the same, since render parameters remain available to the portlet until they are removed or until their value is changed.



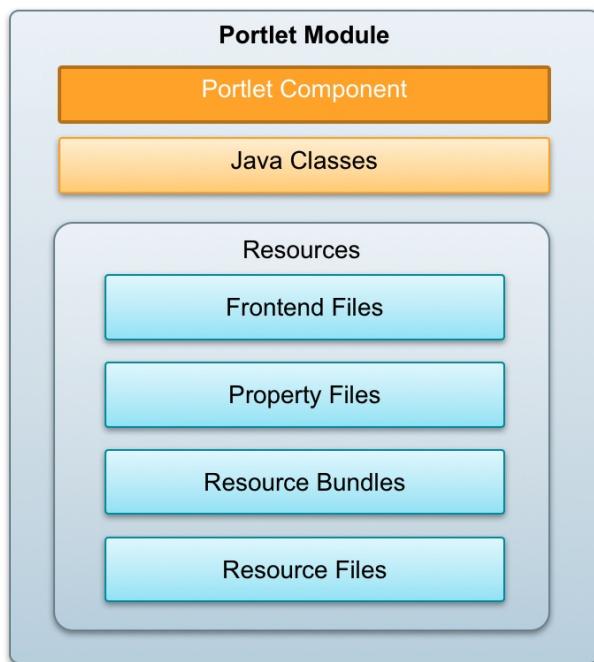
Working with Liferay Portlet Modules

Liferay itself has a portlet container that supports portlets developed under the JSR-168 and JSR-286 portlet specification. If you were to create a portlet using the methodology covered in the previous section, you could get it to deploy and work in Liferay.

Liferay OSGi portlets are based off of the portlet standards but do not follow the Java Standard completely. Liferay has its own implementation of a portlet that extends *GenericPortlet*. Many of the concepts discussed in the previous sections are still in effect, but some of them are implemented a little differently.

What is a Portlet in OSGi?

When we are creating a portlet in OSGi, it's somewhat similar when creating a Java Standard Portlet. There will still be a number of resource files related to the portlet. The main difference will come in the portlet class. When creating the portlet class, we'll actually create it as a component. This component will be registered into the Service Registry as a component that implements the interface *javax.portlet.Portlet*.



Portlet Module Structure

Many of the files that you would expect to see with a portlet and OSGi are there. In the image below, we see the portlet class *TrainingPortlet.java* as well as a few JSP files.

On the OSGi side, the *bnd.bnd* file and *build.gradle*.



Bnd File

The *bnd.bnd* file is used to control the various settings of the bundle project. Functionally, the *bnd.bnd* file will help generate the manifest file of the bundle by adding all the headers to the manifest when the bundle is built. Below, we see a few headers set in *bnd.bnd*.

```
Bundle-Name: Training Portlet
Bundle-SymbolicName: com.liferay.training.portlet
Bundle-Version: 1.0.0
```

build.gradle

Liferay Workspace is configured to use either Gradle or Maven as the build tool. At Liferay, the build tool that we have decided to use is Gradle. The *build.gradle* is what's used to declare the dependencies need for our bundle project. *build.gradle* is also the build script that is used by Gradle to run Gradle commands such as build, deploy, etc.

```
dependencies {
    compileOnly group: "com.liferay.portal", name: "com.liferay.portal.kernel", version: "2.0.0"
    compileOnly group: "javax.portlet", name: "portlet-api", version: "2.0"
    compileOnly group: "javax.servlet", name: "javax.servlet-api", version: "3.0.1"
    compileOnly group: "jstl", name: "jstl", version: "1.2"
    compileOnly group: "org.osgi", name: "osgi.cmpn", version: "6.0.0"
    compileOnly group: "org.osgi", name: "org.osgi.service.component.annotations", version:"1.3.0"
}
```

Portlet Component

When creating portlets in Liferay, we'll be creating them as a component that extends the Liferay portlet framework *MVCPortlet*. We register the component as a component that implements the interface *javax.portlet.Portlet*. In everyday language, we say this is a portlet component. When using Liferay Workspace to create a portlet component, many of the properties of the portlet will be automatically set in the property element.

```
@Component(
    immediate = true,
    property = {
        "com.liferay.portlet.display-category=category.sample",
```

```

    "com.liferay.portlet.header-portlet-css=/css/main.css",
    "com.liferay.portlet.instanceable=true",
    "javax.portlet.display-name=Training Portlet",
    "javax.portlet.init-param.template-path=/",
    "javax.portlet.init-param.view-template=/view.jsp",
    "javax.portlet.name=" + TrainingPortletKeys.TRAINING,
    "javax.portlet.resource-bundle=content.Language",
    "javax.portlet.security-role-ref=power-user,user"
),
service = Portlet.class
)
public class TrainingPortlet extends MVCPortlet {
}

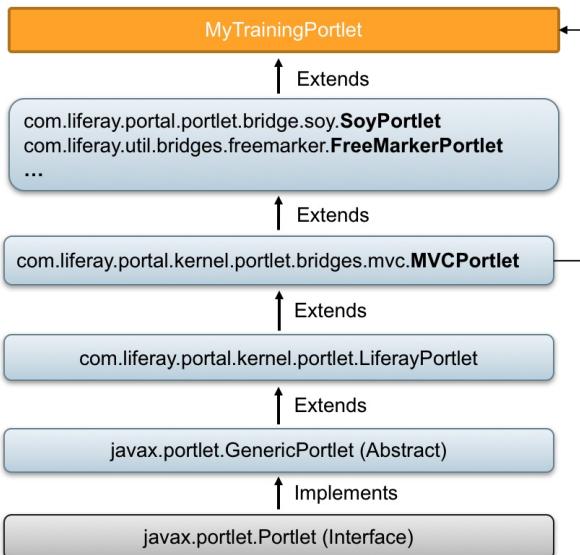
```

Liferay MVC Portlet Class

There are quite a few things that need to be done to get a *GenericPortlet* portlet up and running. Liferay's *MVCPortlet* removes the need to add boilerplate code by already implementing it. Things such as implementing *doView()* are already done for us when using *MVCPortlet*.

Liferay's *MVCPortlet* still inherits from *javax.portlet.Portlet* and extends *javax.portlet.GenericPortlet*, so all the rules and patterns that we learned about *javax.portlet.GenericPortlet* still apply.

Liferay MVC Portlet Class



Portlet Configuration

Portlet configuration is no longer done in *portlet.xml* or *liferay-portlet.xml*. These files don't exist anymore in an OSGi Liferay Module. Instead, all of the properties normally set in those xml files have converted and are set in the property element of the component annotation.

Java Standard Portlet properties are prefixed with *javax.portlet.* and Liferay-specific portlet properties are prefixed with *com.liferay.portlet.*

- Portlet standard - Liferay OSGi Property mapping:

https://dev.liferay.com/develop/reference/-/knowledge_base/7-1/portlet-descriptor-to-osgi-service-property-map

Portlet Configuration

In the first snippet we see the classical way of setting the various attributes of a portlet. In the second snippet we see how the same attributes are set in the portlet component's property element.

portlet.xml

```
<?xml version="1.0"?>
<portlet-app
    xmlns="http://java.sun.com/xml/ns/portlet/portlet-app_2_0.xsd"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/portlet/portlet-app_2_0.xsd http://java.sun.com/xml/ns/portlet/portlet-app_2_0.xsd"
    version="2.0">
    <portlet>
        <portlet-name>l</portlet-name>
        <display-name>Training Portlet</display-name>
        <portlet-class>com.liferay.training.TrainingPortlet</portlet-class>
        <init-param>
            <name>view-jsp</name>
            <value>view.jsp</value>
        </init-param>
        <expiration-cache>0</expiration-cache>
        <supports>
            <mime-type>text/html</mime-type>
        </supports>
        <portlet-info>
            <title>Training Portlet</title>
            <short-title>Training Portlet</short-title>
            <keywords>Training Portlet</keywords>
        </portlet-info>
        <security-role-ref>
            <role-name>administrator</role-name>
        </security-role-ref>
        <security-role-ref>
            <role-name>guest</role-name>
        </security-role-ref>
        <security-role-ref>
            <role-name>power-user</role-name>
        </security-role-ref>
        <security-role-ref>
            <role-name>user</role-name>
        </security-role-ref>
        </portlet>
    </portlet-app>
```

Component properties

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

Portlet Lifecycle Methods

While we still have the classical ways of supporting the various portlet lifecycle methods, in Liferay the preferred way is using MVC Commands. MVC Commands are Liferay's way of implementing the render, action, and serve resource phase of a portlet. MVC Commands are implemented as components and provide a more modular way of handling the portlet lifecycle. Rather than putting everything in the portlet class as is the traditional way, the Liferay way keeps the portlet class lean and makes the various phases of the portlet more manageable.

Lifecycle Handling Example

The code snippets below show how the render phase is handled by an MVC Render Command. The first snippet shows how the renderURL will be invoked by a specific portlet. It will contain the name value pair of *mvcRenderCommandName* and */my_portlet_path/view_entry*.

The second snippet is taken from a component that implements *MVCRenderCommand*. This component will handle the render phase, or, in other words, handle or "listen" to a renderURL. Which renderURL the MVCRenderCommand handles is based on the two properties set in the property element. The *javax.portlet.name* property denotes which portlet is the MVCRenderCommand configured to listen for a renderURL. The second property *mvc.command.name* is the value of *mvcRenderCommandName*, set over on the JSP side.

Once those three criteria are met, a renderURL coming from a specific portlet passes a specific value of mvcRenderCommand, the MVC Render Command that is created for those three criteria is invoked and will trigger the render phase and render a specific JSP.

JSP

```
<portlet:renderURL var="viewEntryUrl">
    <portlet:param name="mvcRenderCommandName" value="/my_portlet_path/view_entry" />
    <portlet:param name="entryId" value="<% String.valueOf(entry.getEntryId()) %>" />
</portlet:renderURL>

<a href="<%= viewEntryUrl %>">Click here to view the entry</a>
```

MVC Render Command Registered to the Portlet and and mvc.command.name

```
@Component(
    immediate = true,
    property = {
        "javax.portlet.name=" + PortletKeys.TRAINING_PORTLET,
        "mvc.command.name=/training_portlet/view_entry"
    },
    service = MVCRenderCommand.class
)
public class ViewMVCRenderCommand implements MVCRenderCommand{

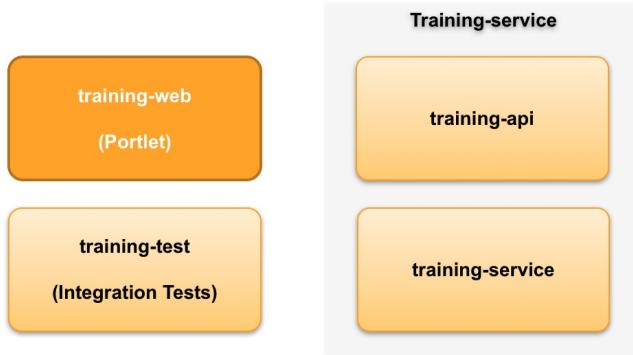
    @Override
    public String render(
        RenderRequest renderRequest, RenderResponse renderResponse) {
        ...
    }
}
```

Portlet Applications in Liferay

In a traditional portlet application, everything was packaged as one war file and that one war file contained everything that was needed for the portlet to run. In a Liferay application, the portlet component only makes up one part of the application. The API and the implementation of the API are usually contained in another module project. For the portlet component, this is typically within a module that ends in -web.

Portlet Application Modules

The diagram below shows how one application can be made up of different modules. In this example, the portlet component is located inside the training-web module. The various services for the training project are found in training-api and training-service, where training-api contains the apis and the implementation of the api is found in training-service.



Example: Liferay Blogs Application

| | | |
|------------------------------|---|--------------|
| blogs-analytics | LPS-74544 Auto SF | 12 hours ago |
| blogs-api | LPS-77699 Update translations | 19 hours ago |
| blogs-demo-data-creator-api | LPS-75049 Auto SF | a month ago |
| blogs-demo-data-creator-impl | LPS-74544 Auto SF | 29 days ago |
| blogs-demo | LPS-75049 Auto SF | a month ago |
| blogs-editor-configuration | LPS-79621 Remove unused code | 22 days ago |
| blogs-item-selector-api | LPS-75049 Auto SF | a month ago |
| blogs-item-selector-web | LPS-74544 Auto SF | 29 days ago |
| blogs-layout-prototype | LPS-77699 Update translations | 5 days ago |
| blogs-reading-time | LPS-75049 Auto SF | a month ago |
| blogs-recent-bloggers-api | LPS-77425 Increment all major versions | 2 months ago |
| blogs-recent-bloggers-test | LPS-77425 Auto SF | 2 months ago |
| blogs-recent-bloggers-web | LPS-79848 Include cancel button | 9 days ago |
| blogs-rest | LPS-75049 Auto SF | a month ago |
| blogs-service | LPS-78354 Apply clay management toolbar to blog entries | a day ago |
| blogs-test-util | LPS-75049 Auto SF | a month ago |
| blogs-test | LPS-79874 Tests are no longer necessary | 5 days ago |
| blogs-uad-test | LPS-80386 blogs-uad-test - removes *UADAggregator* classes | a day ago |
| blogs-uad | LPS-80466 blogs-uad - autogenerated | a day ago |
| blogs-web | LPS-79700 Fixing sorting order issue | 18 hours ago |
| source-formatter.properties | LPS-76110 Add temporary exclusion | 5 months ago |
| subsystem.bnd | LPS-78803 Add Collaboration subsystem.bnd | 2 months ago |
| test.properties | LRQA-40064 Add properties for functional module group tests | 15 days ago |

What Technologies Can Be Used in the Interface?

Although using Liferay's *MVCPortlet* is the preferred way of creating portlets, there are other frameworks that can be used to create a portlet. A more front-end developer way of creating portlets is to leverage *SoyPortlet* and FreeMarker. Sticking with a more traditional framework, Spring MVC and JSF portlets can also be created.

Wrapping it Up

Steps for Creating a Portlet

When creating a portlet in Liferay, there are a few generic steps that apply universally. By following the steps below, you will be on your way to creating your own portlets:

- Decide how you want to implement your portlet, if you're going to be using Liferay's Portlet Framework or a different one.
- If you're creating a Liferay Portlet, you'll be creating an *MVCPortlet* using Liferay Workspace.

- If you're creating a portlet using another framework, you'll extend the base class needed to implement the portlet.
- If applicable, you'll annotate your Java class as a component.
- Within the component annotation, you'll set the necessary properties in the property element.
- You'll then handle your various lifecycle phases either using Liferay's MVC Commands or implementing the lifecycle methods.
- From there, you'll add your business logic.
- Finally, you'll test out what you have with integration tests.

Exercises

Create a Liferay MVC Portlet Module

Introduction

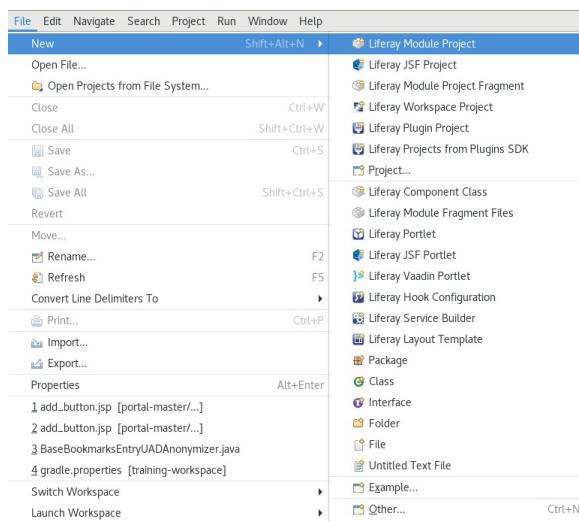
In this exercise, we will implement a basic portlet module, which has similar features as the previously created JSR-286 portlet but makes use of Liferay's MVCPortlet class, Liferay's MVCActionCommand component, and the new modularized OSGi architecture.

Steps Overview

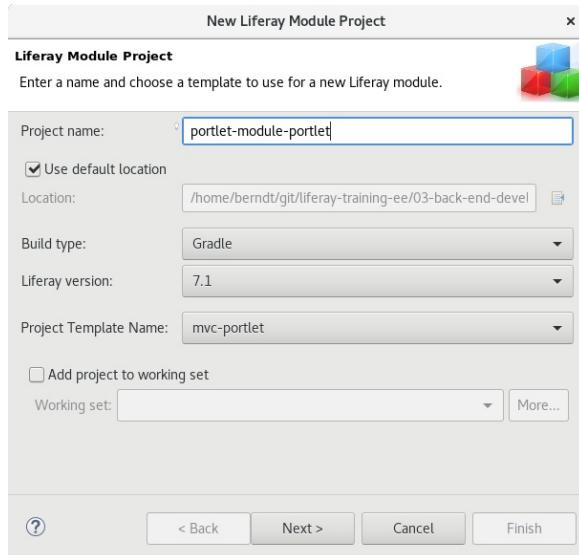
- ① Create a new Liferay module project using the MVC portlet template
- ② Use the auto-deploy feature of Liferay Dev Studio
- ③ Add the portlet to a portal page
- ④ Set and process render parameters in view.jsp
- ⑤ Create a form and actionURL in view.jsp
- ⑥ Handle the ActionRequest with an ActionCommand component

Create a New Liferay Module Project Using the MVC Portlet Template

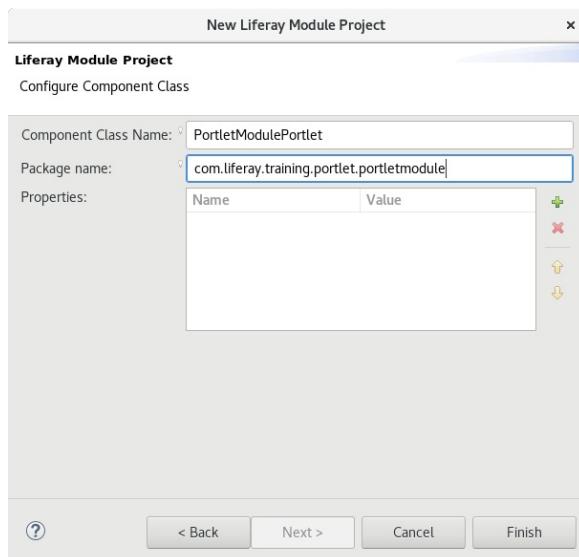
1. Click **File → New → Liferay Module Project** on the Developer Studio menu bar to create a new Liferay Module Project:



2. Type **portlet-module-portlet** for the Project name.
3. Choose **mvc-portlet** for the Project Template Name.
4. Choose Liferay Version 7.1 if it isn't already selected.



5. Click **Next**.
6. Type **PortletModulePortlet** for the Component Class Name.
7. Type **com.liferay.training.portlet.portletmodule.portlet** for the Package Name:



8. Click **Finish**

Use the Auto-Deploy Feature of Dev Studio

1. Drag the *portlet-module-portlet* module from the Project Explorer onto your running server in the Servers view.
 - o Whenever you edit your project and save your modifications, your module will automatically be redeployed to the server and restarted by Liferay's OSGI container. You can see the dependencies for the module in its `build.gradle` file in the Project Explorer.

Add the Portlet to a Portal Page

After the *portlet-module-portlet* has been started:

1. **Login** to Liferay.
2. **Click** on the *Add* button in the upper-right corner.

3. **Expand** the *Widgets* section.
4. **Expand** the *Samples* section.
5. **Drag** the *PortletModule* portlet to a portal page.



Set and Process Render Parameters in view.jsp

1. **Open** the `view.jsp` file in the `src/main/resources/META-INF/resources` folder of the *portlet-module-project*.
2. **Create** two renderURLs with two different parameter values (`viewRedURL`, `viewYellowURL`).
3. **Create** two hyperlinks, which use the renderURLs as their href value.
4. **Retrieve** the background parameter value from the renderRequest and create a corresponding CSS style.
5. **Wrap** the message paragraph into a DIV with a dynamic style attribute.

- o After you have completed these steps, your `view.jsp` should look like:

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

<%
    String cssStyle = "";
    String backgroundColor = renderRequest.getParameter("backgroundColor");

    if (backgroundColor != null && !backgroundColor.isEmpty()) {
        cssStyle = "background-color: " + backgroundColor + ";";
    }
%>
<div style="<%= cssStyle %;>">
    <p class="caption">
        <liferay-ui:message key="jsr-286-portlet.caption" />
    </p>
</div>

<portlet:renderURL var="viewRedURL">
    <portlet:param name="backgroundColor" value="red"/>
</portlet:renderURL>
<portlet:renderURL var="viewYellowURL">
    <portlet:param name="backgroundColor" value="yellow"/>
</portlet:renderURL>

<div class="btn-group">
    <a class="btn btn-default" href="<%= viewRedURL %;>">Set red</a>
    <a class="btn btn-default" href="<%= viewYellowURL %;>">Set yellow</a>
</div>
```

6. **Save** the file.
7. **Refresh** the page in your browser and test your implementation.
 - o You don't have to redeploy the module manually, since you run the portlet in auto-deploy mode.

Note: The implementation of the `view.jsp` is the same as for the JSR-286-portlet. There is no need to explicitly include the `view.jsp`, since `view.jsp` is configured as the default view-template in the Component annotation of the Portlet component. The PortletRequestDispatcher include is performed by the `include()` method of the Component's super class (`MVCPortlet`). We don't have to implement an `init()` or a `destroy()` method, since they're

both implemented in the MVCPortlet class as well.

Create a Form and actionURL in view.jsp

1. **Configure** a named actionURL in view.jsp.
2. **Add** a form element and use the configured actionURL as a parameter for the form's action attribute.
3. **Save** the file.

The implementation of the form is the same as in the JSR-286-portlet exercise. In order to address the HandleFormMVCActionCommand component, we set the `name="handleForm"` attribute of the actionURL, which corresponds to the `"mvc.command.name=handleForm"` property of the HandleFormMVCActionCommand component annotation (see below).

The view.jsp should now look as follows:

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

<%
    String cssStyle = "";
    String backgroundColor = renderRequest.getParameter("backgroundColor");
    if (backgroundColor != null && !backgroundColor.isEmpty()) {
        cssStyle = "background-color: " + backgroundColor + ";";
    }
%>

<div style="<%=cssStyle%>">
    <p>
        <b><liferay-ui:message key="portlet-module-portlet.caption"/></b>
    </p>
</div>

<portlet:renderURL var="viewRedURL">
    <portlet:param name="backgroundColor" value="red" />
</portlet:renderURL>

<portlet:renderURL var="viewYellowURL">
    <portlet:param name="backgroundColor" value="yellow" />
</portlet:renderURL>

<div class="btn-group">
    <a class="btn btn-default" href="<%=viewRedURL%>">Set red</a> <a
        class="btn btn-default" href="<%=viewYellowURL%>">Set yellow</a>
</div>

<portlet:actionURL name="handleForm" var="actionURL"/>

<aui:form action="<%= actionURL %>" style="margin-top: 2rem;">
    <aui:select name="backgroundColor">
        <aui:option label="aqua"/>
        <aui:option label="gray"/>
        <aui:option label="lime" />
        <aui:option label="olive" />
        <aui:option label="silver" />
    </aui:select>
    <aui:button-row>
        <aui:button type="submit" value="send"/>
    </aui:button-row>
</aui:form>
```

Handle the ActionRequest with an ActionCommand Component

Create an ActionCommand component using the component wizard:

1. **Right-click** on the *portlet-module-portlet* project to open the context menu.
2. **Select** New → *Liferay Component Class*.
3. **Type** *com.liferay.training.portlet.portletmodule.action* in the *Package Name* field.
4. **Enter** *HandleFormMVC* in the *Component Class Name*.
 - The name will be automatically added as *ActionCommand* during the creation.
5. **Choose** *Portlet Action Command* component class template.
6. **Click** *Finish* to close the wizard.
7. **Expand** the *com.liferay.training.portlet.portletmodule.action* package found in the *src/main/java* folder of the project.
8. **Delete** the *HandleFormMVCPortlet* class in the created by the wizard.
9. **Open** the *bnd.bnd* file for the *portlet-module-portlet* project.
10. **Click** on the *Source* tab.
11. **Delete** the following lines:

```
-includeresource: \
    @com.liferay.util.bridges-2.0.0.jar!/com/liferay/util/bridges/freemarker/FreeMarkerPortlet.class, \
    @com.liferay.util.taglib-2.0.0.jar!/META-INF/*.tld
-sources: true
```

- The wizard also creates FreeMarker resource file stubs to the META-INF/resource, but we ignore them now.
12. **Update** the component properties in the *HandleFormMVCActionCommand* as follows:

```
"javax.portlet.name=" + PortletModulePortletKeys.PortletModule,
"mvc.command.name=handleForm"
```

13. **Replace** the contents of the *_handleActionCommand()* method with:

```
...
System.out.println("HandleFormMVCActionCommand.doProcessAction()");

String backgroundColor = actionRequest.getParameter("backgroundColor");

System.out.println("backgroundColor = " + backgroundColor);
...
```

14. **Press** `CTRL + SHIFT + O` to resolve imports.
15. **Save** the file.

- Your action command class should look like:

```
/**
 * Copyright 2000-present Liferay, Inc.
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *     http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */
package com.liferay.training.portlet.portletmodule.action;

import com.liferay.portal.kernel.portlet.bridges.mvc.MVCActionCommand;
import com.liferay.training.portlet.portletmodule.constants.PortletModulePortletKeys;
```

```
import javax.portlet.ActionRequest;
import javax.portlet.ActionResponse;
import javax.portlet.PortletException;

import org.osgi.service.component.annotations.Component;

@Component(
    immediate = true,
    property = {
        "javax.portlet.name=" + PortletModulePortletKeys.PortletModule,
        "mvc.command.name=handleForm"
    },
    service = MVCActionCommand.class
)
public class HandleFormMVCActionCommand implements MVCActionCommand {
    @Override
    public boolean processAction(
        ActionRequest actionRequest, ActionResponse actionResponse)
        throws PortletException {
        _handleActionCommand(actionRequest);
        return true;
    }

    private void _handleActionCommand(ActionRequest actionRequest) {
        System.out.println("HandleFormMVCActionCommand.doProcessAction()");
        String backgroundColor = actionRequest.getParameter("backgroundColor");
        System.out.println("backgroundColor = " + backgroundColor);
    }
}
```

16. **Refresh** the portal page and test your implementation.

Chapter 6: Develop a Real-World Application

Chapter Objectives

- Understand How to Build a Complete Real-World Liferay Application
- Understand How to Connect Your Business Logic to Your User Interface
- Learn How to Leverage Time-Saving Liferay Tools and Frameworks in a Single Application

Overview

In this chapter, we'll create a real-world Liferay application using Liferay's time-saving tools and frameworks. Below are our goals and requirements for the exercise application, which will be a course gradebook.

Features

- Teachers can create assignments.
- Students can send submissions to assignments.
- Teachers can grade the submissions.

Functional Requirements

- Both assignments and submissions have to be under access control.
- Assignments have to be listable in the Asset Publisher portlet.
- Assignments have to be searchable with portal search.
- The Application user interface has to be configurable.
- The application has to support localization.

Non-Functional Requirements

- The application has to be modular.
- There have to be basic level integration tests.
- Assignments and submissions have to be persisted in the database.
- Form submissions have to be validated.

Entities to Persist

The following entities should be persisted to the database:

- Assignments
- Submissions

Chosen Development Technologies

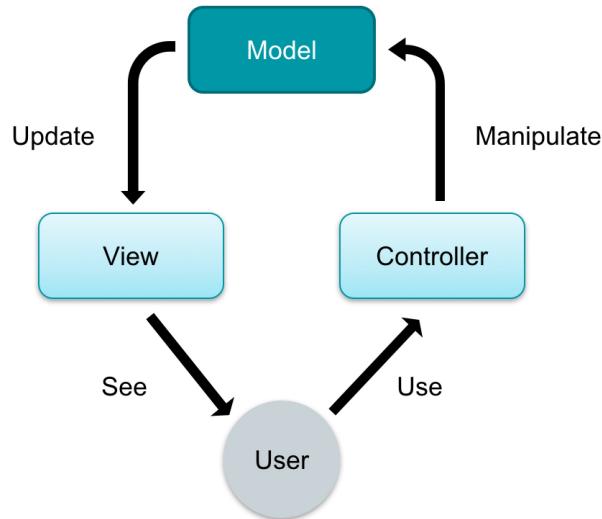
We will build the model and service layer of the Gradebook application using the *Liferay Service Builder* code generation tool, which greatly reduces the need for boilerplate coding, automatically creates the persistence layer, implements caching, and creates both local and remote service APIs.

Liferay provides several options for front-end implementation. We will use the classic Liferay MVC portlet template with JSPs because of the many available tag libraries that simplify user interface implementation.

For the integration test, we will use *Arquillian* and Liferay-provided integration test JUnit rules.

Project Setup

Like Liferay core applications, this application will follow the MVC design pattern:



Module Architecture

The application will be divided in four modules:

- **gradebook-web**: the user interface with portlet component
- **gradebook-api**: the service layer API
- **gradebook-service**: the service layer implementation
- **gradebook-test**: module for integration tests

Implementation Steps Overview

- ① Create the API and service modules.
- ② Define the data model and create the service layer.
- ③ Create the portlet module and implement the user interface.
- ④ Make the application configurable.
- ⑤ Implement access control (permissions).
- ⑥ Integrate with portal Asset, Search, and Workflow frameworks.
- ⑦ Create the integration test module.

Create the Service Layer

Introducing Liferay Service Builder

Liferay Service Builder is a code generation tool that takes an xml configuration file as an input and generates a complete service layer as an output. The generated code includes database schema definition, persistence and caching code, service classes with CRUD methods, and a remote service layer supporting JSON and SOAP web services. The generated code can be complemented and overridden by service implementation classes, which are being created for every entity defined in the service schema.

A service created by the Service Builder defines a zone where all the operations are run within same transaction. Service Builder can also be used without defining any persistence entities just to create a web service.

Service Builder is one of the central development patterns in Liferay and is used in its core services.

Service Builder Features

Service Builder relies on *Hibernate* and *Spring* frameworks. Although service classes generated with it are not OSGi services but Spring beans, they are wired to the OSGi service container and exposed through the OSGi service registry.

The generated database schema can be fine-tuned on a field-mapping level. You can choose to use external datasources for any entity. Although Service Builder-generated code abstracts the database layer providing basic CRUD methods for you, you can completely customize the service implementation classes and use dynamic and custom SQL queries in the code.

The entity and finder level caching layer is generated automatically and is based on EHCache.

Basic Concepts

Here's an overview of the basic concepts of Liferay Service Builder:

- Service and persistence schema definition file `service.xml`
- Database modeling hint file `portlet-model-hints.xml`
- Local service
- Remote service
- Service Implementation classes
- Finders
- Service Wrappers
- Service Context

service.xml

`service.xml` is the main configuration file. With this file, you define:

- Global information for the service (database namespace, package)
- Service entities and their attributes (columns)
- Default order of retrieval
- Entity finder methods
- Generated service variants (local and remote)
- Datasource (internal or external)
- Service exceptions

- Service references available in the generated service classes
- Caching

Service generation, based on the service.xml, is done with *buildService* Gradle task, which generates the code. **Every time service.xml or the generated implementation classes are modified, buildService task has to be re-run.**

Below is an excerpt of service definition with service.xml for the native Liferay Blogs application:

```
<?xml version="1.0"?>
<!DOCTYPE service-builder PUBLIC "-//Liferay//DTD Service Builder 7.1.0//EN"
"http://www.liferay.com/dtd/liferay-service-builder_7_1_0.dtd">

<service-builder auto-import-default-references="false" auto-namespace-tables="false" package-path="com.liferay.blogs">
    <namespace>Blogs</namespace>
    <entity local-service="true" name="BlogsEntry" remote-service="true" trash-enabled="true" uuid="true">

        <!-- PK fields -->

        <column name="entryId" primary="true" type="long" />

        <!-- Group instance -->

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

        <!-- Audit fields -->

        <column name="companyId" type="long" />
        <column name="userId" type="long" />
        <column name="userName" type="String" uad-anonymize-field-name="fullName" />
        <column name="createDate" type="Date" />
        <column name="modifiedDate" type="Date" />
        ...
    ...

```

Source: <https://github.com/liferay/liferay-portal/blob/7.1.x/modules/apps/blogs/blogs-service/service.xml>

portlet-model-hints.xml

The portlet-model-hints.xml file is being generated when Service Builder is run. It lets you customize the entity to SQL column mapping by defining field types, sizes, and validation.

Below is an excerpt of the Liferay Blogs application portlet-model-hints.xml:

```
<?xml version="1.0"?>

<model-hints>
    <model name="com.liferay.blogs.model.BlogsEntry">
        <field name="uuid" type="String" />
        <field name="entryId" type="long" />
        <field name="groupId" type="long" />
        <field name="companyId" type="long" />
        <field name="userId" type="long" />
        <field name="userName" type="String" />
        <field name="createDate" type="Date" />
        <field name="modifiedDate" type="Date" />
        <field name="title" type="String">
            <hint name="max-length">150</hint>
            <sanitize content-type="text/plain" modes="ALL" />
            <validator name="required" />
        </field>
        ...
    ...

```

Source: <https://github.com/liferay/liferay-portal/blob/7.1.x/modules/apps/blogs/blogs-service/src/main/resources/META-INF/portlet-model-hints.xml>

Local Service

When configuring the service, you can define which kind of services are generated. There are two variants available: local and remote. The local service is meant to be an access-point for the application within the same JVM and without any permission checks.

The local service is meant to be the layer where you call the persistence layer to retrieve and store data entities.

Remote Service

The remote service variant serves two purposes. First, it's meant to be a façade layer for the local service, where you can implement permission checking. The other purpose is to provide JSON and SOAP web service APIs.

When developing custom applications, you should generally implement the remote service layer with permissions checks to provide a user level access to the service.

When you don't need permission checks, the local service should be preferred because of better performance.

Notion about Service Builder and OSGi

Service Builder-generated service classes are Spring beans and not OSGi service components. Although they are wired to the OSGi service registry, you can't use the OSGi @Reference annotation inside the Service Builder-generated classes. Making other Liferay core services available in your custom service class should be done by referencing them in the service.xml configuration file. You can also use the Spring @ServiceReference and @BeanReference annotations.

Service Implementation Classes

When you generate the service, implementation classes are, depending on the configuration, created for:

- Local service variant
- Remote service variant
- Entity Model class
- Entity Finders

In the implementation classes, you can override the generated service methods and implement any custom logic needed.

Service implementation classes **are the only classes meant to be modified manually**.

Whenever a modification to these classes is done, the service has to be regenerated by running the *buildService* Gradle task.

Finders

Finders are database querying methods, which you define in the service configuration file service.xml. They are automatically cached and can be customized in the entity-specific finder implementation classes.

There are two kind of finders: *finders*, which work without permission checking and *filtered finders*, which provide permission checking. Service permissions, which will be discussed later in the training, have to be defined in order for filtered finders to be created.

Service Wrappers

When the `buildService` task is run, it not only generates implementation classes for all model entities, it also generates so-called service wrapper APIs, which allow you to override your services from an external module. With this approach, you can also override any Liferay core service. This technique will be discussed in more detail in *Chapter 10*.

Service Context

When you implement Liferay services or, for example, integrate with Liferay frameworks, you'll often see a `ServiceContext` object in the method parameter list.

The `ServiceContext` object holds contextual information for a service. It aggregates information necessary for features used throughout Liferay's portlets and services, such as:

- Actions
- Attributes
- Classification (tags and categories)
- Exceptions
- Scoping (company and group)
- Locale
- Request object
- Permission-related information

For more information, please visit the Liferay Developer Network at https://dev.liferay.com/develop/tutorials/-/knowledge_base/7-1/understanding-servicecontext

Service Builder Caching

Service Builder provides a built-in caching support for all entities and finders defined in the services configuration. Caching is implemented on three levels:

- Hibernate
- Entity
- Finder

Ehcache is the default cache provider for all the levels.

Hibernate cache has two layers: level 1 (L1) and level 2 (L2).

Level 1 is used to cache objects retrieved from the database within the current database session, which is typically tied to the invocation of the Liferay service layer within a single request.

Level 2 stores both database objects (Entity Cache) and results of queries (Query Cache) and is able to span across database sessions.

Liferay provides custom Service Builder Entity and Query (Finder) caches, making the Hibernate L2 cache redundant. L2 cache is disabled by default, and unless you are accessing Hibernate code directly, there shouldn't be a reason to enable it.

Wrapping it Up

Liferay's Service Builder is a code generation tool that takes a `service.xml` configuration file as an input and builds a complete service with persistence and caching code as output. The diagram below illustrates a Service Builder project file structure before and after code generation:

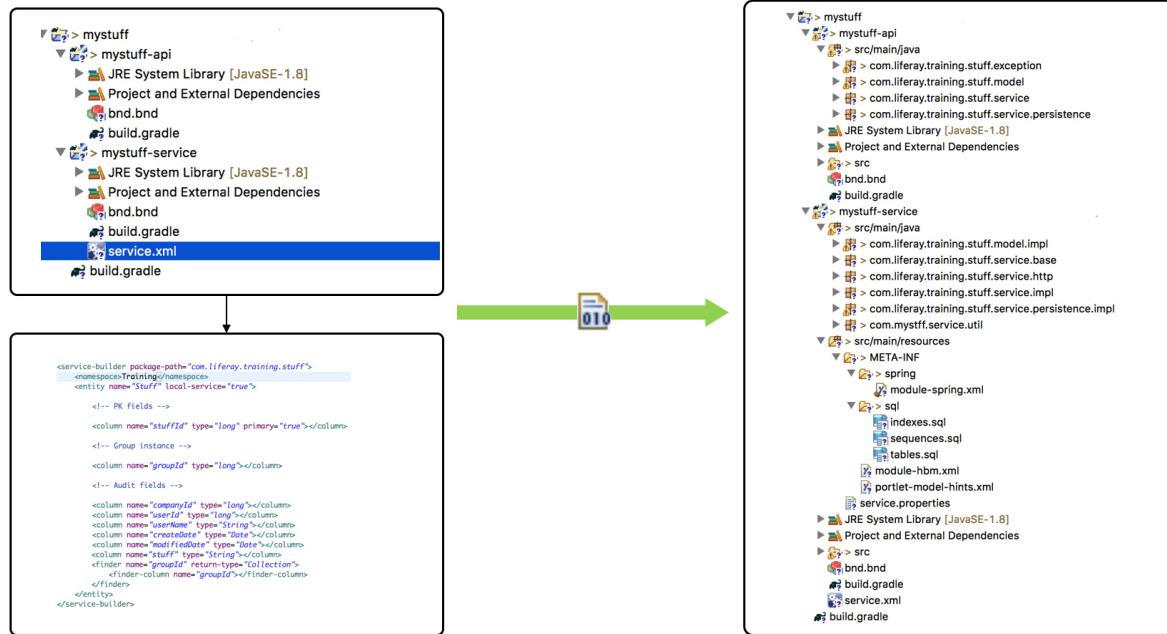


Figure: Service Builder code generation

Service Builder allows you to create two service variants: the **local** for the access from local JVM and without permission checks, and the **remote** for access with permission checks and through JSON and SOAP web service API.

It also allows you to define **finders**, which are database querying methods with caching support.

Additionally, **service wrapper** classes are created for all service implementation classes. Service wrappers allow you to override Service Builder-generated services from within external modules. They can also be used to override core Liferay services.

The diagram below illustrates Service Builder's design and architecture:

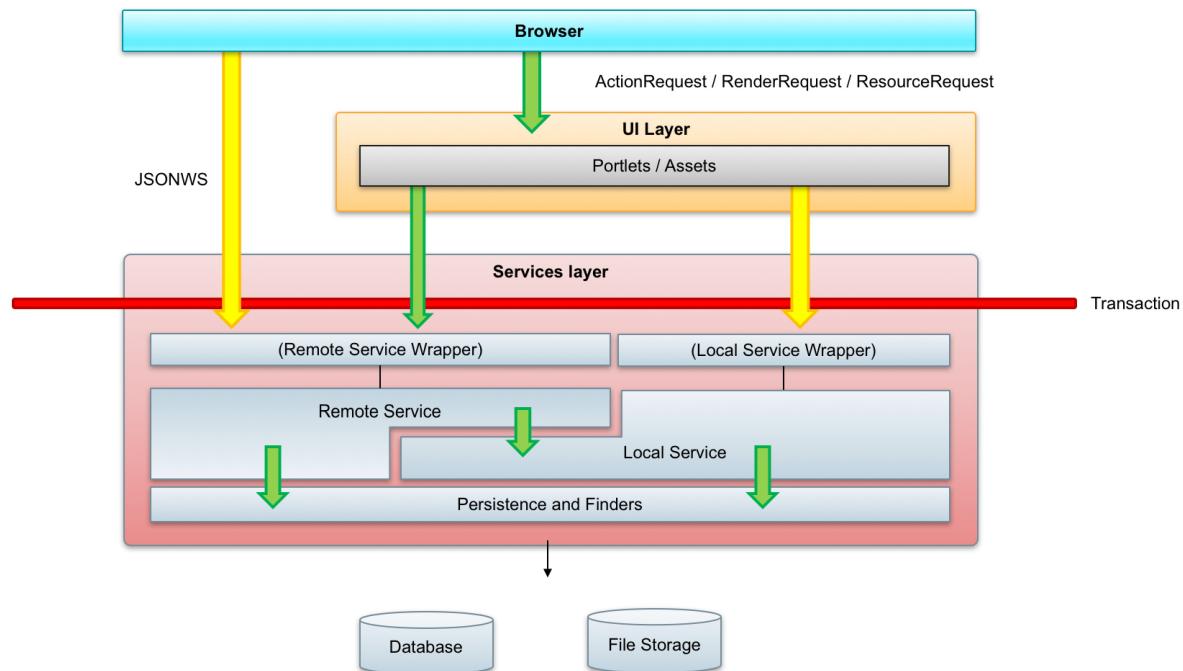


Figure: Service Builder architecture

Using Service Builder

A typical workflow with a Service Builder project:

1. Define model entities and finder queries in service.xml.
2. **Run build service task.**
3. Modify the model and service implementation classes.
4. **Run build service task.**
5. Modify finder implementation for the entity.
6. **Run build service task.**
7. ...

The `buildService` Gradle task must be run whenever service.xml is modified or model, service, or finder implementation class method signatures have been changed.

When you implement your own CRUD logic, the following pattern is recommended:

- `{entity}ServiceImpl.add(userId, groupId, {all entity fields}, {serviceContext});`
 - returns created `{entity}`
- `{entity}ServiceImpl.update({primaryKey}, {all entity fields}, {serviceContext})`
 - returns updated `{entity}`
- `{entity}ServiceImpl.delete({primaryKey});`
 - returns deleted `{entity}`

"Silencing" generated methods

Sometimes it's desirable to silence generated methods or method signatures. Below is an example of overriding a signature in an implementation class:

```
@Override
public class AssignmentLocalServiceImpl extends AssignmentLocalServiceBaseImpl {

    @Indexable(type = IndexableType.REINDEX)
    public Assignment addAssignment(long groupId, long userId, Map titleMap,
        String description, Date dueDate, ServiceContext serviceContext) throws PortalException {
        ..
    }

    public Assignment addAssignment(Assignment assignment) {
        throw new UnsupportedOperationException(
            "please use instead addAssignment(long groupId, long userId, Map titleMap, " +
            "String description, Date dueDate, ServiceContext)");
    }
    ..
}
```

Exercises

Using the Service Builder

Introduction

In this exercise, we will create the service layer for the Gradebook application using the Liferay Service Builder code generation tool.

Overview

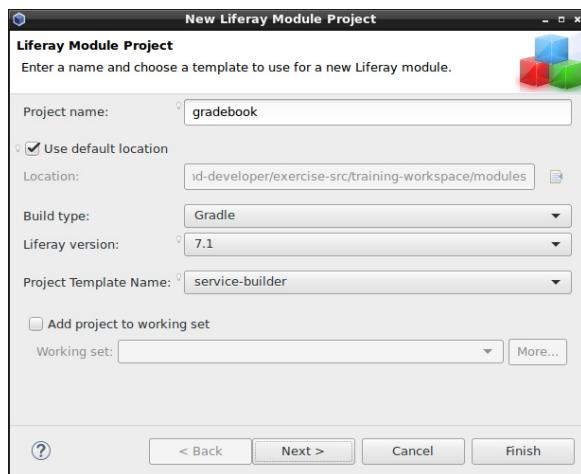
- 1 Create a Liferay module project using a service-builder template
- 2 Define Assignment entity
- 3 Define Submission entity
- 4 Define the relationship between Assignment and Submission
- 5 Define service exceptions
- 6 Build services

Snippets and resources for this exercise are in `snippets/chapter-06/02-create-the-service-layer` of your Liferay Workspace.

The complete Gradebook application **solution** is in `solutions/solutions-chapter-06/solution-06-gradebook`.

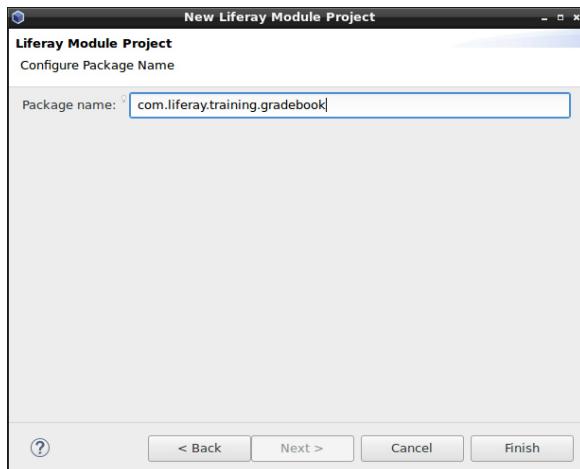
Create a Liferay Module Project Using the Service Builder Template

1. **Click** *New → Liferay Module Project* in Developer Studio.
2. **Type** *gradebook* in the *Project Name* field.
3. **Choose** *service-builder* in the *Project Template Name* field.
4. **Choose** *7.1* in the *Liferay Version* field if it isn't already selected.
5. **Click** *Next*.



6. **Type** *com.liferay.training.gradebook* in the *Package Name* field.

7. **Click** *Finish* to close the wizard.



o Next, let's make sure our dependencies are in order in the *gradebook-api* and *gradebook-service* bundles.

8. **Expand** the *gradebook-api* folder in the *gradebook* project.

9. **Open** the `build.gradle` file.

10. **Add** the following line to the content currently in the `build.gradle` *dependencies* section:

```
compileOnly group: "com.liferay", name: "com.liferay.osgi.util", version: "3.0.0"
```

11. **Expand** the *gradebook-service* folder.

12. **Open** the `build.gradle` file.

13. **Add** the *servlet-api*, *portlet-api*, and *osgi.util* dependencies to the *dependencies* section of the `build.gradle` file.

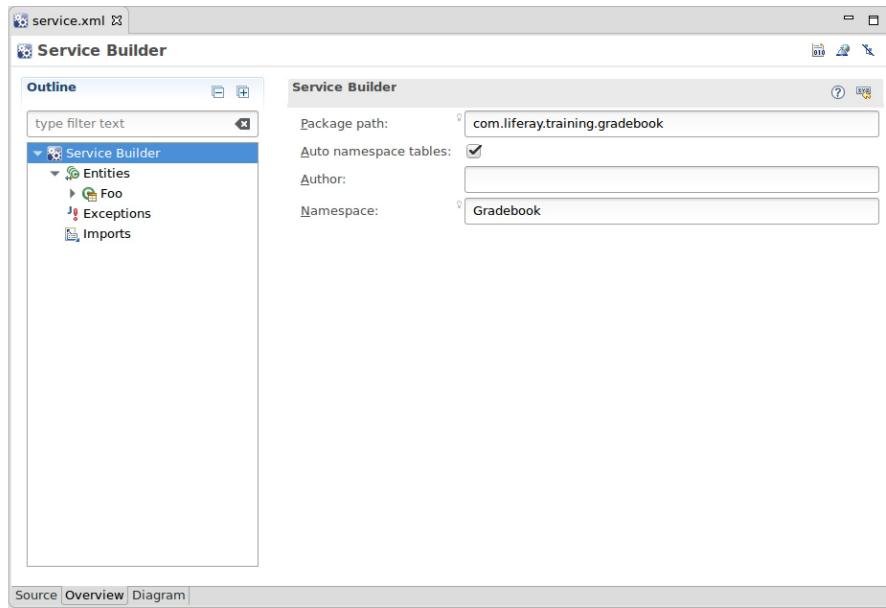
Afterwards the *dependencies* section should look like as follows:

```
dependencies {
    dependencies {
        compileOnly group: "biz.aQute.bnd", name: "biz.aQute.bndlib", version: "3.5.0"
        compileOnly group: "com.liferay", name: "com.liferay.osgi.util", version: "3.0.0"
        compileOnly group: "com.liferay", name: "com.liferay.portal.spring.extender", version: "2.0.0"
        compileOnly group: "com.liferay.portal", name: "com.liferay.portal.kernel", version: "3.0.0"
        compileOnly group: "javax.portlet", name: "portlet-api", version: "3.0.0"
        compileOnly group: "javax.servlet", name: "servlet-api", version: "2.5"
        compileOnly project(":modules:gradebook:gradebook-api")
    }
}
```

Define Assignment Entity

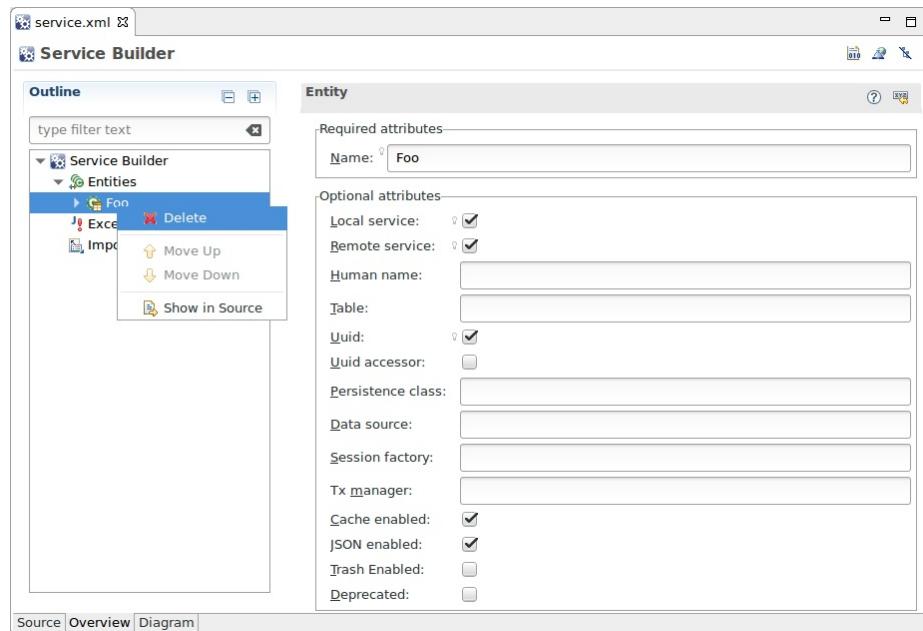
service.xml is the main configuration and definition file of Service Builder. It lets you define, for example, model entities, datasources, finder methods, and service variants to be generated for our service.

1. **Double-click** *service.xml* file in the *gradebook-service* folder.
2. **Click** the *Overview* tab.
3. **Click** *Service Builder* in the outline tree.
4. **Type** *Gradebook* in the *Namespace*, replacing the *FOO*.
 - o Namespace is used for prefixing the database tables the Service Builder generates for the service.



5. Click **Entities** in the outline tree.

6. Delete the default **Foo** entity.

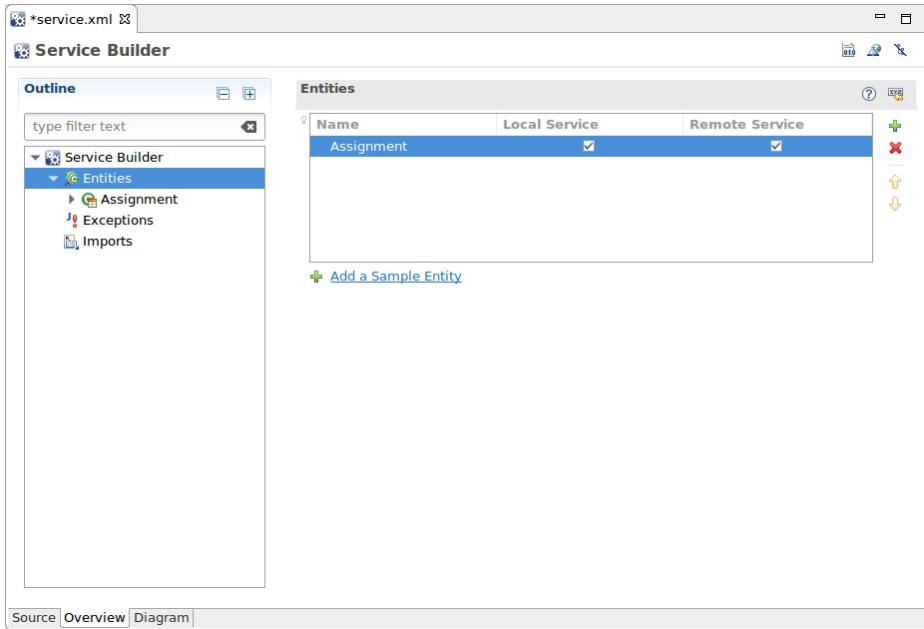


- Next, we will create a new *Assignment* entity.

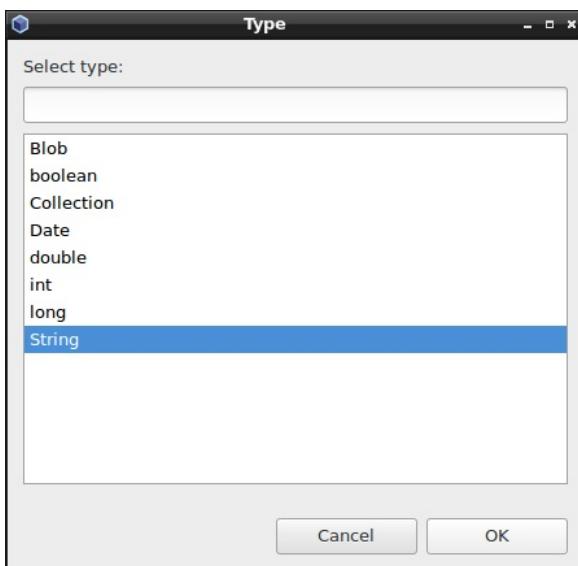
7. Click the green plus sign on the right side of the entities list to add a new entity.

8. Type *Assignment* in the **Name** field.

9. Check both **Local Service** and **Remote Service**.



- We'll also add the `uuid` attribute:
10. Click the *Assignment* entity on the outline tree to open entity properties.
 11. Check *Uuid*.
 - Let's define an entity's properties, which map to database model columns.
 12. Double-click on the *Assignment* entity in the outline tree to open entity properties.
 13. Click on the *Columns*.
 14. Click *Add Default Columns* to add a default set of entity properties.
 - The primary key for the entity, `assignmentId`, should have been created automatically using the *Add Default Columns* functionality. Let's add a few more columns:
 15. Click the green plus sign on the right side of the columns list to add a new column.
 16. Type *title*.
 17. Double-click the *Type* column.
 18. Click the browse icon on the right side of the field and select *String* type.
 19. Click *OK* to close the dialog.

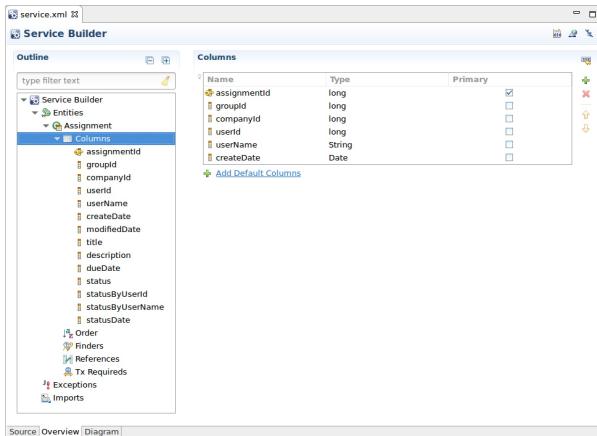


20. Repeat the process and add the rest of the fields for the Assignment entity as follows:

| Field | Type |
|------------------|--------|
| description | String |
| dueDate | Date |
| status | int |
| statusByUserId | long |
| statusByUsername | String |
| statusDate | Date |

The final list of fields, with their descriptions, should look like:

| Field | Type | Description |
|------------------|--------|--|
| assignmentId | long | primary key (<i>default</i>) |
| companyId | long | portal instance id (<i>default</i>) |
| groupId | long | scope id for site / staging / page scope (<i>default</i>) |
| userId | long | creator user id (<i>default</i>) |
| userName | String | creator user name (<i>default</i>) |
| createDate | Date | create date (<i>default</i>) |
| modifiedDate | Date | modified date (<i>default</i>) |
| title | String | Title of assignment |
| description | String | Assignment description |
| dueDate | Date | Due date of the Assignment |
| status | int | Status fields for workflow and recycle bin (<i>status field</i>) |
| statusByUserId | long | (<i>status field</i>) |
| statusByUsername | String | (<i>status field</i>) |
| statusDate | Date | (<i>status field</i>) |



We need to set the **title** field as **Localized**. This can be done in column properties.

1. **Double-click** the **Columns** field in the outline tree to expand it.
2. **Click** the **title** field in the outline tree to show the column properties.

3. Check Localized.

- o So far we have defined the assignment entity properties and service variant to be created. The next step is defining the finders. *Finders* are database query methods that are automatically generated in the service api.

4. Right-click Finders in the outline tree.

5. Choose Add Finder.

6. Type GroupId in the Name field.

7. Type Collection in the Return type field.

8. Double-click the just created *GroupId* in the outline tree.

9. Right-click *Finder Column* in the opened outline tree leaf.

10. Click *Add Finder Column*.

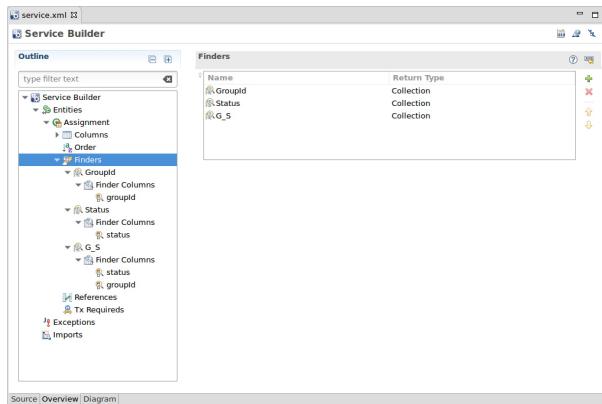
11. Type *groupId* in the *Name* field.

12. Add two more finders following the steps above.

- o Notice that there are two finder columns in *G_S*:

| Name | Columns | Return type | Description |
|--------|-------------------|-------------|---------------------------|
| Status | status | Collection | Finds by entity status |
| G_S | groupId status | Collection | Finds by group and status |

The list of finders should look like the list below:



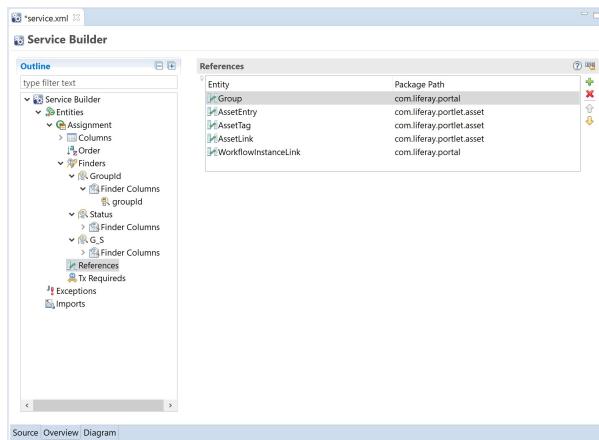
Service references are needed to make the persistence of any references model entities available in our service classes. That way, all model update operations done from within our service classes remain inside the same transaction boundary. We'll create references for group, asset entry, asset tag, and asset link.

1. **Click** *References* in the outline tree.
2. **Click** the green plus sign on the right side of the *References* list.
3. **Type** *Group* in the *Entity* field.
4. **Double-click** the *Package Path* column on the left side of *Group* to edit.
5. **Type** *com.liferay.portal* in the field.
6. **Repeat** the same for the following references:

| Entity | Package Path | Description |
|----------------------|---------------------------|---|
| AssetEntry | com.liferay.portlet.asset | Needed for the integration to the Assets framework |
| AssetTag | com.liferay.portlet.asset | Needed for the integration to the Assets framework |
| AssetLink | com.liferay.portlet.asset | Needed for the integration to the Assets framework |
| WorkflowInstanceLink | com.liferay.portal | Needed for the integration to the Workflows framework |

The final list of references should be:

| Entity | Package path | Description |
|----------------------|---------------------------|---|
| Group | com.liferay.portal | Make's Group Services and persistence available for Assignment service |
| AssetEntry | com.liferay.portlet.asset | Make's AssetEntry Services and persistence available for Assignment service |
| AssetTag | com.liferay.portlet.asset | Make's AssetTag Services and persistence available for Assignment service |
| AssetLink | com.liferay.portlet.asset | Make's AssetLink Services and persistence available for Assignment service |
| WorkflowInstanceLink | com.liferay.portal | Make's WorkflowInstanceLink Services and persistence available for Assignment service |



Define Submission Entity

Let's repeat the entity creation process for submissions.

1. **Click Entities** in the outline tree.
2. **Click** the green plus sign on the right side of the entities list to add a new entity.
3. **Type Submission** in the Name field.
4. **Check** both Local Service and Remote Service.
 - o Define next entity's columns:
5. **Double-click** on the *Submission* entity in the outline tree to open entity properties.
6. **Click** on the *Columns*.
7. **Click** Add Default Columns to add a default set of entity properties.

The primary key for the entity, *submissionId*, should have been created automatically using the *Add Default Columns* functionality.

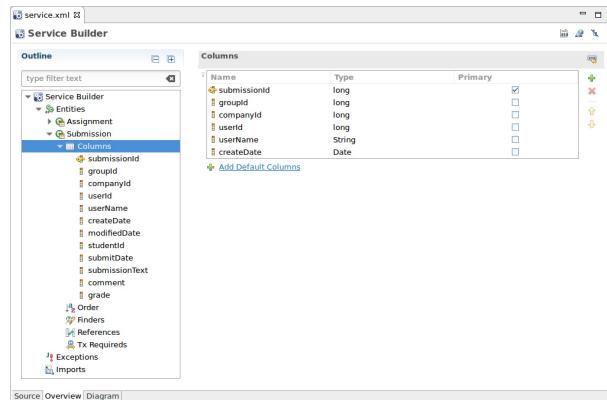
Now, add the following columns:

| Column | Type |
|----------------|--------|
| studentId | long |
| submitDate | Date |
| submissionText | String |
| comment | String |

| | |
|-------|-----|
| grade | int |
|-------|-----|

The final list of the columns for submissions should be:

| Column | Type | Description |
|----------------|--------|---|
| submissionId | long | primary key (<i>default</i>) |
| companyId | long | portal instance id (<i>default</i>) |
| groupId | long | scope id for site / staging / page scope (<i>default</i>) |
| userId | long | creator user id (<i>default</i>) |
| userName | String | creator user name (<i>default</i>) |
| createDate | Date | create date (<i>default</i>) |
| modifiedDate | Date | modified date (<i>default</i>) |
| studentId | long | User id of student |
| submitDate | Date | Student assignment submission date |
| submissionText | String | Student assignment submission text |
| comment | String | Comments for assignment |
| grade | int | Grade of the assignment |



Now follow the process of creating finders for Assignment entity and create the following finders for Submission. You may notice there a reference to assignmentId. We'll create that later:

| Name | Columns | Return type | Description |
|-----------------------|---------------------------|-------------|-------------------------------------|
| GroupId | groupId | Collection | Finds by groupID |
| G_A | groupId assignmentId | Collection | Finds by groupId and assignmentId |
| StudentId | studentId | Collection | Finds by studentId |
| StudentIdAssignmentId | studentId assignmentId | Collection | Finds by studentId and assignmentId |

Next, follow the process of creating references for Assignment entity and create the following references for Submission:

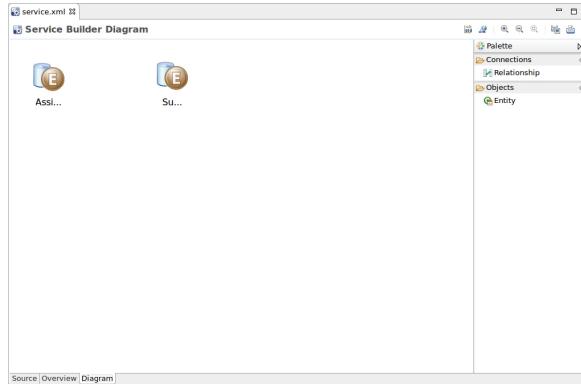
| Entity | Package path | Description |
|--------|--------------|-------------|
| | | |

| | | |
|------------|---------------------------|---|
| AssetEntry | com.liferay.portlet.asset | Make's AssetEntry Services and persistence available for the Assignment service |
| AssetTag | com.liferay.portlet.asset | Make's AssetTag Services and persistence available for the Assignment service |

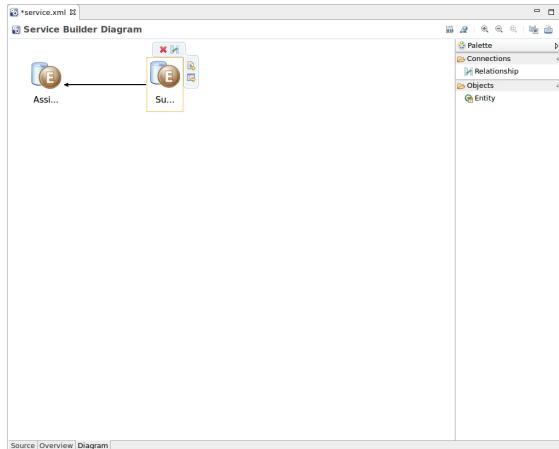
Define the Relationship Between Assignment and Submission

A submission is dependent on an assignment, so we have to add a foreign key of an assignment entity to a submission entity.

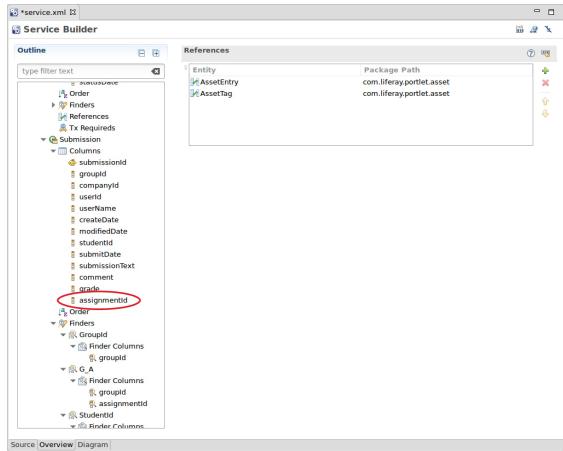
1. Click the *Diagram* tab on the main panel:



2. Click on the *Submission* entity to display the available options.
3. Click the *Connect* icon on the top right corner of the entity
 - o You can hover over the icons to see tooltips.
4. Click on the *Assignment* entity to connect the entities.
 - o The diagram view should now look like:



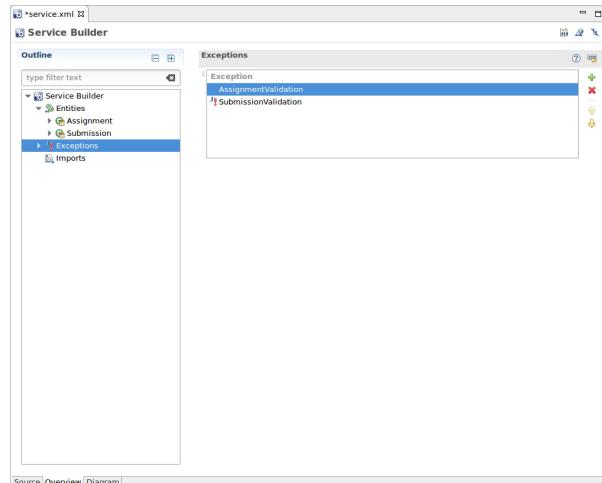
5. Click the *Overview* tab to check the *Submission* entity columns.
 - o There should be a column *assignmentId*, which is the foreign key:



Define Service Exceptions

You can define your custom service exceptions in service.xml. For the Gradebook application, we will define two exceptions for validation purposes.

1. **Click** *Exceptions* in the outline tree (on the bottom of the list).
2. **Click** the green plus sign on the right side of Exceptions list.
3. **Type** AssignmentValidation and hit *Enter*.
4. **Click** the green plus sign again.
5. **Type** SubmissionValidation and hit *Enter*.
6. **Save** the file.



Build Services

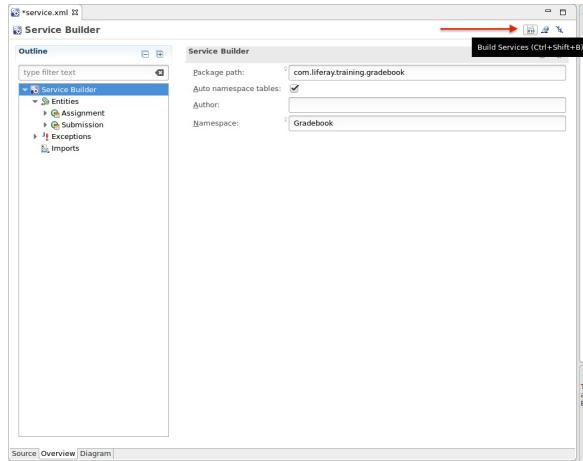
Now it's time to generate the service. When you run the build service task, the following things are generated:

- Database schema (changes to the database done at module deploy time)
- Persistence and caching
- Local and remote service APIs

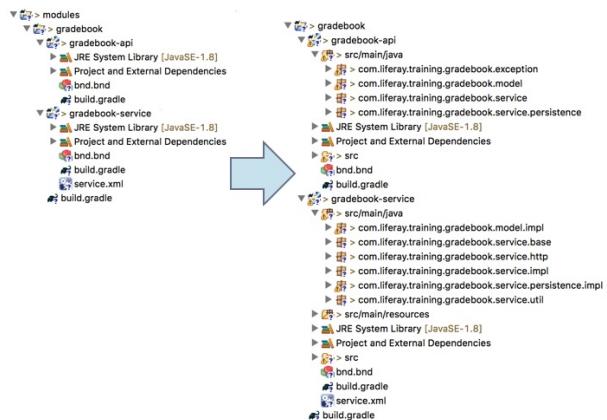
You can run the service generation task from several places in the IDE, like from the Gradle task panel, context menu of the service project, or from the command line using the Gradle wrapper. Here, we are running the task straight from the designer panel.

1. **Click** the *Build the Services* icon on the top right corner of the designer panel.

- You can also run the Service Builder by using the `buildService` task in the Gradle Tasks menu. You can find this task under `training-workspace-[version]/modules/gradebook/build`
2. Right-click on the `training-workspace-[version]` top level folder in the Project Explorer to open a content menu.
 3. Click on `Gradle → Refresh Gradle Project` to update the workspace.



The services have been generated. If you do not see them make sure to refresh the training-workspace:



Exercises

Implement AssignmentLocalServiceImpl

Introduction

In this exercise, we'll be implementing custom logic for local services.

Overview

- 1 Implement custom logic for local services

Snippets and resources for this exercise are in `snippets/chapter-06/02-create-the-service-layer` of your Liferay Workspace.

The complete Gradebook application **solution** is in `solutions/solutions-chapter-06/solution-06-gradebook`.

Implement Custom Logic for Local Services

When services are generated, an implementation class is created for every entity defined in the service.xml. You can not only implement your custom logic but also override the generated code there.

Next, we are going to add our custom logic in the service's implementation classes:

- AssignmentLocalServiceImpl
- AssignmentServiceImpl

Preparations

1. **Open** the exercise snippets folder.
2. **Copy** the contents of the `AssignmentLocalServiceImpl.txt` snippet.
3. **Paste** the contents into the `com.liferay.training.gradebook.service.impl.AssignmentLocalServiceImpl` class, replacing its contents.

Implement AssignmentLocalServiceImpl

The purpose of the Assignment Local Service is to store and find information from Assignments. In this exercise step, we create new method signatures for adding, updating, and deleting assignments. The snippet also demonstrates how any unwanted, generated methods can be "silenced".

1. **Open** `com.liferay.training.gradebook.service.impl.AssignmentLocalServiceImpl`
 - o Follow the numbered tasks and their instructions in the comments of class source code to do the exercise.
2. **Save** the file after making changes.

By the end of the exercise the AssignmentLocalServiceImpl should look as follows:

```
public class AssignmentLocalServiceImpl extends AssignmentLocalServiceBaseImpl {
    public Assignment addAssignment(long groupId, Map<Locale, String> titleMap, String description, Date dueDate,
        ServiceContext serviceContext) throws PortalException {
        //
```

```

// ( 1 ) - Get group.
//

Group group = groupPersistence.findByPrimaryKey(groupId);

//
// ( 2 ) - Get user.
//

long userId = serviceContext.getUserId();

User user = userLocalService.getUserById(userId);

//
// ( 3 ) - Generate primary key for the new assignment.
//

long assignmentId = counterLocalService.increment(Assignment.class.getName());

//
// ( 4 ) - Create a new assignment object.
//

Assignment assignment = createAssignment(assignmentId);

assignment.setCompanyId(group.getCompanyId());
assignment.setGroupId(groupId);
assignment.setUserId(userId);
assignment.setTitleMap(titleMap);
assignment.setDueDate(dueDate);
assignment.setDescription(description);
assignment.setUserName(user.getScreenName());

assignment.setCreateDate(serviceContext.getCreateDate(new Date()));
assignment.setModifiedDate(serviceContext.getModifiedDate(new Date()));

//
// ( 5 ) - Persist the assignment.
//

assignment = super.addAssignment(assignment);

//
// ( 6 ) Return the created Assignment.
//

return assignment;
}

/**
 * Notice: we can "silence" generated, unwanted signatures simply by making an override.
 */
@Override
public Assignment addAssignment(Assignment assignment) {
    throw new UnsupportedOperationException("Not supported.");
}

public Assignment deleteAssignment(long assignmentId) throws PortalException {

    Assignment assignment = getAssignment(assignmentId);

    return deleteAssignment(assignment);
}

public Assignment deleteAssignment(Assignment assignment) {

//
// ( 7 ) - Delete assignment.
//
}

```

```

        return super.deleteAssignment(assignment);
    }

    public List<Assignment> getAssignmentsByGroupId(long groupId) {
        //
        // ( 8 ) - Get assignments by groupId.
        //

        return assignmentPersistence.find(groupId);
    }

    public List<Assignment> getAssignmentsByGroupId(
        long groupId, int start, int end) {

        //
        // ( 9 ) - Get assignments by groupId.
        //

        return assignmentPersistence.find(groupId, start, end);
    }

    public int getAssignmentsCountByGroupId(long groupId) {
        //
        // ( 10 ) - Get assignments by groupId.
        //

        return assignmentPersistence.count(groupId);
    }

    public Assignment updateAssignment(long assignmentId, Map<Locale, String> titleMap, String description,
        Date dueDate, ServiceContext serviceContext) throws PortalException {
        //
        // ( 11 ) - Get Assignment to be updated
        //

        Assignment assignment = getAssignment(assignmentId);

        // Update the changes to assignment

        assignment.setTitleMap(titleMap);
        assignment.setDueDate(dueDate);
        assignment.setDescription(description);
        assignment.setModifiedDate(new Date());

        // ( 12 ) - Persist the assignment

        assignment = super.updateAssignment(assignment);

        // ( 13 ) - Return the updated Assignment

        return assignment;
    }

    /**
     * We can "silence" generated, unwanted signatures simply by making an override.
     */
    @Override
    public Assignment updateAssignment(Assignment assignment) {
        throw new UnsupportedOperationException(
            "Not supported.");
    }
}

```


Exercises

Implement SubmissionLocalServiceImpl

Introduction

In this exercise, we will implement the logic for SubmissionLocalService.

Overview

- 1 Implement custom logic for local services

Snippets and resources for this exercise are in `snippets/chapter-06/02-create-the-service-layer` of your Liferay Workspace.

The complete Gradebook application **solution** is in `solutions/solutions-chapter-06/solution-06-gradebook`.

Preparations

1. **Open** the exercise snippets folder.
2. **Copy** the contents of the `SubmissionLocalServiceImpl.txt` snippet.
3. **Paste** the contents into the `com.liferay.training.gradebook.service.impl.SubmissionLocalServiceImpl` class, replacing its contents.

Implement SubmissionLocalServiceImpl

The purpose of the Submission Local Service is to store and find submissions for assignments and also provide the api to the teacher to grade and comment them.

1. **Open** `com.liferay.training.gradebook.service.impl.SubmissionLocalServiceImpl`
 - o Follow the numbered and the instructions in the comments of class source code to do the exercise.
2. **Save** the file after making changes.
 - o You may see errors in the code. Re-building services and performing a gradle refresh should remove the issues.

By the end of the exercise the `SubmissionLocalServiceImpl` should look as follows:

```
public class SubmissionLocalServiceImpl extends SubmissionLocalServiceBaseImpl {

    @Override
    public Submission addSubmission(long assignmentId, long studentId, String submissionText,
                                    ServiceContext serviceContext) throws PortalException {

        //
        // ( 1 ) - Get Assignment.
        //

        Assignment assignment = assignmentLocalService.getAssignment(assignmentId);

        //
        // ( 2 ) - Get user.
        //

        // Even though we will be not using the user object in this method, fetching it validates
        // its existence.
    }
}
```

```

//  

long userId = serviceContext.getUserId();  

User user = userLocalService.getUser(userId);  

//  

// ( 3 ) - Get student user (studentId).  

//  

// Even though we will be not using the user object in this method, fetching it validates  

// its existence.  

//  

User studentUser = userLocalService.getUser(studentId);  

//  

// ( 4 ) - Generate submission id  

//  

long submissionId = counterLocalService.increment(Submission.class.getName());  

//  

// ( 5 ) - Create Submission  

//  

Submission submission = submissionLocalService.createSubmission(submissionId);  

// Populate submission fields  

submission.setSubmissionId(submissionId);  

submission.setAssignmentId(assignmentId);  

submission.setCompanyId(assignment.getCompanyId());  

submission.setGroupId(assignment.getGroupId());  

submission.setCreateDate(new Date());  

submission.setModifiedDate(new Date());  

submission.setUserId(userId);  

submission.setGrade(-1);  

submission.setStudentId(studentId);  

submission.setSubmissionText(submissionText);  

submission.setSubmitDate(new Date());  

//  

// ( 6 ) - Persist the entity.  

//  

submission = super.addSubmission(submission);  

//  

// ( 7 ) - Return the updated Submission.  

//  

return submission;  

}  

public List<Submission> getSubmissionsByAssignment(long groupId, long assignmentId) {  

//  

// ( 8 ) - Get submissions by groupId.  

//  

return submissionPersistence.findByG_A(groupId, assignmentId);  

}  

public List<Submission> getSubmissionsByAssignment(long groupId, long assignmentId, int start, int end) {  

//  

// ( 9 ) - Get submissions by groupId and assignmentId.  

//
```

```

        return submissionPersistence.findByG_A(groupId, assignmentId, start, end);
    }

    public int getSubmissionsCountByAssignment(long groupId, long assignmentId) {
        //
        // ( 10 ) - Get count by groupId and assignmentId.
        //

        return submissionPersistence.countByG_A(groupId, assignmentId);
    }

    public Submission gradeAndCommentSubmission(long submissionId, int grade, String comment) throws PortalException {
        //
        // ( 11 ) - Get Submission.
        //

        Submission submission = this.getSubmission(submissionId);

        //
        // ( 12 ) - Update following fields: grade, comment, modifiedDate.
        //
        submission.setGrade(grade);
        submission.setComment(comment);
        submission.setModifiedDate(new Date());

        //
        // ( 13 ) - Return updated submission.
        //

        return super.updateSubmission(submission);
    }

    public Submission gradeSubmission(long submissionId, int grade) throws PortalException {
        //
        // ( 14 ) - Get Submission.
        //

        Submission submission = this.getSubmission(submissionId);

        //
        // ( 15 ) - Update following fields: grade, modifiedDate.
        //
        submission.setGrade(grade);
        submission.setModifiedDate(new Date());

        //
        // ( 16 ) - Return updated submission.
        //

        return super.updateSubmission(submission);
    }

    @Override
    public Submission updateSubmission(long submissionId, String submissionText, ServiceContext serviceContext)
        throws PortalException {

        //
        // ( 17 ) - Get Submission.
        //

        Submission submission = this.getSubmission(submissionId);
    }
}

```

```
//  
// ( 18 ) - Update following fields: submissionText, submitDate, modifiedDate.  
//  
  
submission.setSubmissionText(submissionText);  
submission.setSubmitDate(new Date());  
submission.setModifiedDate(new Date());  
  
//  
// ( 19 ) - Persist the entity.  
//  
  
submission = super.updateSubmission(submission);  
  
//  
// ( 20 ) - Return updated submission.  
//  
  
return submission;  
}  
}
```

Rerun the Service Builder Task

After you modify either the service.xml or implementation classes, the service has to be regenerated.

1. **Right-click** on the *gradebook* project.
2. **Choose** *Liferay → build-service*.
3. **Right-click** on the *gradebook* project again.
4. **Choose** *Gradle → Refresh Gradle Project*.

Exercises

Implement and Test Remote Services

Introduction

In this exercise, we'll be implementing and testing remote services.

Overview

- ① Implement custom logic for remote services
- ② Deploy the Service
- ③ Test the service through remote API

Snippets and resources for this exercise are in `snippets/chapter-06/02-create-the-service-layer` of your Liferay Workspace.

The complete Gradebook application **solution** is in `solutions/solutions-chapter-06/solution-06-gradebook`.

Implement Custom Logic for Remote Services

The purpose of the Remote Service layer is, on one hand, to provide a façade layer with permission checks for the local service layer and, on the other hand, to provide remote interfaces for JSON web services and SOAP. We will add permission checks and discuss remote services more in detail later. Right now, we'll add some methods to the remote APIs in order to test the service through the JSON web service and see if that works.

1. **Open** the exercise snippets folder.
2. **Copy** the contents of the `AssignmentServiceImpl.txt` snippet.
3. **Paste** the contents into the `com.liferay.training.gradebook.service.impl.AssignmentServiceImpl` class, replacing its contents.
4. **Copy** the contents of the `SubmissionServiceImpl.txt` snippet.
5. **Paste** the contents into the `com.liferay.training.gradebook.service.impl.SubmissionServiceImpl` class, replacing its contents.
6. **Right-click** on the `gradebook` project.
7. **Choose** `Liferay → build-service`.

Deploy the Service

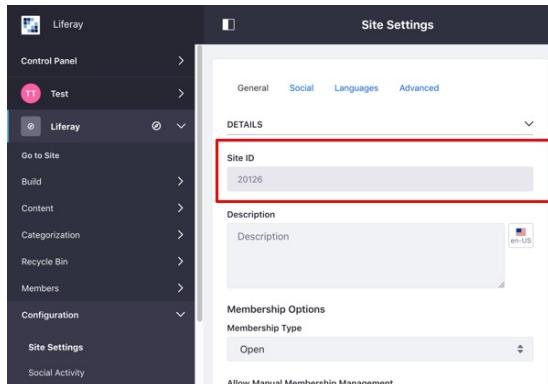
1. **Drag** the `gradebook-api` and `gradebook-service` onto the Liferay server to deploy the modules.



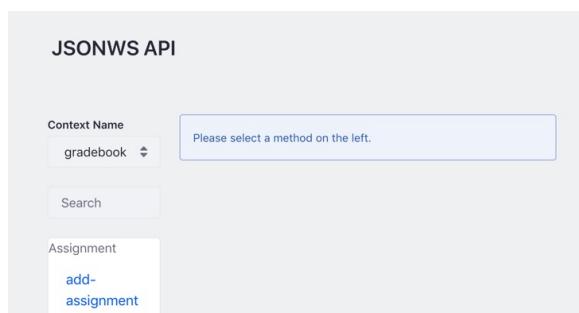
Test the Services Through Remote API

We'll use the JSON web service test page to test our service. Before starting, we have to find an ID for the site where we are going to create our test assignments.

1. [Login](#) to Liferay in your web browser.
2. [Open](#) the *Menu*.
3. [Expand](#) the *Site Administration* panel.
4. [Click](#) on the *Configuration → Site Settings*.
5. [Find](#) Site ID.
 - o This is required for the next step.

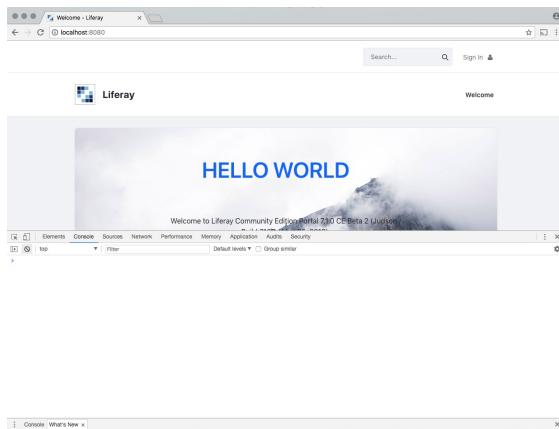


6. [Open](#) a new tab in your browser and navigate to <http://localhost:8080/api/jsonws>.
7. [Choose](#) *Gradebook* in the *Context Name* menu.



- o This is the test page for our Gradebook service. On the menu, you see a list of methods we just added to our remote service classes.
- o We'll now test our service with a browser's JavaScript console.

- You can find the example scripts in `solutions/solutions-chapter-06/solution-06-gradebook/snippets/create-service-layer/json`.
 - Make sure you are still logged into the platform.
8. Open the JavaScript console of your favorite browser.
- In most browsers, you can open the Javascript console by pressing `ctrl+Shift+J` (Windows / Linux) or `Cmd+Opt+J` (OSX).



9. Copy the contents of `add-an-assignment.json` found in the `solutions/solutions-chapter-06/solution-06-gradebook/snippets/02-create-the-service-layer/json` folder.
10. Paste the script into the console and press Enter.

- You should get an assignment as JSON as response.

```
> Liferay.Service('/gradebook.assignment/add-assignment', {
  groupId: 20126,
  title: { 'en_US': 'How to make birthday cake'},
  description: 'Design most delicious and beautiful birthday cake.',
  dueDate: (new Date('2018-05-22')).getTime()
}, function(obj) {
  console.log(obj);
});
<- > {readyState: 1, getResponseHeader: f, getAllResponseHeaders: f, setRequestHeader: f, onreadystatechange: f, status: 200, statusText: "OK", data: 
  ▼ {assignmentId: "101", companyId: "20099", createDate: 1529966843319, des
    ↴ e: 1526947200000, ...} ⓘ
    ▼ assignmentId: "101"
    companyId: "20099"
    createDate: 1529966843319
    description: "Design most delicious and beautiful birthday cake."
    dueDate: 1526947200000
    groupId: "20126"
    modifiedDate: 1529966843319
    status: 0
    statusByUserId: "0"
    statusByUserName: ""
    statusDate: null
    title: "<?xml version='1.0' encoding='UTF-8'?><root available-locales= titleCurrentValue: "How to make birthday cake"
    ▼ userId: "20139" ⓘ
    userName: "test"
    uuid: "40e0b07d-5ff4-dbbe-092d-b4b72e0caeba"
  ▶ __proto__: Object
}
```

11. Find the `assignmentId` and `userId` in the JSON response.
- This is required for the next step.
12. Click `get-assignment` on the JSONWS API test page's menu.
13. Type the `assignmentId` you received in the JSON output from the first step.
14. Click `Invoke`.

- You should see the assignment you created using the example script from the first step.
- Let's now add a submission.

15. **Copy** the contents of `add-a-submission.json` found in the `..snippets/02-create-the-service-layer/json` folder.
16. **Paste** the the script into the console.
17. **Replace** the `studentId` with the `userId` from the JSON response.
18. **Replace** the `assignmentId` with `assignmentId` from JSON response.
19. **Press Enter** to run the script.

- You should get a submission as a JSON as response.

```
> Liferay.Service('/gradebook.submission/add-submission', {
    assignmentId: 101,
    studentId: 20139,
    submissionText: 'Cheesecake with strawberries and blueberries.'
}, function(obj) {
    console.log(obj);
});
<- ▶ {readyState: 1, getResponseHeader: f, getAllResponseHeaders: f, setRequestHeader: f, open: f, send: f}
▼ {assignmentId: "101", comment: "", companyId: "20099", createDate: 1529967358047,
    assignmentId: "101"
    comment: ""
    companyId: "20099"
    createDate: 1529967358047
    grade: -1
    groupId: "20126"
    modifiedDate: 1529967358047
    studentId: "20139"
    submissionId: "1"
    submissionText: "Cheesecake with strawberries and blueberries."
    submitDate: 1529967358047
    userId: "20139"
    userName: ""
    ► __proto__: Object
```

20. **Find** `assignmentId` and `groupId` in the JSON response.
 - This is required for the next step.
21. **Click** `get-submissions-by-assignment` on the test page's menu.
 - There are two methods with this name, make sure to select the one that does not require a start and end integer.
22. **Type** `assignmentId` from the JSON response in the `assignmentId` field.
23. **Type** `groupId` from the JSON response in the `groupId` field.
24. **Click** on the `Invoke`.

You should get a list of submissions to the assignment with specified id as a response.

Implement Access Control

Introduction

Liferay has a robust security model that allows for the configuration of fine-grained access control. The access control system lets you define who can use an application and who is allowed to add and edit a model resource. The access control system also makes sure that these restrictions are checked for every request.

For example, all applications dealing with user and site management are only accessible for Liferay Administrators and not for regular users. Wiki Pages can be created and edited by any member of a site while Blogs posts by default can only be edited by their original author.

The access control system is used to manage access to all of Liferay's resources and can also be used to provide fine-grained access control management for your custom applications.

Access control in Liferay is based on four key concepts:

1. Resources
2. Actions
3. Permissions
4. Roles

We will explore each of them in greater detail in the following sections.

Resources

A resource in Liferay's permission framework is a generic representation of any application or model entity that can be used as an action target.

There are two distinct kinds of resources:

- Portlet resources
- Model resources

Portlet Resources

Portlet resources represent portlet applications. Examples of portlet resources are:

- The Blogs Portlet
- The Wiki Portlet

Portlet resources are identified by the portlet ID, which is either defined in the portlet component's properties with the property `javax.portlet.name` or in the portlet's `portlet.xml` (for legacy or strictly JSR-286 compliant portlets).

When using portlet components, the portlet ID is configured with the `javax.portlet.name` property of the component's properties, as shown in the code snippet taken from the native Blogs application's portlet class:

Blogs Portlet Class

```
@Component(  
    immediate = true,  
    property = {  
        ...  
        "javax.portlet.name=" + BlogsPortletKeys.BLOGS,  
        ...  
    },  
    service = Portlet.class
```

```

    }
    public class BlogsPortlet extends BaseBlogsPortlet {
    ...
}
```

A best practice is to define the portlet's ID as a constant in a dedicated keys class in order to avoid ambiguities and misspellings.

By convention, a Portlet Component's portlet ID is the component's fully qualified classname with the dots replaced by underscores, as can be seen in the code snippet taken from Blogs Portlet's Portlet Keys:

BlogsPortletKeys

```

public class BlogsPortletKeys {

    public static final String BLOGS =
        "com_liferay_blogs_web_portlet_BlogsPortlet";

    ...
}
```

Model Resources

A **Model Resource** represents a model entity managed by a portlet application. Examples of model resources include:

- A wiki page
- A document
- A site

Model resources are usually referenced by the entity's fully qualified class name, for example:

- **Web Content:** com.liferay.journal.model.JournalArticle
- **BlogsEntry:** com.liferay.blogs.model.BlogsEntry

Actions

Actions are operations that can be performed by a Liferay user on a given resource, either on a portlet or a model resource.

Examples for portlet resource actions are:

- ADD_TO_PAGE
- CONFIGURATION
- VIEW

ADD_TO_PAGE gives the user the ability to add a portlet to a page, CONFIGURATION lets a user access a portlet's configuration menu, and VIEW refers to accessing the portlet.

Examples for resource actions are:

- ADD_ENTRY
- UPDATE
- DELETE
- PERMISSIONS
- VIEW

ADD_ENTRY lets a user add an entry to the database, for example, a BlogsEntry or a BookmarksEntry. UPDATE lets a user update an entry and DELETE lets a user delete an entry. PERMISSIONS refers to accessing and editing an entry's permission settings, and VIEW refers to displaying a single entry.

The above examples are taken from Liferay's core applications, but Liferay's flexible permission control mechanism can also be used to define custom actions for your own applications.

Action Types

There are two types of actions:

- **Top-level actions:** Top-level actions are general model actions that can't be applied to an existing resource, for example, ADD_ENTRY.
 - Example: `com.liferay.blogs`
- **Resource actions:** Resource actions are applied to an existing resource, for example, DELETE.
 - Example: `com.liferay.blogs.model.BlogsEntry`

By convention, the top-level actions are referenced by the package name of the respective model's package, e.g., `com.liferay.blogs` for the top-level actions of the `com.liferay.blogs.model.BlogsEntry` model resource.

Permissions

Permissions

A permission is a combination of a *resource* and an *action* or, more formally: a permission is an *action* that can be performed on a *resource*. The permission to update a Blogs post, for example, is defined by the combination of the BlogsEntry's resource identifier and the UPDATE action:

- `com.liferay.blogs.model.BlogsEntry + UPDATE`

Permission Scopes

Permissions can be defined in four different scopes.

1. Company
2. Group
3. Group Template
4. Individual

The scope of a permission determines how broadly it applies to the resources in the portal. *Company* is the broadest, and grants a user permissions for every resource of the type within the company (= portal instance). Likewise, *Group* scope gives users permissions for every resource within a specified group, and *Individual* applies only to a single resource of the type. *Group Template* is similar to Group scope, except that it does not automatically apply to a specific group. A user must be a member of a group (generally either a site or an organization), and they must have been given the role within that group before they are granted its permissions.

<https://docs.liferay.com/ce/portal/7.1-latest/javadocs/portal-impl/com/liferay/portal/model/impl/ResourcePermissionImpl.html>

Roles

Roles are entities that bind all the previously mentioned concepts to a user or a user group.

A Role

Roles are collections of permissions that give users access to different parts of the platform and let them perform certain tasks.

Roles can be assigned to:

- Users
- Sites
- Organizations
- User groups

Roles are always defined in one of the following scopes:

- Global
- Site or organization
- Within a site or organization (Team)

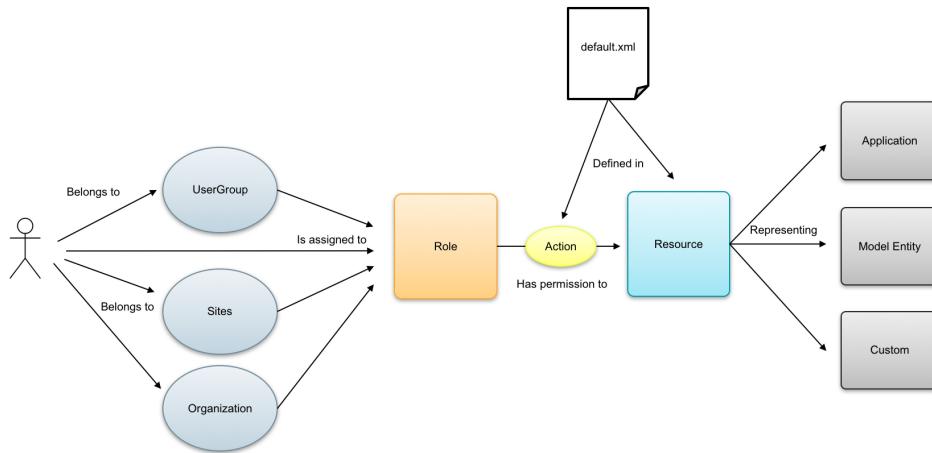
Roles defined for the *Global* scope apply to the entire portal instance. The most prominent global role is the (portal) Administrator role, which gives all its members control over every resource of the portal instance.

Roles defined with the *Site or Organization* scope apply only within the respective site or organization. Examples for Site or Organization roles are the *Site Administrator* or *Organization Administrator* role, which give their members administrative privileges for a certain site or organization.

Permissions are **always granted to roles**, not to users directly.

Permissioning Concepts Summary

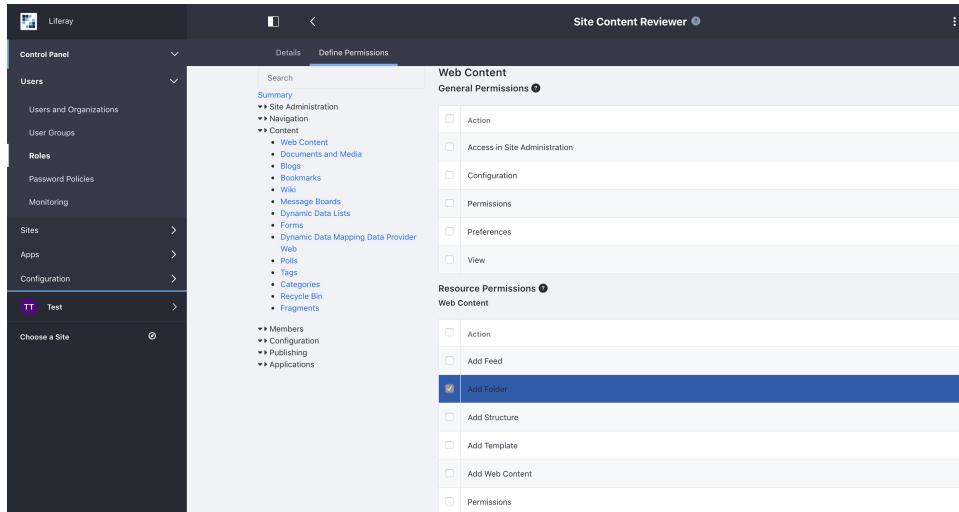
Liferay's access control system can be summarized as follows:



- Permissions are actions that can be performed on a resource.
- Resources represent either an application or a model entity.
- A *Role* collects *permissions* that define a particular function within Liferay.
- Roles are assigned to users either directly or through a site, an organization, or an user group.
- Permissions are inherited from all of a user's roles, so a Liferay user has all permissions of all roles he belongs to.

Managing Permissions

Managing permissions happens in the *Control-Panel -> Roles*.



How to Implement Permissioning

Where and how you define and implement permissioning depends on your application structure. If you are using Service Builder to create your custom entities, you usually define model permissions in the service module. Portlet permissions are defined independently in the portlet, typically the "web" module. Regardless of the case, the steps are the same:

- ① Define the path to the file that defines the resources and permissions. This is usually done in *portlet.properties*.
- ② Define resources and permissions. This is usually done in a file called *default.xml*.
- ③ Manage permission resources creation and deletion. On the service layer, this is usually done in the CRUD methods.
- ④ Implement permission checker class if necessary.
- ⑤ Implement permissions checking where ever necessary.

Step 1 - Define the Permissions and Resources Definition File

Create a file called *portlet.properties* in the *src/main/resources* folder of your module project. Add the property *resource.actions.configs* that defines the path to the definition file.

Below is an example of a portlet properties file:

```
#  
# Input a list of comma delimited resource action configurations that will be  
# read from the class path.  
#  
resource.actions.configs=resource-actions/default.xml
```

Step 2 - Define Resources and Their Permissions

Now you have to implement the actual file that defines module resources and the available permissions. Following the *portlet.properties* example above, this file would be *src/main/resources/resource-actions/default.xml*

Below is an excerpt of a native Blogs application default.xml file, located in the blogs-web module. Notice that there are only *portlet-resource* definitions:

```
<?xml version="1.0"?>
<!DOCTYPE resource-action-mapping PUBLIC
"-//Liferay//DTD Resource Action Mapping 7.0.0//EN" "http://www.liferay.com/dtd/liferay-resource-action
-mapping_7_0_0.dtd">
<resource-action-mapping>
...
<portlet-resource>
<portlet-name>com_liferay_blogs_web_portlet_BlogsPortlet</portlet-name>
<permissions>
<supports>
<action-key>ADD_PORTLET_DISPLAY_TEMPLATE</action-key>
<action-key>ADD_TO_PAGE</action-key>
<action-key>CONFIGURATION</action-key>
<action-key>VIEW</action-key>
</supports>
<site-member-defaults>
<action-key>VIEW</action-key>
</site-member-defaults>
<guest-defaults>
<action-key>VIEW</action-key>
</guest-defaults>
<guest-unsupported>
<action-key>ADD_PORTLET_DISPLAY_TEMPLATE</action-key>
<action-key>CONFIGURATION</action-key>
</guest-unsupported>
</supports>
</permissions>
</portlet-resource>
</resource-action-mapping>
```

Source: <https://github.com/liferay/liferay-portal/blob/7.1.0-ga1/modules/apps/blogs/blogs-web/src/main/resources/resource-actions/default.xml>

The next example is from the native Blogs application's service module blogs-service that defines blog entities permissions. Take note of two things in particular: first, *model-resource* tags instead of *portlet-resource*. Second, notice that model resources that have the package name *com.liferay.blogs* are *top-level actions*, which are applied to generic model actions, while as resources, they have the full class path *com.liferay.blogs.model.BlogsEntry* and are called *resource actions*, applied to existing entities:

```
<?xml version="1.0"?>
<!DOCTYPE resource-action-mapping PUBLIC "-//Liferay//DTD Resource Action Mapping 7.0.0//EN" "http://www.li
feray.com/dtd/liferay-resource-action-mapping_7_0_0.dtd">
<resource-action-mapping>
<model-resource>
<model-name>com.liferay.blogs</model-name>
<portlet-ref>
<portlet-name>com_liferay_blogs_web_portlet_BlogsAdminPortlet</portlet-name>
<portlet-name>com_liferay_blogs_web_portlet_BlogsPortlet</portlet-name>
</portlet-ref>
<root>true</root>
<weight>1</weight>
<permissions>
<supports>
<action-key>ADD_ENTRY</action-key>
<action-key>PERMISSIONS</action-key>
<action-key>SUBSCRIBE</action-key>
</supports>
<site-member-defaults>
<action-key>SUBSCRIBE</action-key>
</site-member-defaults>
<guest-defaults />
<guest-unsupported>
```

```

<action-key>ADD_ENTRY</action-key>
...
</guest-supported>
</permissions>
</model-resource>
<model-resource>
<model-name>com.liferay.blogs.model.BlogsEntry</model-name>
<portlet-ref>
<portlet-name>com_liferay_blogs_web_portlet_BlogsAdminPortlet</portlet-name>
<portlet-name>com_liferay_blogs_web_portlet_BlogsPortlet</portlet-name>
</portlet-ref>
<weight>2</weight>
<permissions>
<supports>
<action-key>ADD_DISCUSSION</action-key>
<action-key>DELETE</action-key>
<action-key>DELETE_DISCUSSION</action-key>
<action-key>PERMISSIONS</action-key>
...
</model-resource>
</resource-action-mapping>
```

Source: <https://github.com/liferay/liferay-portal/blob/7.1.0-ga1/modules/apps/blogs/blogs-service/src/main/resources/resource-actions/default.xml>

Step 3 - Manage Permission Resources Creation and Deletion

Permission resources are permission container objects bound to the model entities. After resources and available permissions are defined, we need to take care of adding and deleting permission resources for our entities.

In the case of a Service Builder project, adding is usually done in the method adding a new entity. Resource deletion is done in the entity deletion method. Portal services for managing permission resources are `com.liferay.portal.kernel.service.ResourceService` and `com.liferay.portal.kernel.service.ResourceLocalService`.

Below is an excerpt of a Blogs application's BlogsEntry service class:

`com.liferay.blogs.service.impl.BlogsEntryLocalServiceImpl`

```

@Override
public void addEntryResources(
    BlogsEntry entry, boolean addGroupPermissions,
    boolean addGuestPermissions)
throws PortalException {

    resourceLocalService.addResources(
        entry.getCompanyId(), entry.getGroupId(), entry.getUserId(),
        BlogsEntry.class.getName(), entry.getEntryId(), false,
        addGroupPermissions, addGuestPermissions);
}

@Override
public void addEntryResources(
    BlogsEntry entry, ModelPermissions modelPermissions)
throws PortalException {

    resourceLocalService.addModelResources(
        entry.getCompanyId(), entry.getGroupId(), entry.getUserId(),
        BlogsEntry.class.getName(), entry.getEntryId(), modelPermissions);
}
```

The example below is from the entry deletion method of the same class:

```
@Indexable(type = IndexableType.DELETE)
```

```

@Override
@SystemEvent(type = SystemEventConstants.TYPE_DELETE)
public BlogsEntry deleteEntry(BlogsEntry entry) throws PortalException {
    // Entry
    blogsEntryPersistence.remove(entry);

    // Resources
    resourceLocalService.deleteResource(
        entry.getCompanyId(), BlogsEntry.class.getName(),
        ResourceConstants.SCOPe_INDIVIDUAL, entry.getEntryId());
    ...
}

```

Step 4 - Implement Permission Checkers

When using permissioning, while not mandatory, implementing permission checker helper classes makes things more convenient. With a Service Builder project, this would mean:

- Creating the permission checker service interface in the API module
- Creating the checker implementation to the implementation module

Below is an example of a Blogs application portlet resource permission checker:

com.liferay.blogs.web.internal.security.permission.resource.BlogsPermission

```

@Component(immediate = true)
public class BlogsPermission {

    public static boolean contains(
        PermissionChecker permissionChecker, long groupId, String actionId) {
        return _portletResourcePermission.contains(
            permissionChecker, groupId, actionId);
    }

    @Reference(
        target = "(resource.name=" + BlogsConstants.RESOURCE_NAME + ")",
        unbind = "!")
    protected void setPortletResourcePermission(
        PortletResourcePermission portletResourcePermission) {
        _portletResourcePermission = portletResourcePermission;
    }

    private static PortletResourcePermission _portletResourcePermission;
}

```

The example below is a model-resource checker for the Blogs application:

com.liferay.blogs.web.internal.security.permission.resource.BlogsEntryPermission

```

@Component(immediate = true, service = BlogsEntryPermission.class)
public class BlogsEntryPermission {

    public static boolean contains(
        PermissionChecker permissionChecker, BlogsEntry entry,
        String actionId)
        throws PortalException {
        return _blogsEntryFolderModelResourcePermission.contains(

```

```

        permissionChecker, entry, actionId);
    }

    public static boolean contains(
        PermissionChecker permissionChecker, long entryId, String actionId)
    throws PortalException {

        return _blogsEntryFolderModelResourcePermission.contains(
            permissionChecker, entryId, actionId);
    }

    @Reference(
        target = "(model.class.name=com.liferay.blogs.model.BlogsEntry)",
        unbind = "_"
    )
    protected void setEntryModelPermission(
        ModelResourcePermission<BlogsEntry> modelResourcePermission) {

        _blogsEntryFolderModelResourcePermission = modelResourcePermission;
    }

    private static ModelResourcePermission<BlogsEntry>
        _blogsEntryFolderModelResourcePermission;
}

```

Step 5 - Implement Permission Checking

The final step is to make use of the permissions. In the user interface, you should check permissions first to show certain data or, for example, controls for adding and deleting entities. Although service modules deployed to the OSGi container can be accessed other places besides your application, you also have to implement permission checking on the service layer. As mentioned in the Service Builder section, permission checking should be done on your remote service implementation class. User level access to services should, therefore, always be done through remote service classes. Local services **should not** implement permission checking.

The example below from a Blog applications BlogsEntryServiceImpl class demonstrates permission checking before letting you add a new Blogs entry:

com.liferay.blogs.service.impl.BlogsEntryServiceImpl

```

@Override
public BlogsEntry addEntry(
    String title, String subtitle, String description, String content,
    int displayDateMonth, int displayDateDay, int displayDateYear,
    int displayDateHour, int displayDateMinute, boolean allowPingbacks,
    boolean allowTrackbacks, String[] trackbacks,
    String coverImageCaption, ImageSelector coverImageImageSelector,
    ImageSelector smallImageImageSelector,
    ServiceContext serviceContext)
throws PortalException {

    _portletResourcePermission.check(
        getPermissionChecker(), serviceContext.getScopeGroupId(),
        ActionKeys.ADD_ENTRY);

    return blogsEntryLocalService.addEntry(
        getUserId(), title, subtitle, description, content,
        displayDateMonth, displayDateDay, displayDateYear, displayDateHour,
        displayDateMinute, allowPingbacks, allowTrackbacks, trackbacks,
        coverImageCaption, coverImageImageSelector, smallImageImageSelector,
        serviceContext);
}

```

In the JSP user interface, checking can be done, for example, with standard JSTL library tags:

[liferay-portal/modules/apps/blogs/blogs-web/src/main/resources/META-INF/resources/blogs/entry_action.jsp](#)

```
...
<c:if test="<%!= BlogsEntryPermission.contains(permissionChecker, entry, ActionKeys.PERMISSIONS) %>">
    <liferay-security:permissionsURL
        modelResource="<%!= BlogsEntry.class.getName() %>"
        modelResourceDescription="<%!= BlogsEntryUtil.getDisplayTitle(resourceBundle, entry) %>"
        resourceGroupId="<%!= String.valueOf(entry.getGroupId()) %>"
        resourcePrimKey="<%!= String.valueOf(entry.getEntryId()) %>"
        var="permissionsEntryURL"
        windowState="<%!= LiferayWindowState.POP_UP.toString() %>"
    />

    <liferay-ui:icon
        label="<%!= true %>"
        message="permissions"
        method="get"
        url="<%!= permissionsEntryURL %>"
        useDialog="<%!= true %>"
    />
</c:if>
...

```

Looking Under the Hood

Let's have a look at how permissions are stored in the database. There are two tables storing the permissions information:

- ResourceAction
- ResourcePermission

Example of ResourceAction table

```
mysql> select * from resourceaction where name like '%blogs%' limit 5;
+-----+-----+-----+-----+-----+
| mvccVersion | resourceId | name          | actionId | bitwiseValue |
+-----+-----+-----+-----+-----+
|      0 |        4 | com.liferay.blogs | ADD_ENTRY |          2 |
|      0 |        5 | com.liferay.blogs | PERMISSIONS |         4 |
|      0 |        6 | com.liferay.blogs | SUBSCRIBE |         8 |
|      0 |       35 | com.liferay.blogs.kernel.model.BlogsEntry | UPDATE_DISCUSSION |         2 |
|      0 |       36 | com.liferay.blogs.kernel.model.BlogsEntry | DELETE |         4 |
+-----+-----+-----+-----+-----+
5 rows in set (0,00 sec)
```

Example of ResourcePermissions table

```
mysql> mysql> select * from resourcepermission where name like '%blogs%' order by resourcepermissionId desc limit 5;
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| mvccVersion | resourcePermissionId | companyId | name          | scope | primKey | primKeyId | roleId | ownerId | actionIds | viewActionId |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
|      0 |        4406 | 20115 | com.liferay.blogs | 4 | 66680 | 66680 | 20130 | 0 | 8 | 0 |
|      0 |        4405 | 20115 | com.liferay.blogs | 4 | 66680 | 66680 | 20123 | 0 | 14 | 0 |
|      0 |        4320 | 20115 | com.liferay.blogs.kernel.model.BlogsEntry | 4 | 63341 | 63341 | 20123 | 0 | 65 | 1 |
|      0 |        4319 | 20115 | com.liferay.blogs.kernel.model.BlogsEntry | 4 | 63341 | 63341 | 20130 | 0 | 65 | 1 |
|      0 |        4318 | 20115 | com.liferay.blogs.kernel.model.BlogsEntry | 4 | 63341 | 63341 | 20123 | 20155 | 127 | 1 |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
5 rows in set (0,00 sec)
```

Exercises

Implement Gradebook Permissioning Support

Introduction

In this exercise, we will implement entity permissioning support and service-level permission checking into the Gradebook application. We will revisit permissioning later when implementing the Gradebook presentation layer.

Overview

- 1 Prepare service implementation classes
- 2 Add the portlet.properties file to the Gradebook service, defining the path to default.xml
- 3 Declare model resources and actions in default.xml
- 4 Define permission checker service interface
- 5 Export permission checker interface
- 6 Implement permission checker class
- 7 Implement resources creation and deletion on the service layer
- 8 Implement permission checking in the remote service implementation class

Snippets and resources for this exercise are in `snippets/chapter-06/03-implement-permissioning-support` of your Liferay Workspace.

The complete Gradebook application **solution** is in the respective module directories in `solutions/solutions-chapter-06/solution-06-gradebook`.

Prepare Service Implementation Classes

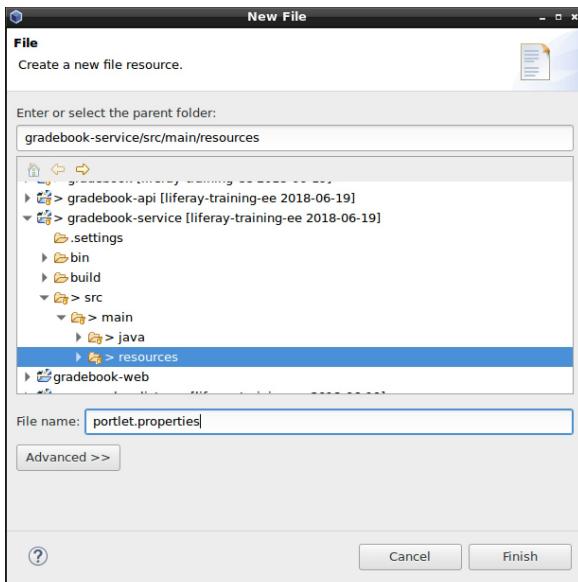
Before starting the actual exercise, we will replace our current service implementation classes with copies containing placeholders and methods for the coming steps in order to have a common starting point for this exercise.

1. **Copy** the four service implementation classes from the exercise snippets `gradebook-service/src/../../service/impl`. (Hint: select the folder `gradebook-service` in the Project Explorer, press the shift key and jam on the right arrow key to expand the tree nodes below.)
2. **Paste** the four files into the `com.liferay.training.gradebook.service.impl` package of the `gradebook-service` module.
3. **Choose Yes To All** when prompted to overwrite existing files.
4. **Run** the `buildService` task in the `gradebook-service` module to recreate the service.

Add the portlet.properties File

1. **Right-click** on the `src/main/resources` folder of the `gradebook-service` module.
2. **Choose New → Other**.
3. **Choose File** from the list of options.
4. **Click Next**.

5. Type `portlet.properties` in the *File name* field.



6. Click *Finish*.

7. Type the following in the `portlet.properties`:

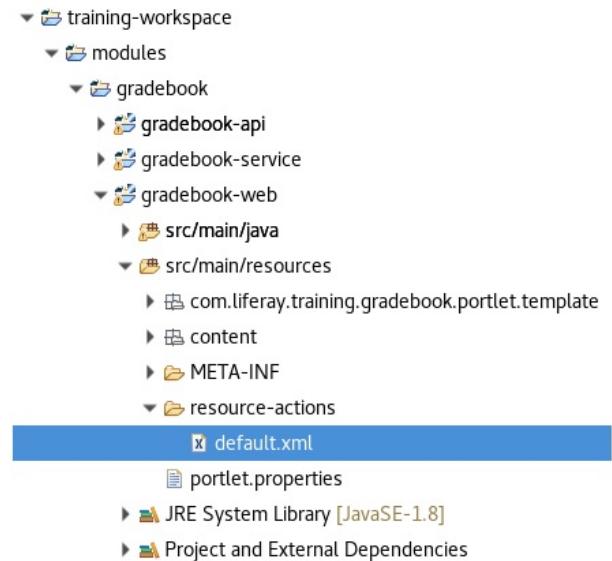
```
resource.actions.configs=/META-INF/resource-actions/default.xml
```

8. Save the file.

Declare Model Resources and Actions in the default.xml

Now we will create the `default.xml` file, which defines the model resources and actions for this service.

1. Right-click on the `META-INF` folder in `src/main/resources/` of the `gradebook-service` module.
2. Choose `New → Folder`
3. Type `resource-actions` for the folder name.
4. Click *Finish*.
5. Copy the `gradebook-service/src/main/resources/META-INF/resource-actions/default.xml` in the exercise snippets folder.
6. Paste the file into the `gradebook-service/src/main/resources/META-INF/resource-actions/` folder:

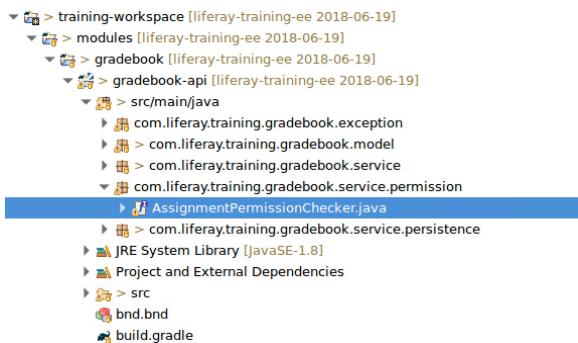


Open and review the *default.xml* after you've added it to the module project.

Define Permission Checker Service Interface

We will create a permission checker utility class, which will be implemented as an OSGi service and referenced from the *gradebook-api* module. For that purpose, we will first create an interface in the *gradebook-api* module.

1. **Right-click** on the *gradebook-api* module to open the context menu.
 2. **Choose** *New → Package*.
 3. **Type** *com.liferay.training.gradebook.service.permission* in the *Name* field.
 4. **Click** *Finish* to close the dialog.
 5. **Copy** the *gradebook-*
- api/src/main/java/com/liferay/training/gradebook/service/permission/AssignmentPermissionChecker.java class in the exercise snippets folder. (Hint: select the *gradebook-api* folder, press the shift key and jam on the right arrow key to expand the tree nodes below.)
6. **Paste** the file in the package created in the previous step.



Make sure to review the interface. Now we have the permission checker API in place.

Export Permission Checker Interface

Remember that OSGi bundles do not expose anything from themselves to other bundles in the container unless explicitly defined. That means that the permission checker you just created is not accessible to any bundle yet. Let's fix that:

1. **Open** the `bnd.bnd` file in `gradebook-api`.
2. **Click** on the `Source` tab.
3. **Add** the `Export-package` header for the `com.liferay.training.gradebook.service.permission` package as follows:

```
Bundle-Name: Gradebook API
Bundle-SymbolicName: com.liferay.training.gradebook.api
Bundle-Version: 1.0.0
Export-Package:\\
    com.liferay.training.gradebook.exception,\\
    com.liferay.training.gradebook.model,\\
    com.liferay.training.gradebook.service,\\
    com.liferay.training.gradebook.service.persistence,\\
    com.liferay.training.gradebook.service.permission
-includeresource: META-INF/service.xml=../gradebook-service/service.xml
```

4. **Save** the `bnd.bnd` file.

The permission package and the interface in it is now exposed and accessible.

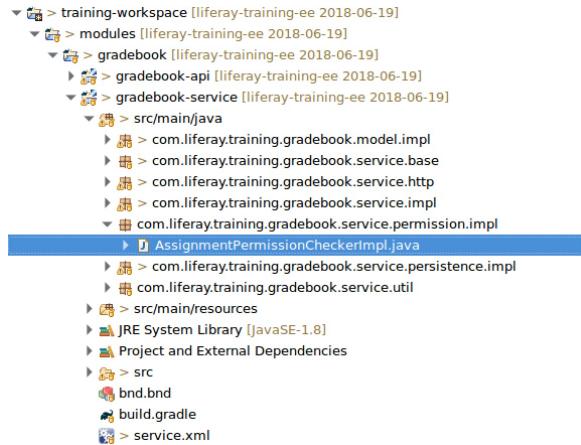
Implement Permission Checker Class

Let's implement the interface we just created.

1. **Right-click** on the `gradebook-service` module to open the context menu.
2. **Choose** `New → Package`.
3. **Type** `com.liferay.training.gradebook.service.permission.impl` in the `Name` field.
4. **Click** `Finish` to close the dialog.
5. **Copy** the `gradebook-`

```
service/src/main/java/com/liferay/training/gradebook/service/permission/impl/AssignmentPermissionCheckerImpl
from exercise snippets folder.
```

6. **Paste** the file to the package created before:



7. **Run** the `buildService` Gradle task to recreate the service.

If you review the persistence layer, you'll notice that the `filterFindBy` methods, which support permissioning, were now created.

Implement Permission Checker Class

We now have our model permissions and resources defined and a permission checker utility in place. Next, we will need to bind the permissions to model entities. This is done by attaching permission resources to any new entities created. We'll also implement deleting of resources to the entity deletion method, as otherwise orphan resources

would remain in the database.

1. **Open** `AssignmentLocalServiceImpl` in `gradebook-service`.
2. **Find** the `addAssignment()` method.

- o You'll find a placeholder comment for adding resources:

```
***** [PLACEHOLDER FOR ADDING PERMISSION RESOURCES] *****
```

3. **Replace** the placeholder with the following code:

```
resourceLocalService.addResources(  
    group.getCompanyId(), groupId, userId, Assignment.class.getName(),  
    assignment.getAssignmentId(), portletActions, addGroupPermissions,  
    addGuestPermissions);
```

- o This will attach resources to the new model instances and make them support permissioning. Now let's take care of cleaning the resources on entity deletion.

4. **Find** the `deleteAssignment()` method.

- o You'll find there a placeholder comment for deleting the resources:

```
***** PLACEHOLDER FOR DELETING PERMISSION RESOURCES *****
```

5. **Replace** the placeholder with the following code:

```
resourceLocalService.deleteResource(  
    assignment, ResourceConstants.SCOPe_INDIVIDUAL);
```

6. **Save** the file.

The last thing to take care of on the service layer is to implement the permission checking.

Implement Permission Checking in the Remote Service

We want certain service calls, like adding, updating, and deleting of assignments to be protected by permissions. When using the Service Builder tool, permission checking is implemented in the remote service implementation class, in this case `AssignmentServiceImpl`. As a rule of thumb, all user-level service calls should use the remote interface to operate under permissioning.

1. **Open** `AssignmentServiceImpl`.
2. **Uncomment** all the commented methods and calls, including the `@ServiceReference` on the bottom of the class.
3. **Type** `CTRL+Shift+O` to use the *Organize Imports* feature and create imports.
4. **Save** the file.
5. **Run** the `buildService` Gradle task to refresh the service.

Now our service layer supports permissions.

Create the Presentation Layer

In modular Liferay applications, the presentation layer is usually located in what's called the *web module*. For example, in the native Blogs application this module is called *blogs-web*. In terms of the MVC pattern, this module typically contains both the view (user interface) and the controller (portlet) layers. Here's a typical set of web module contents:

- Portlet component(s)
- MVC action command components
- JSP files (or any other chosen technology)
- Localization resources
- CSS and JavaScript resources

The implementation steps and their order of execution depend on the chosen technology and approach. For the Gradebook exercise, we will be using the following approach:

1. Create the web module
2. Provide localization resources
3. Implement portlet actions using MVC commands
4. Implement portlet JSPs using tag libraries
5. Implement portlet validation and feedback
6. Add CSS resources
7. Add Javascript resources

Let's discuss the steps and the concepts related to the user interface of our Gradebook application.

Creating the Web Module

Liferay provides several module templates for building the presentation layer:

- **mvc portlet**: a module template with a sample Liferay MVC portlet component, localization resources, and JSP files
- **freemarker-portlet**: a module template for the user interface with FreeMarker templating language and Freemarker portlet back-end. The FreeMarkerPortlet class is an extension of Liferay MVC portlet.
- **NPM portlets**: several Liferay MVC portlet-based templates for building the user interface with JavaScript frameworks like Angular and React; package management with NPM
- **soy-portlet**: a starter template for building a Google SOY templates-based user interface backed with a SoyPortlet component. The SoyPortlet class is an extension of Liferay MVC portlet.
- **spring-mvc-portlet**: for building a Spring MVC portlet
- **war-mvc-portlet**: a template for legacy WAR-style portlets

In addition to these, *JSF* and *Vaadin* portlet-type user interfaces are supported out of the box.

Internationalizing the Application

Internationalization (i18n) means designing an application so that it isn't hardwired to one language only.

Localization (i10n) is conceptually deriving from internationalization and means adding support to a specific language.

Even if you don't intend to have your application supporting multiple languages, internationalization as a design pattern make applications much easier to maintain.

When doing internationalization, you assign a string key to each display message. These keys are then used in the code, instead of the actual messages. A separate resource file, a language resource bundle, holds a list of display messages as key-value pairs. This approach makes it possible to have all the display messages in a single place, making updating and management easier. Adding support for an additional language is then usually just a matter of translating the language file.

Language Keys

Language keys are unique string identifiers for the display messages. The same keys can be reused in the code as many times as needed. As a good practice, you should use descriptive keys like `submit-form` to improve code readability.

Parametrization of keys is supported.

Language Files

Language files, or localization files, contain a list of key value pairs for a single language. The default `Language.properties` file, which serves as a fallback default file, is usually automatically generated by a module template and is located in the `src/main/resources/content` folder.

Language files should use UTF-8 encoding.

Language files have the following naming syntax: `Language_LANGUAGE_CODE.properties`, where `LANGUAGE_CODE` is replaced either by:

- ISO 639-1 standard language code or
- ISO 639-1 language code and ISO 3166 country code, separated by an underscore

Examples:

- `Language_en.properties` (all the English variants)
- `Language_en_US.properties` (US English)

Making Portlets Aware of Language Resources

In order to use the language files, portlets and other components have to be made aware of the available resources. In case of portlets, this is done by defining the `javax.portlet.resource-bundle` component property. When using Liferay portlet module templates, default resources and properties are created automatically.

Let's have a look at a concrete localization example below. First, we have a portlet component and the `javax.portlet.resource-bundle` property. Notice that the value is actually a path pointing to `resources/content/Language.properties`.

A portlet

```
@Component(
    immediate = true,
    property = {
        ...
        "javax.portlet.resource-bundle=content.Language",
        ...
    },
    service = Portlet.class
)
public class LocalizationExamplePortlet extends MVCPortlet {
    ...
}
```

The language file contains a list of key value pairs for a specific language.

Language.properties

```
hello-stranger=Hello stranger
hello-you=Hello you
my-favourite-dogs-are-x-and-x=My favourite-dogs are {0} and {1}
```

Display messages can be referenced by their respective keys. For example, by using Liferay tag libraries:

JSP

```
<liferay-ui:message key="hello-stranger" />
<liferay-ui:message key="my-favourite-dogs-are" arguments="<% new String[] {"Ryder", "Marshall"} %>" />
```

When using tag libraries, notice that when a matching language key is not found in the portlet language file, the lookup falls back to the portal resource bundle.

Localizing Portlet Names

Portlet standard message localization follows a special pattern.

Pattern in Language properties

```
javax.portlet.<portlet-name-field>.<portlet-id>=<translation>
```

Where the *portlet-name-field* can be one of the following:

- description
- display-name
- keywords
- short-title
- title

Example: portlet-name *gradebook-web*

```
javax.portlet.display-name.gradebook-web=Gradebook Portlet
```

Localization in the Back-End

Displaying localized messages can be done in the user interface. You can use tag libraries or the *Liferay.Language* JavaScript object. Accessing the resources in a Java class requires loading the resource bundle manually. Below is an example:

```
public void doView(RenderRequest renderRequest, RenderResponse renderResponse) throws IOException, PortletException {
    ...
    String helloStranger = getResourceBundleLocalization("hello-stranger", locale);
    String myFavoriteDogsAre = getResourceBundleLocalization("my-favourite-dogs-are-x-and-x", locale, "Ryder", "Marshall");
    ...
}

private String getLanguageLocalization(String key, Locale locale, Object...objects) {
    ResourceBundle resourceBundle = _resourceBundleLoader.loadResourceBundle(locale);

    String value = ResourceBundleUtil.getString(resourceBundle, key, objects);

    return value==null ? _language.format(locale, key, objects) : value;
}
```

```
}

@Reference(target = "(bundle.symbolic.name=com.liferay.training.localization.example)", unbind = "-")
private ResourceBundleLoader _resourceBundleLoader;

@Reference
private Language _language;
...
```

Sharing Resource Bundles Between Modules

In modular OSGi design, a single module is self-contained and usually contains all the resources it needs. In a multi-module application, however, you'll sometimes want to centralize all localization resources in a dedicated module or let modules access localization resources from each other. Liferay provides a bnd directive *liferay-aggregate-resource-bundles* for this purpose:

bnd.bnd

```
-liferay-aggregate-resource-bundles: \
[bundle.symbolic.name1], \
[bundle.symbolic.name2]
```

Summarizing Internationalization

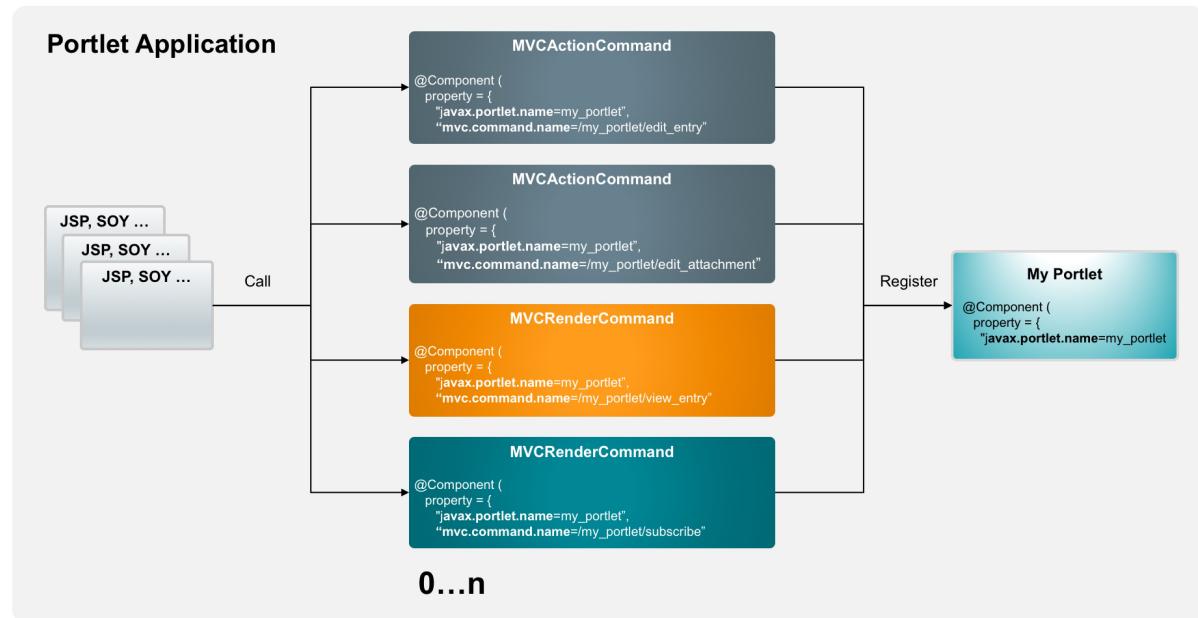
Internationalization is not just about localizing your application, but about a preferable design pattern for any user interface-centric applications. This lets you create a loose coupling between the display messages and the business logic, making the application more manageable.

Implementing Portlet Actions

The interaction between a portlet back-end and a user is handled by portlet lifecycle methods as described in *Chapter 5 - Java Standard Portlet*.

MVC commands are Liferay-provided, portlet liferay handler components meant to be used in conjunction with Liferay MVC portlets. They reduce the need for boilerplate portlet coding and provide a more modular application design compared to legacy style lifecycle handlers. MVC commands are single class OSGi components, responsible for handling an action defined by components in the *mvc.command.name* property. MVC commands register as lifecycle handlers for a certain portlet with their *javax.portlet.name* component property.

There are three types of MVC commands: MVC render commands, MVC action commands, and MVC resource commands:



MVC Render Commands

MVC render commands handle portlet render phase. They are called by setting the *mvcRenderCommandName* in a calling URL, as in the example below:

Calling from JSP Example

```
<portlet:renderURL var="viewEntryUrl">
    <portlet:param name="mvcRenderCommandName" value="/my_portlet/view_entry" />
    <portlet:param name="entryId" value="<% String.valueOf(entry.getId()) %>" />
</portlet:renderURL>
<a href="<% viewEntryUrl %>">Click here to view the entry</a>
```

MVC Action Commands

MVC action commands handle a portlet's action phase. Actions are typically form submits that trigger events, like entity update, on the model layer. The action to respond to is defined in the calling URL's name parameter.

Calling from JSP Example

```
<portlet:actionURL name="/my_portlet/edit_entry" var="editEntryURL" />
<aui:form action="<% editEntryURL %>" method="post" name="fm">
    ...
</aui:form>
```

MVC Resource Commands

MVC resource commands handle the resource serving phase. As the resource serving lifecycle phase doesn't invoke the render phase, they are typically used for operations that don't need page refresh. Such operations can be, for example:

- Subscribing
- Updating a list with Ajax call, without refreshing page
- Captcha checking

Calling from JSP Example

```
<portlet:resourceURL id="/login/captcha" var="captchaURL" />
```

Using Tab Libraries

Tag libraries are collections of user interface components called tags for JSP development, allowing a clean separation between the look-and-feel business logic. Tag libraries have a namespace, like:

- c
- portlet
- liferay-ui
- aui

Tag libraries can significantly reduce development time and remove boilerplate coding. Tag libraries provided by Liferay contain tags for the most common user interface components (like forms) are designed to be responsive.

The **tags** inside a library are JSP markup elements, which are replaced by a respective markup at render times. Tags available in a library can be browsed in the library's descriptor TLS file (inside taglibs jar).

Below is an example of a liferay-ui tag library declaration, a single tag for showing user details and rendered HTML.

Tag library declaration (init.jsp)

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

Tag usage (view.jsp)

```
<liferay-ui:user-display
    markupView="lexicon"
    showUserDetails="true"
    showUserName="true"
    userId="<% themeDisplay.getRealUserId() %>"
    userName=""<%= themeDisplay.getRealUser().getFullName() %>">
/>
```

Rendered HTML

```
<div class="profile-header">
<div class="nameplate">
<div class="nameplate-field">
<div class="user-icon-color-0 user-icon user-icon-default">
<span>TT</span>
</div>
</div>
<div class="nameplate-content">
<div class="heading4">
<a href="http://localhost:8080/web/test"> Test Test </a>
</div> </div> <div class="nameplate-content">
</div>
</div>
</div>
```

Standard Libraries

The portlet standard libraries are:

- **portlet**: standard portlet JSR 168 tag library (overrun by portlet_2_0)
- **portlet_2_0**: standard portlet JSR 286 tag library (included in liferay_portlet_2_0_ext)
- **portlet_3_0**: standard portlet JSR 362 tag library

- **The standard JSTL tag libraries**
 - Core Tags
 - Formatting tags
 - SQL tags
 - XML tags
 - JSTL Functions

Standard JSTL Libraries contain general purpose tags for example for:

- Variables, conditionals (if-else), loops
- Formatting strings, dates, and numbers
- Basic string functions (replace, join, lowercase etc.)
- Parsing XML
- Showing date

Below is an example of standard JSTL tags usage:

```
<c:choose>
    <c:when test="${themeDisplay.isSignedIn()}">
        <p>You appear to be signed in. Your groups: </p>
        <ul>
            <c:forEach items="${themeDisplay.getUser().getGroups()}" var="group">
                <li><c:out value = "${group.getName(locale)}"/></li>
            </c:forEach>
        </ul>
    </c:when>
    <c:otherwise>
        <p>You are not signed in.
    </c:otherwise>
</c:choose>
```

The **Standard JSR 362 Library (portlet_3_0)** library provides portlet lifecycle url generation and portlet namespacing. Below are examples of creating an action url and render url with the portlet 3.0 tag library:

Taglib Declaration

```
<%@ taglib uri="http://xmlns.jcp.org/portlet_3_0" prefix="portlet" %>
```

An Action URL

```
<portlet:actionURL name="/gradebook/assignment/delete"
    var="deleteAssignmentURL">
    <portlet:param name="redirect" value="${currentURL}" />
    <portlet:param name="assignmentId" value="${assignment.assignmentId}" />
</portlet:actionURL>
```

A Render URL

```
<portlet:renderURL var="viewSubmissionsURL">
    <portlet:param name="mvcRenderCommandName" value="/gradebook/submissions/view" />
    <portlet:param name="redirect" value="${currentURL}" />
    <portlet:param name="assignmentId" value="${assignment.getAssignmentId()}" />
</portlet:renderURL>
```

Liferay's Tag Libraries

Liferay provides a rich set of its own libraries:

- **liferay-theme**: provides theme components
- **liferay-portlet**: a wrapper for standard portlet_2_0 library
- **liferay-ui**: contains numerous display tags like alert, icons, search-container
- **liferay-util**: provide page level tags like includes
- **aui**: Responsive Alloy UI tags for forms and containers (deprecated)
- **liferay-security**: provides doAsURL and permissionURL
- **clay**: set of tags for creating Liferay Clay UI components
- **chart**: charting tags

Each library has its own use case. The **Clay** library supersedes the AUI library of many parts, leveraging the Liferay Clay framework and providing tags, for example, for:

- Alerts
- Badges
- Buttons
- Cards
- Dropdown Menus and Action Menus
- Form Elements
- Icons
- Labels and links
- Navigation Bars
- Progress Bars
- Stickers

Below is an example of using Clay:

Taglib Declaration

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

```
<clay:stripe
    message="This is a success message."
    style="success"
    title="Success"
/>
```

Rendered Output

The **Chart** tag library contains tags for modeling data graphically. Below is an example:

JSP

```
<%@ page import="com.liferay.frontend.taglib.chart.model.point.bar.BarChartConfig" %>

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

<%
    BarChartConfig barChartConfig = (BarChartConfig)request.getAttribute("chartSample");
%>
```

Back-End Code

```
@Override
```

```

    public void doView(RenderRequest renderRequest, RenderResponse renderResponse) throws IOException, PortletE
xception {

    renderRequest.setAttribute("chartSample", getBarChartConfig());
    super.doView(renderRequest, renderResponse);
}

private BarChartConfig getBarChartConfig() {

    BarChartConfig barChartConfig = new BarChartConfig();
    barChartConfig.addColumn(
        new MultiValueColumn("data1", 100, 20, 30),
        new MultiValueColumn("data2", 20, 70, 100));

    return barChartConfig;
}

```

Validation and Feedback

When dealing with user input, it's important to have control both of the input and output. A robust validation is done on multiple layers, and it never relies on the user interface only.

Before discussing validation and feedback, let's take a quick overview at Liferay's utility classes, which not only provide tools for validation but for many other common tasks, like:

- String manipulation
- Date formatting
- User input validation
- Output sanitizing
- Request parameter handling
- JSON

General Utilities

- **arrayUtil**: array manipulation
- **dateFormatFactory**: date formatting
- **languageUtil**: internationalization and localization tools
- **StringUtil**: string manipulation
- ...

Core Utilities

- **paramUtil**: used to retrieve parameters from Servlet or Portlet requests
- **portalUtil**: gives access to almost all of the current context
- **propsUtil**: for fetching configuration values from your properties file(s)
- **serviceLocator**: provides access to core or custom services
- **utilLocator**: helps to locate a specific utility bean by name
- ...

Other Helper Utilities

- arrayUtil
- browserSniffer
- calendarFactory
- dateFormatFactory
- dateUtil
- getterUtil
- htmlUtil

- httpUtil
- imageToolUtil
- jsonFactoryUtil
- languageUtil
- locateUtil
- staticFieldGetter
- stringUtil
- timeZoneUtil
- unicodeFormatter
- unicodeLanguageUtil
- saxReaderUtil

Most of the utilities can be found in the `com.liferay.portal.kernel.util` package. The available classes and the documentation is available at <https://docs.liferay.com/dxp/digital-enterprise/7.0-sp1/javadocs/portal-kernel/com/liferay/portal/kernel/util/package-summary.html>

Implementing Validation and Feedback

There are many good reasons to implement proper validation and user interface feedback in your applications:

1. **Security:** to protect against malicious input, no user input should be allowed to enter the model layer without validation
2. **Usability:** early validation and feedback provides a better user experience
3. **Resource Usage:** validation on user interface reduces faulty form submissions and server load

Liferay provides tools and utilities for both client and server-side validation. On the client-side, validation is implemented in many of the tag libraries. The following tags are supporting validators:

- 
- 
- 
- 

Input Validation on the User Interface

Below is an example of using the Alloy UI Validator:

```
<aui:form>
    <aui:input name="title" >
        <aui:validator name="required" errorMessage="Please enter the title."/>
    </aui:input>
    ...
<aui:form>
```

Alloy UI libraries also allow you to write your own JavaScript validator functions, which simply return true or false.

Below is an example:

```
<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>
```

As the Liferay Clay user interface framework is based on Twitter Bootstrap, its validation methods are available:

```
<form data-toggle="validator" role="form">
    <div class="form-group">
        <label for="inputName" class="control-label">Name</label>
        <input type="text" class="form-control" id="inputName" placeholder="Your Name" required>
    </div>
    <div class="form-group">
        <button type="submit" class="btn btn-primary">Submit</button>
    </div>
</form>
```

Input Validation on the Back-End

On the back-end side, a Validator utility class is available, providing methods for common validation tasks:

```
String name = ParamUtil.getString(request, "title");
If (Validator.isNull(name)) {
    SessionErrors.add(request, "title-empty");
    return false;
}
```

Output Validation

In addition to input validation, an output validation for malicious content should be done. Usually this means at least escaping content prior to displaying it to prevent malicious code from executing.

On the client side, escaping can be done with the standard JSTL library:

```
<c:out escapeXML="true">${bodyText}</c:out>
```

On the server side, Liferay HTMLUtil class can be used:

```
request.setAttribute("bodyText", HtmlUtil.escape(bodyText));
```

Note on AntiSamy

There is an additional security module that protects against malicious user input. The **AntiSamy** module leverages the OWASP AntiSamy library, processing user input on form submit and stripping away all the HTML elements and content not explicitly allowed. The module is configurable through Control Panel->System Settings->Foundation->AntiSamy Sanitizer.

Validation Guidelines

Here's a summary of the best practices for validation:

- Establish validation on both the client and server side.
 - User interface validation is not for securing but for usability.
- Establish control over all input.
- Establish control over all output; escape the output on the user interface.
- Escaping of user-provided input should generally be done on render time, not storing time.

Showing Feedback

In Liferay portlet JSP applications, the feedback from the back-end to the user interface is transported typically with SessionErrors and SessionMessages objects. Below is an example demonstrating setting a message key in the back-end, doing a localization in Language.properties and showing the message on the user interface using the liferay-ui:success and liferay-ui:error tags:

Portlet Class

```
String name = ParamUtil.getString(request, "title");
If (Validator.isNull(name)) {
    SessionErrors.add(request, "title-empty");
    return false;
}

if (SessionErrors.isEmpty()) {
    SessionMessages.add(request, "assignment-updated-successfully");
}
```

Language.properties

```
assignment-updated-successfully=Assignment was updated successfully
please-enter-title=Please enter the title.
```

view.jsp

```
<liferay-ui:success key="assignment-updated-successfully" message="assignment-updated-successfully" />
<liferay-ui:error key="title-empty" message="please-enter-title" />
```

Rendered HTML on error

Please enter the title.

Hiding Default Messages

Liferay sets default success messages for successful portlet actions. These messages can be hidden by setting the property `add-process-action-success-action` in portlet component properties to `false`:

```
@Component(
    immediate = true,
    property = {
        ...
        "javax.portlet.init-param.add-process-action-success-action=false",
        ...
    },
    service = Portlet.class
)
```

Wrapping Up Validation and Feedback Flow

Below are steps for implementing a basic validation in your application:

- ① Validate user input on the user interface
- ② Customize AntiSamy to filter input on form submit
- ③ Do back-end validation and add messages to the session objects
- ④ Show validation feedback on the screen using liferay-ui tag library

- 5 Validate user-contributed output

Adding CSS

Portlet CSS files are defined in portlet component properties. The following properties are available, each of them allowing multiple values:

- com.liferay.portlet.header-portal-css
- com.liferay.portlet.header-portlet-css
- com.liferay.portlet.footer-portal-css
- com.liferay.portlet.footer-portlet-css

By default, header and footer CSS files are injected respectively in page header and footer. The difference between *portal-css* and *portlet-css* is the context path with which the former is set to portal and the latter to the portlet context. Portlet CSS files are typically placed in the META-INF/resources/css folder.

Liferay uses SASS compiler. Files suffixed with .scss are SASS compiled automatically. Notice that the reference to a CSS file in the portlet properties is still .css even if you use .scss as a suffix. Here's an example:

CSS file

```
css-portlet/src/main/resources/META-INF/resources/css/main.scss
```

Portlet Component Properties

```
...
"com.liferay.portlet.header-portlet-css=/css/main.css",
...
```

Generated HTML source

```
<head>
...
<link data-senna-track="temporary" href="http://localhost:8080/o/com.liferay.training.css/css/main.css?
browserId=other&themeId=classic_WAR_classictheme&languageId=en_US&b=7100&t=1526032024000" id="6
05088bd" rel="stylesheet" type="text/css">
...
```

Adding JavaScript

Following the logic of defining CSS files, portlet JavaScript files are defined in the component properties with the following properties:

- com.liferay.portlet.header-portal-javascript
- com.liferay.portlet.header-portlet-javascript
- com.liferay.portlet.footer-portal-javascript
- com.liferay.portlet.footer-portlet-javascript

Typically, JavaScript files are placed in the META-INF/resources/js folder.

Portlet JavaScript files are merged and minified at compile time and served through combo servlet. Here's an example:

JavaScript files

```
css-portlet/src/main/resources/META-INF/resources/js/main1.js
```

```
css-portlet/src/main/resources/META-INF/resources/js/main2.js
```

main1.js contents

```
console.log("This is the main1.js file.")
```

main2.js contents

```
console.log("This is the main2.js file.")
```

Portlet Component Properties

```
"com.liferay.portlet.header-portlet-javascript=/js/main1.js",
"com.liferay.portlet.header-portlet-javascript=/js/main2.js",
```

Generated HTML source

```
<head>
  ...
  <script data-senna-track="temporary" src="/combo?browserId=other&minifierType=&themeId=classic_WAR_classictheme&languageId=en_US&b=7100&CssAndJavascriptExample_INSTANCE_3sSKq0IS8KsX:%2Fjs%2Fmain1.js&CssAndJavascriptExample_INSTANCE_3sSKq0IS8KsX:%2Fjs%2Fmain2.js&t=1526034802000" type="text/javascript">&lt;/script>
</head>
```

Final (Bundled), Single JavaScript File Contents

```
console.log("This is the main1.js file.")
console.log("This is the main2.js file.")
```

Configuring JavaScript

Portal properties contains many properties for JavaScript configuration, which you can see at

<https://github.com/liferay/liferay-portal/blob/7.1.x/portal-impl/src/portal.properties>. Look for the properties with the following prefixes:

- javascript
- minifier
- combo

Liferay JavaScript API

The Liferay JavaScript object is available on all the pages and contains helpful tools like:

- **Liferay.Browser**: browser capabilities detection
- **Liferay.ThemeDisplay**: ThemeDisplay object
- **Liferay.Service**: invokes Liferay services
- **Liferay.Language**: localization

An example of calling a Liferay service with Liferay.Service:

```
Liferay.Service(
  '/user/get-user-by-email-address',
  {
    companyId: Liferay.ThemeDisplay.getCompanyId(),
```

```
        emailAddress: 'test@liferay.com'  
    },  
    function(obj) {  
        console.log(obj);  
    }  
);  
);
```

Good to Know

Alloy UI and jQuery libraries are globally available, but the platform doesn't restrict you from using any other preferred library. Also, a full support for ES2015 as well as NPM modules is available as well as a configurable AMD loader. These topics are discussed more in detail in the Front-End Developer course.

Links and Resources

- **Alloy UI Validators**

[](https://github.com/liferay/alloy-ui/blob/master/src/aui-form-validator/js/aui-form-validator.js)

- **Using JavaScript in Portlets**

[](https://dev.liferay.com/develop/tutorials/-/knowledge_base/7-1/using-javascript-in-your-portlets)

- **Liferay taglibs:**

<https://docs.liferay.com/portal/7.1-latest/taglibs/util-taglib/>

Exercises

Create the Gradebook Web Module

Introduction

Over the next few exercise sets we will create the user interface for the Gradebook application. We will be using coding conventions and patterns recommended for Liferay development to leverage the high-level superclasses and components, removing the need for boilerplate portlet coding.

As a portlet component, we will use the Liferay MVC portlet. The portlet lifecycle and communication between the portlet back-end and user interface will be handled by MVC command components.

The user interface will be implemented with JSP technology. We will be using Liferay tag libraries, which both minimize the need for HTML coding but also guarantee a responsive layout.

In this exercise, we'll start by creating the gradebook-web module.

Overview

1 Create the Web Module

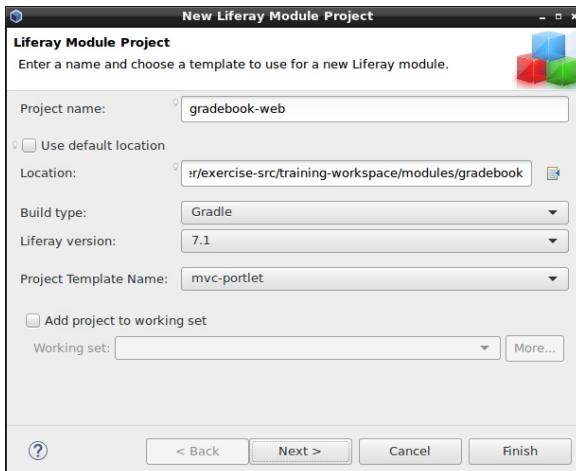
Snippets and resources for this exercise are in `snippets/chapter-06/04-create-the-presentation-layer` of your Liferay Workspace.

As before, if you get stuck, you can take a sneak peek from the **gradebook-solution** as to how the final gradebook has been completed. The solution is in `solutions/solutions-chapter-06/solution-06-gradebook`.

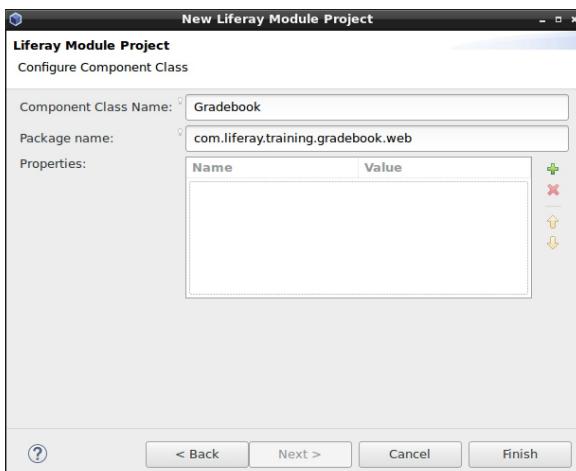
Create the Web Module

In the first step, we will create the MVC portlet module project from a template and deploy it to the portal to the hot deploy feature of DevStudio.

1. **Click** *File → New → Liferay Module Project* in DevStudio.
2. **Type** `gradebook-web` in the *Project Name* field.
3. **Uncheck** *Use default location*.
4. **Type** to set the *Location* to `modules/gradebook`.
5. **Choose** 7.1 in the *Liferay Version* field if it isn't already selected.
6. **Choose** `mvc-portlet` in the *Project Template Name* field if it isn't already selected.
7. **Click** *Next*.



8. Type **Gradebook** in the *Component Class Name*.
9. Type **com.liferay.training.gradebook.web** in the *Package Name* field.
10. Click **Finish** to close the wizard.



The following classes were automatically created by the template:

- **GradebookPortlet**: the portlet component
- **GradebookPortletKeys**: contains portlet name constants

Before deploying, let's change the portlet name. The portlet name serves as an identifier, and to avoid misspellings, it is a good practice to use a descriptive and easily identifiable name like the full package name for the portlet class. Also, constants are usually written in uppercase in Liferay development. The portlet name is referenced from a constant class *GradebookPortletKeys*, created by the template:

1. Go to **com.liferay.training.gradebook.web.constants.GradebookPortletKeys** in the *Project Explorer* panel.
 - **GradebookPortletKeys**

```
package com.liferay.training.gradebook.web.constants;

/**
 * @author liferay
 */
public class GradebookPortletKeys {

    public static final String Gradebook = "gradebook";
}
```

2. **Right-click** on the `Gradebook` variable name to open the context menu.
 - o You can select the variable name by double-clicking on it before right-clicking.
3. **Choose Refactor → Rename.**
4. **Type GRADEBOOK** and hit *Enter*.
5. **Replace** the variable value `gradebook` with `"com_liferay_training_gradebook_portlet_GradebookPortlet"`.
6. **Save** the class.

- o The class should now look like:

```
package com.liferay.training.gradebook.web.constants;

/**
 * @author liferay
 */
public class GradebookPortletKeys {

    public static final String GRADEBOOK =
        "com_liferay_training_gradebook_portlet_GradebookPortlet";

}
```

- o We don't want the Gradebook application to be instanceable as the data needs to be scoped under a site. Also, we'd like our application to appear in the *Liferay Training* applications category instead of *Sample*. Let's change portlet component properties to match that:
7. **Open** the `GradebookPortlet` class in the `com.liferay.training.gradebook.web.portlet`.
 8. **Edit** the properties as follows:

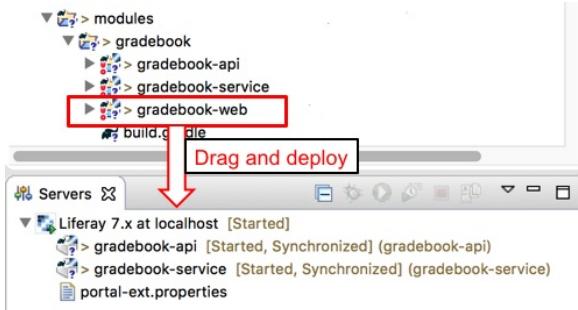
| Field | New value |
|---|------------------------------------|
| <code>com.liferay.portlet.instanceable</code> | <code>false</code> |
| <code>com.liferay.portlet.display-category</code> | <code>category.training</code> |
| <code>javax.portlet.display-name</code> | <code>gradebook-web Portlet</code> |

The portlet class should now look like:

```
@Component(
    immediate = true,
    property = {
        "com.liferay.portlet.display-category=category.training",
        "com.liferay.portlet.instanceable=false",
        "javax.portlet.display-name=gradebook-web Portlet",
        "javax.portlet.init-param.template-path=/",
        "javax.portlet.init-param.view-template=/view.jsp",
        "javax.portlet.name=" + GradebookPortletKeys.GRADEBOOK,
        "javax.portlet.resource-bundle=content.Language",
        "javax.portlet.security-role-ref=power-user,user"
    },
    service = Portlet.class
)
public class GradebookPortlet extends MVCPortlet { }
```

DevStudio's hot deploy feature allows us to see the changes on the user interface in almost real-time as we work with the code. Let's deploy our new web module, add it on a page, and do a quick test to see if the feature works:

1. **Check** the dependencies in `build.gradle`.
2. **Drag** the `gradebook-web` project onto the Liferay server to deploy the project.



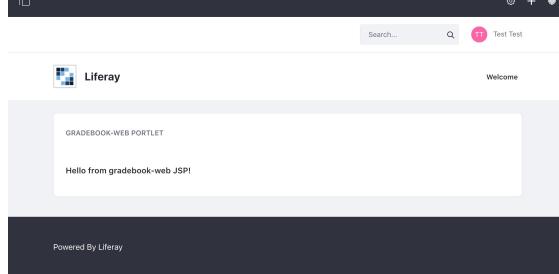
- In few moments, you should see a message on the console telling you that the module was started successfully:

```
STARTED com.liferay.training.gradebook.web_1.0.0
```

- Next, add the Gradebook portlet on a portal page:

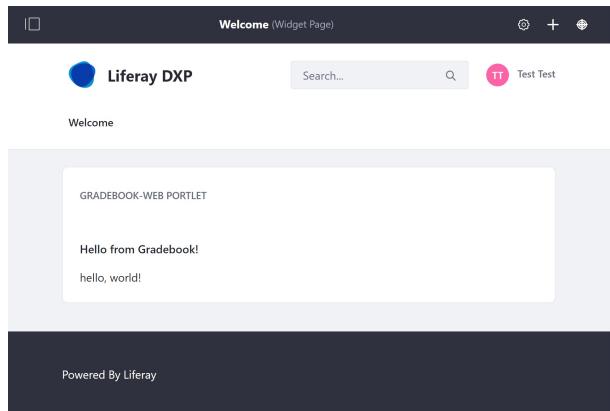
3. Open your browser and navigate to the portal home page at <http://localhost:8080>
4. Sign in to Liferay with test@liferay.com.
5. Click the Add button on the top right corner of the page.
6. Click Widgets to open the panel of available applications.
7. Click on the *category.training* category (we will localize the term later).
8. Drag the *gradebook-web Portlet* on the page.

- The portlet on a page should now look like this:



- Every time you do a change and save a file in the project, the module refreshes and changes are published to the server automatically.

9. Go to `src/main/resources/META-INF/resources` folder of the *gradebook-web* project.
10. Double-click the `view.jsp` file.
11. Type "hello, world!" into the `view.jsp`, under the closing `</p>` tag.
12. Save the file.
13. Refresh the page in your browser to see changes, after the console message shows the bundle has been restarted:



Exercises

Add Localization Resources

Introduction

In this exercise we'll be providing localization resources for our Gradebook-web module.

Overview

- ① Provide Localization Resources

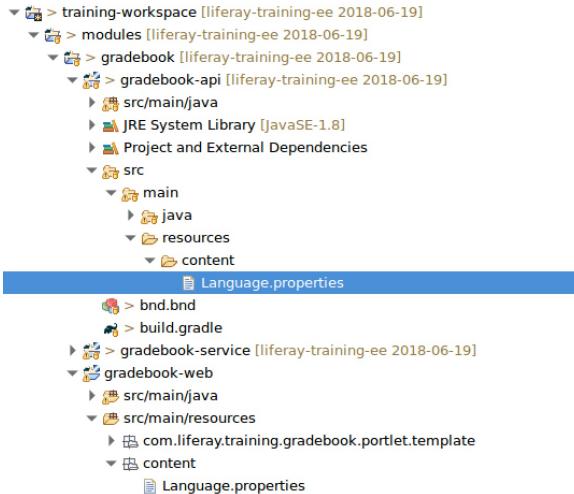
Snippets and resources for this exercise are in `snippets/04-create-the-presentation-layer`.

As before, if you get stuck, you can take a sneak peek from the **gradebook-solution** as to how the final gradebook has been completed. The solution is in `solutions/solutions-chapter-06/solution-06-gradebook`.

Provide Localization Resources

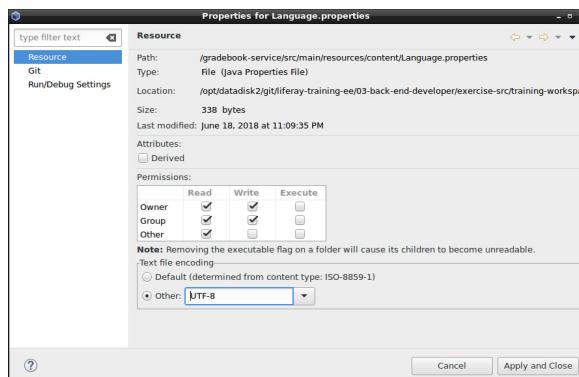
The internationalization design pattern makes your application easier to maintain by externalizing display terms from the actual code itself. In this step, we will create `Language.properties` files that provide English localizations for `gradebook-web` and `gradebook-api` modules. The API module has its own localization file for its configuration user interface, which we will create later during the exercises.

1. **Expand** the `snippets/04-create-the-presentation-layer/step-2` folder.
2. **Copy** the `gradebook-web/src/main/resources/content/Language.properties` file in the `step-2` folder.
3. **Paste** the file into the `src/main/resources/content` folder in the `gradebook-web` project.
4. **Click** Yes when prompted to overwrite the existing file.
5. **Right-click** on the `src/main` folder in the `gradebook-api` project to open the context menu.
6. **Choose** `New → Folder`.
7. **Type** `resources/content` in the folder name field.
8. **Click** `Finish`.
 - You may have to refresh Dev Studio (F5) before you are able to view new folders.
9. **Copy** the `gradebook-api/src/main/resources/content/Language.properties` file in the `step-2` folder.
10. **Paste** the file into the newly created `src/main/resources/content` folder in the `gradebook-api` project.



- Remember that the localization files should use UTF-8 encoding. Although the default ISO-8859-1 usually works for English, other localizations with non-standard ASCII special characters might have complications.
- Check the encoding of **both** Language.properties files:

- Right-click on the Language.properties file in the gradebook-api project.
- Click on Properties.
- Click on the Other radio button under the Text file encoding section.
- Choose UTF-8 from the drop-down menu.
- Click the Apply and Close button.
- Click Yes when prompted for confirmation.
- Repeat the above steps to update the encoding for the Language.properties file in the gradebook-web project.



Let's take a look at the localization file for the gradebook-web. It contains all the display messages used by the module and by the *Real World Application* exercise. In your own projects, if you ever wanted to add a display message to your module, reference it first by a key in the code and then add the mapping to the localization file.

```
#  
# This is the default localization file containing key to display messages mappings.  
#  
#  
# Standard portlet display messages:  
#  
javax.portlet.description.com_liferay_training_gradebook_portlet_GradebookPortlet=Gradebook  
javax.portlet.display-name.com_liferay_training_gradebook_portlet_GradebookPortlet=Gradebook  
javax.portlet.keywords.com_liferay_training_space_gradebook_portlet_GradebookPortlet=Gradebook  
javax.portlet.short-title.com_liferay_training_gradebook_portlet_GradebookPortlet=Gradebook  
javax.portlet.title.com_liferay_training_gradebook_portlet_GradebookPortlet=Gradebook
```

```

javax.portlet.display-name.com_liferay_training_gradebook_portlet_GradebookPortlet=Gradebook

#
# Application category
#
category.training=Liferay Training

#
# Asset name localization (Asset Publisher and search)
#
model.resource.com.liferay.training.gradebook.model.Assignment=Assignment

#
# Application Display Template localization
#
application-display-template-type=Gradebook template

#
# Permission localizations
#
action.ADD_SUBMISSION=Add Submission
action.DELETE_SUBMISSION=Delete Submission
action.EDIT_SUBMISSION>Edit Submission
action.GRADE_SUBMISSION=Grade Submissions
action.VIEW_SUBMISSIONS=View Submissions

#
# Other messages
#
add-assignment=Add New Assignment
add-submission=Add New Submission

```

Let's also check the reference to the localization file in the Gradebook portlet component.

1. [Open](#) the `com.liferay.training.gradebook.web.portlet.GradebookPortlet` class in the `gradebook-web` project.
2. [Find](#) the `javax.portlet.resource-bundle` property:

```

@Component(
    immediate = true,
    property = {
        "com.liferay.portlet.display-category=category.training",
        "com.liferay.portlet.instanceable=false",
        "javax.portlet.display-name=gradebook-web Portlet",
        "javax.portlet.init-param.template-path=/",
        "javax.portlet.init-param.view-template=/view.jsp",
        "javax.portlet.name=" + GradebookPortletKeys.GRADEBOOK,
        "javax.portlet.resource-bundle=content.Language",
        "javax.portlet.security-role-ref=power-user,user"
    },
    service = Portlet.class
)
public class GradebookPortlet extends MVCPortlet {
}

```

The value of the property is a path to `src/main/resources/content/Language.properties`. By setting this property, localization resources of the `Language.properties` are available for Liferay JSP tags and JavaScript libraries automatically. To add a support for another language, just add a new language properties file to the `src/main/resources/content` folder, for example `Language_de_DE.properties` to add a localization for German (as written and spoken in Germany).

Exercises

Implement Portlet Actions

Introduction

In this exercise, we'll be implementing the portlet's actions via MVC commands.

Overview

- ① Manually create and implement a `MVCRenderCommand`.
- ② Configure required compile time dependencies in `build.gradle`.
- ③ Copy and review prepared MVCCCommands and utility classes.

Snippets and resources for this exercise are in `snippets/04-create-the-presentation-layer`.

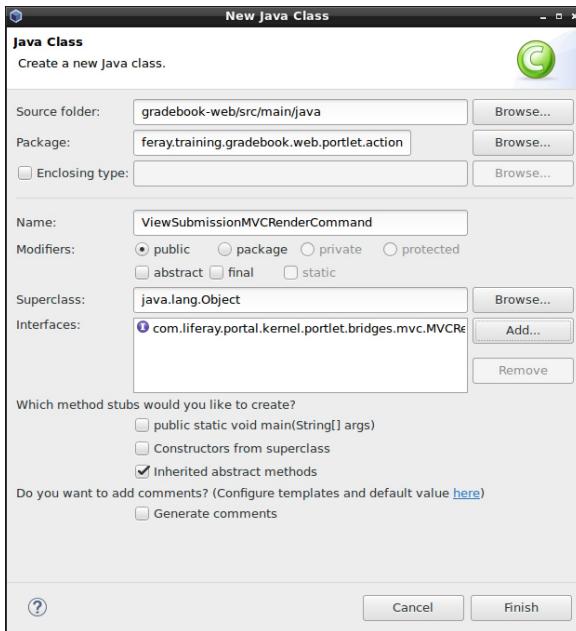
As before, if you get stuck, you can take a sneak peek from the **gradebook-solution** as to how the final gradebook has been completed. The solution is in `solutions/solutions-chapter-06/solution-06-gradebook`.

Implement Gradebook Portlet Actions Using MVC Commands

Next, we will prepare our portlet back-end classes for the user interface implementation. We will create MVC commands to take care of the portlet lifecycle and the interaction between the user interface and back-end. Three things wire an MVC command to the right action in the right place. First, MVC commands are wired to a certain portlet lifecycle (render, action, resource). Second, they respond to certain `mvc.command.name`, defined in component properties. Third, they register to a certain portlet with the `javax.portlet.name` component property. It's also useful to remember that a single MVC command can respond to multiple `mvc.command.name` and, on other hand, can register to multiple portlets. MVC commands are called from the user interface with corresponding portlet URLs, which are in JSP files typically generated by the standard `portlet` tag library. By convention, the MVC command classes are named by the following syntax, corresponding to the portlet lifecycle: `*MVCRenderCommand`, `*MVCACTIONCommand`, and `*MVCResourceCommand`. Commands are usually placed in the `*portlet.action` sub-package, for example: `com.liferay.training.gradebook.web.portlet.action`:

We will now create one action class for showing the submission view.

1. **Right-click** on the `gradebook-web` module.
2. **Choose** New → Class.
3. **Type** `com.liferay.training.gradebook.web.portlet.action` in the Package field.
4. **Type** `ViewSubmissionMVCRenderCommand` in the Name field.
5. **Click** on the Add button next to the Interfaces field.
6. **Type** MVC in the Choose interfaces field.
7. **Choose** the `MVCRenderCommand` interface.
8. **Click** Ok.
9. **Click** Finish to create the class and close the wizard.



- Next, let's make our class an OSGi component class.

10. Add a `@Component` annotation and properties as follows, just before the class declaration.

```
@Component(
    immediate = true,
    property = {
        "javax.portlet.name=" + GradebookPortletKeys.GRADEBOOK,
        "mvc.command.name=/gradebook/submission/view"
    },
    service = MVCRenderCommand.class
)
```

11. Press **CTRL + SHIFT + O** to use the *Organize Imports* to generate the imports automatically.

12. Save the file.

- The `render()` method should return a path to the JSP file. For now, just add the return path to the JSP file for showing a single submission.

13. Edit the `render()` method to return the following:

```
@Override
public String render(RenderRequest renderRequest, RenderResponse renderResponse)
    throws PortletException {

    return "/submission/view_submission.jsp";
}
```

- Before adding the rest of the commands, let's add the required dependencies for the Gradle:

14. Copy the `step-3/gradebook-web/build.gradle` from the exercise snippets folder.

15. Paste the file into the `gradebook-web` project.

16. Click Yes when prompted to overwrite.

17. Right-click on the `gradebook-web` project to open the context menu.

18. Choose `Gradle → Refresh Gradle Project` to refresh the project dependencies.

19. Review the `build.gradle`.

- Notice that we added dependencies for Application Display Templates. An Application Display Template will be discussed in detail later in this training, but we will be using them to render the initial assignments list view.

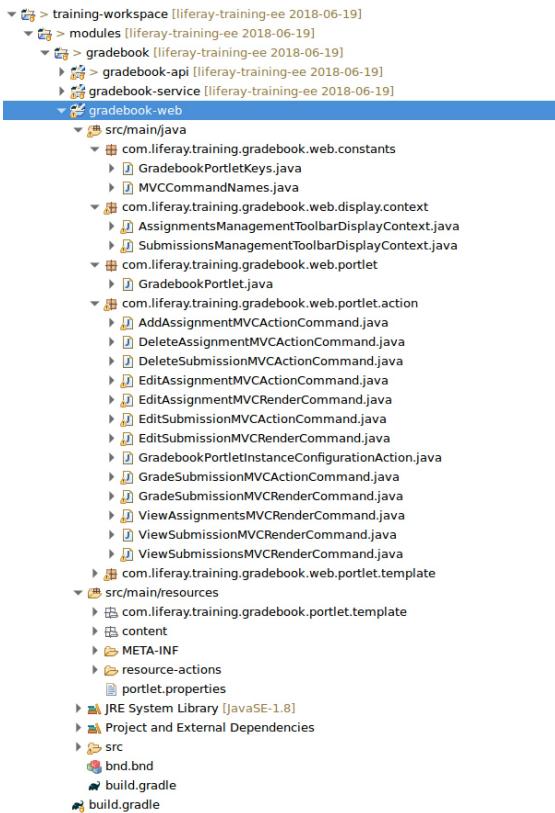
Maybe the most important thing to notice is the dependency to the *gradebook-api* project:

```
compile project(":modules:gradebook:gradebook-api")
```

Now all the packages exported by *gradebook-api* are automatically made available to the web-module by the Bndtools at build time. Now we have both dependencies and the service layer ready for our MVC commands.

In order to save time, let's copy the rest of the commands and the dependent classes:

1. **Copy** all classes (create packages as necessary) from the exercise snippets folder `step-3/gradebook-web/src/.../gradebook/web/` to the respective locations in the *gradebook-web* project.
2. **Click** Yes when prompted to overwrite.
3. **Copy** the snippets folder `step-3/gradebook-web/src/main/resources/com` to the respective place in *gradebook-web* project.
 - o This folder contains the default template for the Application Display Template renderer.
 - o Your project layout should now look like this:



4. **Open** the browser and refresh the page.
 - o Check that you can see the portlet.

Let's review what happened.

First, two kind of MVC commands were added to the *com.liferay.training.gradebook.web.portlet.action* package. There are *MVC render* commands for showing the different views and *MVC action* commands to respond to actions from the user interface.

You'll also notice that two classes in *com.liferay.training.gradebook.web.display.context* were created:

- *AssignmentsManagementToolbarDisplayContext*
- *SubmissionsManagementToolbarDisplayContext*

These are context POJOs for transporting values from the back-end to the user interface, used by the *Clay* tag library and removing the need to write business logic in the user interface JSP files. A constants class *MVCCCommandNames* was created to store the MVC command names to avoid misspellings.

Finally, a display template handler *com.liferay.training.gradebook.template.GradebookPortletDisplayTemplateHandler* was created. The Application Display Templates framework allows you to customize the user interface of an application directly with templates written in FreeMarker directly from the Control Panel and without a deployment process. We will discuss this topic in detail later in this training.

Exercises

Implement Portlet Permissions

Introduction

In this exercise we'll be implementing the portlet's permissions.

Overview

- 1 Implement Gradebook portlet permissions

Snippets and resources for this exercise are in `snippets/04-create-the-presentation-layer`.

As before, if you get stuck, you can take a sneak peek from the **gradebook-solution** as to how the final gradebook has been completed. The solution is in `solutions/solutions-chapter-06/solution-06-gradebook`.

Implement Gradebook Portlet Permissions

We already implemented permissioning support on the model layer, but the application layer has to be protected too. Portlet permissions define how and who can access the portlet. The process is similar to defining permissions on the model layer. We will just need to add `portlet.properties` and `default.xml` files. Notice that we are now not putting the `resource-actions` folder inside the `META-INF` to reserve that folder just for the user interface resources.

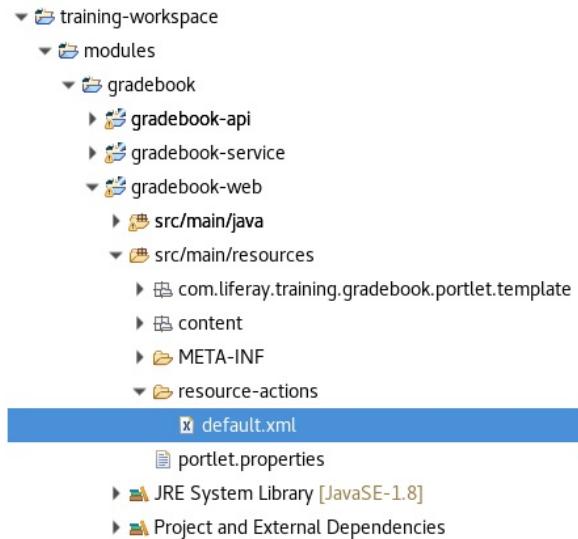
1. Go to the `src/main/resources` folder of *gradebook-web* module.
2. Right-click to open a context menu.
3. Choose *New → File*.
4. Type `portlet.properties` in the *File name* field.
5. Click *Finish*.
6. Type the following in the `portlet.properties`:

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

7. Save the file.

Let's create the `default.xml` file defining portlet resources and actions:

1. Right-click on the `src/main/resources` folder in the *gradebook-web* module.
2. Choose *New → Folder*.
3. Type `resource-actions` for the folder name.
4. Click *Finish*.
5. Copy the snippet `step-4/gradebook-web/src/main/resources/resource-actions/default.xml` into the newly created `resource-actions` folder.



6. Review the *default.xml*.

Exercises

Using JSP Tag Libraries

Introduction

In this exercise we'll implement the Gradebook-web's JSPs using tag libraries.

Overview

- 1 Implement Gradebook portlet JSPs using tag libraries

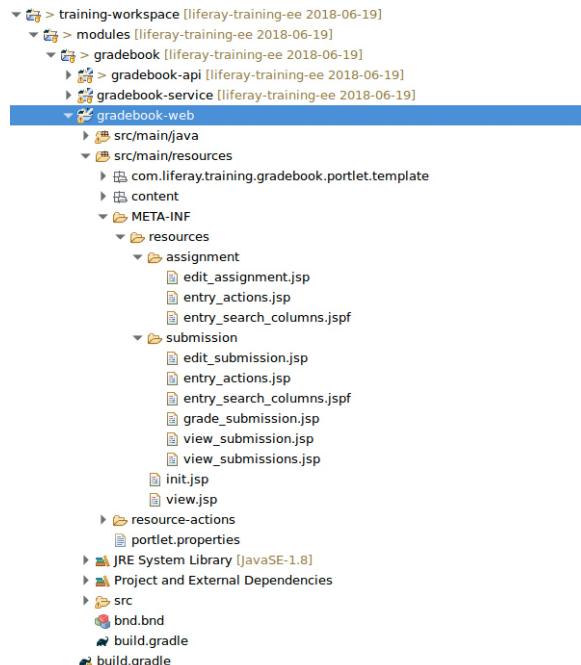
Snippets and resources for this exercise are in `snippets/04-create-the-presentation-layer`.

As before, if you get stuck, you can take a sneak peek from the **gradebook-solution** as to how the final gradebook has been completed. The solution is in `solutions/solutions-chapter-06/solution-06-gradebook`.

Implement Gradebook Portlet JSPs Using Tag Libraries

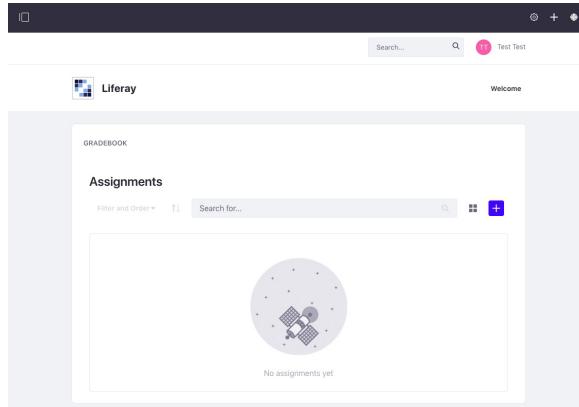
Now our portlet and service back-end are ready and we can implement the user interface. We implement the user interface using the JSP technology. Inside the JSPs, we heavily rely on tag libraries for producing the HTML. That way, we can automatically produce a responsive layout and avoid writing boilerplate code, making our user interface more maintainable at the same time. At this step, we will mostly use files provided to us. To get the most of this step, review the provided files and don't hesitate to ask your trainer if you have any questions.

1. **Copy** the contents of the snippets folder `step-5/gradebook-web/src/main/resources/META-INF/resources` in the respective location in the **gradebook-web** module.
2. **Click Yes To All** when prompted to overwrite existing files. Your project layout should now look like this:



If you open your browser and refresh the page with the Gradebook portlet, it should now have a complete user

interface that looks like this:



Note: You may see that an assignment already exists in the portlet. This is the assignment we created via web services in an earlier exercise. Delete this assignment via the services to avoid any issues going forward.

Let's take a look at some of the JSP files:

3. Open the `init.jsp`, which typically contains all taglib declarations, class imports, and common variable initializations. Take particular note of:

```
<liferay-frontend:defineObjects />
<liferay-theme:defineObjects />
<portlet:defineObjects />
```

These three tags initialize the required variables and objects, like the `themeDisplay` object for us.

4. Open the `view.jsp`. This is the default view the user gets when landing on the Gradebook portlet page. At the top of the page, we include the initialization JSP:

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

Then we have a block for feedback messages from the back-end. We will discuss those in the next exercise step:

```
<liferay-ui:success key="assignment-deleted" message="assignment-deleted-successfully" />
```

The rest of the page is taken by the Application Display Templates renderer tag:

```
<liferay-ddm:template-renderer
  className="${assignmentClassName}"
  displayStyle="${ADTdisplayStyle}"
  displayStyleGroupId="${ADTdisplayStyleGroupId}"
  entries="${assignments}">
  ...
</liferay-ddm:template-renderer>
```

We're going to discuss Application Display Templates later in the training, but what this tag effectively does is allow you to customize and replace the contents of this tag dynamically with your own FreeMarker scripts from within the Control Panel. This design not only makes the user interface much more customizable, but allows you to delegate administrative responsibilities in a real-world environment. What you see inside the renderer tag is the default content. There is a Liferay Clay management toolbar that creates the menu, with search and other

controls above the assignments list. Then there's a Liferay UI library search-container tag that takes care of the assignments list HTML creation. The search container tag is used frequently in all the lists in Liferay. See the documentation at <https://docs.liferay.com/ce/portal/7.1-latest/taglibs/util-taglib/liferay-ui/tld-summary.html>.

```
<liferay-ui:search-container
    emptyResultsMessage="no-assignments"
    id="assignmentEntries"
    iteratorURL="${portletURL}"
    total="${assignmentCount}">

    <liferay-ui:search-container-results results="${assignments}" />

    <liferay-ui:search-container-row
        className="com.liferay.training.gradebook.model.Assignment"
        modelVar="entry">

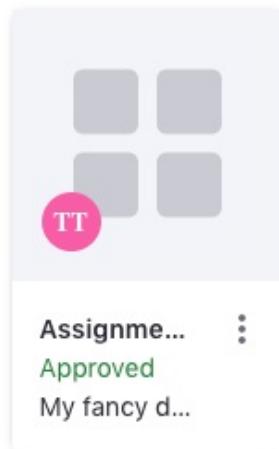
        <%@ include file="/assignment/entry_search_columns.jspf" %>

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

```

Let's now create our first assignment.

5. **Click** the *Add* button on the top right corner of the assignments list on the web page.
6. **Type** a title and description.
7. **Click** to set the due date to tomorrow's date.
8. **Click Save**. You should now see your assignment on the list. Before going to the next step, you can try out all the other controls of the Clay management toolbar, on the left side of the plus sign. Now we have a working user interface and the first assignment. Take a look at the assignment card on the list:



If you try to click on the assignment title to see the the assignment details, it doesn't work. There's a portlet render URL missing.

9. **Open** `assignment/entry_search_columns.jspf`. At the top of the file, you can see a placeholder:

```
<%-- [PLACEHOLDER FOR GENERATING SUBMISSIONS VIEW URL --%>
```

10. **Replace** the placeholder with:

```
<portlet:renderURL var="viewSubmissionsURL">
    <portlet:param name="mvcRenderCommandName" value="<%=MVCCommandNames.VIEW_SUBMISSIONS %>" />
    <portlet:param name="redirect" value="${currentURL}" />
    <portlet:param name="assignmentId" value="${entry.assignmentId}" />
</portlet:renderURL>
```

11. **Save** the file.
12. **Refresh** the page in your web browser. Wait for the module to restart, and refresh the web page. Now start to click the title again. You should be redirected to the submissions list page. Go ahead and add a new submission from the plus sign to test the functionality. Now when we have JSPs in place, it's time to get back to our still-incomplete MVC command for viewing a single submission. Let's complete the class and practice getting and setting request parameters and attributes as well as using the service layer:
13. **Open** `com.liferay.training.gradebook.web.portlet.action.ViewSubmissionMVCRenderCommand.java`. The class should look like this:

```

@Component(
    immediate = true,
    property = {
        "javax.portlet.name=" + GradebookPortletKeys.GRADEBOOK,
        "mvc.command.name=/gradebook/submission/view"
    },
    service = MVCRenderCommand.class
)
public class ViewSubmissionMVCRenderCommand implements MVCRenderCommand {

    @Override
    public String render(RenderRequest renderRequest, RenderResponse renderResponse)
        throws PortletException {

        return "/submission/view_submission.jsp";
    }
}

```

Before completing the `render()` method, we need to make some services available for the class by using the OSGi `@Reference` annotation:

14. **Type** the following at the bottom of the class, after the `render()` method. Remember to use *Organize Imports* (`CTRL+Shift+O`) to create imports, when necessary:

```

@Reference
protected AssignmentPermissionChecker _assignmentPermissionChecker;

@Reference
private AssignmentService _assignmentService;

@Reference
private SubmissionLocalService _submissionLocalService;

@Reference
private UserLocalService _userLocalService;

```

The JSP page is sending us the ID of the submission we should show. We can get the parameter using Liferay's `ParamUtil` class.

```
long submissionId = ParamUtil.getLong(renderRequest, "submissionId", 0);
```

Then we should call our service to get the submission. That will be done using the `SubmissionService`:

```

Submission submission =
    _submissionLocalService.fetchSubmission(submissionId);

```

We also have to get the assignment this submission belongs to. We can get the assignment ID from the submission just fetched and use the `AssignmentService` to get the assignment:

```

long assignmentId = submission.getAssignmentId();
Assignment assignment =
    _assignmentService.getAssignment(assignmentId);

```

1. **Type** to complete the `render()` as follows:

```

@Override
public String render(
    RenderRequest renderRequest, RenderResponse renderResponse)
throws PortletException {

    long submissionId = ParamUtil.getLong(renderRequest, "submissionId", 0);

    try {

        Submission submission =
            _submissionLocalService.fetchSubmission(submissionId);

        long assignmentId = submission.getAssignmentId();

        Assignment assignment =
            _assignmentService.getAssignment(assignmentId);
    }
}

```

Now we have our entities and only have to set the parameters to the request so that they will be available for the JSP pages.

1. **Add** the following to continue with completing the `render()` method. Be careful with attribute names. In a complex application, it would be a preferred practice to use a dedicated constants file for them:

```

...
DateFormat dateFormat = DateFormatFactoryUtil.getSimpleDateFormat(
    "EEEE, MMMMM dd, yyyy", renderRequest.getLocale());

// Set attributes to the request.

renderRequest.setAttribute("assignment", assignment);
renderRequest.setAttribute("submission", submission);
renderRequest.setAttribute("submissionClass", Submission.class);
renderRequest.setAttribute(
    "assignmentPermissionChecker", _assignmentPermissionChecker);
renderRequest.setAttribute(
    "createDate", dateFormat.format(submission.getCreateDate()));
renderRequest.setAttribute(
    "student", _userLocalService.getUser(
        submission.getStudentId()).getFullName());
renderRequest.setAttribute(
    "dueDate", dateFormat.format(assignment.getDueDate()));
...

```

We want to have a back link from our submission view page. That has to be told to the `PortletDisplay` object. Notice the `ThemeDisplay` object. `ThemeDisplay` is a container object for requesting context information. Typically, you'll use that a lot in Liferay's portlet development. `ThemeDisplay` provides, for example, information about the current user, group, and path.

2. **Finish** the class with the following code:

```

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

```

```

PortletDisplay portletDisplay = themeDisplay.getPortletDisplay();

String redirect = renderRequest.getParameter("redirect");

portletDisplay.setShowBackIcon(true);
portletDisplay.setURLBack(redirect);
...

```

Our class is now ready and should look like this:

```

@Component(
    immediate = true,
    property = {
        "javax.portlet.name=" + GradebookPortletKeys.GRADEBOOK,
        "mvc.command.name=/gradebook/submission/view"
    },
    service = MVCRenderCommand.class
)
public class ViewSubmissionMVCRenderCommand implements MVCRenderCommand {

    @Override
    public String render(
        RenderRequest renderRequest, RenderResponse renderResponse)
        throws PortletException {

        long submissionId = ParamUtil.getLong(renderRequest, "submissionId", 0);

        try {
            Submission submission =
                _submissionLocalService.fetchSubmission(submissionId);

            long assignmentId = submission.getAssignmentId();

            Assignment assignment =
                _assignmentService.getAssignment(assignmentId);

            DateFormat dateFormat = DateFormatFactoryUtil.getSimpleDateFormat(
                "EEEE, MMMMM dd, yyyy", renderRequest.getLocale());

            renderRequest.setAttribute("assignment", assignment);
            renderRequest.setAttribute("submission", submission);
            renderRequest.setAttribute("submissionClass", Submission.class);
            renderRequest.setAttribute(
                "assignmentPermissionChecker", _assignmentPermissionChecker);
            renderRequest.setAttribute(
                "createDate", dateFormat.format(submission.getCreateDate()));
            renderRequest.setAttribute(
                "student", _userLocalService.getUser(
                    submission.getStudentId()).getFullName());
            renderRequest.setAttribute(
                "dueDate", dateFormat.format(assignment.getDueDate()));

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

            PortletDisplay portletDisplay = themeDisplay.getPortletDisplay();

            String redirect = renderRequest.getParameter("redirect");

            portletDisplay.setShowBackIcon(true);
            portletDisplay.setURLBack(redirect);

            return "/submission/view_submission.jsp";
        }
        catch (PortalException e) {
            throw new PortletException(e);
        }
    }
}

```

```
}

@Reference
protected AssignmentPermissionChecker _assignmentPermissionChecker;

@Reference
private AssignmentService _assignmentService;

@Reference
private SubmissionLocalService _submissionLocalService;

@Reference
private UserLocalService _userLocalService;

}
```

1. **Save** the class. You can now test creating and viewing submissions in your browser.

Exercises

Implement Portlet Validation

Introduction

In this exercise, we'll be implementing portlet validation.

Overview

- 1 Implement Gradebook portlet validation and feedback

Snippets and resources for this exercise are in `snippets/04-create-the-presentation-layer`.

As before, if you get stuck, you can take a sneak peek from the **gradebook-solution** as to how the final gradebook has been completed. The solution is in `solutions/solutions-chapter-06/solution-06-gradebook`.

Implement Gradebook Portlet Validation and Feedback

When you created the assignment in the previous step, you may have noticed that no feedback was given. The new assignment just popped up on the list. Giving feedback is implemented in the provided back-end MVC command classes, but the user interface counterpart tag for showing the assignment creation message is missing.

1. Open the default view file `view.jsp` located in `source/main/resources/META-INF/resources` for the **gradebook-web** project.

- o You can see there are two placeholders:

```
<%-- [PLACEHOLDER FOR ASSIGNMENT ADDED SUCCESS MESSAGE --%>
<%-- [PLACEHOLDER FOR ASSIGNMENT ADDED UPDATED MESSAGE --%>
```

2. Replace the placeholders with `liferay-ui:message` tags, which are used to show messages set to the `SessionMessages` object in the back-end:

```
<liferay-ui:success key="assignment-added" message="assignment-added-successfully" />
<liferay-ui:success key="assignment-updated" message="assignment-updated-successfully" />
```

3. Save the file.

Now when you try to add and update an assignment again you should see a success message. Let's talk a little more about the validation.

1. Open the `assignment/edit_assignment.jsp` file, which is the view for editing assignments.

- o Take a look at the form and the title field:

```
<aui:input name="title">
<aui:validator name="required" />
<%-- Custom AUI validator. --%>
```

```

<aui:validator errorMessage="error.assignment-title-format" name="custom">
    function(val, fieldNode, ruleValue) {
        var wordExpression =
            new RegExp("^[^\\[\\]\\]$");
        return wordExpression.test(val);
    }
</aui:validator>
</aui:input>

```

There are two validators for the *title* field. The second one is for the title format. It doesn't, for example, allow the dollar (\$) character. If you master regular expressions, try to alter the validator and test it by creating assignments.

Remember that user interface validation is not really about security but more about usability: if you disable page JavaScripts, your security is gone. If you really want to secure your application, validation has to be done on the back-end. So where should you put the back-end validation, on the portlet controller layer or on the service layer?

Remember again that a service can be accessed from other modules besides your portlet module. There's also a web service API available. That's why putting validation in your portlet module won't necessarily protect your service, and it's better to do that on the service layer.

Let's first create a validation utility for validating assignments.

1. **Create** a new package `com.liferay.training.gradebook.service.validation` in the `gradebook-service` module.
2. **Copy** `step-6/src/main/java/com/liferay/training/gradebook/service/validation/AssignmentValidator.java` snippet to the package created previously.

Take a look at the class. We do just a basic validation, which you could improve with your own logic. You can also see an error in the class. We will fix that next.

When you validate on the service layer, you usually create *exceptions* that are sent to the controller, where they are possibly processed and dispatched to the user interface. When we created our service with Service Builder, we also defined two exception classes:

- `com.liferay.training.gradebook.exception.AssignmentValidationException`
- `com.liferay.training.gradebook.exception.SubmissionValidationException`

These classes were created automatically in the `gradebook-api` module.

When you do validation, it would be good if your validation class supported adding multiple messages in your exception so that you wouldn't have to write an exception for every possible error. These automatically created classes don't have a constructor for that, so let's add it.

1. **Open** the `com.liferay.training.gradebook.exception.AssignmentValidationException` class in the `gradebook-api` module.
2. **Add** the following code to the end of the class:

```

public AssignmentValidationException(List<String> errors) {
    super(String.join(", ", errors));
    _errors = errors;
}

public List<String> getErrors() {
    return _errors;
}

private List<String> _errors;

```

3. **Press** `CTRL+SHIFT+O` to resolve any missing imports.
4. **Repeat** the same for the `com.liferay.training.gradebook.exception.SubmissionValidationException`, just replacing the constructor name with `SubmissionValidationException`.
 - o Now our exception classes support adding multiple messages.
5. **Run** the `buildService` task to refresh the service and API.

Now, let's implement the actual validation on the service implementation classes. First, let's validate the assignments on creation and update time.

1. **Open** the `com.liferay.training.gradebook.service.impl.AssignmentLocalServiceImpl.java` class in the `gradebook-service` module and find the following placeholder in the `addAssignment()` method:

```
***** [PLACEHOLDER FOR ASSIGNMENT VALIDATION] *****
```

2. **Replace** the placeholder with:

```
AssignmentValidator.validate(titleMap, description, dueDate);
```

- o If necessary, use the *Organize Imports* (`CTRL+Shift+O`) to create the imports.
- 3. **Repeat** the same for the placeholder in the `updateAssignment()` method.

Let's also add a basic validation for submissions:

1. **Open** the `com.liferay.training.gradebook.service.impl.SubmissionLocalServiceImpl.java` class in the `gradebook-service` module and find the following placeholder in the `addSubmission()` method:

```
***** [PLACEHOLDER FOR SUBMISSION VALIDATION] *****
```

2. **Replace** the contents with:

```
validateSubmission(
    serviceContext.getCompanyId(), studentId, assignment,
    submissionText);
```

3. **Repeat** the same for the `updateSubmission()`, replacing the `studentId` with `submission.getStudentId()`:

```
validateSubmission(
    serviceContext.getCompanyId(), submission.getStudentId(), assignment,
    submissionText);
```

Now we have a basic validation on the service layer in place. As we have already implemented the messages on the user interface, there's only one more thing to do in this exercise: let's make the MVC commands to support our custom exception class.

The required code is already in place, but in comments. Take a look at the `doProcessAction()` method and the catch block (`AssignmentValidationException` and `SubmissionValidationException`) in the following classes in `gradebook-web`:

- `AddAssignmentMVCActionCommand.java`
- `EditAssignmentMVCActionCommand.java`
- `EditSubmissionMVCActionCommand.java`
- `GradeSubmissionMVCActionCommand.java`
- **Remove** the following comment from each of the classes above.

```
e.getErrors().forEach(key → SessionErrors.add(actionRequest, key));
```

Let's review what we've accomplished. You first customized the validation exception classes to support multiple messages. Then you implemented the validation in the service implementation classes in the add() and update() methods. Then you implemented reading multiple service error messages in MVC action commands. The messaging with *liferay-ui:message* tags in the JSP files was already in place.

Our application has now implemented validation and feedback both on the user interface and on the service layer.

Exercises

Add CSS Resources

Introduction

In this exercise, we'll add CSS resources to the portlet.

Overview

- 1 Add Gradebook CSS Resources

Snippets and resources for this exercise are in `snippets/04-create-the-presentation-layer`.

As before, if you get stuck, you can take a sneak peek from the **gradebook-solution** as to how the final gradebook has been completed. The solution is in `solutions/solutions-chapter-06/solution-06-gradebook`.

Add Gradebook CSS Resources

The default styling of the Gradebook portlet is good, but the author column styling of assignments list (table layout), for example, could be improved.

In this step, we will provide CSS resources for the Gradebook portlet. This topic is handled more in detail in the *Front-End Developer* training.

Let's first copy the provided CSS file in place:

1. **Copy** the `step-7/gradebook-web/src/main/resources/META-INF/resources/css` folder from the snippets.
2. **Paste** the folder to the respective location in the `gradebook-web` module.
 - Now the only thing to do is to tell the portlet component where the CSS resources are. This is done by defining the `com.liferay.portlet.header-portlet-css` property. We also want to wrap the portlet inside a CSS class `gradebook-portlet` to make styling more explicit.
3. **Open** the `GradebookPortlet` class from the `com.liferay.training.gradebook.web.portlet` package of the `gradebook-web` project.
4. **Add** the following properties in the component properties:

| Field | New value |
|---|--------------------------------|
| <code>com.liferay.portlet.css-class-wrapper</code> | <code>gradebook-portlet</code> |
| <code>com.liferay.portlet.header-portlet-css</code> | <code>/css/main.css</code> |

1. **Save** the class.

Wait for the module to hot deploy, and refresh and refresh the assignments list page in your browser

The Gradebook portlet properties should now look like this:

```
@Component(  
    immediate = true,  
    property = {  
        "com.liferay.portlet.display-category=category.training",  
        "com.liferay.portlet.css-class-wrapper=gradebook-portlet",  
        "com.liferay.portlet.header-portlet-css=/css/main.css",  
        "com.liferay.portlet.instanceable=false",  
        "javax.portlet.display-name=gradebook-web Portlet",  
        "javax.portlet.init-param.template-path=/",  
        "javax.portlet.init-param.view-template=/view.jsp",  
        "javax.portlet.name=" + GradebookPortletKeys.GRADEBOOK,  
        "javax.portlet.resource-bundle=content.Language",  
        "javax.portlet.security-role-ref=power-user,user"  
    },  
    service = Portlet.class  
)  
public class GradebookPortlet extends MVCPortlet {
```

You should see that the author column is now better aligned and that the links have underlining:

A screenshot of a web application interface titled 'GRADEBOOK'. Below it, a section titled 'Assignments' contains a table with two rows of data. The table has columns for 'Title', 'Author', 'Create Date', 'Status', and 'Actions'. The first row is for 'Assignment_2' with author 'Test Test', created '3 Hours Ago', status 'Approved', and an ellipsis '...' in the Actions column. The second row is for 'Assignment_1' with the same details. The 'Author' column is aligned to the right, and the 'Status' column contains a link labeled 'Approved' with blue underlining, indicating it is a clickable link.

| Title | Author | Create Date | Status | Actions |
|--------------|-----------|-------------|--------------------------|---------|
| Assignment_2 | Test Test | 3 Hours Ago | Approved | ⋮ |
| Assignment_1 | Test Test | 3 Hours Ago | Approved | ⋮ |

Exercises

Add JavaScript Resources

Introduction

In this exercise, we'll add JavaScript resources to the portlet.

Overview

- 1 Add Gradebook JavaScript Resources

Snippets and resources for this exercise are in `snippets/04-create-the-presentation-layer`.

As before, if you get stuck, you can take a sneak peek from the **gradebook-solution** as to how the final gradebook has been completed. The solution is in `solutions/solutions-chapter-06/solution-06-gradebook`.

Add Gradebook JavaScript Resources

Sooner or later, you usually have to add JavaScripts to your portlet module projects. Although this topic is beyond the Back-End Developer training topics and more thoroughly handled in the Front-End Developer training, we will demonstrate the basics. In this case, we will use an external, Asynchronous Module Definition (AMD) compatible JavaScript library to create a nice help tooltip for the submissions list view.

In this final presentation layer exercise step, we will:

- Create the AMD module loader configuration file
- Implement the JavaScript-based tooltip
- Add a portlet component property for the JavaScripts

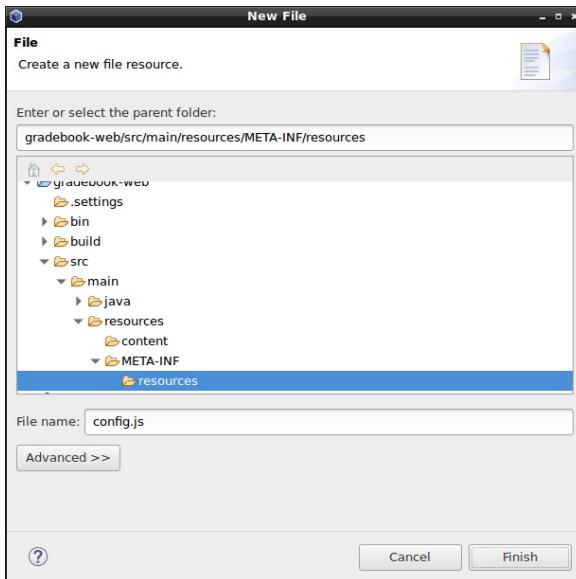
You can find more information about JavaScripts in Liferay in the Liferay Developer Network website.

Let's create the AMD loader configuration file and tell the Bndtools where to find the config file:

1. **Go to** the `src/main/resources/META-INF/resources` folder in the **gradebook-web** project.
2. **Right-click** on the `resources` folder to open the context menu.
3. **Choose** `New → File`.
4. **Type** `config.js` in the `File name` field.
5. **Click** `Finish` to close the wizard.
6. **Add** the following line to the `bnd.bnd` file of **gradebook-web** module:

```
Liferay-JS-Config: /META-INF/resources/config.js
```

7. **Save** the file.



- o Now let's define the 'tippy' tooltip module.
8. Type the following definition in the *config.js* and save the file.

```
Liferay.Loader.addModule({
    dependencies: [],
    exports: 'tippy',
    name: 'tippy',
    path: 'https://unpkg.com/tippy.js@2.5.2/dist/tippy.all.min.js'
});
```

- o Next, let's create our help tooltip element, where we want to show the tooltip:
9. Open the `src/main/resources/META-INF/resources/submission/view_submissions.jsp` file and find the following in the file:

```
<!-- PLACEHOLDER FOR HELP TOOLTIP -->
```

10. Replace the placeholder with the following:

```
<p>
    <a class="gradebook-tip" href="javascript:void(0);"
        title=<liferay-ui:message key="submissions-help-t
        ext" />>
        <liferay-ui:message key="help" />
        <clay:icon symbol="question-circle" />
    </a>
</p>
```

- o Now the module and the help element are available, and we will create the script to initialize the nice tooltips.
11. Create a subfolder `src/main/resources/META-INF/resources/js` in the *gradebook-web* module.
12. Create an empty file `main.js` in the `js` folder.
13. Type the following in the *main.js* and save the file:

```
AUI().ready(
    function() {
        Liferay.Loader.require('tippy', function(tippy) {
            tippy('.gradebook-tip', {
                hideOnClick: false
            });
        });
    }
);
```

What is happening here? `AUI().ready()`* callback function gets executed when all the HTML of the portlet is loaded.

In the callback function, we first load the `tippy` module, which we defined in `config.js` before, and then we run the `tippy` function to instantiate the tooltips, identified by `gradebook-tip` CSS class.

Only one more thing is still missing. We have to tell our portlet component that it should load the JavaScript resources from the `js` folder.

1. [Open](#) the portlet component `GradebookPortlet`.
2. [Add](#) the following component property:

| Field | New value |
|--|--------------------------|
| <code>com.liferay.portlet.footer-portlet-javascript</code> | <code>/js/main.js</code> |

The Gradebook portlet component properties should now look like:

```
@Component(
    immediate = true,
    property = {
        "com.liferay.portlet.display-category=category.training",
        "com.liferay.portlet.css-class-wrapper=gradebook-portlet",
        "com.liferay.portlet.header-portlet-css=/css/main.css",
        "com.liferay.portlet.footer-portlet-javascript=/js/main.js",
        "com.liferay.portlet.instanceable=false",
        "javax.portlet.display-name=gradebook-web Portlet",
        "javax.portlet.init-param.template-path=/",
        "javax.portlet.init-param.view-template=/view.jsp",
        "javax.portlet.name=" + GradebookPortletKeys.GRADEBOOK,
        "javax.portlet.resource-bundle=content.Language",
        "javax.portlet.security-role-ref=power-user,user"
    },
    service = Portlet.class
)
public class GradebookPortlet extends MVCPortlet {
```

1. [Save](#) the class.
2. [Open](#) the browser and navigate to the page with the Assignments portlet.
3. [Click](#) on an assignment to see the submissions list.
4. [Hover](#) your cursor over the `Help` text or the question mark icon.

You should see something like:

The screenshot shows a Liferay portlet titled "Assignment 1". Under "Assignment Information", it displays "Created" on Thursday, June 21, 2018, by "test". The "Assignment Due Date" is Friday, June 22, 2018. The "Description" is "My fancy description". A tooltip message at the bottom left says: "Please click the plus sign to add a new submission for an assignment. Notice that depending on the configuration, only one submission per assignment is allowed." A "Help" link with a question mark icon is located at the bottom right.

Make the Application Configurable

Introducing Liferay Configuration API

Liferay's configuration API is a configuration management framework for both the Liferay platform and applications on it. It is based on the OSGi configuration Admin service that allows you to dynamically set and manage configuration data for OSGi bundles and components. The configuration API is part of the OSGi Compendium specification.

Although standard portlet preferences can still be used, the configuration API, being more flexible and feature-rich, supersedes them as the preferred portlet application configuration framework. Features include:

- Key value pairs
- Value typing
- Scoping
- Programmatic and file-based management

As with portlet preferences, the data is stored in **key-value** pairs. With the configuration API, however, the values have strong typing. Values can have any Java types like:

- Integer
- String
- Long
- URL
- Collection
- Custom type

Also, similar to portlet preferences on the Liferay platform previously, the configuration can be scoped. The available scopes are:

- **System**: unique for the complete system
- **Virtual Instance**: instance-wide
- **Site**: configuration can vary per site
- **Portlet Instance**: single application (portlet) instance

When an application configuration is implemented using the configuration API, a **management user interface is generated automatically**. The management user interface is in Control Panel -> System Settings:

Configuration data can be exported and imported and is thus transportable. This is useful, for example, in replicating settings between environments. Importing configuration data can be done by copying the exported configuration file to the `Liferay Home / osgi / configs`-folder. It is also worth noting that, as the configuration API is using the standard OSGi configuration Admin service, the component runtime configuration data can be read with standard OSGi management tools like the Gogo Shell and Felix Web Console.

Generally, the minimum steps required for making a Liferay application configurable with the configuration API are as follows:

- ① Add required Bndlib dependencies to the build.gradle
- ② Add metatype instruction to bnd.bnd
- ③ Create the configuration interface
- ④ Make the configuration data available in an OSGi component

Example: Creating a System-Wide Portlet Configuration

Step 1 - Add Metatype Dependency

```
dependencies {
    ...
    // This is needed when using the ConfigurableUtil class or @ExtendedObjectClassDefinition annotation
    compileOnly group: "com.liferay", name: "com.liferay.portal.configuration.metatype.api", version: "1.0.0"
    ...
}
```

Step 2 - Add Metatype Bnd Instruction

This instructs the Bndtools to create a configuration XML file. You can check the file by extracting the generated JAR file and taking a look at the OSGI-INF folder. Notice that while this is good to know, you don't have to add this manually with Liferay 7.1 anymore:

```
-metatype: *
```

Step 3 - Create the Configuration Interface

Creating the configuration interface automatically creates the configuration user interface in the Control Panel -> System Settings.

Configuration *id* property **has to match** the interface fully qualified name.

```
package com.liferay.training.configuration;

import aQute.bnd.annotation.metatype.Meta;

@Meta.OCD(
    id = "com.liferay.training.configuration.ModuleConfiguration",
    localization = "content/Language",
    name = "configuration-api-example-portlet",
)
public interface ModuleConfiguration {

    @Meta.AD(
        deflt = "false",
        description = "show-hello-description",
        name = "show-hello-name",
        required = false
    )
    public boolean showHello();

}
```

Step 4 - Make the Configuration Data Available in an OSGi Component

Reference the configuration by id using the component property configurationPid. The configuration variable has to be volatile. After that, the configuration can be consumed, for example, by putting values into the request:

```

@Component(
    configurationPid = "com.liferay.training.configuration.ModuleConfiguration",
    immediate = true,
    property = {
        "com.liferay.portlet.display-category=category.sample",
        "com.liferay.portlet.instanceable=true",
        "javax.portlet.display-name=Make-application-configurable Portlet",
        "javax.portlet.init-param.template-path=/",
        "javax.portlet.init-param.view-template=/view.jsp",
        "javax.portlet.name=" + ConfigurationExamplePortletKeys.CONFIGURATION_EXAMPLE,
        "javax.portlet.resource-bundle=content.Language",
        "javax.portlet.security-role-ref=power-user,user"
    },
    service = Portlet.class
)
public class ConfigurationExamplePortlet extends MVCPortlet {

    @Activate
    @Modified
    protected void activate(Map<String, Object> properties) {

        _moduleConfiguration = ConfigurableUtil.createConfigurable(
            ModuleConfiguration.class, properties);
    }

    @Override
    public void render(
        RenderRequest renderRequest, RenderResponse renderResponse)
        throws IOException, PortletException {

        renderRequest.setAttribute("showHello", _moduleConfiguration.showHello());

        super.render(renderRequest, renderResponse);
    }

    private volatile ModuleConfiguration _moduleConfiguration;
}

```

Notes on the Annotations

@Meta.OCD and @Meta.AD are part of the bnd library, while @ObjectClassDefinition and @AttributeDefinition are OSGi core equivalents. Both can be used, but only bnd annotations are available at runtime.

Links and Resources

- **OSGi Compendium Specification**
<https://osgi.org/specification/osgi.cmpn/7.0.0/service.cm.html>

Exercises

Make the Gradebook Configurable

Introduction

In this exercise, we will create a Gradebook application configuration in the API module and consume that from both the service and web modules. The configuration will be scoped to be system-wide. We will learn how to use the `configurationPid` component property as well as the `ConfigurationProvider` in the Service Builder-generated class.

The configuration we will implement contains the following settings:

- Minimum and maximum length for submission text
- Whether a user is allowed to do multiple submissions for a single assignment

Overview

- ① Add required dependencies to the build.gradle of the API module
- ② Create the configuration interface in the API module
- ③ Consume configuration in the web module
- ④ Consume configuration in the service module
- ⑤ Deploy and test

Snippets and resources for this exercise are in `snippets/05-make-gradebook-configurable`.

For troubleshooting, the complete Gradebook application **solution** is in `solutions/solutions-chapter-06/solution-06-gradebook`.

Add Required Dependencies to the build.gradle of the API Module

1. **Open** the `build.gradle` of the `gradebook-api` module.
2. **Type** the following line in the dependencies section:

```
...  
compileOnly group: "com.liferay", name: "com.liferay.portal.configuration.metatype.api", version:  
"1.0.0"  
...
```

3. **Save** the file.
4. **Right-click** on the `gradebook-api` module.
5. **Choose** `Gradle → Refresh Gradle Project`.

Create the Configuration Interface in the API Module

Create the configuration interface.

1. **Right-click** on the `gradebook-api` project to open the context menu.
2. **Choose** `New → Package`.
3. **Type** `com.liferay.training.gradebook.configuration` in the `Name` field.

4. **Click** *Finish* to close the wizard.

5. **Copy** the `gradebook-`

```
api/src/main/java/com/liferay/training/gradebook/configuration/GradebookSystemServiceConfiguration.java
snippets to the package created previously.
```

Let's take a look at the interface. First, the naming of the interface uses the naming convention as follows: [Application][Scope][Layer]Configuration.

Take a look at the `@ExtendedObjectClassDefinition`. The `category` property defines the category where the configuration is located in the user interface. What's more important, however, is the `scope`. By default, the scope is system-wide. We are defining it here explicitly just for the sake of clarity. A common pattern for the configuration ID is the fully qualified name of the interface.

The `localization` property defines the resource bundle, which, if you remember, we created when implementing the presentation layer resource bundles. The value of the `name` property is a localization key.

```
@ExtendedObjectClassDefinition(
    category = "Gradebook",
    scope = ExtendedObjectClassDefinition.Scope.SYSTEM
)
@Meta.OCD(
    id = "com.liferay.training.gradebook.configuration.GradebookPortletInstanceConfiguration",
    localization = "content/Language",
    name = "gradebook-portlet-instance-configuration-name"
)
public interface GradebookPortletInstanceConfiguration {
```

Our interface does not have any methods yet. Every method we'll add there and annotate with `@Meta.AD` will be transformed into a configuration setting.

Let's add our settings.

1. **Add** the following method into the interface's body:

```
@Meta.AD(
    deflt = "10",
    description = "submission-min-length-description",
    name = "submission-min-length-name",
    required = false
)
public int submissionMinLength();

@Meta.AD(
    deflt = "200",
    description = "submission-max-length-description",
    name = "submission-max-length-name",
    required = false
)
public int submissionMaxLength();

@Meta.AD(
    deflt = "false",
    description = "allow-multiple-user-submissions-description",
    name = "allow-multiple-user-submissions-name",
    required = false
)
public boolean allowMultipleUserSubmissions();
```

2. **Save** the file.

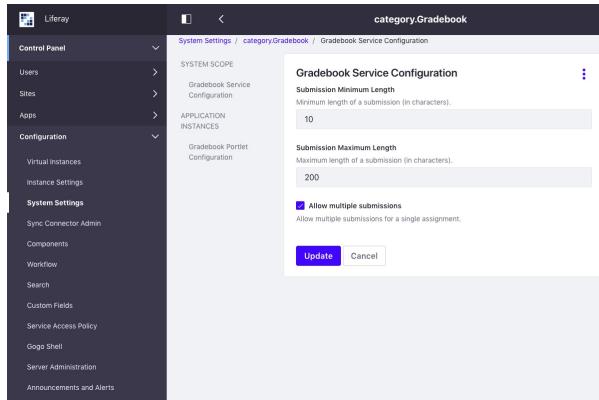
- o Next, we will expose the configuration package to the other modules.

3. Open the `bnd.bnd` file of the `gradebook-api` module.
4. Add the configuration package to the end of the list of Export-Packages as follows:

```
Export-Package: \
    com.liferay.training.gradebook.exception, \
    com.liferay.training.gradebook.model, \
    com.liferay.training.gradebook.service, \
    com.liferay.training.gradebook.service.persistence, \
    com.liferay.training.gradebook.service.permission, \
    com.liferay.training.gradebook.configuration
```

- Wait for the API module to refresh. Let's test our automatically created configuration user interface.
5. Open the browser and go to the *Control Panel* → *Configuration* → *System Settings*.
 - You should see the Gradebook configuration in the *Other* category. You can also search for *Gradebook*.
 6. Click the configuration icon.

You should see:



Next, we will implement reading the configuration in the web and service modules.

Consume Configuration in the Web Module

1. Open the `build.gradle` of the `gradebook-web` module.
2. Type the following line in the dependencies section:

```
...
compileOnly group: "com.liferay", name: "com.liferay.portal.configuration.metatype.api", version:
"1.0.0"
...
```

3. Right-click on the `gradebook-web` module.
4. Choose *Gradle* → *Refresh Gradle Project*.
 - Our user interface has to know whether the user is allowed to have multiple submissions for a single assignment. Adding submissions is done from the plus sign in the Clay management toolbar. The toolbar context is constructed in the `ViewSubmissionsMVCRenderCommand`.
5. Open `com.liferay.training.gradebook.web.portlet.action.ViewSubmissionsMVCRenderCommand` in the `gradebook-web` module.
6. Add the class variable holding the configuration to the end of the class, after the `userLocalService` variable.
 - Notice that the configuration variable **has** to be `volatile`.

```
protected volatile GradebookSystemServiceConfiguration _gradebookSystemServiceConfiguration;
```

- Use *Organize Imports* (`CTRL+Shift+O`) to create the import:

- Next, we have to instantiate or refresh the configuration when the component is activated or modified.
7. Add the following method to the class (after the `public render()` method and before the `private` methods. Sorting methods by scope and then alphabetically by name is the default pattern in Liferay development.):

```
@Activate
@Modified
protected void activate(Map<String, Object> properties) {

    _gradebookSystemServiceConfiguration = ConfigurableUtil.createConfigurable(
        GradebookSystemServiceConfiguration.class, properties);
}
```

- Now we can consume the configuration.
8. Find the following placeholder in the class:

```
***** [PLACEHOLDER FOR READING THE CONFIGURATION] *****
```

9. Replace the placeholder with:

```
isAllowMultipleSubmissions =
    _gradebookSystemServiceConfiguration.allowMultipleUserSubmissions();
```

10. Press `CTRL + SHIFT + O` to resolve imports.
 11. Add the `configurationPid` property to the `ViewSubmissionsMVCRenderCommand`'s component configuration:

```
...
@Component(
    configurationPid = "com.liferay.training.gradebook.configuration.GradebookSystemServiceConfiguration",
    immediate = true,
    property = {
        "javax.portlet.name=" + GradebookPortletKeys.GRADEBOOK,
        "mvc.command.name=" + MVCCmdNames.VIEW_SUBMISSIONS
    },
    service = MVCRenderCommand.class
)
...
```

- To avoid typos in the `configurationPid` property, copy the id configuration from the `GradebookSystemServiceConfiguration` interface (in the `gradebook-api` module) and change the property key from `id` to `configurationPid`.

You can test the configuration by changing the relevant setting in the Control Panel and check whether the plus sign is visible on the assignment submissions list page.

Consume Configuration in the Service Module

To consume the configuration from a Service Builder service implementation class, we have to create a *configuration bean declaration* component first.

1. Right-click on the `gradebook-service` module to open the context menu.
2. Choose `New → Class`.
3. Type `com.liferay.training.gradebook.configuration.definition` in the `Package` field.
4. Type `GradebookSystemServiceConfigurationBeanDeclaration` in the `Name` field.
5. Click `Finish` to close the wizard.
6. Type to implement the class as follows:

```
@Component
```

```
public class GradebookSystemServiceConfigurationBeanDeclaration  
    implements ConfigurationBeanDeclaration {  
  
    @Override  
    public Class<?> getConfigurationBeanClass() {  
  
        return GradebookSystemServiceConfiguration.class;  
    }  
}
```

- When you import (`CTRL+Shift+O`), be sure that you choose the `org.osgi.service.component.annotations.Component` for Component.
- If you choose the other, the component will not register correctly.

7. **Save** the class.
8. **Open** the `SubmissionLocalServiceImpl` class, where we consume the configuration in our validation method.
 - On the end of the class, you can see that the ConfigurationProvider is referenced already.
 - Notice the Spring `@ServiceReference` instead of the OSGi `@Reference` annotation. Remember that Service Builder services are Spring beans, although wired to the OSGi service registry.
9. **Find** the `validateSubmission()` method.
10. **Type** to uncomment the commented validation code.
11. **Press** `CTRL + SHIFT + O` to resolve imports.
12. **Save** the class.

Wait for the modules to refresh, and test the submission min and max length setting from your browser.

- For example, try to add submissions with fewer than 10 characters.

You may notice that we didn't implement the checking of multiple submissions.

Now the Gradebook application is configurable.

Integrate with Liferay Frameworks

In this section, we will discuss how to integrate your custom entities with the following Liferay frameworks:

- Asset Framework
- Search Framework
- Workflows

Integrate With Asset Framework

The Asset Framework is a Liferay platform framework that makes it possible to publish and manage any kind of content in a unified way and through a standard API. It provides ways of associating and linking content with, for example, other portal assets, tags, and categories, and makes it possible to integrate it with portal search and workflows.

The central concepts in integrating with the Asset Framework are:

- An asset
- Asset renderer
- Asset renderer factory

An Asset

An asset is an abstract, generic representation of any (model) entity. An asset is a wrapper class for the actual model entity, guaranteeing a certain set of metadata (fields) to the consuming applications and APIs, for example:

- Title, description, and date fields for the *Asset Publisher* portlet
- Entry class name for the portal search

To be able to show custom entities in the Liferay *Asset Publisher* portlet, they have to be made assets.

An Asset Renderer

Asset renderer is a class responsible for rendering the asset for display. It is a class implementing the `com.liferay.asset.kernel.model.AssetRenderer` interface.

The Asset renderer class provides the calling API access to the wrapped entity, which in the example below is the Gradebook Assignment:

```
@Override
public Assignment getAssetObject() {

    return _assignment;
}
```

The Asset renderer class is also responsible for rendering the URLs for viewing and editing an asset:

```
@Override
public String getURLViewInContext(
    LiferayPortletRequest liferayPortletRequest,
    LiferayPortletResponse liferayPortletResponse,
    String noSuchEntryRedirect)
throws Exception {

    try {
        long plid =
```

```

_assignmentAssetRendererFactory.getPortal().getPlidFromPortletId(
    _assignment.getGroupId(), GradebookPortletKeys.GRADEBOOK);

PortletURL portletURL;
if (plid == LayoutConstants.DEFAULT_PLID) {
    portletURL = liferayPortletResponse.createLiferayPortletURL(
        getControlPanelPlid(liferayPortletRequest),
        GradebookPortletKeys.GRADEBOOK,
        PortletRequest.RENDER_PHASE);
}
else {
    portletURL =
        _assignmentAssetRendererFactory.getPortletURLFactory().create(
            liferayPortletRequest, GradebookPortletKeys.GRADEBOOK,
            plid, PortletRequest.RENDER_PHASE);
}

portletURL.setParameter(
    "mvcRenderCommandName", MVCCmdNames.VIEW_SUBMISSIONS);
portletURL.setParameter(
    "assignmentId", String.valueOf(_assignment.getAssignmentId()));

...

```

And checking the entity permissions, when it's delivered, for example, through the Asset Publisher:

```

@Override
public boolean hasEditPermission(PermissionChecker permissionChecker)
    throws PortalException {

    return _assignmentAssetRendererFactory.assignmentPermissionChecker().contains(
        permissionChecker, _assignment.getGroupId(),
        _assignment.getAssignmentId(), ActionKeys.UPDATE);
}

@Override
public boolean hasViewPermission(PermissionChecker permissionChecker)
    throws PortalException {

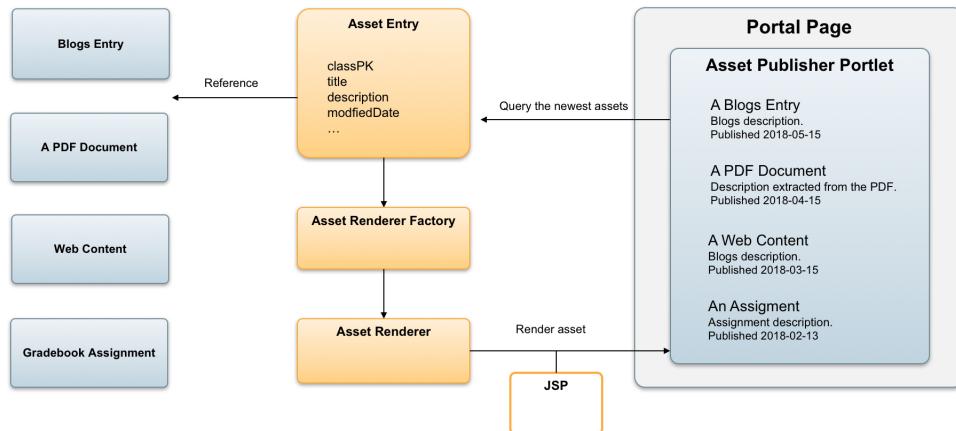
    return _assignmentAssetRendererFactory.assignmentPermissionChecker().contains(
        permissionChecker, _assignment.getGroupId(),
        _assignment.getAssignmentId(), ActionKeys.VIEW);
}

```

Asset Renderer Factory

The Asset Renderer factory is an OSGi component class that makes an Asset Renderer available to the calling application or API. The factory pattern makes it theoretically possible to have multiple renderers for a single asset type.

The diagram below illustrates and summarizes the components of the Asset Framework: On a portal page, there is an *Asset Publisher* portlet querying the newest assets. The Asset Framework gets a list of assets that contain the required set of metadata and references to the actual content items. When rendering the assets, the *Asset Publisher* portlet first finds the Asset Renderer Factory (OSGi service) for the model type and then asks for an Asset Renderer from the Asset Renderer factory. The Asset Renderer uses the data from the actual content item, wrapped by an asset, to render the item on the type specific JSP files provided. There are JSP files for different *Asset Publisher* views, like abstracts, full content, and table views.



Assets Seen From the Database

Below is an example of database table structures for the BlogsEntry and AssetEntry. You can see that the AssetEntry contains a subset of the data of BlogsEntry:

Blogs Entry Table

```

mysql> describe blogsentry;
+-----+-----+-----+-----+-----+
| Field          | Type       | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+
| _uuid_          | varchar(75) | YES  | MUL | NULL    |       |
| entryId        | bigint(20)  | NO   | PRI | NULL    |       |
| groupId         | bigint(20)  | YES  | MUL | NULL    |       |
| companyId       | bigint(20)  | YES  | MUL | NULL    |       |
| userId          | bigint(20)  | YES  |      | NULL    |       |
| userName        | varchar(75) | YES  |      | NULL    |       |
| createDate      | datetime(6) | YES  |      | NULL    |       |
| modifiedDate    | datetime(6) | YES  |      | NULL    |       |
| title           | varchar(150) | YES  |      | NULL    |       |
| subtitle         | longtext    | YES  |      | NULL    |       |
| urlTitle        | varchar(150) | YES  |      | NULL    |       |
| description      | longtext    | YES  |      | NULL    |       |
| content          | longtext    | YES  |      | NULL    |       |
| displayDate     | datetime(6) | YES  | MUL | NULL    |       |
| allowPingbacks  | tinyint(4)  | YES  |      | NULL    |       |
| allowTrackbacks | tinyint(4)  | YES  |      | NULL    |       |
| trackbacks       | longtext    | YES  |      | NULL    |       |
| coverImageCaption | longtext   | YES  |      | NULL    |       |
| coverImageFileEntryId | bigint(20) | YES  |      | NULL    |       |
| coverImageURL    | longtext    | YES  |      | NULL    |       |
| smallImage        | tinyint(4)  | YES  |      | NULL    |       |
| smallImageFileEntryId | bigint(20) | YES  |      | NULL    |       |
| smallImageId      | bigint(20)  | YES  |      | NULL    |       |
| smallImageURL    | longtext    | YES  |      | NULL    |       |
| lastPublishDate   | datetime(6) | YES  |      | NULL    |       |
| status            | int(11)     | YES  |      | NULL    |       |
| statusByUserId    | bigint(20)  | YES  |      | NULL    |       |
| statusByUserName   | varchar(75) | YES  |      | NULL    |       |
| statusDate        | datetime(6) | YES  |      | NULL    |       |
+-----+-----+-----+-----+-----+
29 rows in set (0,01 sec)
  
```

Asset Entry Table

```

mysql> describe assetentry;
+-----+-----+-----+-----+-----+
| Field          | Type       | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+
  
```

```
+-----+-----+-----+-----+-----+
| entryId | bigint(20) | NO   | PRI | NULL    |       |
| groupId | bigint(20) | YES  | MUL | NULL    |       |
| companyId | bigint(20) | YES  | MUL | NULL    |       |
| userId  | bigint(20) | YES  |      | NULL    |       |
| userName | varchar(75) | YES  |      | NULL    |       |
| createDate | datetime(6) | YES  |      | NULL    |       |
| modifiedDate | datetime(6) | YES  |      | NULL    |       |
| classNameId | bigint(20) | YES  | MUL | NULL    |       |
| classPK   | bigint(20) | YES  |      | NULL    |       |
| classUuid  | varchar(75) | YES  |      | NULL    |       |
| classTypeId | bigint(20) | YES  |      | NULL    |       |
| listable   | tinyint(4)  | YES  |      | NULL    |       |
| visible    | tinyint(4)  | YES  | MUL | NULL    |       |
| startDate  | datetime(6) | YES  |      | NULL    |       |
| endDate    | datetime(6) | YES  |      | NULL    |       |
| publishDate | datetime(6) | YES  | MUL | NULL    |       |
| expirationDate | datetime(6) | YES  | MUL | NULL    |       |
| mimeType   | varchar(75) | YES  |      | NULL    |       |
| title      | longtext   | YES  |      | NULL    |       |
| description | longtext   | YES  |      | NULL    |       |
| summary    | longtext   | YES  |      | NULL    |       |
| url        | longtext   | YES  |      | NULL    |       |
| layoutUuid | varchar(75) | YES  | MUL | NULL    |       |
| height     | int(11)    | YES  |      | NULL    |       |
| width      | int(11)    | YES  |      | NULL    |       |
| priority   | double     | YES  |      | NULL    |       |
| viewCount  | int(11)    | YES  |      | NULL    |       |
+-----+-----+-----+-----+-----+
27 rows in set (0,00 sec)
```

Below is an example of how a blogs entry instance is referenced in an asset entry: the `entryId` of a blogs entry is the class primary key, `classPK` (Class Primary Key) value in the corresponding asset entry:

Blogs Entry in the BlogsEntry Table

```
mysql> select entryId, groupId, companyId, userId, userName, title from blogsentry where entryId=63341;
+-----+-----+-----+-----+-----+
| entryId | groupId | companyId | userId | userName   | title          |
+-----+-----+-----+-----+-----+
| 63341  | 47971  | 20115  | 20155 | Liferay Demo | New Great Blog Entry |
+-----+-----+-----+-----+-----+
1 row in set (0,00 sec)
```

The corresponding AssetEntry in the AssetEntry Table

```
mysql> select entryId, classPK, groupId, companyId, userId, userName, title from assetentry where classPK=63341;
+-----+-----+-----+-----+-----+-----+
| entryId | classPK | groupId | companyId | userId | userName   | title          |
+-----+-----+-----+-----+-----+-----+
| 63342  | 63341  | 47971  | 20115  | 20155 | Liferay Demo | New Great Blog Entry |
+-----+-----+-----+-----+-----+
1 row in set (0,00 sec)
```

Asset entry fields and their descriptions:

- **userId**: the user updating the content
- **groupId**: the scope group of the created content
- **createDate**: the date the entity was created

- **modifiedDate**: the date the entity was last modified
- **className**: identifies the entity's class
- **classPK**: the primary key of the model entity
- **classUuid**: a secondary identifier that's guaranteed to be universally unique
- **classTypeId**: identifies the particular variation of this class (if any, default 0)
- **categoryIds**: the asset category ids for the entity
- **tagNames**: tag names for the entity
- **listable**: specifies whether the entity can be shown in dynamic lists of content (Asset Publisher)
- **visible**: specifies whether the entity is approved
- **startDate**: when the entity should be visible
- **endDate**: when the entity should stop being visible
- **publishDate**: the date the entity will be published (visible)
- **expirationDate**: the date the entity will be archived (not visible)
- **mimetype**: the Mime type such as ContentTypes.TEXT_HTML of the content
- **title**: the entity's name
- **description**: a String-based textual description of the entity
- **summary**: a shortened or truncated sample of the entity's content
- **url**: a URL to optionally associate with the entity
- **layoutUuid**: the universally unique ID of the layout of the entry's default display page
- **height**: this can be set to 0
- **width**: this can be set to 0
- **priority**: specifies how the entity is ranked among peer entity instances; the lower numbers take priority

The Benefits of Integrating the Asset Framework

To leverage most of the Liferay platform native features, a custom entity must be integrated into the Asset Framework. Integration allows you to:

- Show custom entities in the Asset Publisher portlet
- Associate tags and categories
- Associate comments and ratings
- Integrate to portal search
- Integrate to portal workflows
- Enable staging on the entities
- Link assets to each other
- Assign social bookmarks like Facebook likes to the entity
- Add custom fields (Liferay Expando API)
- Track the number of times an asset is viewed
- Implement Recycle Bin support

How to Integrate the Asset Framework

Generally, the steps to integrate with the Asset Framework are as follows:

- ① Add the required fields, references, and finders to the model entity definitions
- ② Manage asset resources (usually in the CRUD methods on the service layer)
- ③ Create an Asset Renderer factory for the model entities
- ④ Create an Asset Renderer for the model entities
- ⑤ Implement the JSP files to support the different display modes of the Asset Publisher

Example: Integrating a Custom Entity to the Asset Framework

Step 1 - Add the Required Fields and References

Ensure that the custom service.xml defines the following fields:

- userId
- userName
- status
- statusByUserId
- statusByUserName
- statusDate
- createDate
- modifiedDate

The entity's `uuid` attribute should be set to true to guarantee global uniqueness, for example, in a staged environment.

A reference to `AssetEntry` has to be made to bring the corresponding service available and inside the same transaction boundary.

A status finder is also needed for the workflow support:

service.xml

```

<?xml version="1.0"?>
<!DOCTYPE service-builder PUBLIC "-//Liferay//DTD Service Builder 7.0.0//EN" "http://www.liferay.com/dtd/liferay-service-builder_7_0_0.dtd">

<service-builder package-path="com.liferay.training.gradebook">
<namespace>GradeBook</namespace>
<entity name="Assignment"
       uuid="true"
       local-service="true"
       remote-service="true"
       >

    <!-- PK fields -->
    <column name="assignmentId" type="long" primary="true" />

    <!-- Group instance -->
    <column name="groupId" type="long" />

    <!-- Audit fields -->
    <column name="companyId" type="long" />
    <column name="userId" type="long" />
    <column name="userName" type="String" />
    <column name="createDate" type="Date" />
    <column name="modifiedDate" type="Date" />

    <!-- Status fields -->
    <column name="status" type="int" />
    <column name="statusByUserId" type="long" />
    <column name="statusByUserName" type="String" />
    <column name="statusDate" type="Date" />

    <!-- Permission fields -->
    <!-- Other fields -->

    <column name="title" type="String" localized="true" />

```

```

<column name="description" type="String" />
<column name="dueDate" type="Date" />

<!-- Order -->

<order by="asc">
    <order-column name="title" order-by="asc" />
</order>

<!-- Finder methods -->

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

<finder name="Status" return-type="Collection">
    <finder-column name="status" />
</finder>
<finder name="G_S" return-type="Collection">
    <finder-column name="groupId" />
    <finder-column name="status" />
</finder>

<!-- References -->

<reference package-path="com.liferay.portlet.asset"
    entity="AssetEntry" />
<reference package-path="com.liferay.portlet.asset"
    entity="AssetTag" />
<reference package-path="com.liferay.portlet.asset"
    entity="AssetLink" />
<reference package-path="com.liferay.portal"
    entity="Group" />
</entity>

...

```

Step 2 - Manage Asset Resources

Whenever you modify (create, update, or delete) the custom model entity, the corresponding asset entry has to be updated correspondingly.

In the example below, taken from the native Blogs application, there is a method for adding blogs entries. In the end of the method, you can see the call to update the asset:

```

@Indexable(type = IndexableType.REINDEX)
@Override
public BlogsEntry addEntry(
    long userId, String title, String subtitle, String urlTitle,
    String description, String content, Date displayDate,
    boolean allowPingbacks, boolean allowTrackbacks,
    String[] trackbacks, String coverImageCaption,
    ImageSelector coverImageImageSelector,
    ImageSelector smallImageImageSelector,
    ServiceContext serviceContext)
throws PortalException {

    // Entry

    User user = userLocalService.getUser(userId);
    long groupId = serviceContext.getScopeGroupId();
    int status = WorkflowConstants.STATUS_DRAFT;

    validate(title, urlTitle, content, status);

    ...

```

```
// Asset

updateAsset(
    userId, entry, serviceContext.getAssetCategoryIds(),
    serviceContext.getAssetTagNames(),
    serviceContext.getAssetLinkEntryIds(),
    serviceContext.getAssetPriority());
```

Correspondingly, in the delete entry method, the asset entry is removed:

```
@Indexable(type = IndexableType.DELETE)
@Override
@SystemEvent(type = SystemEventConstants.TYPE_DELETE)
public BlogsEntry deleteEntry(BlogsEntry entry) throws PortalException {

    // Entry

    blogsEntryPersistence.remove(entry);

    // Resources

    resourceLocalService.deleteResource(
        entry.getCompanyId(), BlogsEntry.class.getName(),
        ResourceConstants.SCOPe_INDIVIDUAL, entry.getEntryId());

    ...

    // Asset

    assetEntryLocalService.deleteEntry(
        BlogsEntry.class.getName(), entry.getEntryId());
    ...
}
```

Step 3 - Create an Asset Renderer Factory

An Asset Renderer factory is an OSGi service component for the *AssetRendererFactory* interface, extending the *BaseAssetRendererFactory* class. Wiring to the specific model is made with the Generics type and wiring the component to the portlet with the *javax.portlet.name* property:

```
@Component(
    immediate = true, property = "javax.portlet.name=" + BlogsPortletKeys.BLOGS,
    service = AssetRendererFactory.class
)
public class BlogsEntryAssetRendererFactory
    extends BaseAssetRendererFactory<BlogsEntry> {

    public static final String TYPE = "blog";

    public BlogsEntryAssetRendererFactory() {
        setClassName(BlogsEntry.class.getName());
        setLinkable(true);
        setPortletId(BlogsPortletKeys.BLOGS);
        setSearchable(true);
    }
    ...
}
```

Step 4 - Create an Asset Renderer

The next thing to do is to implement an Asset Renderer:

```
public class BlogsEntryAssetRenderer
    extends BaseJSPAssetRenderer<BlogsEntry> implements TrashRenderer {
```

```

public BlogsEntryAssetRenderer(
    BlogsEntry entry, ResourceBundleLoader resourceBundleLoader) {

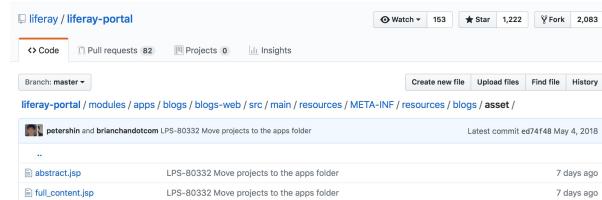
    _entry = entry;
    _resourceBundleLoader = resourceBundleLoader;
}
...

```

For detailed instructions and samples of how to implement an Asset Renderer, see the Liferay Developer Network tutorial at https://dev.liferay.com/develop/tutorials/-/knowledge_base/7-1/rendering-an-asset

Step 5 - Implement the JSP files to support the Asset Publisher

In the final steps, the JSP files for displaying the Asset in different Asset Publisher views are created, as necessary. Typically, abstracts and full content view JSPs are created:



Integrate with the Search Framework

The Liferay platform uses an external search engine, which is by default Elasticsearch. When Liferay content items are indexed to the search engine, they have to be transformed to search engine documents. This conversion is done by model entity type specific indexers.

How to Make a Custom Entity Searchable

General steps for making an entity searchable:

- ① Create an Indexer component for your entity
- ② Update service layer to refresh index on entity modification event
- ③ If needed, create a custom search user interface

Step 1 - Create an *Indexer Component for Your Entity

An indexer is an OSGi component extending the `com.liferay.portal.kernel.search.BaseIndexer` class. In the service builder projects, this component is typically in the service module of an application. Below is an excerpt of an indexer:

```

@Component(
    immediate = true,
    service = Indexer.class
)
public class AssignmentIndexer extends BaseIndexer<Assignment> {
    public AssignmentIndexer() {
        setDefaultSelectedFieldNames(
            Field.COMPANY_ID, Field.ENTRY_CLASS_NAME, Field.ENTRY_CLASS_PK,
            Field.UID, Field.DESCRIPTION);
        setDefaultSelectedLocalizedFieldNames(Field.TITLE);
        setFilterSearch(true);
        setPermissionAware(true);
    }
}

```

```

@Override
public String getClassName() {
    return CLASS_NAME;
}
...

```

Step 2 - Update the Service Layer to Refresh Index on Entity Modification Event

It is possible to call your entity indexer directly, but when you use Service Builder, the Liferay-provided method level `@Indexable` annotation automates the task for you. Just annotate those methods in the local service implementation classes, which should trigger an index update. The only requirement for the methods is that they have to return the target entity. The `@Indexable` annotation has two action types:

- **REINDEX**: on add or update
- **DELETE**: on entity delete

Below is an example from a Service Builder project implementation class, where an index document gets created on entity add:

```

@Indexable(
    type = IndexableType.REINDEX
)
public Assignment addAssignment(
    long groupId, Map<Locale, String> titleMap, String description,
    Date dueDate, ServiceContext serviceContext)
throws PortalException {

    AssignmentValidator.validate(titleMap, description, dueDate);
    ...

    return assignment;
}

```

Accordingly the index document gets deleted when the entity is deleted:

```

@Indexable(
    type = IndexableType.DELETE
)
public Assignment deleteAssignment(Assignment assignment)
throws PortalException {

    ...
    return super.deleteAssignment(assignment);
}

```

Step 3 - If Necessary, Create a Custom Search User Interface

To get your custom entities to show up in standard portal search, you **have to** integrate with the Liferay Asset Framework. This will be discussed in the next section.

If you, however, would like to create your own custom search user interface, you can call the extensive portal search API directly.

In an example scenario, you would:

1. Send the query parameters from the user interface to the back-end.
2. Catch the parameters and build a `SearchContext` object that transports all the required information to the search

engine adapter.

3. Call the *IndexSearcherHelper* service and execute the search.
4. Get the *Hits* response objects back and format the hits for your user interface.

In the center of communicating with the portal search API is the *SearchContext* object. Below is an example excerpt of populating the object and executing the search:

```
...
SearchContext searchContext = new SearchContext();
searchContext.setCompanyId(themeDisplay.getCompanyId());
searchContext.setStart(_queryParams.getStart());
searchContext.setEnd(_queryParams.getEnd());
searchContext.setSorts(_queryParams.getSorts());

searchContext.setBooleanClauses(new BooleanClause[] {
    myBooleanClause
});

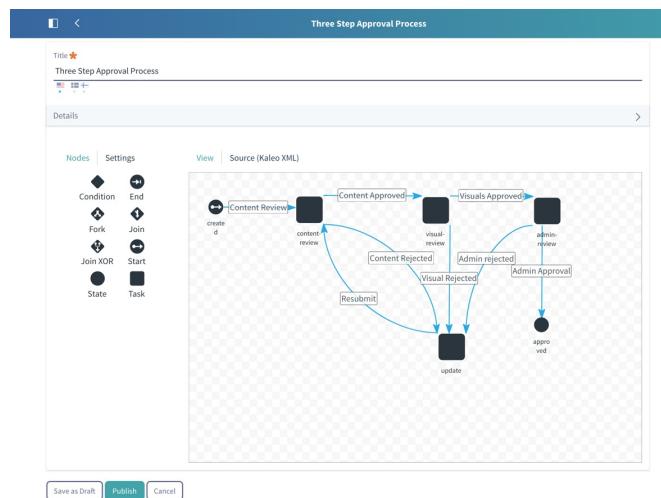
Hits hits = _indexSearcherHelper.search(searchContext, query);
...
```

Integrate with Workflows

A workflow is an orchestrated and repeatable pattern or sequence of operations. Typically, they are used in review processes but often also in integrating to other systems.

In Liferay, any registered asset can be assigned to a workflow. For handling the workflows, Liferay is using its own Kaleo workflow engine, but integrations to other workflow engines are available by third parties.

For designing the workflows, there is a workflow editor available in the Control Panel -> System Settings.



Here are the steps for enabling workflows for a custom entity:

- ① Ensure that the model entity has status fields.
- ② Add workflow instance creation and deletion handling to the service layer.
- ③ Handle status in the getter methods of the service layer (as parameters or explicitly).

- ④ Create a workflow handler component and an update status method on the service layer.
- ⑤ If needed, modify the user interface to show workflow status.

class: bd-subsection bd-demonstration

Chapter 6 Develop a Real-World Application Integrate with Liferay Frameworks

Example: Blogs Portlet Integration with Workflow

Step 1 - Ensure that the Model Entity has Status Fields

service.xml

```
<?xml version="1.0"?>
<!DOCTYPE service-builder PUBLIC "-//Liferay//DTD Service Builder 7.1.0//EN" "http://www.liferay.com/dtd/liferay-service-builder_7_1_0.dtd">

<service-builder auto-import-default-references="false" auto-namespace-tables="false" package-path="com.liferay.blogs">
    <namespace>Blogs</namespace>
    <entity local-service="true" name="BlogsEntry" remote-service="true" trash-enabled="true" uuid="true">

        ...

        <column name="status" type="int" />
        <column name="statusByUserId" type="long" />
        <column name="statusByUserName" type="String" uad-anonymize-field-name="fullName" />
        <column name="statusDate" type="Date" />
    </entity>
</service-builder>
```

<https://github.com/liferay/liferay-portal/blob/7.1.x/modules/apps/blogs/blogs-service/service.xml>

Step 2 - Add Workflow Instance Creation and Deletion Handling to the Service Layer

Workflow instance creation is done when an instance is created:

com.liferay.blogs.service.impl.BlogsEntryLocalServiceImpl

```
@Indexable(type = IndexableType.REINDEX)
@Override
public BlogsEntry addEntry(
    long userId, String title, String subtitle, String urlTitle,
    String description, String content, Date displayDate,
    boolean allowPingbacks, boolean allowTrackbacks,
    String[] trackbacks, String coverImageCaption,
    ImageSelector coverImageImageSelector,
    ImageSelector smallImageImageSelector,
    ServiceContext serviceContext)
throws PortalException {
    ...

    return startWorkflowInstance(userId, entry, serviceContext);
}
```

<https://github.com/liferay/liferay-portal/blob/7.1.x/modules/apps/blogs/blogs-service/src/main/java/com/liferay/blogs/service/impl/BlogsEntryLocalServiceImpl.java>

When an entity is deleted, the workflow instance is also deleted:

com.liferay.blogs.service.impl.BlogsEntryLocalServiceImpl

```

@Indexable(type = IndexableType.DELETE)
@Override
@SystemEvent(type = SystemEventConstants.TYPE_DELETE)
public BlogsEntry deleteEntry(BlogsEntry entry) throws PortalException {
    ...
    workflowInstanceLinkLocalService.deleteWorkflowInstanceLinks(
        entry.getCompanyId(), entry.getGroupId(),
        BlogsEntry.class.getName(), entry.getEntryId());
    return entry;
}

```

<https://github.com/liferay/liferay-portal/blob/7.1.x/modules/apps/blogs/blogs-service/src/main/java/com/liferay/blogs/service/impl/BlogsEntryLocalServiceImpl.java>

Step 3 - Handle Status in the Getter Methods of the Service Layer**com.liferay.blogs.service.impl.BlogsEntryLocalServiceImpl**

```

@Override
public List<BlogsEntry> getCompanyEntries(
    long companyId, Date displayDate,
    QueryDefinition<BlogsEntry> queryDefinition) {
    if (queryDefinition.isExcludeStatus()) {
        return blogsEntryPersistence.findByC_LtD_Nots(
            companyId, displayDate, queryDefinition.getStatus(),
            queryDefinition.getStart(), queryDefinition.getEnd(),
            queryDefinition.getOrderByComparator());
    }
    else {
        return blogsEntryPersistence.findByC_LtD_S(
            companyId, displayDate, queryDefinition.getStatus(),
            queryDefinition.getStart(), queryDefinition.getEnd(),
            queryDefinition.getOrderByComparator());
    }
}
...

```

<https://github.com/liferay/liferay-portal/blob/7.1.x/modules/apps/blogs/blogs-service/src/main/java/com/liferay/blogs/service/impl/BlogsEntryLocalServiceImpl.java>

Step 4 - Create a Workflow Handler Component and an Update Status Method on the Service Layer**com.liferay.blogs.internal.workflow.BlogsEntryWorkflowHandler**

```

@Component(
    property = "model.class.name=com.liferay.blogs.model.BlogsEntry",
    service = WorkflowHandler.class
)
public class BlogsEntryWorkflowHandler extends BaseWorkflowHandler<BlogsEntry> {
    ...
    @Override
    public BlogsEntry updateStatus(
        int status, Map<String, Serializable> workflowContext)
        throws PortalException {
        long userId = GetterUtil.getLong(

```

```

        (String)workflowContext.get(WorkflowConstants.CONTEXT_USER_ID));
    long classPK = GetterUtil.getLong(
        (String)workflowContext.get(
            WorkflowConstants.CONTEXT_ENTRY_CLASS_PK));

    ServiceContext serviceContext = (ServiceContext)workflowContext.get(
        "serviceContext");

    return _blogsEntryLocalService.updateStatus(
        userId, classPK, status, serviceContext, workflowContext);
}
...

```

The actual status updating method is in the service:

com.liferay.blogs.service.impl.BlogsEntryLocalServiceImpl

```

@Indexable(type = IndexableType.REINDEX)
@Override
public BlogsEntry updateStatus(
    long userId, long entryId, int status,
    ServiceContext serviceContext,
    Map<String, Serializable> workflowContext)
throws PortalException {

    // Entry

    User user = userLocalService.getUser(userId);
    Date now = new Date();

    BlogsEntry entry = blogsEntryPersistence.findByPrimaryKey(entryId);

    validate(
        entry.getTitle(), entry.getUrlTitle(), entry.getContent(), status);

    int oldStatus = entry.getStatus();

    if ((status == WorkflowConstants.STATUS_APPROVED) &&
        now.before(entry.getDisplayDate())) {

        status = WorkflowConstants.STATUS_SCHEDULED;
    }

    ...

    return entry;
}

```

<https://github.com/liferay/liferay-portal/blob/7.1.x/modules/apps/blogs/blogs-service/src/main/java/com/liferay/blogs/internal/workflow/BlogsEntryWorkflowHandler.java>

Step 5 - If Needed, Create a Custom Search User Interface

...blogs-web/src/main/resources/META-INF/resources/blogs_admin/entry_search_columns.jspf

```

<c:choose>
    <c:when test='<%= displayStyle.equals("descriptive") %>'>
        <liferay-ui:search-container-column-user
            showDetails="<%= false %>"
            userId="<%= entry.getUserId() %>">
    />

    <liferay-ui:search-container-column-text
        colspan="<%= 2 %>">
    >

```

```
<%  
Date modifiedDate = entry.getModifiedDate();  
  
String modifiedDateDescription = LanguageUtil.getTimeDescription(request, System.currentTimeMillis() - modifiedDate.getTime(), true);  
%>  
  
<h5 class="text-default">  
    <liferay-ui:message arguments="<%= new String[] {entry.getUserName(), modifiedDateDescription} %>" key="x-modified-x-ago" />  
</h5>  
  
<h4>  
    <aui:a href="<%= rowURL.toString() %>">  
        <%= BlogsEntryUtil.getDisplayTitle(resourceBundle, entry) %>  
    </aui:a>  
</h4>  
  
<h5 class="text-default">  
    <aui:workflow-status markupView="lexicon" showIcon="<%= false %>" showLabel="<%= false %>" status="<%= entry.getStatus() %>" />  
</h5>  
</liferay-ui:search-container-column-text>  
  
<liferay-ui:search-container-column-jsp  
    path="/blogs_admin/entry_action.jsp"  
/>  
</c:when>  
</c:choose>
```

https://github.com/liferay/liferay-portal/blob/7.1.x/modules/apps/blogs/blogs-web/src/main/resources/META-INF/resources/blogs_admin/entry_search_columns.jspf

Exercises

Integrate the Gradebook Application with Portal Search

Introduction

In this exercise, you'll integrate the Gradebook application with the Liferay Search framework, making the Gradebook assignments searchable with portal search.

Overview

- ① Implement the AssignmentIndexer component in gradebook-service module.
- ② Add @Indexable annotations to the assignment service CRUD methods.

Snippets and resources for this exercise are in `snippets/06a-integrate-with-portal-search`.

The complete Gradebook application **solution** is in `solutions/solutions-chapter-06/solution-06-gradebook`.

Implement the AssignmentIndexer Component in gradebook-service Module

First, create a package for the assignment indexer class:

1. **Right-click** on the `gradebook-service` module to open the context menu.
2. **Choose** `New → Package`.
3. **Type** `com.liferay.training.gradebook.search` in the `Name` field.
4. **Click** `Finish` to close the wizard.
5. **Copy** the `gradebook-service/src/main/java/com/liferay/training/gradebook/search/AssignmentIndexer.java` `class` snippet to the package created previously.
6. **Review** the class.
 - o Notice that the indexer registers itself as an indexer for certain entity types with two properties: the `service = Indexer.class` component property and with the `BaseIndexer Generics` parameter `Assignment`.

Add @Indexable Annotations to Assignment Service CRUD Methods

Next, we have to make our Assignment local service implementation index-aware.

1. **Open** the `com.liferay.training.gradebook.service.impl.AssignmentLocalServiceImpl` class in the `gradebook-service` module.
 - o Just above the `addAssignment()` method, you can find a placeholder for the `@Indexable` annotation:

```
***** [PLACEHOLDER FOR INDEXABLE ANNOTATION (REINDEX)] *****

public Assignment addAssignment( ...
```

2. **Replace** the placeholder with (use *Organize Imports* as necessary):

```
@Indexable(
    type = IndexableType.REINDEX
)
public Assignment addAssignment( ...
```

3. **Repeat** for the placeholders before `deleteAssignment()` and `updateAssignment()` methods.
 - o Notice the `IndexableType` for each.

Now our entities get indexed when they are either added or updated or removed from the index when the corresponding object is deleted.

Exercises

Integrate the Gradebook Application with Liferay's Asset Framework

Introduction

In this exercise, you'll integrate the Gradebook application with Liferay's Asset framework.

Overview

- ① Add required dependencies to the build.gradle of the web module
- ② Ensure that the Gradebook schema has the required fields
- ③ Add asset CRUD logic to the Gradebook service
- ④ Create an asset renderer factory component
- ⑤ Create the asset renderer for assignments
- ⑥ Create JSP files for the views
- ⑦ Deploy and test

Snippets and resources for this exercise are in `snippets/06b-integrate-with-asset-framework`.

The complete Gradebook application **solution** is in `solutions/solutions-chapter-06/solution-06-gradebook`.

Add Required Dependencies to the build.gradle of Web Module

1. **Open** the `build.gradle` of the *gradebook-web* module.
2. **Type** the following line to the dependencies section:

```
...  
// Needed by the Asset Renderer  
compileOnly group: "com.liferay", name: "com.liferay.asset.api", version: "1.0.1"  
...
```

3. **Save** the file.
4. **Right-click** on the *gradebook-web* module.
5. **Choose** *Gradle → Refresh Gradle Project*.

Ensure that Gradebook Schema has the Required Fields

1. **Open** the `service.xml` in *gradebook-service* module.
2. **Find** that the following fields are defined for the *Assignment* entity:
 - companyId
 - groupId
 - userId
 - userName
 - createDate

- modifiedDate
 - status
3. **Check** that the *Assignment* entity *uuid* attribute is set to true and that there is a reference to *AssetEntry*, making the *AssetEntryService* available in the Assignment local and remote services. There should also be a status finder.

Add Asset CRUD Logic to the Gradebook Service

Next, we will complete the Assignment local service class so that every time an Assignment gets created, a corresponding Asset entry gets created. We will also take care of updating and deleting the Asset entry. The method for updating an Asset entry is already provided and we just have to call it:

1. **Open** the *com.liferay.training.gradebook.service.impl.AssignmentLocalServiceImpl* in the *gradebook-service* module.
2. **Find** the *addAssignment()* method.
3. **Find** the following placeholder at the end of the method:

```
***** [PLACHOLDER FOR UPDATING ASSET] *****
```

4. **Replace** the placeholder with:

```
updateAsset(assignment, serviceContext);
```

5. **Repeat** the same for the *updateAssignment()* method: replace the placeholder with *updateAsset()*.
 - Now, every time you create or update an Assignment, a corresponding Asset entry gets created or updated.
6. **Review** the *updateAsset()* method before proceeding.
 - Next, we will take care of deleting the Asset entry when deleting an Assignment.
7. **Find** the *deleteAssignment()* method.
8. **Find** the following placeholder at the end of the method:

```
***** [PLACHOLDER FOR DELETING ASSET] *****
```

9. **Replace** the placeholder with:

```
assetEntryLocalService.deleteEntry(
    Assignment.class.getName(), assignment.getAssignmentId());
```

10. **Save** the file.

Create an Asset Renderer Factory Component

Next, we will create the factory for the Assignment renderer. This component will register to the OSGi service registry and will be called whenever we need to have a renderer for the Assignment Asset.

1. **Right-click** on the *gradebook-web* module to open the context menu.
2. **Choose** *New → Package*.
3. **Type** *com.liferay.training.gradebook.web.asset* in the *Name* field.
4. **Click** *Finish* to close the wizard.
5. **Copy** the *gradebook-*
`web/src/main/java/com/liferay/training/gradebook/web/asset/AssignmentAssetRendererFactory.java` snippet to the package created previously.
6. **Review** the class.
 - Notice the *getAssetRenderer()* method. This is the method we will be calling to get the actual renderer (created in the next step):

```

public AssetRenderer<Assignment> getAssetRenderer(long classPK, int type)
    throws PortalException {

    Assignment assignment = _assignmentLocalService.getAssignment(classPK);

    AssignmentAssetRenderer assignmentAssetRenderer =
        new AssignmentAssetRenderer(assignment, this);

    assignmentAssetRenderer.setAssetRendererType(type);
    return assignmentAssetRenderer;
}

```

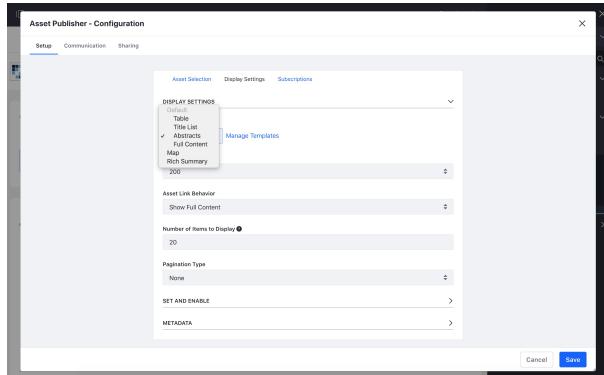
Create the Asset Renderer for Assignments

Now let's create the Assignment renderer. This class is responsible for rendering the asset, for example, in *Asset Publisher* portlet views.

1. **Copy** the `gradebook-web/src/main/java/com/liferay/training/gradebook/web/asset/AssignmentAssetRenderer.java` snippet to the package created in the previous step with the factory component.
2. **Review** the class.

Create JSP files for the Views

The final step in integrating with the Asset Framework is to create the JSP files for different views. Typically, you would create your custom JSP for the abstracts and full content view. This corresponds to the options you see in the *Asset Publisher* portlet display configuration:

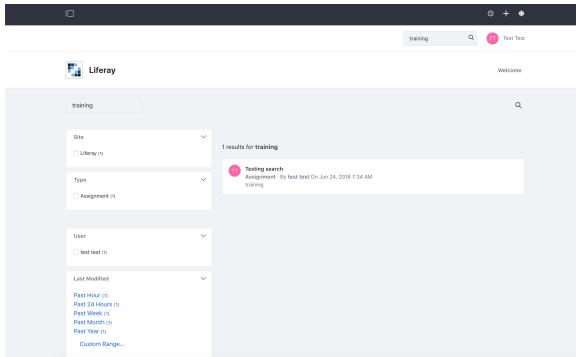


1. **Copy** the `gradebook-web/src/main/resources/META-INF/resources/asset` snippet folder to the corresponding location in the *gradebook-web* module.
2. **Review** the 'full_content.jsp' file.

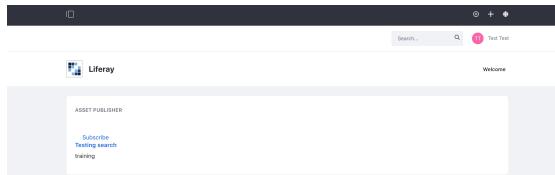
Deploy and Test

Let's verify that everything works correctly. We will check the following things:

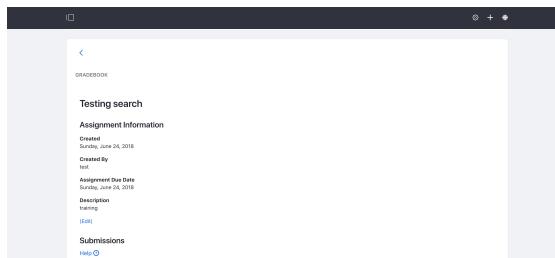
- Assignments appear in the search results (Asset integration is required).
- Assignments appear in the *Asset Publisher* and custom JSPs are working.
- Assignments appear in the *Asset Publisher Create New* menu.
- **Open** your browser, login if necessary, and go to the *Gradebook* portlet page.
- **Create** a new Assignment, typing 'training' in the *Description* field.
- **Type** 'training' in the search field at the top right corner of the page. The assignment you just created should appear on the results list:



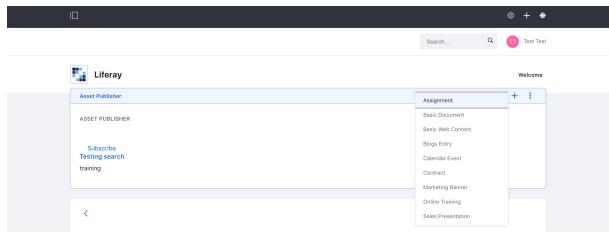
- In order to index Assignments which were added before we had implemented the AssignmentIndexer, Open the *Control Menu* → Expand the *Control Panel* → Expand the *Configuration Panel* → Select *Search* → Execute *Reindex all search indexes* - afterwards your previously entered Assignments will be found by the global search, too.
 - If everything works so far, let's proceed.
- **Go back** to the *Gradebook* page.
- **Create** a new portal page called *Asset Publisher*.
 - Expand the Site Administration Menu → Expand the *Build* panel → Select *Pages* → Add *Public Page* → Use *Widget Page* → Enter *Asset Publisher* as page name.
- **Add** an *Asset Publisher* portlet on the page (the *Widgets* menu on the right side of the page) created previously.
 - The assignment just created should appear on the list of recent assets:



- **Click** the Assignment on the list.
 - You should see the Assignment full content (*full_content.jsp*).
- **Click** *View in Context* on the bottom of the view.
 - You should now be seeing the related Assignment in the Gradebook portlet.



- If everything works, the last thing to check is that the Assignment is appearing on the *Create New* menu of the *Asset Publisher* portlet.
- **Go back** to the *Asset Publisher* page.
- **Click** the plus sign on the top right corner of the *Asset Publisher* portlet. There should be an option for an *Assignment*.



Exercises

Enable Workflows for Gradebook Assignments

Introduction

Workflows allow creating automated and complex processes on asset entries. In the Gradebook application, workflows could be used, for example, to allow drafting and reviewing assignments before publishing them or in case of enabling workflows for submissions, to notify a student when a submission has been approved and graded. In this exercise, we will enable workflow support for the Gradebook assignments.

Overview

- 1 Check Assignment model entity readiness for workflows
- 2 Handle creating and deleting WorkflowInstance in the Assignment local service
- 3 Create Assignment workflow handler
- 4 Check user interface readiness for workflows
- 5 Deploy and test

Snippets and resources for this exercise are in `snippets/06c-enable-workflow-support`.

The complete Gradebook application **solution** is in `solutions/solutions-chapter-06/solution-06-gradebook`.

Check Assignment Model Entity Readiness for Workflows

1. **Review** the Gradebook service definition file `service.xml` for the fields required by workflows:
 - o status
 - o statusByUserId
 - o statusByUserName
 - o statusDate The status field, contains the workflows status of an entity, defaulting to '0' which equals to *Approved*. That's why you have been seeing the approved status on the assignments list in the Gradebook portlet. We want the WorkflowInstanceLink service to be available in our Assignment local service.
2. **Check** in `service.xml` that there's the following service reference for the Assignment entity:

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

To be able to make the user interface work with workflows, we have to be able to call Assignments by their status. That's why there are finders that have status fields as parameters:

```
<finder name="Status" return-type="Collection">
  <finder-column name="status"></finder-column>
</finder>
<finder name="G_S" return-type="Collection">
  <finder-column name="status"></finder-column>
  <finder-column name="groupId"></finder-column>
</finder>
```

Handle Creating and Deleting a WorkflowInstance in the Assignment Local Service

Let's add status and workflow instance handling to the `addAssignment()` and `deleteAssignment()` methods. The method for starting the workflow instance has been provided in the class for you. We just have to add a call for it:

1. **Open** `com.liferay.training.gradebook.service.impl.AssignmentLocalServiceImpl` in the `gradebook-service` module.
2. **Find** the `addAssignment()` method and uncomment the following code:

```
/*
assignment.setStatus(WorkflowConstants.STATUS_DRAFT);
assignment.setStatusByUserId(userId);
assignment.setStatusByUserName(user.getFullName());
assignment.setStatusDate(serviceContext.getModifiedDate(null));
*/
```

3. **Find** the following placeholder in the end of the method:

```
***** [PLACEHOLDER FOR STARTING WORKFLOW INSTANCE *****]

return assignment;
```

4. **Replace** the placeholder **and** return statement with:

```
return startWorkflowInstance(userId, assignment, serviceContext);
```

- o Now, whenever an assignment gets created, it will be workflowable. Let's also take care of cleaning the workflow data when an Assignment is deleted:

5. **Find** the `deleteAssignment()` method and the following placeholder:

```
***** PLACEHOLDER FOR DELETING WORKFLOW INSTANCE *****
```

6. **Replace** the placeholder with:

```
workflowInstanceLinkLocalService.deleteWorkflowInstanceLinks(
    assignment.getCompanyId(), assignment.getGroupId(),
    Assignment.class.getName(), assignment.getAssignmentId());
```

7. **Save** the file.

We still have to take care of updating the workflow status. We will create a Workflow Handler for that.

Create Assignment Workflow Handler

A workflow handler is an OSGi component that registers to the OSGi service registry as the handler responsible for handling workflow status changes on certain model entities. The registration for an entity type is made with Generics typing:

```
@Component(
    immediate = true,
    service = WorkflowHandler.class
)
public class AssignmentWorkflowHandler extends BaseWorkflowHandler<Assignment> {
```

The logical location for the component is in the service module:

1. **Right-click** on the `gradebook-service` module to open the context menu.

2. Choose New → Package.
 3. Type `com.liferay.training.gradebook.service.workflow` in the Name field.
 4. Click Finish to close the wizard.
 5. Copy the `gradebook-`
- `service/src/main/java/com/liferay/training/gradebook/service/workflow/AssignmentWorkflowHandler.java` snippet to the package created previously.

Let's take a look at the class. The `updateStatus()` method takes care of updating the workflow status of a model instance. The actual update code could be in the workflow handler class, too, but a more logical place is the `AssignmentLocalServiceImpl`, where the rest of the CRUD functionality is. That's why you can see, in the end of the method, a call to the method we provided in the `AssignmentLocalServiceImpl` already:

```
return _assignmentLocalService.updateStatus(
    userId, resourcePrimKey, status, serviceContext);
```

Now workflows are supported for the Assignments. For your convenience, we have already provided the status fields in the user interface and have already passed the status information within the service class, so you don't have to take care of adding it anymore. Let's just review:

1. Open the file `/src/main/resources/META-INF/resources/assigment/entry_search_column.jspf` in the `gradebook-web` module.

- o You can see the status column:

```
<liferay-ui:search-container-column-status
    name="status"
/>
```

2. Open the class `com.liferay.training.gradebook.web.portlet.action.ViewAssignmentsMVCRenderCommand.java`, which is responsible for listing the assignments.

- o In the `addAssignmentListAttributes()`, notice the following section, where we resolve the workflow status for the service list call:

```
// Get the workflow status for the list.

int status = getWorkflowStatus(renderRequest);

// Call the service to get the list of assignments.

List<Assignment> assignments =
    _assignmentService.getAssignmentsByKeywords(
        themeDisplay.getScopeGroupId(), keywords, start, end, status, comp);
```

In the `getWorkflowStatus()` we simply allow admins to see Assignments of any status while other users can only see the approved ones:

```
private int getWorkflowStatus(RenderRequest renderRequest) {

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

    PermissionChecker permissionChecker = themeDisplay.getPermissionChecker();

    int status;

    if (permissionChecker.isCompanyAdmin()) {
        status = WorkflowConstants.STATUS_ANY;
    } else {
        status = WorkflowConstants.STATUS_APPROVED;
    }
}
```

```

        }

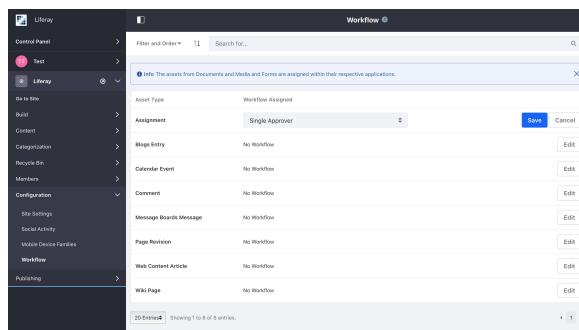
        return status;
    }
}

```

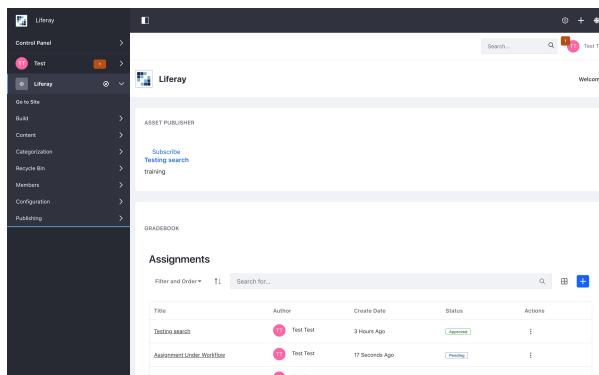
Deploy and Test

First, we have to enable and define a workflow for Assignments:

1. **Open** your browser and sign in if necessary.
2. **Go to** Site Administration → Configuration → Workflow.
3. **Set** Assignment workflow to *Single Approver*.



4. **Go to** the page where the Gradebook is.
5. **Create** a new assignment. The status on the list should now be pending.
 - o After page refresh, you should see a notification on the menu indicating a new workflow event:



Now you can manage the workflows for Assignments as for any other assets.

Integrate with External Systems

Introduction

Liferay Service Builder, when defined, automatically generates **JSON and SOAP web services APIs** for a service. As Liferay platform core services are created using the Service Builder pattern, they have web service APIs available.

In addition to Service Builder generated web services, it is possible to publish **JAX-RS** and **JAX-WS** endpoints for any ad-hoc services.

Service Builder and JSON Web Services

remote-service=true

Setting the remote-service attribute to true in the Service Builder entity definitions automatically creates the remote service variant for the entity. For each remote-enabled *Service.java interface, the @JSONWebService annotation is added, and all the public methods of that interface become registered and available as JSON web services.

Below is an example of an entity definition having remote service generation enabled:

service.xml

```
<service-builder package-path="com.liferay.training.gradebook">
    <namespace>Gradebook</namespace>
    <entity name="Assignment" uuid="true" local-service="true" remote-service="true">
        ...
    </entity>
</service-builder>
```

The generated service interface has the @JSONWebService annotation added:

The Generated AssignmentService Interface

```
@AccessControlled
@JSONWebService
@OSGiBeanProperties(
    property = {
        "json.web.service.context.name=gradebook",
        "json.web.service.context.path=Assignment"
    },
    service = AssignmentService.class
)
@ProviderType
@Transactional(isolation = Isolation.PORTAL, rollbackFor = {
    PortalException.class, SystemException.class})
public interface AssignmentService extends BaseService {

    ...

    public Assignment addAssignment(long groupId, long userId,
        Map<Locale, java.lang.String> title, java.lang.String description,
        Date dueDate, ServiceContext serviceContext) throws PortalException;
    ...
}
```

Ignoring a Method

A method can be prevented from being exposed as a web service by setting the mode attribute to **JSONWebServiceMode.IGNORE**:

```
@JSONWebService(mode = JSONWebServiceMode.IGNORE)
```

Defining HTTP Methods

All JSON web services are mapped to either GET or POST HTTP methods with the following logic:

- **GET**: if method name starts with get, is, or has
- **POST**: all other method prefixes

HTTP methods can, however, be explicitly defined on a method level by setting the method attribute:

```
@JSONWebService(
    value = "do-some-thing",
    method = "PUT"
)
public void doSomething()

...
```

Explicit Method Registration

By setting the JSONWebService mode to manual, the methods to be exposed have to be declared manually. In the example below, only the getAssignment() method is exposed to the web service API.

```
@JSONWebService(
    mode = JSONWebServiceMode.MANUAL
)
public class AssignmentServiceImpl extends AssignmentServiceBaseImpl{
    ...
    @JSONWebService
    public Assignment getAssignment(
        ...
    )

    public void addAssignment(
        ...
    )
}
```

Configuring the JSON Web Service API

Global JSON Web Service configuration is done in the portal-ext.properties. For example, the following settings are available:

portal_ext.properties

```
# Enable / disable JSON web service API
json.web.service.enabled=false

# Discoverability through the test page http://[address]:[port]/api/jsonws
jsonws.web.service.api.discoverable=false

# Restricted HTTP methods
jsonws.web.service.invalid.http.methods=DELETE,POST,PUT

# By default, the HTTP method is not checked when invoking a service call. This setting enables the strict mode.
jsonws.web.service.strict.http.method=true

# Web service paths that are accessible
jsonws.web.service.paths.includes=get*,has*,is*,
```

```
# Web service paths that aren't allowed. This setting takes precedence over the jsonws.web.service.paths.in
cludes
jsonws.web.service.paths.excludes=set*, add*
```

Testing the JSON Web Service API

A JSON web service test page is available at <http://localhost:8080/api/jsonws>. URL, cURL, and JavaScript examples access the services that are provided.

The screenshot shows the JSONWS API interface. On the left, there's a sidebar with a search bar and a list of assignment-related methods: add-assignment, update-assignment, delete-assignment, get-assignments-by-group-id, get-assignments-count-by-group-id, and get-assignment. The main panel displays the details for the `/space.assignment/get-assignments-by-group-id` endpoint. It shows the `HTTP Method` as `get`, the `com.liferay.training.space.gradebook.service.impl.AssignmentServiceImpl` class, and the `getAssignmentsByGroupId` method. It includes sections for **Parameters** (`p_auth` String, `groupid` long, `start` int, `end` int), **Return Type** (`java.util.List`), **Exception** (`p_auth` String), and **Execute** parameters (`groupid` long, `Start` int, `End` int). At the bottom is a blue `Invoke` button.

Service Builder and SOAP API

Liferay uses Apache Axis for the SOAP services. To generate the SOAP API for your custom application, the WSDD builder task has to be added to the project to create the web service definition.

The WSDD creation process is described in detail in https://dev.liferay.com/develop/tutorials/-/knowledge_base/7-1/creating-remote-services

Configuring the SOAP API

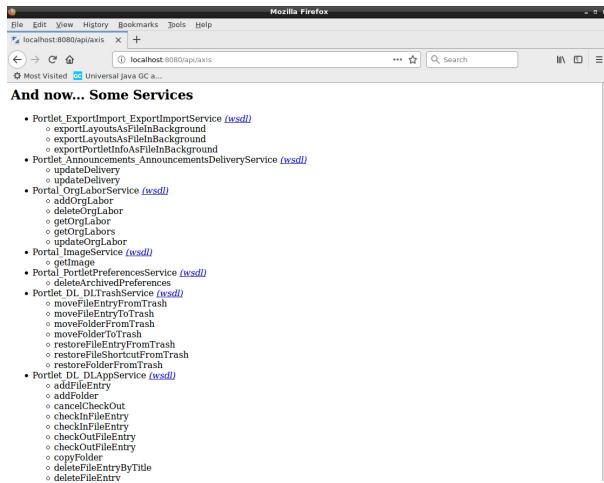
The hosts allowed to access the SOAP API can be defined explicitly in the portal-ext.properties:

portal-ext.properties

```
axis.servlet.hosts.allowed=192.168.100.100, 127.0.0.1, [SERVER_IP]
```

Testing SOAP API

A test page is available at <http://localhost:8080/api/axis>



Publishing JAX-RS and JAX-WS Services

Liferay supports publishing JAX-WS and JAX-RS services via the Apache CXF implementation. Publishing JAX-WS and JAX-RS services requires defining an endpoint and an extender.

The application can publish JAX web services to the CXF **endpoints**. CXF endpoints are context paths the JAX web services are deployed to and accessible from.

Extenders specify where the services are deployed:

- **SOAP Extenders:** for publishing JAX-WS web services. Each SOAP extender can deploy the services to one or more CXF endpoints.
- **REST Extenders:** for publishing JAX-RS web services. REST extenders for JAX-RS services are analogous to SOAP extenders for JAX-WS services.

Steps for Publishing a JAX Web Service:

- ① Create an OSGi service component for the web service.
- ② Configure a Liferay endpoint to access the REST service.
- ③ Map the endpoint to your REST service using the REST or SOAP extender.

Exercises

Publish a REST Service for the Gradebook Application

Introduction

If you have been following along with our real-world application exercises, you should have a fully-working real-world application that incorporates Liferay's service layer, presentation layer, permissions, workflow, search, and asset management framework. In this exercise, we will be adding a REST API to the service layer by way of a JAX-RS Whiteboard. We will be using the standard JAX-RS Whiteboard annotations. You can find more information regarding JAX-RS Whiteboard in the official [OSGi documentation](#).

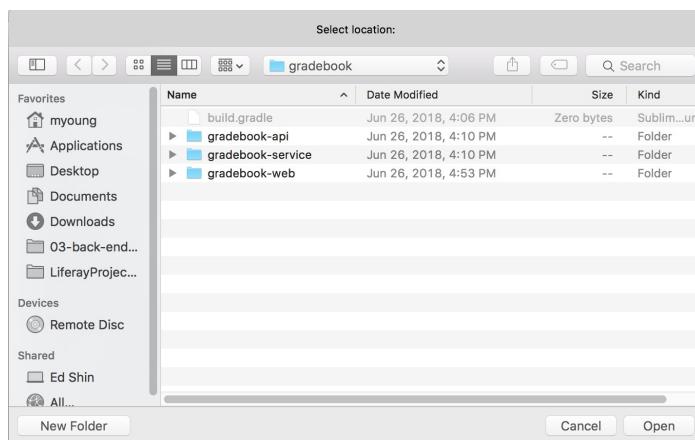
We'll be exposing two REST methods here, one to get all the assignments in our gradebook, and another one to look up a specific assignment with its assignment ID.

Overview

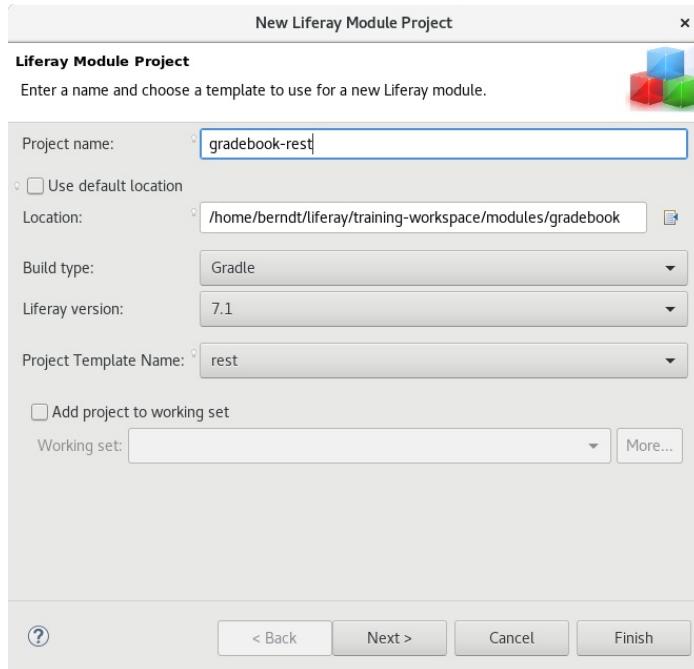
- ① Create a Liferay Module project using the Rest template
- ② Resolve Dependencies
- ③ Implement the AssignmentRestApplication class
- ④ Deploy and test

Create a Liferay Module Project using the Rest Template

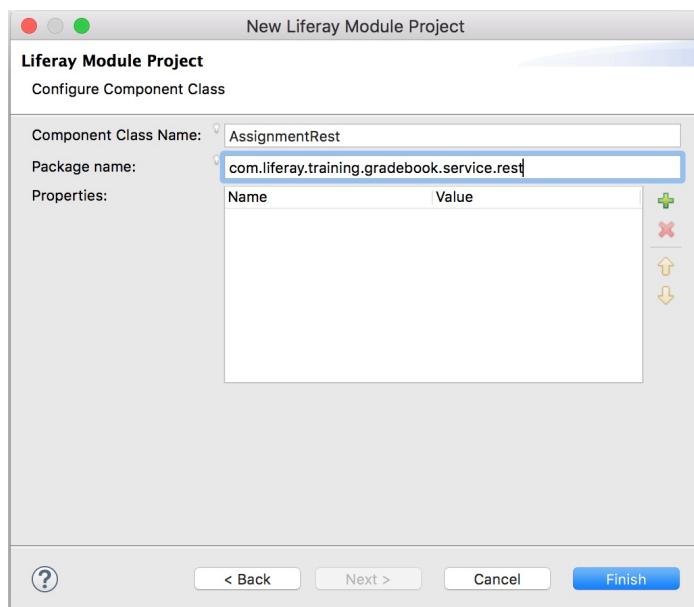
1. [Go to training-workspace](#).
2. [Right-click](#) to open the context menu.
3. [Choose New → Liferay Module Project](#).
4. [Type gradebook-rest](#) in the *Project Name* field.
5. [Uncheck Use Default Location](#).
6. [Choose](#) the *gradebook* directory that your other modules are located in.



7. [Choose rest](#) in the *Project Template Name* field.
8. [Click Next](#).



9. Type `AssignmentRest` in the Component Class Name field.
10. Type `com.liferay.training.gradebook.service.rest` in the Package name field.
11. Click **Finish**.



Resolve Dependencies

We'll need to fix the `bnd.bnd` and `build.gradle` file generated by Liferay Studio. Liferay Studio generated these files using the activator template, so they don't have the dependencies set up for our particular project.

1. Open the `build.gradle` file and replace it with the following code:

```
dependencies {
    compileOnly group: "javax.ws.rs", name: "javax.ws.rs-api", version: "2.0.1"
    compileOnly group: "org.osgi", name: "org.osgi.service.component.annotations", version: "1.3.0"
    compileOnly group: "com.liferay.portal", name: "com.liferay.portal.kernel", version: "3.0.0"
    compileOnly group: "javax.servlet", name: "javax.servlet-api", version: "3.0.1"
```

```
compileOnly group: "javax.portlet", name: "portlet-api", version: "3.0.0"
compile project(":modules:gradebook:gradebook-api")
}
```

2. Open the `bnd.bnd` file and replace it with the following.

- o We are removing the Bundle-Activator that we deleted in an earlier step.
- o We are also renaming the bundle. This step is not necessary, but it's helpful to use a more human-readable *Bundle-Name* in case you need to troubleshoot the bundle in the Gogo Shell, for example.

```
Bundle-Name: Gradebook REST
Bundle-SymbolicName: com.liferay.training.gradebook.service.rest
Bundle-Version: 1.0.0
Liferay-Configuration-Path: /configuration
```

Implement the GradebookRestApplication Class

1. Open the `AssignmentRestApplication` class.

2. Replace contents of the class with the following.

```
public class AssignmentRestApplication extends Application {

    public Set<Object> getSingletons() {

        return Collections.<Object> singleton(this);
    }

    public String getAssignments() {

        Company company;
        List<Assignment> assignments = new ArrayList<Assignment>();

        try {
            company = _companyService.getCompanyById(
                PortalUtil.getDefaultCompanyId());
            List<Group> groups =
                _groupLocalService.getGroups(company.getCompanyId(), 0, true);
            for (Group group : groups) {
                assignments.addAll(
                    _assignmentLocalService.getAssignmentsByGroupId(
                        group.getGroupId()));
            }
        } catch (PortalException e) {
            return "[{}]";
        }
    }

    public String getAssignments(@PathParam("assignmentid") long assignmentId) {

        try {
            return JSONFactoryUtil.serialize(
                _assignmentLocalService.getAssignment(assignmentId));
        } catch (Exception e) {
            return "{}";
        }
    }

    @Reference
    private AssignmentLocalService _assignmentLocalService;

    @Reference
}
```

```

    private CompanyService _companyService;

    @Reference
    private GroupLocalService _groupLocalService;
}

```

3. Implement the OSGi component annotations.

- The `"osgi.jaxrs.application.base=" + "/gradebook-rest"` property tells the OSGi container that our REST API will be located at `http://localhost:8080/o/gradebook-rest`.

```

@Component(
immediate = true,
property = {
    "liferay.auth.verifier=false",
    "liferay.oauth2=false",
    "osgi.jaxrs.application.base=" + "/gradebook-rest",
    "osgi.jaxrs.name=Gradebook.Rest"
},
service = Application.class)
public class AssignmentRestApplication extends Application {

```

4. Implement the following JAX-RS method annotations. The annotations below instruct the container to place two rest services at the path `/o/gradebook-rest/assignments` and `/gradebook-rest/assignment/{assignmentid}`. They will both produce JSON as their output format.

```

@GET
@Path("/assignments")
@Produces({
    MediaType.APPLICATION_JSON
})
public String getAssignments() {...}

---

@GET
@Path("/assignment/{assignmentid}")
@Produces({
    MediaType.APPLICATION_JSON
})
public String getAssignments(@PathParam("assignmentid") long assignmentId) {...}

```

5. Press `CTRL+SHIFT_O` to organize imports.

6. Save the file.

Deploy and Test

|  build | |
|--|---|
|  assemble | Assembles the Java code. |
|  build | Assembles all Java code. |
|  buildCSS | Build CSS file. |
|  buildDependents | Assembles all dependencies. |
|  buildLang | Runs Liferay. |
|  buildNeeded | Assembles all needed dependencies. |
|  buildSoy | Compiles Clojure files. |
|  classes | Assembles the class files. |
|  clean | Deletes the build artifacts. |
|  configJSModules | Generates the configuration for JavaScript modules. |
|  deploy | Assembles the deployment artifacts. |
|  jar | Assembles a JAR file. |
|  replaceSoyTranslation | Replaces 'go' files with translations. |
|  testClasses | Assembles the test class files. |
|  testIntegrationClasses | Assembles the integration test class files. |
|  transpileJS | Transpiles JS files. |

1. Go to the *Gradle Tasks* view in *Liferay Developer Studio*.

2. Double-click `deploy` from `gradebook-rest → build`.

- o This will deploy `gradebook-rest` to the Liferay Server.



```

{
  "status": "ok",
  "data": [
    {
      "id": 1,
      "title": "Assignment 1",
      "description": "Assignment 1 description",
      "status": "PENDING",
      "modifiedDate": "2020-01-01T00:00:00Z"
    },
    {
      "id": 2,
      "title": "Assignment 2",
      "description": "Assignment 2 description",
      "status": "PENDING",
      "modifiedDate": "2020-01-02T00:00:00Z"
    },
    {
      "id": 3,
      "title": "Assignment 3",
      "description": "Assignment 3 description",
      "status": "PENDING",
      "modifiedDate": "2020-01-03T00:00:00Z"
    }
  ]
}
  
```

3. Open your browser to <http://localhost:8080/o/gradebook-rest/assignments>.

4. Verify that you see a JSON listing of assignments.

5. Locate an assignment ID in the JSON output.



```

{
  "id": 1,
  "title": "Assignment 1",
  "description": "Assignment 1 description",
  "status": "PENDING",
  "modifiedDate": "2020-01-01T00:00:00Z"
}
  
```

6. Open your browser to [http://localhost:8080/o/gradebook-rest/assignment/\[assignment-id\]](http://localhost:8080/o/gradebook-rest/assignment/[assignment-id]) and log in using the assignment ID you located from the previous step.

- o You should see a JSON listing of assignments.

7. Verify that you see a JSON form of the assignment that you requested.

Takeaways

You've seen in this exercise how easy it is to create a REST API for your Liferay Services. Because JAX-RS Whiteboard is a standard OSGi specification, you now have another convenient tool at your disposal to create APIs that others can use to easily integrate with your applications.

Logging

Logging is a powerful method for example for emitting application metrics and statistics but also to help in troubleshooting and resolving production issues. Properly used it improves application quality and maintainability.

Examples of use cases:

- Adding additional, contextual information like user ID or thread ID to stack traces
- Collecting and sending metrics like execution timers
- Emit warning when a threshold, for example for group count, is exceeded

Although the role of logging in troubleshooting can be essential, it in many cases doesn't expose the root cause behind but just detects a problem. Debugging takes it from there forward.

Implementing Logging in Custom Modules

Following out of the box options are available to implement logging in your custom modules :

- Java native logger (java.util.logging)
- Liferay native logger (com.liferay.kernel.log)
- SLF4J logger
- OSGi Log Service

Except of the Java native logger, all the options use natively Log4J as a SLF4J implementation in the background.

Using Java native logging is usually not recommended because of its limitations and performance compared to SLF4J implementations.

Let's take a look at the different ways of invoking logging.

Using Liferay Native Logger

Liferay native logger is called via `LogFactoryUtil.getLog(CLASS_NAME)`.

```
package com.liferay.training.sample.log;

import com.liferay.portal.kernel.log.Log;
import com.liferay.portal.kernel.log.LogFactoryUtil;

import java.math.BigDecimal;

public class EmployeeHandler {

    public void setSalary(BigDecimal salary) {

        if (_log.isInfoEnabled()) {
            if (salary.compareTo(MAX_SALARY) == 1) {
                _log.info("Alert: suspiciously high salary: " + salary + ".");
            }
        }

        _salary = salary;
    }

    private BigDecimal _salary;

    public static final BigDecimal MAX_SALARY;

    static {
        MAX_SALARY = new BigDecimal(100000.0);
    }
}
```

```

    }

    private static final Log _log = LogFactoryUtil.getLog(EmployeeHandler.class);

}

```

Using SLF4J Logger

In this approach you have to declare the dependency for SLF4J API:

build.gradle

```

dependencies {

    ...

    compileOnly group: "org.slf4j", name: "slf4j-api", version: "1.7.2"
    ...

}

```

Logger can be invoked by calling the SLF4J *LoggerFactory* service as in the example below. This logging approach is the recommended as it doesn't bind logger to any specific implementation.

```

package com.liferay.training.sample.log;

import java.math.BigDecimal;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class EmployeeHandler {

    public void setSalary(BigDecimal salary) {

        if (_log.isInfoEnabled()) {
            if (salary.compareTo(MAX_SALARY) == 1) {
                _log.info("Alert: suspiciously high salary: " + salary + ".");
            }
        }

        _salary = salary;
    }

    private BigDecimal _salary;

    public static final BigDecimal MAX_SALARY;

    static {
        MAX_SALARY = new BigDecimal(10000.0);
    }

    private static final Logger _log = LoggerFactory.getLogger(EmployeeHandler.class);
}

```

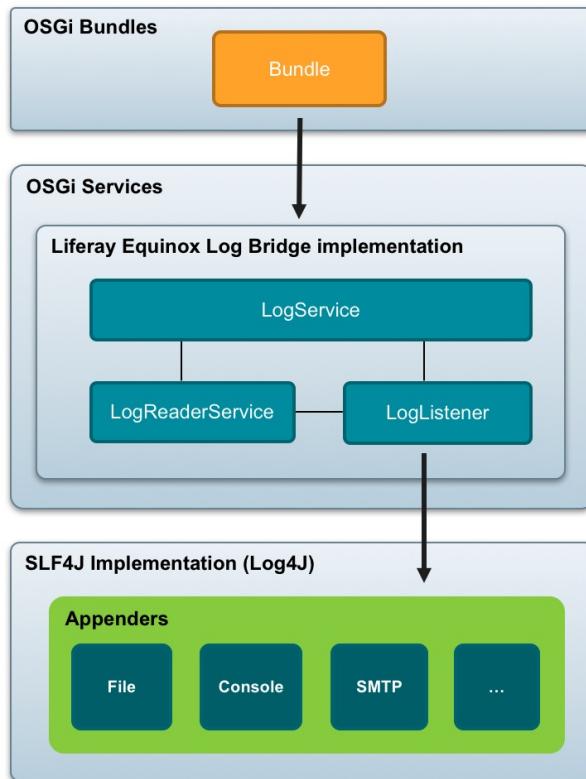
OSGi Log Service

OSGi Log Service is a message logger service for the OSGi framework, specified in the OSGi Compendium. It consists of three main components:

- LogService: service interface for storing logs

- LogReaderService: service interface for reading and dispatching log entries
- LogListener: interface for the listener of log entry objects

In Liferay's Equinox Log Bridge implementation there's a SLF4J log listener implementation using Log4J as backend:



OSGi LogService can be called with Declarative Services via OSGi @Reference annotation. This approach is limited to OSGi components.

```

package com.liferay.training.sample.log;

import java.math.BigDecimal;

import org.osgi.service.component.annotations.Component;
import org.osgi.service.component.annotations.Reference;
import org.osgi.service.log.LogService;

@Component(
    service = EmployeeHandlerService.class
)
public class EmployeeHandlerServiceImpl implements EmployeeHandlerService {

    @Override
    public void setSalary(BigDecimal salary) {
        if (salary.compareTo(MAX_SALARY) == 1) {
            _log.log(LogService.LOG_INFO, "Alert: suspiciously high salary: " + salary + ".");
        }
    }

    private BigDecimal _salary;

    public static final BigDecimal MAX_SALARY;

    static {
        MAX_SALARY = new BigDecimal(10000.0);
    }
}

```

```

@Reference
private LogService _log;
}

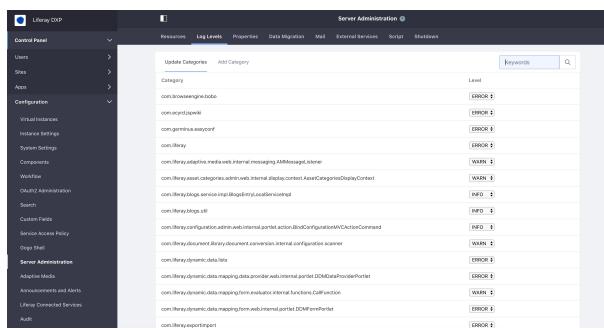
```

Worth noting is that for OSGi LogService to start logging there has to be a level definition. There are two ways to define log levels when using OSGi Log Service:

- *MODULE_ROOT/src/main/META-INF/module-log4j.xml*
- *Control panel -> Server Administration -> Log Levels*

Configuring Logging

Logging levels can be configured platform wide from Control Panel. Changes made there won't persist after restart, however:



Persistent settings as well as all the other LOG4J configuration settings, like appenders can be set with *LIFERAY_WEBAPP_ROOT/WEB-INF/classes/META-INF/portal-log4j-ext.xml*. An example configuration, setting package *com.liferay.blogs* level to *INFO*:

```

<?xml version="1.0"?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">

<log4j:configuration xmlns:log4j="http://jakarta.apache.org/log4j/">

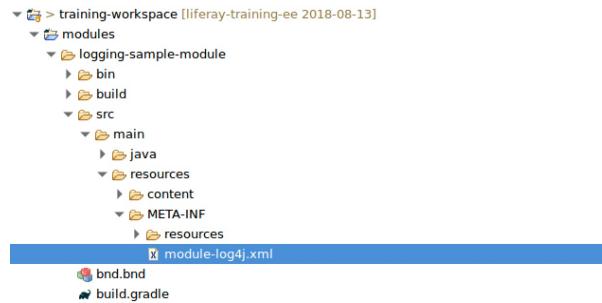
    <appender name="FILE" class="org.apache.log4j.rolling.RollingFileAppender">
        <rollingPolicy class="org.apache.log4j.rolling.TimeBasedRollingPolicy">
            <param name="FileNamePattern" value="@liferay.home@/logs/sql.%d{yyyy-MM-dd}.log" />
        </rollingPolicy>
        <layout class="org.apache.log4j.PatternLayout">
            <param name="ConversionPattern" value="%d{ABSOLUTE} %-5p [%c{1}:%L] %m%n" />
        </layout>
    </appender>

    <logger name="com.liferay.blogs">
        <priority value="INFO" />
        <appender-ref ref="FILE" />
    </logger>

</log4j:configuration>

```

On module level, you can define settings for a single, or multiple modules within a module, in *MODULE_ROOT/src/main/resources/META-INF/module-log4j.xml* or independently from module code, in *LIFERAY_HOME/osgi/log4j/BUNDLE_SYMBOLIC_NAME-log4j-ext.xml*.



In case of fragment bundles, levels for the host bundle can be defined in the fragment bundles `MODULE_HOME/META-INF/module-log4j-ext.xml`

Below is an example of `module-log4j.xml` configuration file. Note the package naming syntax difference when configuring OSGi Log Service and native or SLF4J loggers: OSGi Log Service related categories have to begin with `osgi.logging` and the dots in the package name have to be replaced with underscores. This same syntax applies when configuring through Control panel, too:

```
<?xml version="1.0"?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">
<log4j:configuration xmlns:log4j="http://jakarta.apache.org/log4j/">

    <!-- OSGi Log Service logger configuration -->

    <category name="osgi.logging.com_liferay_training_sample.logservice">
        <priority value="INFO" />
    </category>

    <!-- Native or SLF4J API logger configuration -->

    <category name="com.liferay.training.sample.">
        <priority value="INFO" />
    </category>

</log4j:configuration>
```

Other Logging Related Settings

While SQL logging is as such not directly Liferay platform related, you can enable logging for Hibernate queries in `portal-ext.properties` with the following setting:

```
hibernate.show_sql=true
```

Javascript logging can accordingly be enabled with:

```
javascript.log.enabled=true
```

Remember to disable logging in production systems.

Wrapping It Up

If you choose to use platform provided logging functionalities, using SLF4J Logger is recommended for the best code portability.

Logging approach should always be designed and standardized. Misuse of logging severities, not readable log messages or missing contextual information do not necessarily bring any extra value or quality for the application. Properly designed logging can, on the other hand, make application much easier maintainable and dramatically help

in troubleshooting issues.

Too extensive logging may severely affect application performance. Also as a thumb of rule, remember to check if logging level is enabled before calling the logger because the clause inside the log call is otherwise evaluated and may impact performance:

```
if (_log.isDebugEnabled()) {  
    _log.debug(DO_AN_EXPENSIVE_FUNCTION_CALL);  
}
```

Exercises

Implement Gradebook Logging

Introduction

In this exercise, you will add logging to Gradebook service so that every time an assignment is created, a log entry is created. We will also create a module specific Log4J configuration file where you can define persistent logging levels as well as configure appenders.

Overview

- ① Add SLF4J API dependency to service module's build.gradle
- ② Add logging code to Gradebook service
- ③ Create and configure module-log4j.xml

Snippets and resources for this exercise are in `snippets/chapter-06/08-implement-logging`.

The complete Gradebook application **solution** is in `solutions/solutions-chapter-06/solution-06-gradebook`.

Add Required Dependencies to the build.gradle

1. **Open** the `build.gradle` of the *gradebook-service* module.
2. **Add** the following line to the dependencies section:

```
...  
compileOnly group: "org.slf4j", name: "slf4j-api", version: "1.7.2"  
...
```

3. **Save** the file.
4. **Right-click** on the *gradebook-service* module.
5. **Choose** *Gradle → Refresh Gradle Project*.

Add Logging Code to Gradebook Service

1. **Open** the `com.liferay.training.gradebook.service.impl.AssignmentLocalServiceImpl` class in the *gradebook-service* module.
2. **Add** logger variable instantiation to the very end of the class:

```
...  
  
private void updateAsset(  
    Assignment assignment, ServiceContext serviceContext)  
throws PortalException {  
  
    assetEntryLocalService.updateEntry(  
        serviceContext.getUserId(), serviceContext.getScopeGroupId(),  
        assignment.getCreateDate(), assignment.getModifiedDate(),  
        Assignment.class.getName(), assignment.getAssignmentId(),  
        assignment.getUuid(), 0, serviceContext.getAssetCategoryIds(),  
        serviceContext.getAssetTagNames(), true, true,  
        assignment.getCreateDate(), null, null, null,
```

```

        ContentTypes.TEXT_HTML,
        assignment.getTitle(serviceContext.getLocale()),
        assignment.getDescription(), null, null, null, 0, 0,
        serviceContext.getAssetPriority());
    }

    private static final Logger _log = LoggerFactory.getLogger(AssignmentLocalServiceImpl.class);
}

```

1. Add logging code the end of `addAssignment()` method:

```

...
updateAsset(assignment, serviceContext);

if (_log.isInfoEnabled()) {
    _log.info("User " + userId + " added an assignment.");
}

// Start workflow instance and return the assignment.

return startWorkflowInstance(userId, assignment, serviceContext);
}

...

```

1. Press `CTRL+SHIFT_O` to resolve imports.
2. Save the class. Changes should be deployed automatically.

If you test adding assignment you won't see any messages in the log yet because the logging level for the package `com.liferay` is set by default to **ERROR**. Next we will add a Log4J configuration file and set the logging level for debugging purposes to **DEBUG**.

Create and Configure module-log4j.xml

1. Copy the file `module-log4j.xml` file from snippets folder `snippets/chapter-06/09-implement-logging` to the `src/main/resources/META-INF` in `gradebook-service`.

Changes should be deployed automatically. Test logging functionality by creating a new assignment. You should a similar entry in your log:

```
2018-09-24 09:18:50.880 INFO [http-nio-8080-exec-2][AssignmentLocalServiceImpl:148] User 20139 added an assignment.
```

Testing

In this section we'll briefly discuss some general principles, practices and guidelines of software testing. The practical part of the section will focus on integration testing with Arquillian framework (<http://arquillian.org>).

Why to Test?

Obviously humans make mistakes and create bugs in the software but it's not everything what software testing is about. Here are some reasons what testing is for:

- Finding bugs and defects in the software (functional testing)
- Verifying interaction between software components (integration testing)
- Checking if both functional and non-functional requirements are met (system testing)
- Checking if user acceptance criteria are met (user acceptance testing)
- Managing development lifecycle (regression testing)

Testing provides concrete metrics about the state of your software. When properly done, it inevitably improves software quality and finally, customer satisfaction.

What to Test?

Things that generally should be tested include:

- The "business logic"
- CRUD functionalities
- Code or functionalities that are intensively used everywhere in the application
- Code or functionalities developed by multiple developers
- Code or functionality that changes often

Quantity vs. quality should however be considered: having less, but good quality and well targeted tests is often better than the opposite.

Functional vs Non-Functional Testing

On the top level, software testing is often divided in two main categories: functional and non-functional testing.

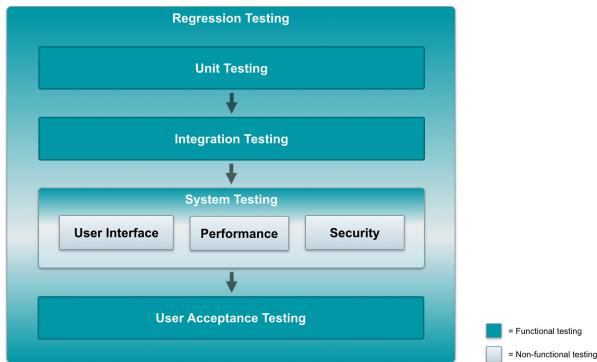
The purpose of **functional testing** is to verify *what* the system does against known requirements. We give tests some input and expect certain kind of output. Examples of functional testing are unit and integration testing.

Non-functional testing is to verify the way the system works. It's supposed to check that all of the components are interacting as designed, the system as a whole works as defined, within agreed performance and capacity levels and is ready for production. Examples of this type of testing are for example performance and security testing.

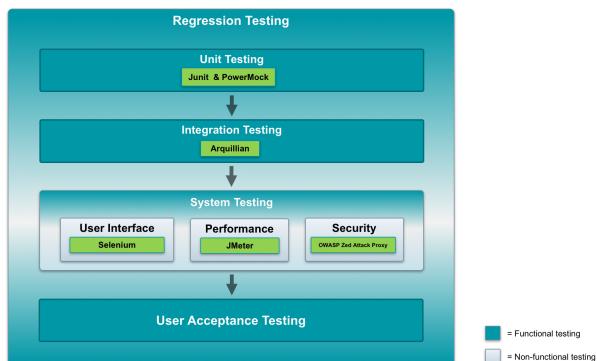
Both functional and non-functional testing categories are needed. While functional testing often takes the major role in testing, performance and load tests can expose defects that otherwise would pass the process and go to production. Fixing issues in production is often more difficult - and more expensive.

Stages of Testing

Properly done, testing is always done in multiple stages. The smallest software components like methods are tested first, then their interaction, next the complete system and lastly, the user perception.



The diagram below shows some tooling examples for the different stages:



General Testing Recommendations

Do a proper **testing plan** and try to test on all the stages of the development cycle.

A good plan takes care of **regression**. Especially in the beginning of application lifecycle there's typically lots of changes in the code, making it prone to regression.

It's practically not possible to cover everything and every possible use case. **Focus on the critical parts** first and then extend the coverage but not at the expense of quality. Rely on automation tools.

Test **borderline cases**. Boundary values when it comes to input fields on the user interface or calculations are a popular living area of bugs. This is also area, where certain kind of security breaches happen.

Keep tests **simple**. Creating tests can be time taking. Try to keep tests simple so that they provide exactly the information they should to and remain maintainable over code changes and lifecycle. If your tests seem to be too complex it might be a sign of too complex code in the application itself.

Testing Types

Let's take a brief overview of some of the most common functional testing types.

Unit Testing

Unit testing is the first stage of testing. It's meant for the smallest units of code, usually done against methods. Unit tests should be atomic and have only minimal dependencies to other units of code.

A good unit test should:

- Test an isolated component only, with no integrations.
- Be able to fail. If a test cannot fail, it's useless.

- Have only one unambiguous reason to fail (often hard to reach).
- Not affect the data other test cases rely upon.
- Not rely on other tests.
- Test just one thing and test it well.
- Not have any conditional logic.
- Be reliable.
- Be repeatable.
- Be platform independent.
- Have self-documenting method names

Liferay uses JUnit framework for unit testing. Below is a simple example of a JUnit test:

```
public class SampleTest {

    @Before
    public void setUp() {

        _car = new Car();
    }

    @Test
    public void testStartCar() {

        _car.start();

        assertEquals(true, car.isEngineRunning());
    }

    protected Car _car;
}
```

It's often not possible to isolate code under test from its dependent components. Mocking is an approach to "fake" those dependent components and help to keep the tested code isolated.

Mocking often increases test code complexity and decreases maintainability. In such cases using integration testing instead should be considered.

Below is an example of JUnit test using PowerMock Mockito extension:

```
@PrepareForTest(
{
    CarService.class
}
)
@RunWith(PowerMockRunner.class)

public class SampleTest {

    @Before
    public void setUp() {

        when(carService.getOilLevel(Car.class)).thenReturn(4);
    }

    @Test
    public void testStartCar() {

        Car car = new Car();

        assertTrue(carService.getOilLevel(car) > 0);
    }
}
```

```

    @Mock
    CarService carService;
}

```

Integration Testing

In integration testing a component is tested as a whole with all its integrations to other modules and services. Integration testing indeed tries to expose defects in interfaces and interaction between integrated components

Liferay uses Arquillian Framework (<http://arquillian.org>) and its own Arquillian extension for integration testing. We'll discuss Arquillian testing more in detail later but below is an example, how an Arquillian tests could look like:

```

@RunWith(Arquillian.class)
public class BlogsEntryLocalServiceImplTest {

    @ClassRule
    @Rule
    public static final AggregateTestRule aggregateTestRule =
        new LiferayIntegrationTestRule();

    @Test
    public void testAddDiscussion() throws Exception {
        ServiceContext serviceContext =
            ServiceContextTestUtil.getServiceContext();

        BlogsEntry blogsEntry = BlogsEntryLocalServiceUtil.addEntry(
            TestPropsValues.getUserId(), StringUtil.randomString(),
            StringUtil.randomString(), new Date(), serviceContext);

        _blogsEntries.add(blogsEntry);

        long initialCommentsCount = CommentManagerUtil.getCommentsCount(
            BlogsEntry.class.getName(), blogsEntry.getEntryId());

        CommentManagerUtil.addComment(
            TestPropsValues.getUserId(), TestPropsValues.getGroupId(),
            BlogsEntry.class.getName(), blogsEntry.getEntryId(),
            StringUtil.randomString(),
            new IdentityServiceContextFunction(serviceContext));

        Assert.assertEquals(
            initialCommentsCount + 1,
            CommentManagerUtil.getCommentsCount(
                BlogsEntry.class.getName(), blogsEntry.getEntryId()));
    }
    ...
}

```

End-to-End Testing

End-to-End testing, sometimes also called user interface or browser testing strives to emulate and test the (human) interaction with an application. Probably the most common testing framework in this category is Selenium (<https://www.seleniumhq.org>).

Selenium's core component, WebDriver emulates browsers and is also behind Arquillian's Graphene extension (<http://arquillian.org/arquillian-graphene/>). Below is an example using Arquillian Graphene for Liferay portlet user interface testing. The `testInstallPortlet()` get the current webpages source and checks if the portlet is found. The other test, `testAdd()` tests the simple calculator portlet by setting form values:

```

@RunAsClient
@RunWith(Arquillian.class)
public class BasicPortletFunctionalTest {

```

```

@Deployment
public static JavaArchive create() throws Exception {
    ...
}

@Test
public void testAdd() throws InterruptedException, IOException, PortalException {
    _browser.get(_portletURL.toExternalForm());
    _firstParameter.clear();
    _firstParameter.sendKeys("2");
    _secondParameter.clear();
    _secondParameter.sendKeys("3");

    _add.click();
    Thread.sleep(5000);
    Assert.assertEquals("5", _result.getText());
}

@Test
public void testInstallPortlet() throws IOException, PortalException {
    _browser.get(_portletURL.toExternalForm());

    final String bodyText = _browser.getPageSource();

    Assert.assertTrue("The portlet is not well deployed", bodyText.contains("Sample Portlet is working!"));
});

}

@FindBy(css = "button[type=submit]")
private WebElement _add;

@Drone
private WebDriver _browser;

@FindBy(css = "input[id$='firstParameter']")
private WebElement _firstParameter;

@PortalURL("arquillian_sample_portlet")
private URL _portletURL;

@FindBy(css = "span[class='result']")
private WebElement _result;

@FindBy(css = "input[id$='secondParameter']")
private WebElement _secondParameter;

}

```

Introducing Arquillian

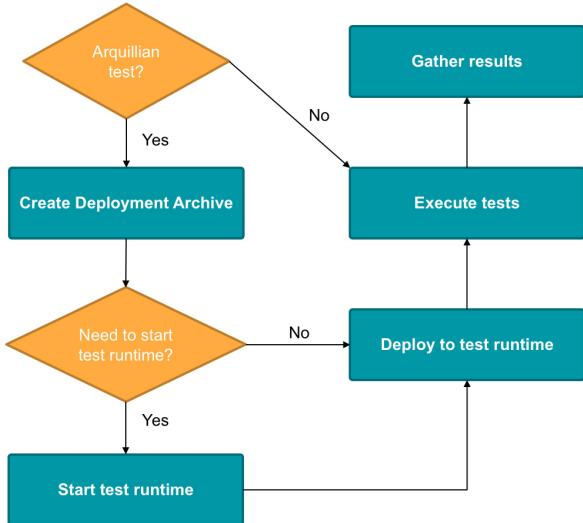
Arquillian is an integration and functional testing platform for Java. Running tests against a dedicated container, it provides a way to test the Liferay modules comprehensively, in a real environment and with real dependencies.

Arquillian can:

- Start and stop the test container
- Create a deployable test archive
- Deploy and un-deploy test archive
- Enrich test case with dependency injections

- Execute tests inside the container
- Captures the results

Below is a diagram of Arquillian test:



Arquillian components

Arquillian framework has four main component:

- Test runners
- Containers
- Test enrichers
- Run modes

Arquillian **Test Runner** extends the JUnit Runner class and takes care of running the Arquillian decorated test class.

Runner is set with @RunWith annotation:

```

@RunWith(Arquillian.class)
public class AssignmentLocalServiceTest {
    ...
}
    
```

Test container is the server runtime where tests are run. Test container should not be referenced directly in the tests.

Container types support by Arquillian are:

- Remote
- Managed
- Embedded

Embedded container is not supported by Liferay Arquillian extension.

Test Enrichers are injections resources or service injections into Arquillian test classes:

- **@Resource** - Java EE resource injections
- **@EJB** - EJB session bean reference injections
- CDI-supported injections

If you inject Liferay service references in your test classes, just use **@Inject** instead of **@Reference**:

```
@Inject
private AssignmentLocalService _assignmentLocalService
```

Run Mode defines the run context for the test:

- Container mode (default):
 - Deploy test to test container
 - Execute tests in the test container
- Client mode:
 - Deploy required components to test container
 - Execute tests as a client, outside the container
 - Suitable for testing web services or user interfaces
 - Is set with `@RunAsClient` annotation

An example of Arquillian test class in client run mode.

```
@RunAsClient
@RunWith (Arquillian.class)
public class GradebookPortletTest {
    ...
}
```

Arquillian Liferay Extension

Arquillian Liferay Extension helps is an extension for Liferay OSGI in-container deployments. It has two components:

- Arquillian Remote Container for Liferay
- Arquillian Deployment Scenario Generator

Steps to Implement Liferay Arquillian Tests

- ① Set up Liferay Tomcat server bundle with JMX support enabled
- ② Define Arquillian test dependencies
- ③ Define BND file for creating the test archive
- ④ Configure container with arquillian.xml
- ⑤ Create test cases

Links and Resources

- **Arquillian Testing Framework**
<http://arquillian.org>
- **Arquillian Graphene Documentation**
<http://arquillian.org/arquillian-graphene/>
- **Arquillian Extension for Liferay** https://dev.liferay.com/develop/tutorials/-/knowledge_base/7-0/arquillian-extension-for-liferay-example
- **Selenium**
<https://www.seleniumhq.org>

Debugging

Your application might be failing integration tests or crashing in the production because of unknown reason. The code complexity might have gotten to the point where you would not be sure any more, how it all works. Developers make mistakes and tests can't cover every possible use scenario and there debugging comes to help.

Although literally meaning resolving bugs, debugging covers a great amount of different approaches, tools and methodologies under its umbrella. Here, for the sake of training, we limit the scope to basics of using IDE debugger tool which enables monitoring and controlling the execution of a program, setting breakpoints and changing values in memory. We'll also briefly discuss some methods of troubleshooting and debugging issues in production, where there's no source code or IDE available.

Introducing JPDA

JPDA is the backbone of Java debugging. It stands for **J**ava **P**latform **D**ebugger **A**

JDI stands for the **J**ava **D**ebug **I**nterface and is the user interface definition layer. In our training context the implementation is Eclipse based Dev Studio.

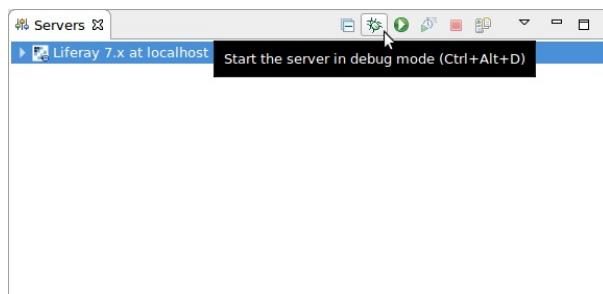
JDWP (**J**ava **D**ebug **W**ire **P**rotocol) defines the communication between the debuggee and debugger.

JVM TI (**J**ava **V**M **T**ool **I**nterface) API defines the debugging services a VM provides and a native programming interface for use by debugging, monitoring and profiling tools. It provides a way to inspect the state of running JVM and to modify and control execution of applications running in the JVM. JVM TI provides an event based support for bytecode instrumentation, the ability to alter the Java virtual machine bytecode instructions. JVM TI clients are called *agents*.

While JVM TI is a native programming interface, requiring agents to be written in C / C++, **java.lang.instrument** is a higher level API on top of JVM TI, providing a Java programming interface for bytecode instrumentation which is one of the most important enablers of Java debugging.

Enabling Debug Mode in IDE

To enable debugging, the JVM has to start in debug mode with JPDA enabled. If you are using Tomcat server adapter in Liferay Dev Studio, IDE takes care of enabling JPDA and you only need start the server in debug mode.



Tomcat can also be started manually in debug mode simply by adding "jpda" startup option to catalina script:

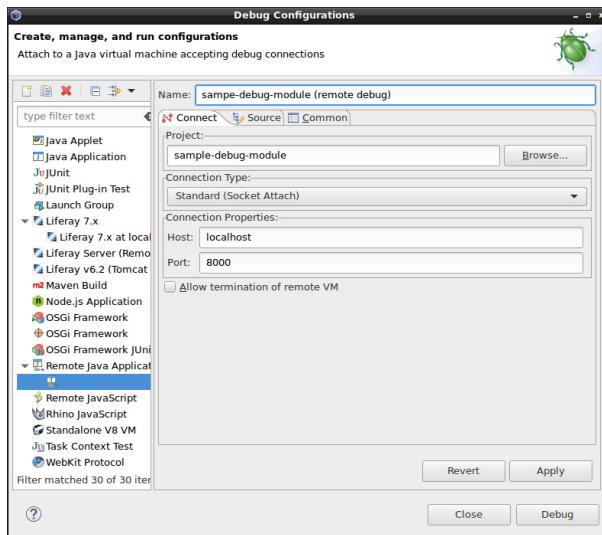
```
TOMCAT_HOME/catalina.sh jpda start
```

Generally, servlet containers and Java EE servers can be started in debug mode by adding appropriate Xagentlib options for the JVM:

Enabling manually in JVM options:

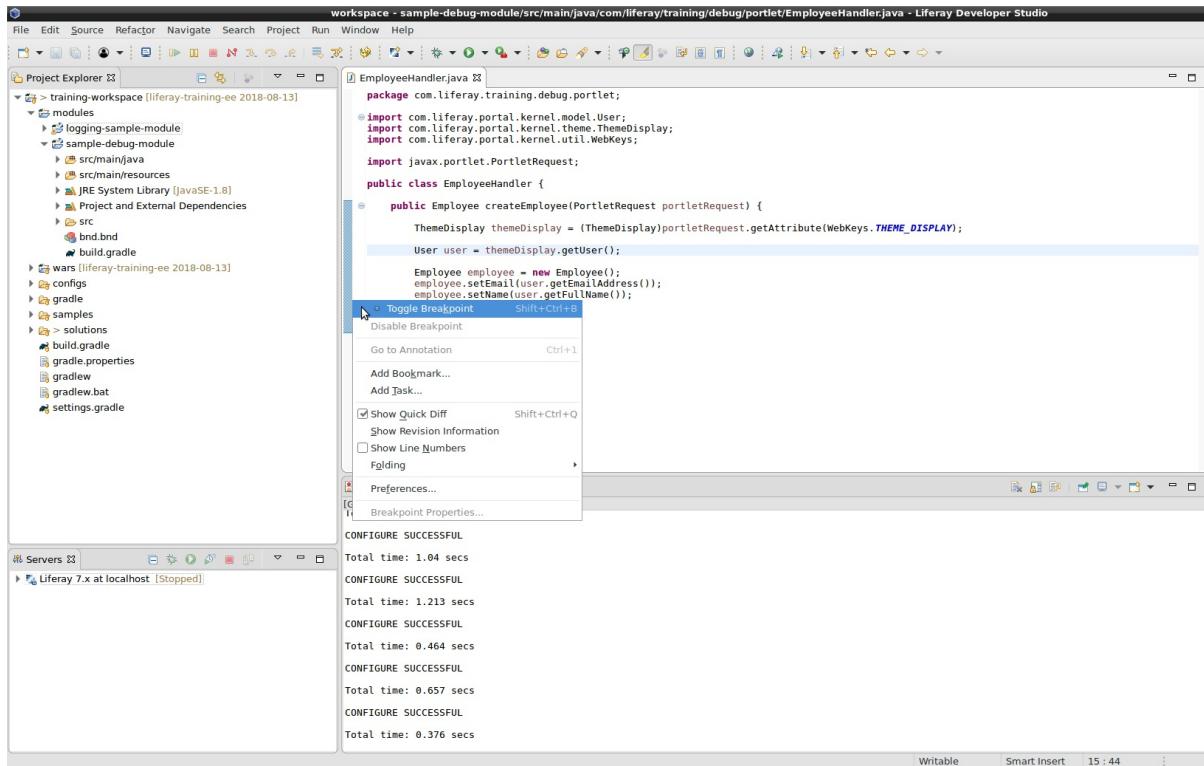
```
-Xagentlib:jdwp=transport=dt_socket,server=y,suspend=y,address=8000
```

When debugging a remote server (including server not launched from IDE), a remote debugging connection profile, corresponding the JDWP options of JVM, has to be added in the IDE :

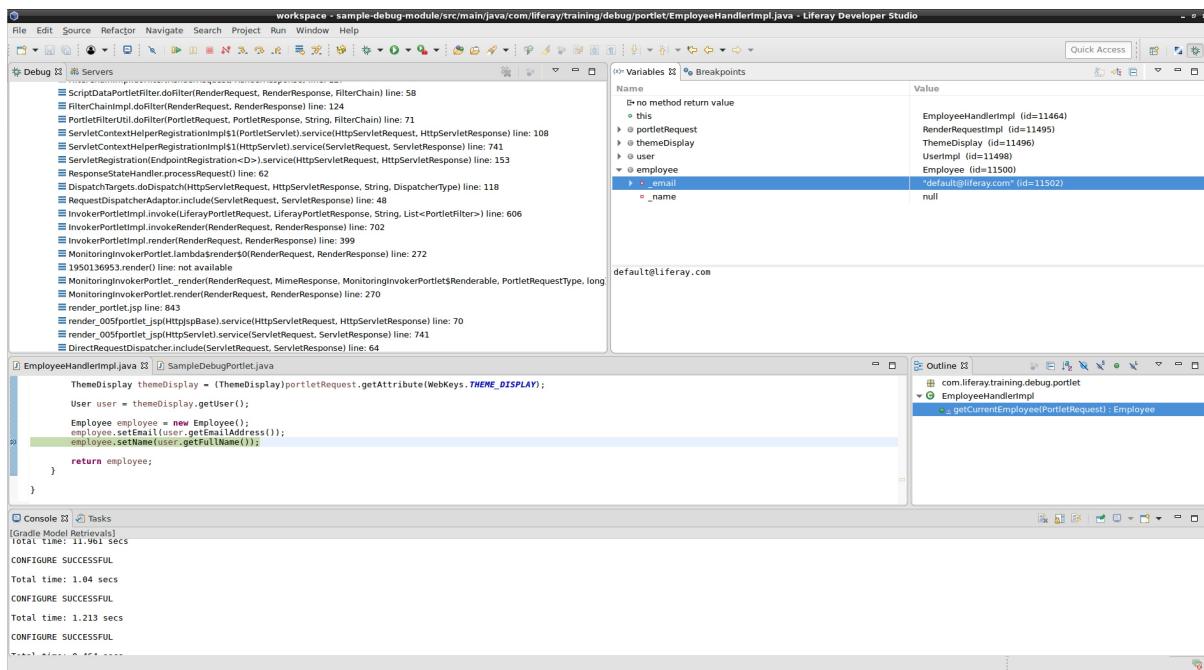


Debugging Basics

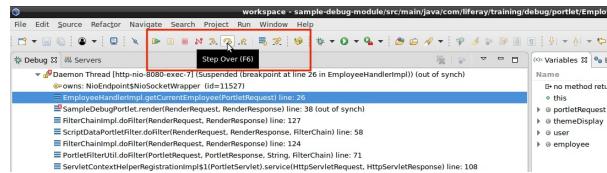
In running debugger we are often interested to see why a certain variable gets an unprecedented value and what's the application states when some code block, method or call is reached. For that purpose we set **breakpoints**, which can be static, meaning that we always stop the execution at that point or conditional, meaning that the execution only stops when a certain condition is met. We can also set variable watchpoints which are triggered when a variable value is changed.



At the breakpoint we can not only watch but also **change variable values**, for example to check if the issues we were resolving is dependent on that specific value. Double-clicking the value allows to change it.



When we have an idea about the area where the issue comes from, we can set a breakpoint and use debugger stepping functionalities to execute the code line by line forward, step into a method call or even go backwards in the execution:



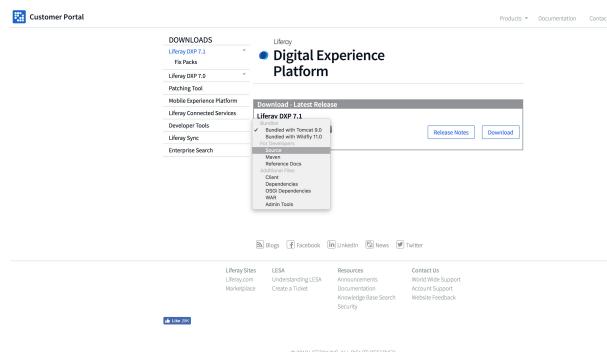
Setting Up Liferay Source Code for Debugging

Sometimes you might want to put a breakpoint at Liferay platform code to see what happens at some specific point. For that purpose you have to setup the portal source code for the IDE. Liferay provides source code packages for all the portal releases, including fix packs. To setup the source code, you need to:

1. Download and extract source code package
2. Import source code project into workspace . Set up debugger configuration (if remote)

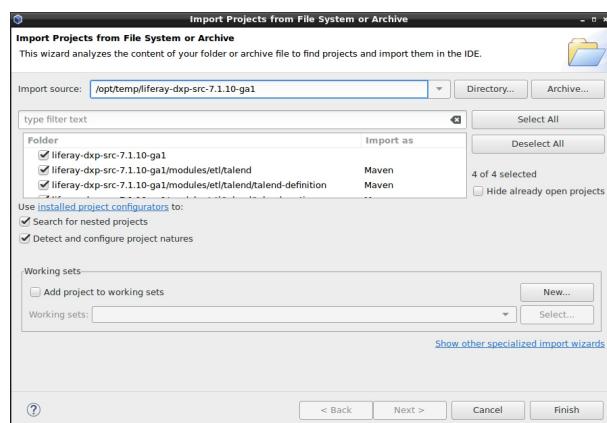
1. Download Source Code

Liferay commercial release source codes can be downloaded from the customer portal at <https://customer.liferay.com>:



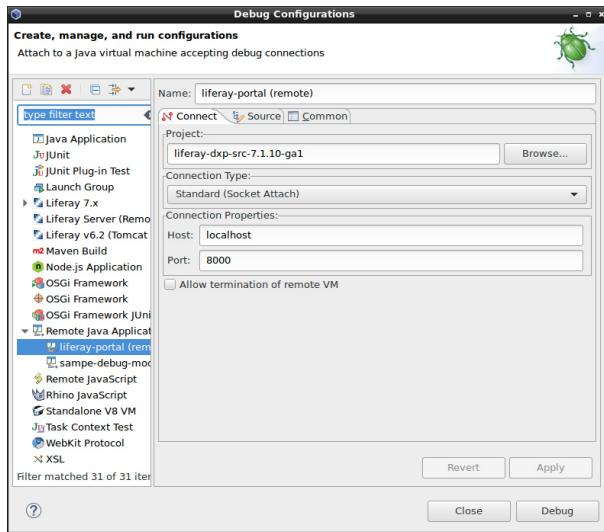
2. Import Source Code into Workspace (Eclipse)

When you extract the source code package and try to import that into Eclipse, you might notice that there's no Eclipse project files present. To import the code onto workspace and to auto create the project files, use *File->Open Projects From File System*:



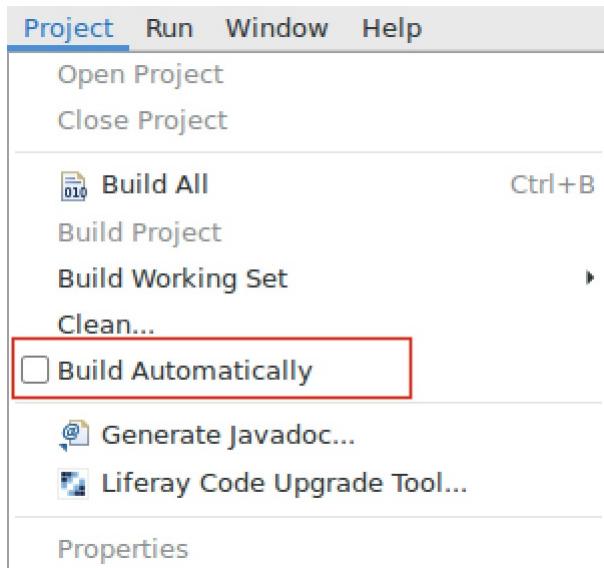
3. Set Up Debugger Configuration (Remote Debugging)

After the source code project is setup in the workspace, you need to create a debugger configuration for that:

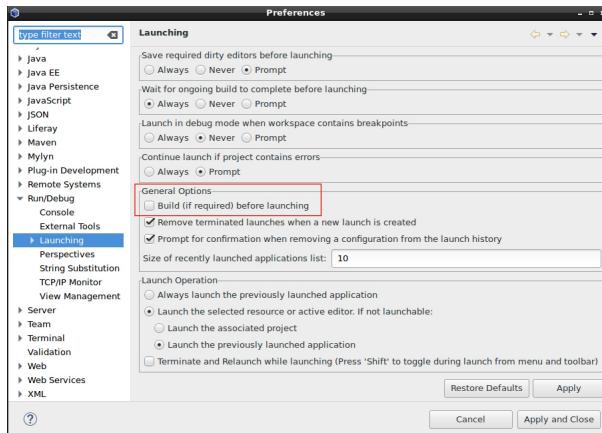


Now you can launch the debugging session profile but before launching there's one more thing to check. When you alter the portal source code or launch the debugging profile, the IDE tries to build the project automatically by default.

Disable the automatic project building in the project settings:



Also disable building at launch at Eclipse workspace preferences in *Window -> Preferences-> Run/Debug -> Launching:*



Overview of Debugging in Production

Debugging in production is not only different of its nature but also of its toolset. At the development stage the development environment, source code and iDE debugger are available. Often there's more time to resolve issues. The nature of debugging is forward tracing where you can run the program line by line to see what's causing or would possibly be causing an issue. Testing also falls into this category of forward tracing debugging.

Debugging issues in production systems on the other hand often relies on interpreting the issue symptoms from logs, stack traces and thread dumps. Often there are strict time constraints and also because of security policies, a limited tool set available. Other important categories of debugging in production involve monitoring and profiling JVM and applications with the means of, for example, APM tools. Here we limit the scope of discussion to introducing thread dumps and JVM getting memory statistics with Oracle JDK tools.

Thread Dumps

Thread dump shows information what each thread is doing at a given time. This information includes the exact method call being executed and the thread state at the time point.

Thread dumps are crucial in troubleshooting when system is having performance issues and freezes. They show which threads and which function calls might be blocking the execution.

To create a thread dump of JVM with JDK tools, you first have to find the process ID of the running JVM. Depending on the operating system there are multiple ways to get that information but running JDK tool *jps* from command line is handy for the purpose:

```
jps -v
```

After you've found the process id (PID), you can use the JDK tool Jstack to create a thread dump of running JVM:

```
liferay@liferay> jstack -l PID > /opt/tmp/thread_dump.txt
```

The internals of thread dumps are beyond the scope of discussion here but the excerpt below shows an example of a thread being at WAITING state:

```
...
"liferay/monitoring-1" #174 daemon prio=5 os_prio=0 tid=0x00007f45f1067800 nid=0x2be1 waiting on condition
[0x00007f45a31fe000]
    java.lang.Thread.State: WAITING (parking)
        at sun.misc.Unsafe.park(Native Method)
```

```

    - parking to wait for  <0x00000000c7bcd0d0> (a java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject)
      at java.util.concurrent.locks.LockSupport.park(LockSupport.java:175)
      at java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject.await(AbstractQueuedSynchronizer.java:2039)
      at com.liferay.portal.kernel.concurrent.TaskQueue.take(TaskQueue.java:258)
      at com.liferay.portal.kernel.concurrent.ThreadPoolExecutor._getTask(ThreadPoolExecutor.java:548)
      at com.liferay.portal.kernel.concurrent.ThreadPoolExecutor.access$500(ThreadPoolExecutor.java:37)
      at com.liferay.portal.kernel.concurrent.ThreadPoolExecutor$WorkerTask.run(ThreadPoolExecutor.java:672)
      at java.lang.Thread.run(Thread.java:748)

Locked ownable synchronizers:
- None

"LCS Worker 4" #171 prio=5 os_prio=0 tid=0x00007f45e002b800 nid=0x2bde waiting on condition [0x00007f45ac16d000]
  java.lang.Thread.State: WAITING (parking)

...

```

If you had a stability issue in the system, you'd probably start looking for threads in BLOCKED state, stealing the available threads and blocking the program execution, possibly caused by certain method calls. Then you would start to investigate, why those calls are blocking the execution.

When debugging issues in production it good to remember the **time aspect**: a thread could for example be blocked at some timepoint and that would be completely normal. If that very same thread would be blocked after 30 seconds, you'd probably be having an issue. That's why, when taking thread dumps, it's important to take several dumps during the problematic application state.

Monitoring Memory with JStat

While thread dumps give invaluable information of CPU state, often JVM problems are memory related. You application might be leaking memory resources or there might just be an resource allocation problem, during a peak load.

JDK tool JStat shows statistics of JVM memory usage and garbage collection. It both helps in detecting the memory leaks but also in tuning the JVM memory and garbage collection parameters.

The command line example below show the garbage collection and memory statistics with the sample rate of 500ms.

```

liferay@liferay-VirtualBox $ jstat -gcutil -t 27284 500ms
  Timestamp   S0    S1     E    O     M    CCS    YGC    YGCT    FGC    FGCT    GCT
  193093.6  24.19  0.00  95.19  57.11  91.63  82.49   450   10.111    8   3.835  13.947
  193094.1  24.19  0.00  95.19  57.11  91.63  82.49   450   10.111    8   3.835  13.947
  193094.6  24.19  0.00  95.19  57.11  91.63  82.49   450   10.111    8   3.835  13.947
  193095.1  24.19  0.00  95.19  57.11  91.63  82.49   450   10.111    8   3.835  13.947
  193095.6  24.19  0.00  95.19  57.11  91.63  82.49   450   10.111    8   3.835  13.947
  193096.1  24.19  0.00  95.19  57.11  91.63  82.49   450   10.111    8   3.835  13.947
  193096.6  24.19  0.00  97.09  57.11  91.63  82.49   450   10.111    8   3.835  13.947
  193097.1  24.19  0.00  98.98  57.11  91.63  82.49   450   10.111    8   3.835  13.947
  193097.6  24.19  0.00  98.98  57.11  91.63  82.49   450   10.111    8   3.835  13.947
  193098.1  24.19  0.00  98.98  57.11  91.63  82.49   450   10.111    8   3.835  13.947
  193098.6  24.19  0.00 100.00  57.11  91.63  82.49   450   10.111    8   3.835  13.947
  193099.1  0.00  95.58   1.89  57.14  91.63  82.49   451   10.125    8   3.835  13.960
  193099.6  0.00  95.58   1.99  57.14  91.63  82.49   451   10.125    8   3.835  13.960
  193100.1  0.00  95.58   1.99  57.14  91.63  82.49   451   10.125    8   3.835  13.960
  193100.6  0.00  95.58   1.99  57.14  91.63  82.49   451   10.125    8   3.835  13.960
  193101.1  0.00  95.58   1.99  57.14  91.63  82.49   451   10.125    8   3.835  13.960
  193101.7  0.00  95.58   1.99  57.14  91.63  82.49   451   10.125    8   3.835  13.960
  193102.2  0.00  95.58   1.99  57.14  91.63  82.49   451   10.125    8   3.835  13.960
  193102.7  0.00  95.58   1.99  57.14  91.63  82.49   451   10.125    8   3.835  13.960

```

For more information about JStat, please see the JDK documentation at
<https://docs.oracle.com/javase/8/docs/technotes/tools/unix/jstat.html>.

Exercises

Debug the Gradebook

Introduction

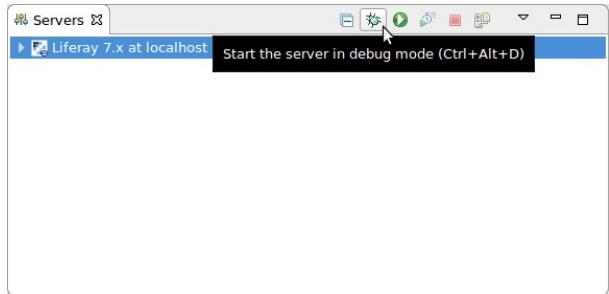
In this exercise, you will learn debugging basics: how to start Tomcat server in debug mode, how to set breakpoints and control program execution with debugger steps.

Overview

- 1 (Re)start Tomcat in debug mode
- 2 Add breakpoint
- 3 Use debugger steps to control program execution

(Re)start Tomcat in Debug Mode

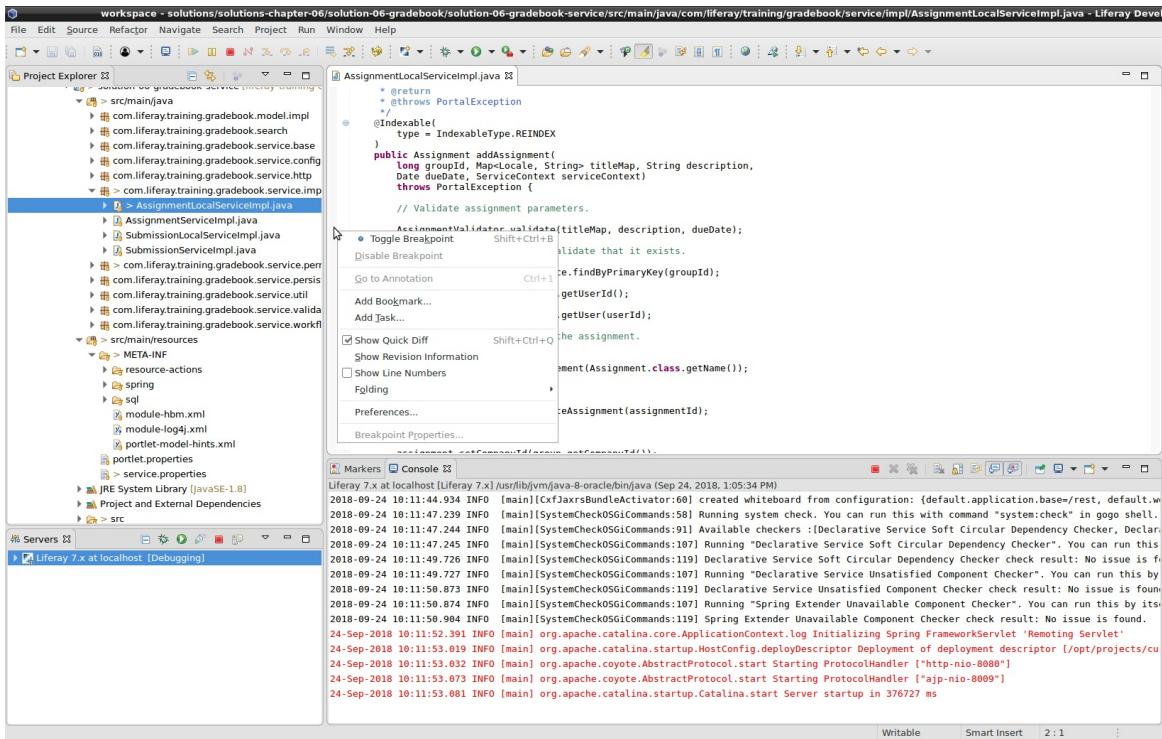
1. **Stop** the Liferay server.
2. **Start** the Liferay server in debug mode by clicking the "bug" icon:



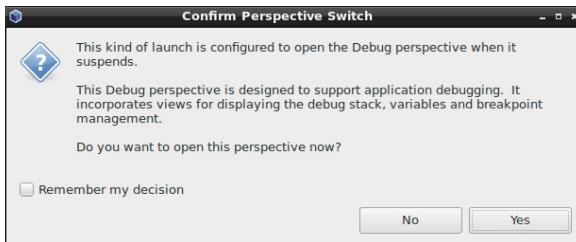
While waiting the server to start, please go ahead and set a breakpoint.

Add Breakpoint

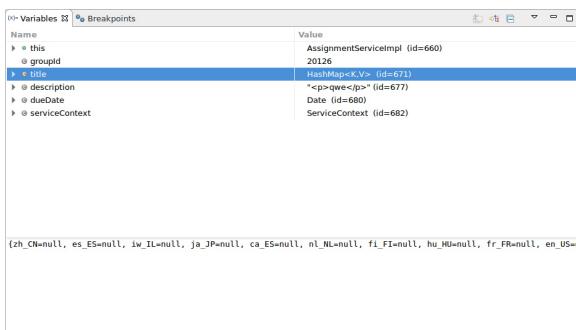
1. **Open** the `com.liferay.training.gradebook.service.impl.AssignmentLocalServiceImpl` class in the `gradebook-service` module.
2. **Find** the `addAssignment()` method in the beginning of class.
3. **Find** the `AssignmentValidator.validate()` call in the beginning of the method.
4. **Right-click** on the left side of the line (in margin) and select `Toggle Breakpoint` from the context menu:



5. Add a new assignment in the Gradebook app in your browser. Adding an assignment should pause in the breakpoint you just defined. In the IDE there should be a dialog asking to switch to the debug perspective.

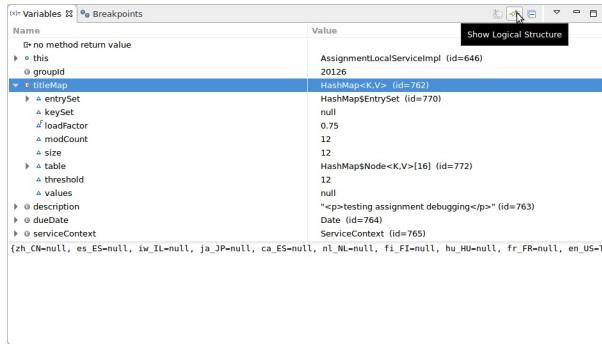


6. Click Yes to switch to the debug perspective. The application execution is now halted at the breakpoint. Let's check what are values for our assignment title.
7. Click the Variables to show the variables stack:

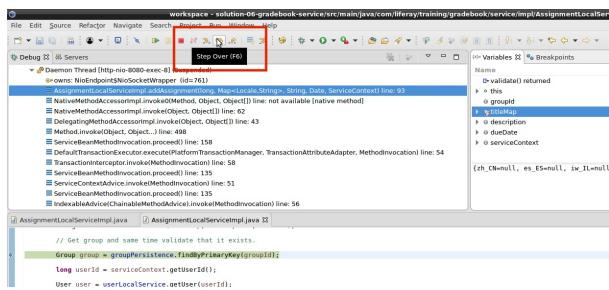


From this panel you can check and manipulate variable values on the fly. Let's click the *titleMap* variable and browse it. In this case you can see the Map values easily in the bottom panel but often it can be arduous to find some specific Array or Map value there.

8. Click Show Logical Structure button to show Map's logical structure and browse the *titleMap* variable again.



In debug mode you can fully control application execution. You can move back and forward as well as dive into the method calls. Try *Step over* to take one step forward in the method execution. Try *Step into* to go to the method behind the call. Watch variables panel while you move. Click *Resume* to continue application execution:



After you've done with the exercise you can restart the server in normal mode.

Managing Deployment Issues

In this section we will discuss module deployment related issues and introduce some methods for resolving them. We've already learned that only the Liferay web application itself is deployed to and managed by the Java application server. All the Liferay applications run in an Liferay embedded OSGi container and are deployed there. That's why resolving module deployment issues typically involves Gogo shell.

Before going into typical deployment issues and resolving them, let's first have an overview on module deployment and undeployment in Liferay.

Module Deployment Overview

Depending on the situation and development stage, there are multiple ways to deploy modules:

- Using autodeploy folder
- Deploy from Control Panel
- Deploy from Gogo shell
- In development environment:
 - Deploy from Dev Studio
 - Use Blade CLI
 - Use Gradle command line

Autodeploy

Autodeploy is the default method of deploying modules to Liferay. You copy the module JAR or legacy WAR into the predefined autodeploy folder and it gets automatically deployed to the OSGi container. By default this folder locates to *LIFERAY_HOME/deploy* but it can be configured in the portal properties. Look for *auto.deploy* prefixed setting in the *portal.properties* file (<https://github.com/liferay/liferay-portal/blob/7.1.x/portal-impl/src/portal.properties>)

In a clustered environment it should be noticed that autodeploy mechanism is not cluster aware: modules deployed to a node's autodeploy folder are deployed only to that node.

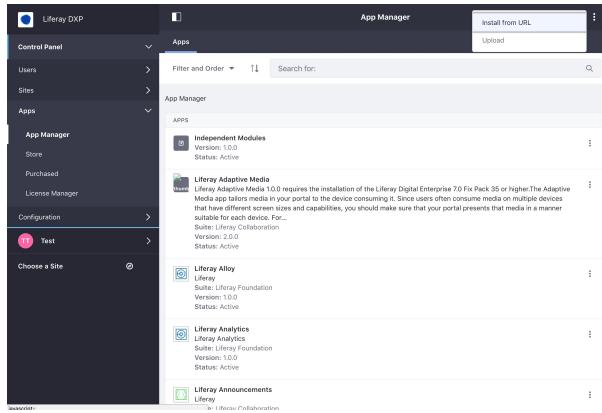
When module JAR is read from the autodeploy folder, several things happen in the file system:

- JAR are copied to *LIFERAY_HOME/osgi/modules*
- JSPs are compiled to *LIFERAY_HOME/work*
- Module state is copied to *LIFERAY_HOME/osgi/state*
- Static resources cached in:
 - *TOMCAT_HOME/work*
 - *TOMCAT_HOME/temp*

It's worth noticing that while autodeploy method copies module JARs to *LIFERAY_HOME/osgi/modules*, not all the deployment methods do that and for the OSGi container runtime, it's not even needed. Bytecode in *LIFERAY_HOME/osgi/state* is from where the OSGi container uses the module and you shouldn't ever edit contents of that folder runtime. However, if you delete the module JAR in *modules* folder, the module gets also undeployed.

Deploying from Control Panel

In scenarios where autodeploy folder access is restricted or policies otherwise dictate, custom modules can also be deployed platform control panel. This method still uses the *autodeploy* folder internally, meaning that module JARs deployed this way will also be copied to the *LIFERAY_HOME/osgi/modules* folder:



Deploying from Gogo Shell

Modules can also be installed from within the Gogo shell directly using standard Felix shell command.

```
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.

Welcome to Apache Felix Gogo

g! install file:/opt/install/com.liferay.training.bundle-1.0.jar
Bundle ID: 961
```

There are few things to notice:

- This method doesn't start the bundle automatically
- Module JAR is **not** copied to *LIFERAY_HOME/osgi/modules*.

Deploying in Dev Studio

When you deploy the modules from Liferay IDE, using one of its alternatives, *autodeploy* folder is, again, not used. This method copies the compiled bytecode directly to *LIFERAY_HOME/osgi/state/BUNDLE_ID*.

Deploying Using Blade

Modules can be deployed using Blade CLI. This method detects the running JVM automaticall, builds and deploys the module directly to *LIFERAY_HOME/osgi/state/BUNDLE_ID*. This method can only be used for JARS and not legacy WARS:

```
liferay@liferay$ blade deploy
```

In development environment there are some benefits in using this method. Run in the folder hierarchy, this command can deploy all the modules in the subdirectories, too. The *watch* command is useful is you want to have a "hot deploy" feature directly from command line. It watches changes in the source folder and deploys automatically:

```
liferay@liferay$ blade deploy -w
```

Deploying Using Gradle Command Line

There's also a Gradle deploy task available:

```
liferay@liferay$ ./gradlew -w
```

This method tries to copy the compiled module JAR into *WORKSPACE_DIR/bundles/osgi/modules*.

Deploying Legacy Modules

Liferay supports deploying legacy Java EE WAR style web applications into OSGi container. When a WAR file is copied to autodeploy folder, it gets automatically converted into WAB bundle which is then deployed to the OSGi container. The WAB is OSGi compatible bundle type and stands for **Web Application Bundle**. It is defined in the OSGi Compendium.

What Liferay's WAB conversion mechanism does, is practically creating an OSGi compatible *MANIFEST.MF* file, based mostly on the legacy XML configuration files and creating a compatible folder structure.

Wrapping a WAR deployment up, from file system's perspective:

1. A WAR archive is copied to *LIFERAY_HOME/deploy*
2. Liferay generates a WAB bundle on the fly
3. Bundle runtime bytecode is created in *LIFERAY_HOME/osgi/state*
4. WAR archive gets stored in *LIFERAY_HOME/osgi/war*

By default, the platform doesn't store the generated WAB. However, you can change that behaviour in *portal.properties*:

```
module.framework.web.generator.generated.wabs.store=true
module.framework.web.generator.generated.wabs.store.dir=${module.framework.base.dir}/wabs
```

Note on Marketplace Applications

Liferay Marketplace applications are packaged in Liferay's proprietary LPKG package format. While the autodeployment process of LPKG packages is the same as for any deployable package type, the LPKG packages are stored in their own folder *LIFERAY_HOME/osgi/marketplace*

Overview of Undeploying Modules

There are also multiple ways to undeploy modules:

- Deleting *LIFERAY_HOME/osgi/modules/MODULE.jar*
- Undeploy from Control panel
- Uninstall from Gogo shell

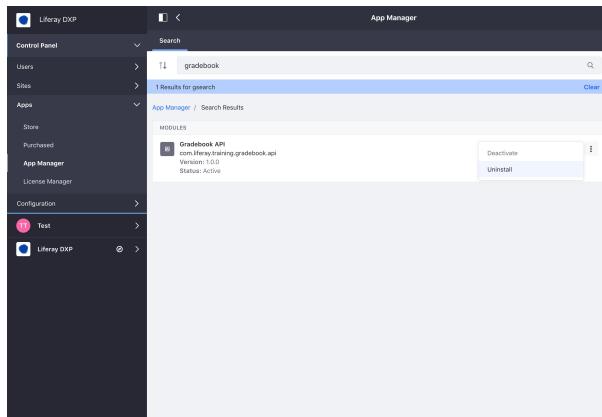
Let's take a look at these methods and some caveats in using them.

Deleting Module JAR

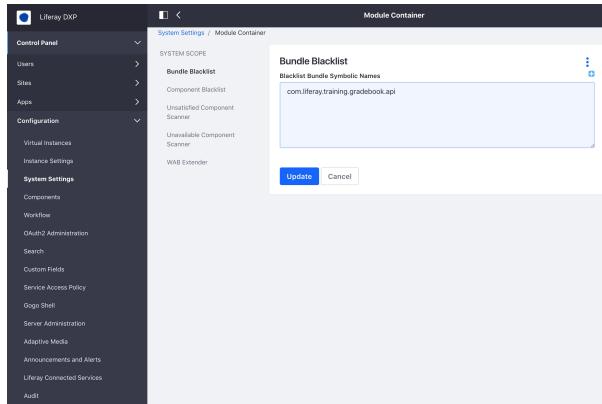
If the deployment method copied the JAR to *LIFERAY_HOME/osgi/modules/MODULE.jar*, removing this file undeploys the module automatically.

Undeploying from Control Panel

When you undeploy the module from Control Panel it removes the module state from *LIFERAY_HOME/osgi/state* but it **does not remove** the module JAR from *LIFERAY_HOME/osgi/modules*:



What's worth noting here are the security related restrictions: undeploying from Control Panel adds an entry for the Blacklister module preventing a reinstall of the module. Thus, before you can reinstall, you have to remove the blacklisting entry. Another thing worth noting here is that removing the Blacklister entry automatically reinstalls the module if the JAR is still present in *LIFERAY_HOME/osgi/modules*:



Undeploying from Gogo Shell

Modules can be uninstalled directly from OSGi container using the *uninstall* command. This method also **does not remove** the module JAR from *LIFERAY_HOME/osgi/modules*:

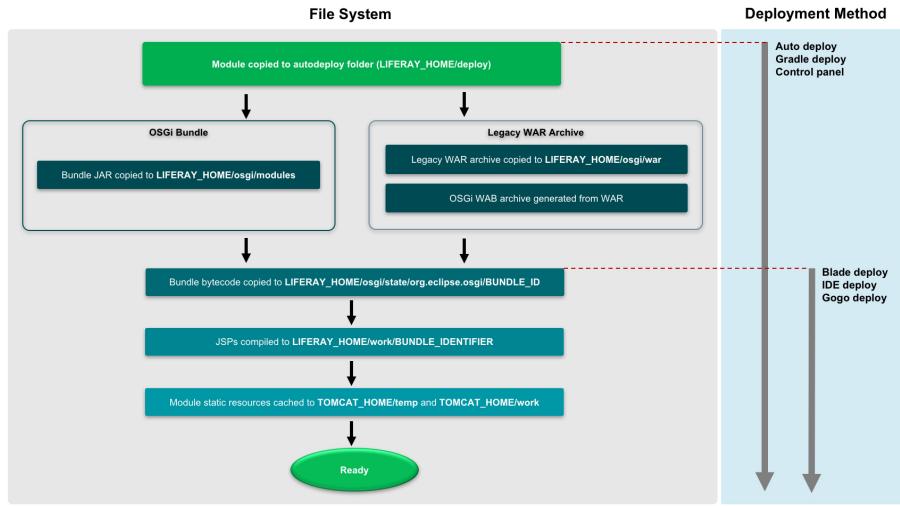
```
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.

Welcome to Apache Felix Gogo

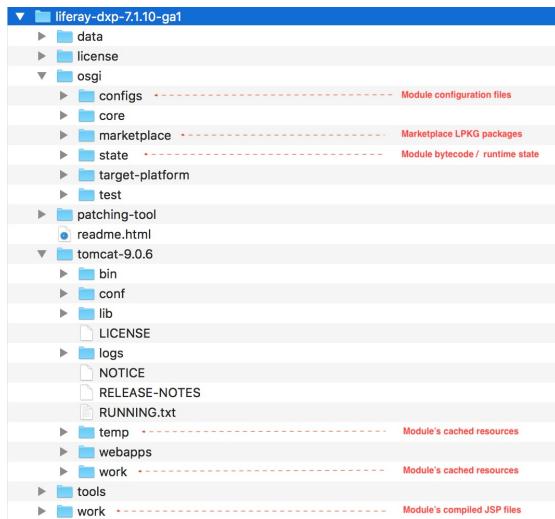
g! uninstall 961
```

Module Deployment Recap (Tomcat)

Understanding the file level changes during the deployment helps to troubleshoot deployment related issues. The following diagrams illustrate what happens in the file system during the deployment process:



Seen in the folder structure:



In case of any, not code related module problems, steps to clean module related resources are generally as follows:

1. Undeploy module
2. Shutdown portal
3. Delete module JAR, WAR or LPKG
4. Delete module resources from `LIFERAY_HOME/osgi/state`
5. Delete module configuration file `LIFERAY_HOME/osgi/configs`
6. Delete compiled JSPs from `LIFERAY_HOME/work`
7. Delete `TOMCAT_HOME/temp/*`
8. Delete `TOMCAT_HOME/work/*`

Troubleshooting Deployment Issues

Let's discuss case by case some of the most common deployment related issues:

- Undeployed module reappears after restart
- Module redeployment fails
- Module resources not refreshing
- Module doesn't start
 - Unsatisfied module dependencies

- Unsatisfied module requirements

Undeployed Module Reappears After Restart

Symptoms

Undeployed module is installed and active after restart.

Possible Cause

The module JAR, WAR, or LPKG file was not removed by the undeploy method.

Suggested Solutions

- Delete module JAR, LPKG or WAR file from the respective folder.

Module Redeployment Fails

Symptoms

Redeploying a module fails. Module is either not deployed at all or just doesn't start.

Possible Cause

Old module JAR file might be blocking the redeployment. This might happen for example, if you were using Gradle deploy or autodeploy and undeployed the module from within Gogo shell - which doesn't remove the module JAR. There might also be a blacklist entry for the module.

Possible Solutions

- Delete module JAR from *LIFERAY_HOME/osgi/modules*
- Check blacklisting settings in *Control Panel -> System Settings -> Platform -> Bundle Blacklist*

Module Resources not Refreshing

Symptoms

Redeploying module won't refresh static resources like CSS and Javascript files or compiled JSP files.

Possible Cause

Caching settings might prevent resources from refreshing. There might also be a system time skew or faulty timezone settings letting system to know that current resources are newer than the updated ones.

Possible Solutions

- Check that portal is running on development mode
- Check that both development and server machine clocks are synchronized
- Check that portal JVM is running on GMT or UTC timezone

To clear JSP cache in production you can also try clear the direct servlet cache from *Control Panel -> Server Administration -> Resources -> Clear the direct servlet cache*.

Module Gets Installed but Doesn't Start

This issue type usually falls in two possible categories:

1. Unsatisfied module dependencies
2. Unsatisfied module requirements

Unsatisfied Module Dependencies

Symptoms

Module deploys but remains in the *installed* state and doesn't become *active*.

Possible Cause

Module didn't get into *resolved* state because some of the module's dependencies is missing or couldn't get activated. This might also happen because of wrong dependency compile scope.

Possible Solutions

- Check the compile scopes in *build.gradle* if any resource should be included in the build (compileInclude)
- Use Gogo *diag* to find any missing dependencies in the container

Unsatisfied Module Requirements

Symptoms

Module is in *active* state but some of its service components seems not to be working.

Possible Cause

OSGi bundles and components have different lifecycles ; a bundle may start even if some of its components does not. Usually this happens when some of the components references (@Reference) is not satisfied i.e. not found or started. There might also be a circular reference.

Possible Solutions

- Find the failing component id by running *ds:unsatisfied BUNDLE_ID*.
Then run *scr:info COMPONENT_ID* and find unsatisfied references.
Find out why the reference was not available
- Run *ds:softCircularDependency* to detect circular references

Exercises

Troubleshoot Failing Module Deployment

Introduction

In this exercise, you will be provided an application consisting of two modules. Your task is to deploy these modules to Liferay and resolve **two issues** preventing the modules from starting and the application from working. Use Liferay log and Gogo shell commands for troubleshooting. Exercise is completed when you get a log message "**Task completed successfully!**" printed in the log.

Snippets and resources for this exercise are in [snippets/chapter-06/11-troubleshoot-failing-module-deployment](#).

Overview

- ① Copy exercises modules
- ② Deploy the modules
- ③ Resolve the issues

Copy Exercises Modules

1. **Open** the exercise snippets folder
2. **Copy** the *failing-api* and *failing-service* folders to the workspace's `modules` folder
3. **Right click** on the modules folder and select *Gradle -> Refresh Gradle Project*

Deploy the Modules

1. **Select** the two modules in the Project Explorer and drag & drop them on the Liferay server to deploy them.

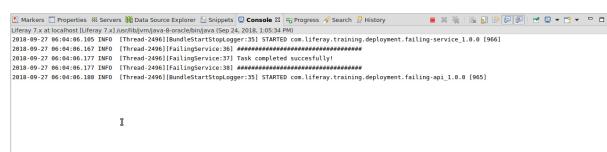
As there are still problems in the modules right now, they won't start and you won't get any log message.

Resolve the Issues

1. **Open** the Gogo shell and list bundles. You'll notice that the *failing-api* and *failing-service* are in installed state. Remember that if a bundle won't get into 'resolved' state, there's a dependency problem. Now, use the Gogo commands we learned in this training section and find out, what are the problems.

Hint: You might want to start with the tool for diagnosing dependency problems. Please ask your trainer for more exercise hints.

As you work with the code, the changes will be hot deployed automatically. The exercise is completed when you get a following message in the log:



```
Liferay 7.x at localhost [Liferay 7.4] JBoss Seam Java EE 7 Enterprise (Start 24. 2018, 10:34 PM)
2018-09-27 05:04:06.165 [Thread-2496] INFO [com.liferay.training.deployment.failing-service_1.0.0 [966]] STARTED com.liferay.training.deployment.failing-service_1.0.0 [966]
2018-09-27 05:04:06.165 [Thread-2496] INFO [com.liferay.training.deployment.failing-service_1.0.0 [966]] Thread-2496|FailingService::start()
2018-09-27 05:04:06.177 [Thread-2496] INFO [com.liferay.training.deployment.failing-service_1.0.0 [966]] Thread-2496|FailingService::start() #####
2018-09-27 05:04:06.177 [Thread-2496] INFO [com.liferay.training.deployment.failing-service_1.0.0 [966]] Thread-2496|[FailingService::start()] Task completed successfully!
2018-09-27 05:04:06.177 [Thread-2496] INFO [com.liferay.training.deployment.failing-service_1.0.0 [966]] Thread-2496|[FailingService::start()]
2018-09-27 05:04:06.188 [Thread-2496] INFO [com.liferay.training.deployment.failing-service_1.0.0 [966]] Thread-2496|[BundleStartStopLogger::start()]
2018-09-27 05:04:06.188 [Thread-2496] INFO [com.liferay.training.deployment.failing-service_1.0.0 [966]] Thread-2496|[BundleStartStopLogger::start()] STARTED com.liferay.training.deployment.failing-service_1.0.0 [966]
```


Chapter 7: Liferay Platform Architecture Overview

Chapter Objectives

- Introduce core technologies and standards

Core Technologies and Standards

The Liferay platform core builds on the following technologies:

- **Java EE:** the runtime and programming environment
- **OSGi:** modular application runtime and development framework
- **Spring:** for transactions and dependency injection in the core
- **Hibernate:** for database access (along with direct JDBC access for optimized queries)
- **Ehcache:** caching
- **Elasticsearch** indexing and searching

Liferay is compliant with many industry-proven standards. For example, the following standards, are supported:

- **Portlets 1.0 (JSR-168)** and **Portlets 2.0 (JSR-286)** Liferay Portal can run any portlets that follow these two portlet specifications.
- **JSF** (JSR-127, JSR-314, JSR-344) The Java standard for building component-based web applications. Liferay is an active contributor to the standard and lead of the JSF-Portlet Bridge specification.
- **EcmaScript 2015 (ES6)** Liferay's tooling supports the 6th edition of ECMAScript specification and provides the ability to use it in all modern browsers with the integration of the Babel JS transpiler.
- **JAX-WS and JAX-RS** Java API for XML Web Services (JAX-WS) and for building RESTful services (JAX-RS) are incorporated as the preferred tooling to create web services.

Liferay supports the OSGi family of standards through its own implementations and also integrates implementations of the Apache Felix and Eclipse Equinox projects. Here are some of the most relevant supported standards:

- **OSGi runtime:** Allowing any OSGi module to run in Liferay
- **Declarative Services:** Supports a dynamic component model for Liferay development
- **Configuration Admin:** Lets you create configurable applications that can be reconfigured on the fly. Liferay provides an auto-generated UI to change the configuration of any component that leverages this standard.



Liferay Platform Architecture

The platform architecture can be presented in different ways. Below are two examples. The first one focuses on the technologies and the second one on the logical architecture.

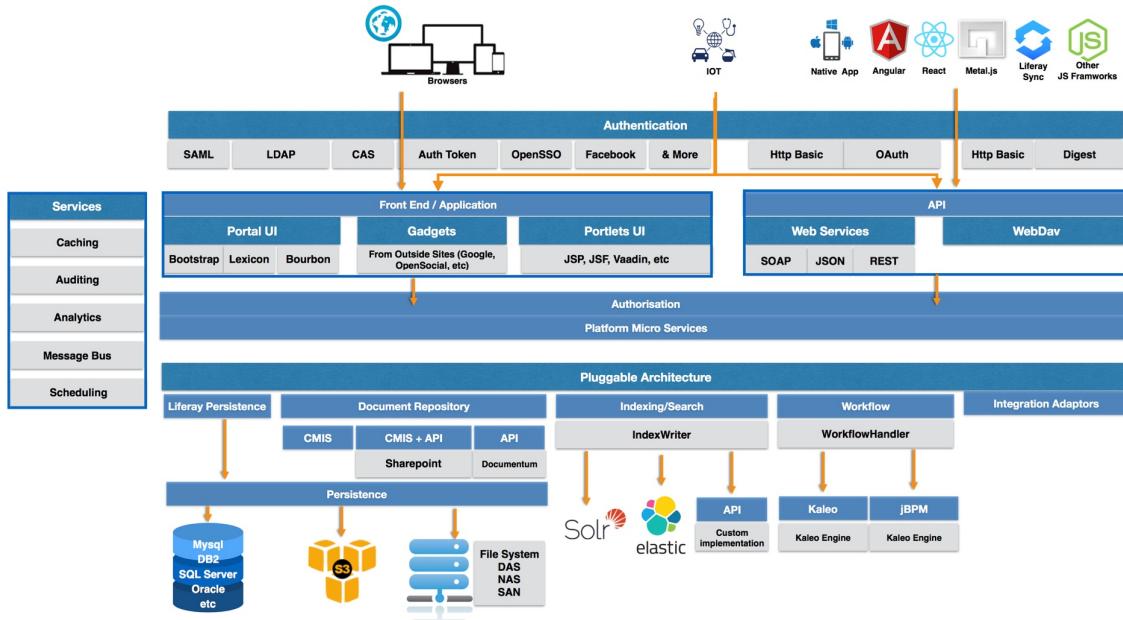


Figure: Liferay architecture, technology view.

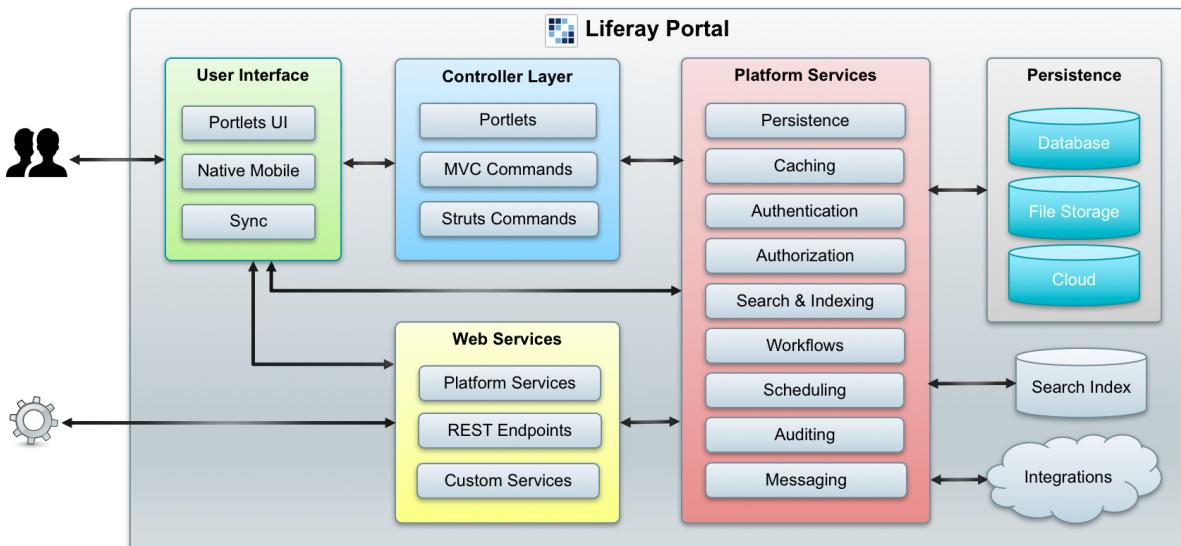
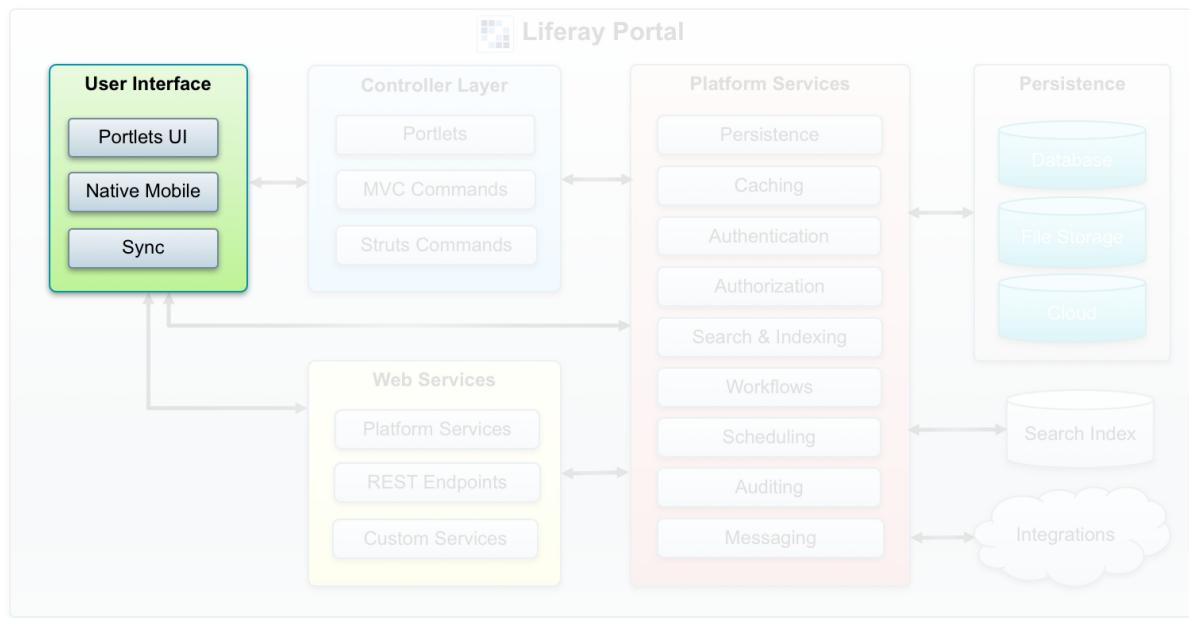


Figure: Liferay architecture, logical view.

Chapter 8: Customize the User Interface



Chapter Objectives

- Understand Liferay User Interface Architecture Basics
- Learn How to Approach Portal User Interface Customization

Liferay User Interface Technologies Overview

Liferay User Interface Types

Because of the comprehensive web services APIs, the Liferay platform can be accessed from different kind of user interfaces. In addition to the portal web user interface, Liferay provides desktop and mobile Sync clients for accessing the document repository and provides tools like Liferay Screens and Mobile SDK for creating native mobile applications. In the following sections, we will discuss the basic concepts and means of customizing the portal web user interface.

Web User Interface Technologies Overview

How is the portal web interface implemented? The web user interface is built mainly on JSP technology. Styling and user interface responsiveness is accomplished with **Liferay Lexicon** and **Clay** frameworks. Three main JavaScript frameworks are used as well: **Alloy UI**, **jQuery** and the emerging **Metal.js**. Additionally, the JSP files use Liferay custom tag libraries.

Liferay Lexicon

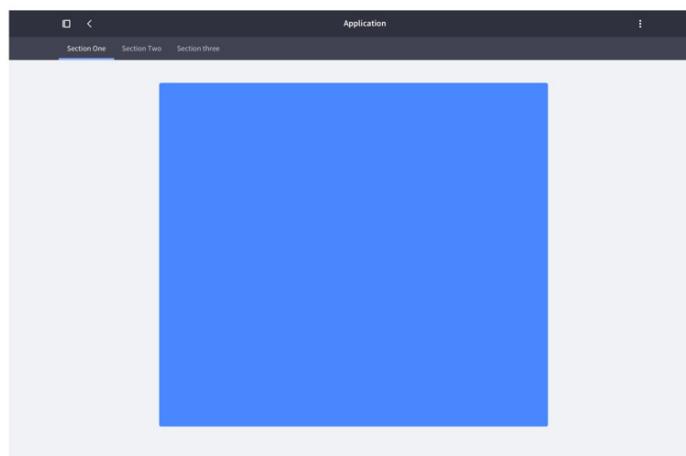
Liferay Lexicon is an abstract user interface **design language**. It doesn't dictate the implementation but provides principles, patterns, and tools to produce a common design framework and a consistent style and user experience.

Below is an example of Lexicon's definition of form:

Form Box

Description

The Form Box is an specific layout used to contain form structures. This layout is composed of a box that occupies 8 columns of the grid with 2 offset columns on both sides.



Liferay Clay

Liferay Clay is an **implementation** of the Lexicon Experience Design Language. It is an extension of the Twitter Bootstrap framework and is built with HTML, CSS, and JavaScript. The dependency of Twitter Bootstrap means that jQuery is always available in Liferay by default.

Below is an example of a dropdown menu created with Clay:

HTML

```
<div aria-labelledby="theDropdownToggleId" class="dropdown-menu">
  <ul class="list-unstyled">
    <li><a class="active dropdown-item" href="#1">Selected Option</a></li>
    <li><a class="dropdown-item" href="#3">Normal Option</a></li>
    <li><a class="disabled dropdown-item" href="#4">Disabled Option</a></li>
  </ul>
</div>
```



Selected Option

Normal Option

Disabled Option

Alloy UI

AlloyUI is an extensive UI framework incorporating HTML, CSS, and JavaScript and built on Yahoo YUI3. It has over 350 YUI and 150 modules. As Yahoo decided to stop maintaining the YUI, AlloyUI was already deprecated in Liferay DXP. Although legacy, it is still commonly used in the portal native user interface and will be supported throughout the whole Liferay 7.x product lifecycle.

Below is an example of dropdown menu created with AlloyUI:

```
<ul>
  <li id="myDropdown">
    <a id="myTrigger" href="#">Dropdown</a>
  </li>
</ul>
```

Javascript

```
YUI().use(
  'aui-dropdown',
  function(Y) {
    new Y.dropdown(
      {
        boundingBox: '#myDropdown',
        trigger: '#myTrigger'
      }
    ).render();
  }
);
```



Dropdown ▾

Action

Metal.js

The development of Liferay-developed lightweight JavaScript framework Metal.js started after Yahoo YUI deprecation in 2014. Metal.js integrates with Google Closures (SOY) and Facebook (JSX) templating languages, supporting ECMAScript 2015 and 6 (ES 2015, ES6). Metal.js also has an isomorphic, Server-side rendering (SSR) and AMD loader support.

Below is an example of a Metal.js component with a SOY template. When compiled and transpiled, they are merged into a single JavaScript file:

SOY template

```
{namespace View}

/**
 * Prints the portlet main view.
 */
{template .render}
<div id="${$id}">
  {call Header.render data="all"}{/call}

  <p>{msg desc=""}here-is-a-message{/msg}</p>

  {call Footer.render data="all"}{/call}
</div>
{/template}
```



Metal.js component registering the SOY template

```
import Component from 'metal-component/src/Component';
import Footer from './Footer.es';
import Header from './Header.es';
import Soy from 'metal-soy/src/Soy';
import templates from './View.soy';

/**
 * View Component
 */
class View extends Component {}

// Register component
Soy.register(View, templates);

export default View;
```

Tag Libraries

In addition to JavaScript and CSS libraries, Liferay offers a set of fully integrated taglibs for use in JSP files. In addition to reducing boilerplate code, taglibs provide other advantages, too, like a consistent and responsive user interface. The documents for most of the Liferay taglibs can be accessed on <https://docs.liferay.com/ce/portal/7.1-latest/taglibs/util-taglib/>.

| Tag Libraries | |
|----------------------------------|--|
| JSTL core | JSTL 1.2 core library |
| aui | Provides the AUI component tags, prefixed with aui:. |
| liferay-portlet | <i>No Description</i> |
| portlet | <i>No Description</i> |
| liferay-security | <i>No Description</i> |
| liferay-theme | <i>No Description</i> |
| liferay-ui | Provides the Liferay UI component tags, prefixed with liferay-ui:. |
| liferay-util | <i>No Description</i> |

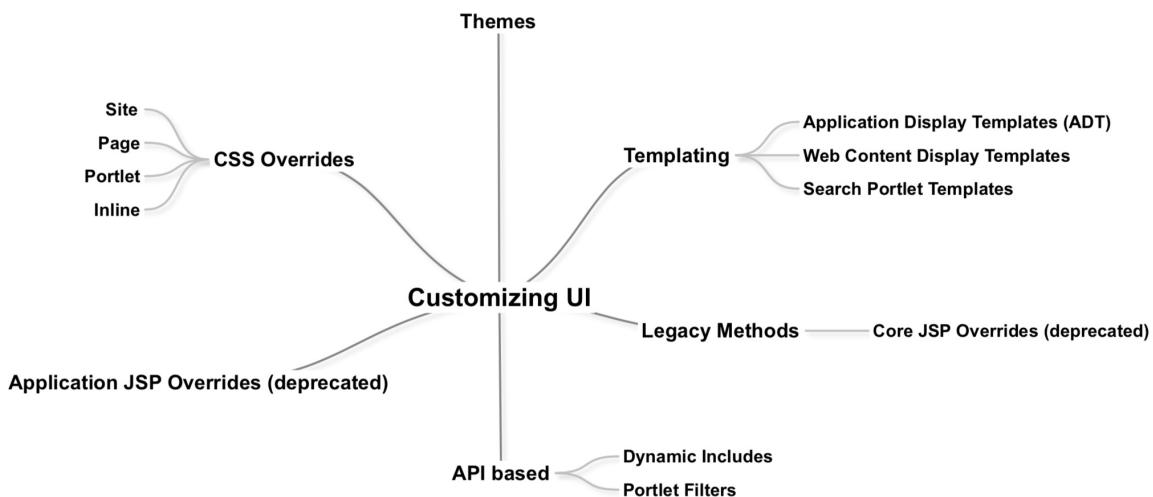
Summary

The Liferay portal web user interface still relies on Alloy UI, but Metal.js is increasingly being introduced in the new native applications.

As a development platform, Liferay is framework-agnostic. Developers can use any preferred JavaScript libraries.

User Interface Customization Overview

The following mindmap summarizes the typical areas of user interface customization in Liferay:



Links and Resources

- **Alloy UI Project Website**
<https://alloyui.com>
- **Lexicon Project Website**
<https://lexicondesign.io>

- **Clay Project Website**
<https://clayui.com>
- **Javascript in Liferay**
https://dev.liferay.com/develop/tutorials/-/knowledge_base/7-0/javascript-in-liferay
- **Metal.js Project Website**
<https://metaljs.com>
- **Metal.js on Liferay Developer Network**
https://dev.liferay.com/develop/tutorials/-/knowledge_base/7-0/metal-js

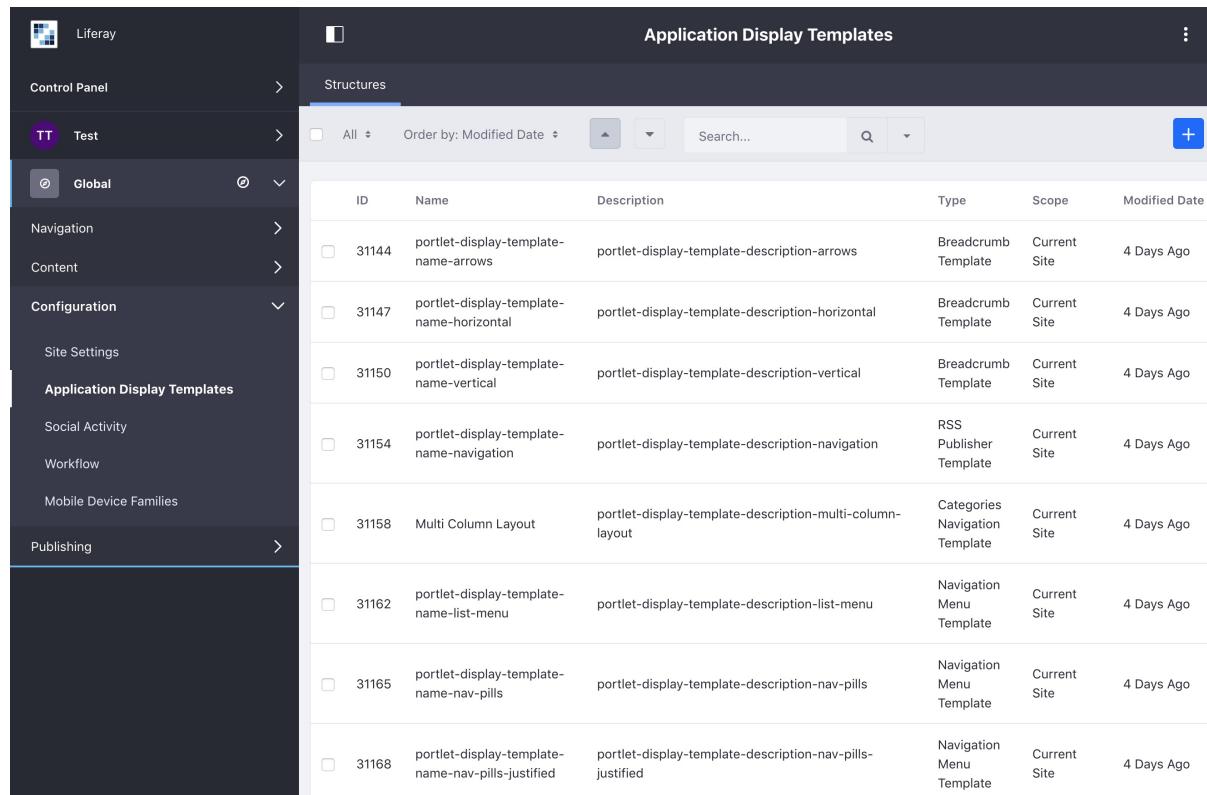
Change the Default Application UI with Application Display Templates

Liferay Application Display Templates (ADT) is a templating framework that allows you to customize and override portal application user interfaces. Native Portlets Supporting the ADT are:

- Asset Publisher
- Asset Categories Navigation
- Asset Tags Navigation
- Blogs
- Media Gallery
- RSS
- Breadcrumb
- Language
- Navigation Menu
- SiteMap
- Wiki

Application Display Templates should be implemented using the FreeMarker template language and are managed in the Control Panel in the Site Panel -> Build -> Application Display Templates. Templates can exist in the portal global or site scope.

Templates List in Global Scope



The screenshot shows the Liferay Control Panel with the 'Global' scope selected. The left sidebar includes links for Control Panel, Test, Global, Navigation, Content, Configuration, Site Settings, Application Display Templates (which is currently selected), Social Activity, Workflow, Mobile Device Families, and Publishing. The main content area is titled 'Application Display Templates' and shows a table of templates. The table has columns for ID, Name, Description, Type, Scope, and Modified Date. The data is as follows:

| ID | Name | Description | Type | Scope | Modified Date |
|-------|---|--|--------------------------------|--------------|---------------|
| 31144 | portlet-display-template-name-arrows | portlet-display-template-description-arrows | Breadcrumb Template | Current Site | 4 Days Ago |
| 31147 | portlet-display-template-name-horizontal | portlet-display-template-description-horizontal | Breadcrumb Template | Current Site | 4 Days Ago |
| 31150 | portlet-display-template-name-vertical | portlet-display-template-description-vertical | Breadcrumb Template | Current Site | 4 Days Ago |
| 31154 | portlet-display-template-name-navigation | portlet-display-template-description-navigation | RSS Publisher Template | Current Site | 4 Days Ago |
| 31158 | Multi Column Layout | portlet-display-template-description-multi-column-layout | Categories Navigation Template | Current Site | 4 Days Ago |
| 31162 | portlet-display-template-name-list-menu | portlet-display-template-description-list-menu | Navigation Menu Template | Current Site | 4 Days Ago |
| 31165 | portlet-display-template-name-nav-pills | portlet-display-template-description-nav-pills | Navigation Menu Template | Current Site | 4 Days Ago |
| 31168 | portlet-display-template-name-nav-pills-justified | portlet-display-template-description-nav-pills-justified | Navigation Menu Template | Current Site | 4 Days Ago |

Template Edit View

The screenshot shows the 'portlet-display-template-name-icon-menu' configuration page. The 'Name' field is set to 'portlet-display-template-name-icon-menu'. The 'Script' tab is selected, displaying the following JSP code:

```

1 <#if entries?has_content>
2   <div class="truncate-text"> @liferay.ui["icon-menu"] direction="left-side" icon=normalizedDefaultLanguageId
3   markupView="lexicon" showWhenSingleIcon=true triggerLabel=normalizedDefaultLanguageId triggerType="button"
4   <#list entries as entry> <if entry.isSelected() && entry.isDisabled()> #assign normalizedDefaultLanguageId =
5   stringUtil.toLowerCase(stringUtil.replaceEntry.getLanguageId(), "-", "-") /> @liferay.ui["icon"] icon=normalize
6   dDefaultLanguageId iconCssClass="inline-item inline-item-before" markupView="lexicon" message=normalizedDefaultLan
7   guageId url=entry.getURL() /> </if> </#list> </@> </div> </#if>

```

The sidebar on the left includes sections for Details, Script, Fields (Languages), General Variables (Current URL, Locale, Portlet Preferences, Template ID, Theme Display), and Util (HTTP Request, Render Request, Render Response). A 'Script File' input field shows 'Choose file No file chosen'. At the bottom are 'Save', 'Save and Continue', and 'Cancel' buttons.

Template Selection in the Portlet

The screenshot shows the 'Language Selector - Configuration' dialog. The 'Setup' tab is selected. The 'Display Template' dropdown is set to 'portlet-display-template-name-icon-menu'. A link to 'Manage Display Templates for Liferay' is shown. The 'Display Current Locale' switch is set to 'YES'. The 'Languages' section is partially visible. A 'Save' button is located at the bottom-left of the dialog.

Below are some example cases of how you can use the Application Display Template:

- Create a custom list layout for the Asset Publisher.
- Create a custom item view layout for Asset Publisher.
- Create a custom layout for navigation.
- Add audience targeting support for navigation layout.

Developing ADTs

A variety of Liferay utilities and services are available for Application Display Templates. It should, however, be noticed that access to some variables is restricted by default. The access is controlled in Control Panel -> System Settings -> FreeMarker Engine. Especially to consume Liferay service API via ADTs, users must be given access to the **serviceLocator** utility.

Templates also have some common variable available, for example:

- currentUrl
- locale
- themeDisplay
- portletPreferences
- templatedId
- renderRequest

Custom objects can be made available using Template Handlers.

Adding ADT support for Custom Applications

Application Display Template support can also be implemented in custom applications. Below is an overview of the steps required. For more information, please see the tutorial in Liferay Developer Network at
https://dev.liferay.com/develop/tutorials/-/knowledge_base/7-0/implementing-application-display-templates:

1. Create a service component extending BasePortletDisplayTemplateHandler
2. Define the template permissions (default.xml)
3. Create the configuration ui using <liferay-ui:ddm-template-selector>
4. Render using <liferay-ddm:template-renderer>

Links and Resources

- **Implementing Application Display Template Support for Custom Applications**

https://dev.liferay.com/develop/tutorials/-/knowledge_base/7-0/implementing-application-display-templates

Exercises

Create a Media Gallery Application Display Template

Introduction

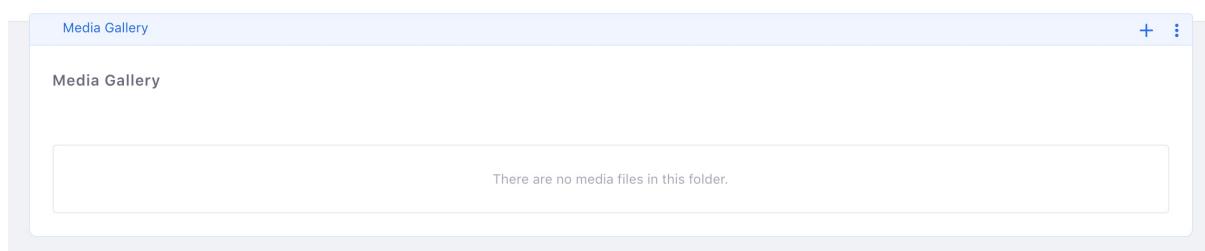
In this exercise, we will be adding an Application Display Template to the Media Gallery to display a custom carousel. Carousels are a common website design pattern that allow you to display more than one piece of content in the same place. Liferay includes a built-in carousel Application Display Template, but the one we will be building can be used, for example, to display rotating hero images on top of web pages.

Overview

- 1 Add the Media Gallery
- 2 Upload Images
- 3 Create a custom Application Display Template
- 4 Verify your custom carousel

Add the Media Gallery

1. Go to <http://localhost:8080> in your browser.
 - o Login to the platform if needed.
2. Click the *Add* button in the top right corner.
3. Expand the *Widgets* section.
4. Type `Media Gallery` in the search bar.
5. Click *Add* next to the `Media Gallery`.



Upload Images

1. Click *Add* in the Media Gallery.
2. Choose *Basic Document*.
3. Click *Choose File/Browse* under the *File* form field.
4. Go to `training-workspace/solutions/solutions-chapter-08/solution-08-application-display-template`.
5. Click to choose `yellowstone-1.jpg`.
6. Click *Open*.
7. Click *Publish*.
8. Repeat for `yellowstone-2.jpg`, `yellowstone-3.jpg`, `yellowstone-4.jpg`.

Media Gallery



Create a Custom Application Display Template

1. **Click** the Options icon above the *Media Gallery*.
2. **Choose** Configuration.
3. **Click** the Setup tab.
4. **Click** Manage Templates under the *Display Template* section.
5. **Click** the Add button.
6. **Type** My Custom Carousel as the Name.
7. **Click** Script File: Choose File
8. **Select** the `custom-carousel-adt.ftl` file located in the `training-workspace/solutions/solutions-chapter-08/solution-08-application-display-template` folder.
9. **Click** Save and Continue and review the uploaded freemarker template.
10. **Close** the pop-up.
11. **Choose** My Custom Carousel as the *Display Template* from the drop-down menu.

Note: you may have to refresh the page to see the new option in the drop-down menu.

12. **Click** the Save button.
13. **Close** the pop-up.

Application Display Templates

Script

```
1 <style>
2 .slides {
3   margin: auto;
4   width: 100%;
5   height: auto;
6   border-radius: 5%;
7 }
8
9 h1 {
10   position: absolute;
11   color: white;
12   top: 100px;
13   width: 100%;
14   text-align: center;
15   font-size: 50px;
16 }
17 </style>
18
19 <if entries?has_content>
20   <div id="<@portlet.namespace />carousel">
21     <#assign imageMimeTypes = propsUtil.getArray("dl.file.entry.preview
22       .image.mime.types") />
23     <#list entries as entry>
24       <if imageMimeTypes?seq_contains(entry.getMimeType())>
```

Search

Fields

Documents *

General Variables

- Current URL
- Locale
- Portlet Preferences
- Template ID
- Theme Display

Util

- HTTP Request
- Render Request
- Render Response

Document Util

- DL Util

Script File No file chosen

Save Save and Continue Cancel

Verify and Test

1. **Close** the Configuration for the *Media Gallery*, and you should see the images that you uploaded in your custom carousel Application Display Template.

Media Gallery



Customize the Application JSPs

Not all the platform native applications support Application Display Templates. Also, Application Display Template support only reaches view templates and not, for example, edit templates.

In customization cases where there is no ADT support available or the support is not enough, you need to customize the JSP. There are a few ways to do this, varying from API-based content injection to overriding the JSP files completely:

- Dynamic includes
- Custom JSP bag
- Fragment modules
- Portlet filters

Dynamic Includes

Dynamic includes allow you to inject content in the JSP files at the places where there is a <liferay-util:dynamic-include> tag. Dynamic includes are supported in content editors, Login portlets, and Asset Publisher portlets.

Below is an example of using dynamic includes:

Blogs Portlet view_entry.jsp

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

<liferay-util:dynamic-include key="com.liferay.blogs.web#/blogs/view_entry.jsp#pre" />

<%
    String redirect = ParamUtil.getString(request, "redirect");
    ...
%>
<liferay-util:dynamic-include key="com.liferay.blogs.web#/blogs/view_entry.jsp#post" />
```

Dynamic Include Component

```
@Component(
    immediate = true,
    service = DynamicInclude.class
)
public class BlogsDynamicInclude implements DynamicInclude {

    @Override
    public void include(HttpServletRequest request, HttpServletResponse response, String key) throws IOException {
        PrintWriter printWriter = response.getWriter();
        printWriter.println("<h2>Added by Blogs Dynamic Include!</h2><br />");
    }

    @Override
    public void register(DynamicIncludeRegistry dynamicIncludeRegistry) {
        dynamicIncludeRegistry.register(
            "com.liferay.blogs.web#/blogs/view_entry.jsp#pre");
    }
}
```

The resulting Blogs portlet entry view

Blogs

Added by Blogs Dynamic Include!

Blogs Test

 Test Test
0 Seconds Ago

See the Dynamic Include on the top of the page.

Custom JSP Bag

Custom JSP bags is a JSP customization method for legacy portal core JSP files. This method is deprecated as of Liferay DXP. You can read more information about it at https://dev.liferay.com/en/develop/tutorials-/knowledge_base/7-1/jsp-overrides-using-custom-jsp-bag.

Fragment Modules

Fragment modules allow you to override entire JSPs. This is a fast and effective method, but it also requires utmost attention on patches and upgrades because there might be breaking changes in the original JSP. That's why this method should be used only very carefully and only in situations where the other strategies can't be applied.

The process of creating a fragment module, which is equivalent to an OSGi Fragment Bundle, is simple:

With the Fragment Module Wizard, you:

- Create a Liferay fragment module for the desired module JSP
- Override the JSP

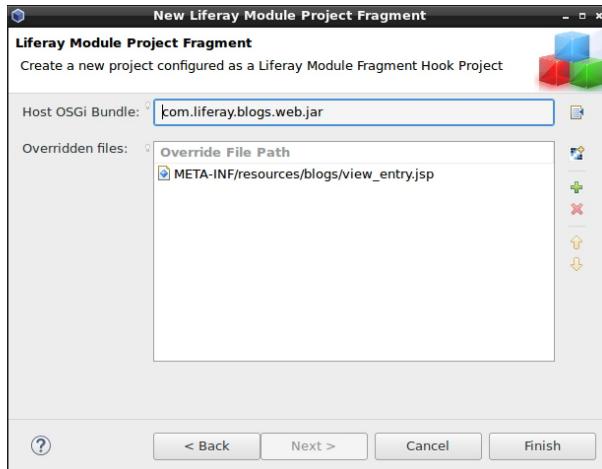
Without the Fragment Module Without the Wizard, you:

- Create a Liferay module
- Declare the dependencies in build gradle and host bundle in bnd.bnd
- Create and override the desired JSP in the same path as the original

Example: Overriding Blogs Portlet view_entry.jsp

Step 1 - Create a Fragment Module

Select the host bundle and file to override:



Step 2 - Override the JSP

Blogs Portlet view_entry.jsp

```

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

<h2>This is a JSP overridden by a Fragment Module</h2>

<liferay-util:dynamic-include key="com.liferay.blogs.web#/blogs/view_entry.jsp#pre" />

```

Step 3 - Deploy and Test



The Generated bnd.bnd:

```

Bundle-Name: JSP Fragment Module Example
Bundle-SymbolicName: com.liferay.training.jsp.fragment.module
Bundle-Version: 1.0.0
Fragment-Host: com.liferay.blogs.web;bundle-version="3.0.0"

```

Exercise

Please see the *Customize Announcements Portlet Using a JSP Fragment Module* in the *Exercises* section.

Portlet Filter

Portlet filter is an API-based and Liferay recommended method of overriding the application JSPs. It operates directly on the request and response content, giving access to all the content sent back to the client.

Below is an example of portlet render filter:

```
@Component(
    immediate = true,
    property = {
        "javax.portlet.name=" + PortletKeys.BLOGS
    },
    service = PortletFilter.class
)
public class BlogsRenderFilter implements RenderFilter {

    @Override
    public void doFilter(
        RenderRequest request, RenderResponse response, FilterChain chain)
        throws IOException, PortletException {

        // Stream manipulation code here.
    }
}
```

Exercises

Customize the Announcements Portlet JSP Using a Fragment Module

Introduction

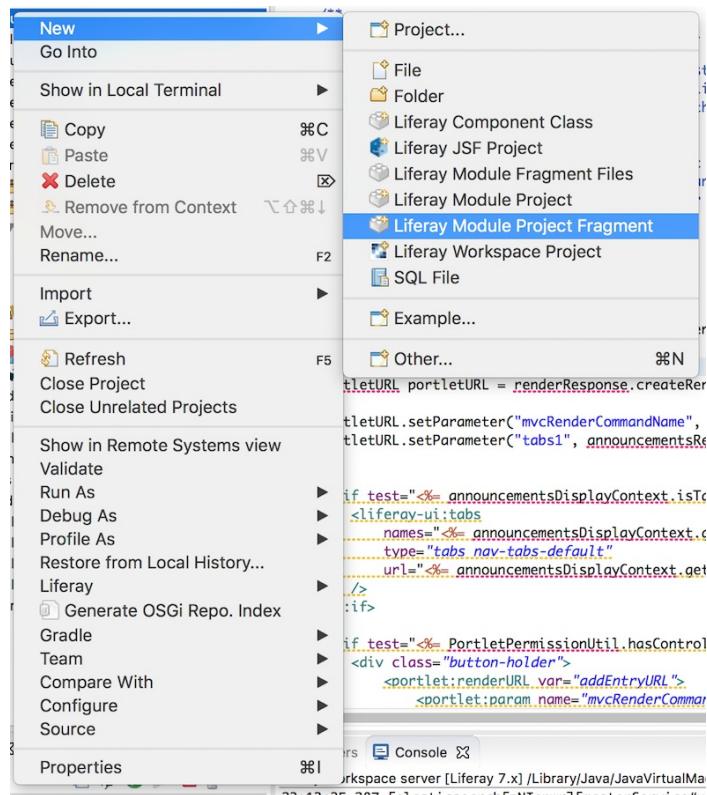
In this exercise, we will be customizing a jsp file in the *Announcements* Portlet with a Fragment Module.

Overview

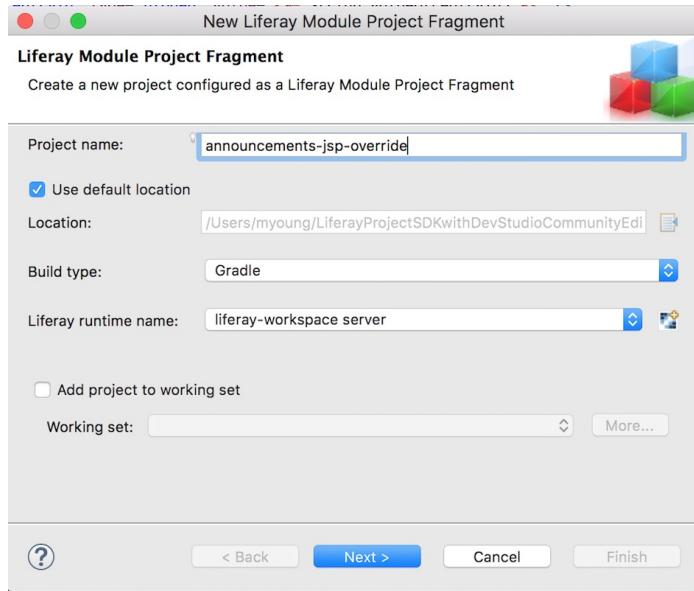
- ① Create the Fragment Module
- ② Choose the Module and JSP to Override
- ③ Implement the JSP
- ④ Deploy your Fragment Module
- ⑤ Verify the Results

Create the Fragment Module

1. [Go to training-workspace](#).
2. [Right-click](#) the workspace to open the context menu.
3. [Choose New → Liferay Module Project Fragment](#).

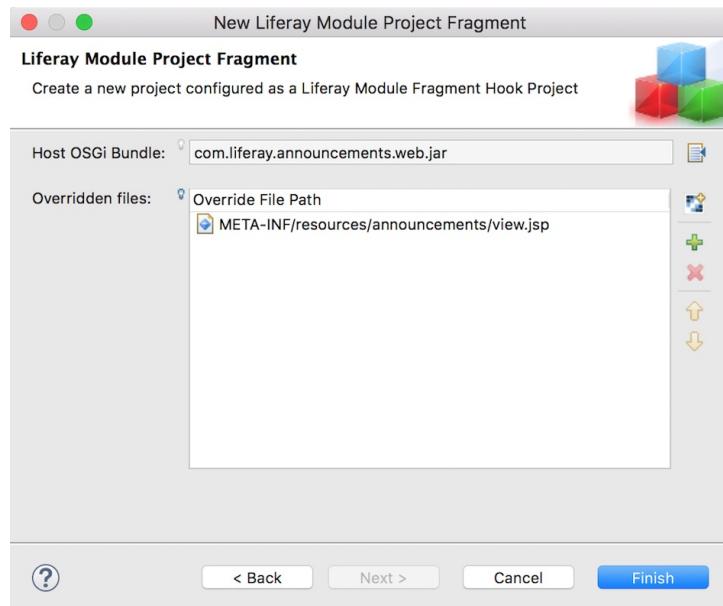


4. Type `annoucements-jsp-override` in the *Project Name* field.
5. Click *Next*.



Choose the Module and JSP to Override

1. Click on the *Browse* button.
2. Type `com.liferay.announcements.web` in the *Select Host OSGi Bundle* field.
3. Choose the `com.liferay.announcements.web-[version].jar`.
4. Click *Ok*.
5. Click the *Add Files from OSGi bundle* icon (Liferay logo).
6. Choose `META-INF/resources/announcements/view.jsp`.
7. Click *Finish*.



Implement the JSP

1. Find the `annoucements-jsp-override` project.

2. **Expand** the `src/main/resources/META-INF/resources/announcements` folder.
3. **Double-click** on `view.jsp`.
4. **Open** the `view.jsp` file found in your `training-workspace/solutions/solutions-chapter-08/solution-08-announcements-jsp-override/src/main/resources/META-INF/resources/announcements` folder.
5. **Copy** the contents in `view.jsp`.
6. **Replace** the contents in the `view.jsp` file in the `announcements-jsp-override` project with the contents of the `view.jsp` in the exercises folder.

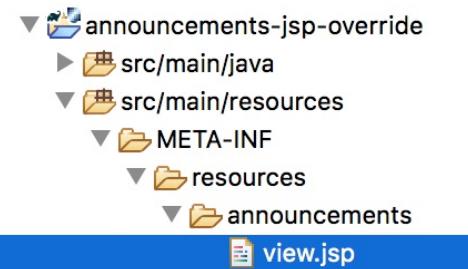
```
<%@ include file="/announcements/init.jsp" %>

<h2>This is the announcements view.jsp overridden by a Fragment Module</h2>

<%-- The following allows you to include the original JSP that was overridden --%>

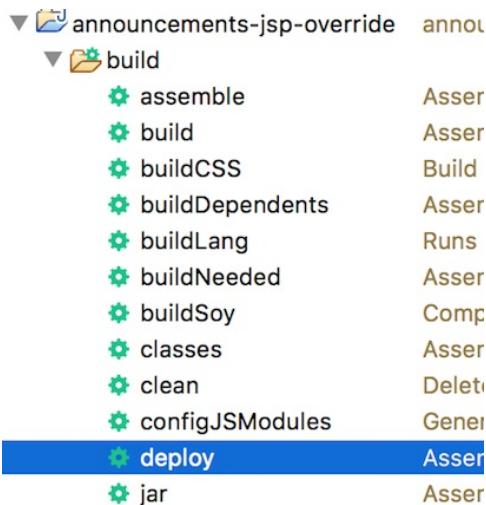
<liferay-util:include
    page="/announcements/view.original.jsp"
    servletContext="<%= application %>"
/>
```

7. **Save** the file.



Deploy Your Fragment Module

1. **Go to** the *Gradle Tasks* view in *Developer Studio*.
2. **Double-click** the `deploy` task from `modules/announcements-jsp-override/build`.
 - o This will deploy `announcements-jsp-override` to the Liferay Server.

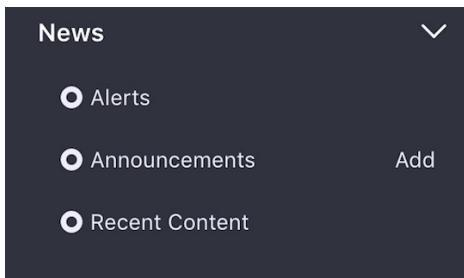


Deploy and Test

1. **Open** your browser to `http://localhost:8080` and log in.

2. **Click** Add in the top right corner.

3. **Go to** `Widgets → News`.



4. **Click** Add next to the *Announcements* widget.

A screenshot of a portlet titled 'ANNOUNCEMENTS'. The content area displays the text: 'This is the announcements view.jsp overridden by a Fragment Module'. Below this, there are two tabs: 'Unread' (underlined) and 'Read'.

Exercises

Customize Blogs Portlet Entry View Using a Portlet Filter

Introduction

In this exercise, we will be creating a RenderFilter to customize the JSP output from the Blogs Portlet. In order to do this, we will need to create an OSGi bundle:

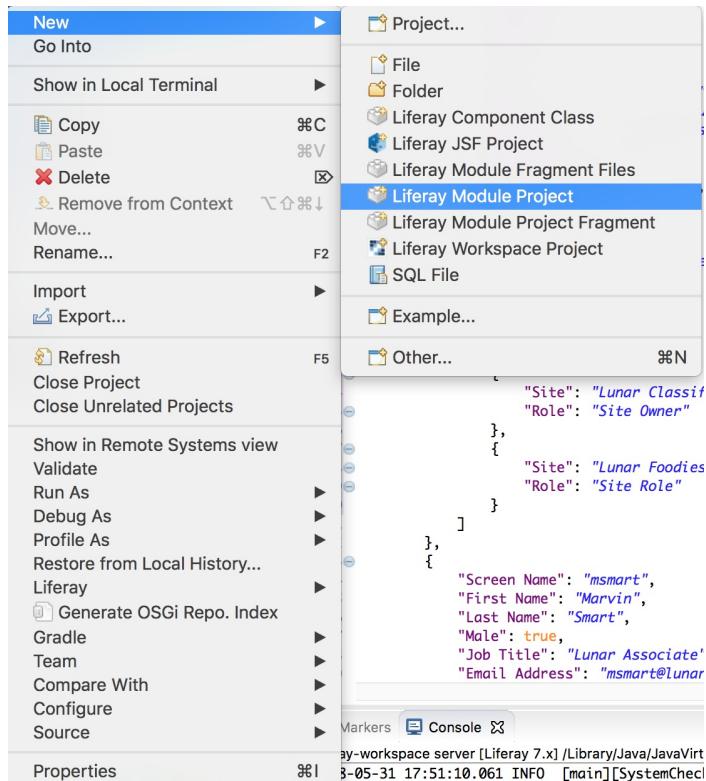
- Capture the output from the Blogs Portlet using a portlet filter
- Search and augment the content
- Push the new content to the Blogs Portlet

Overview

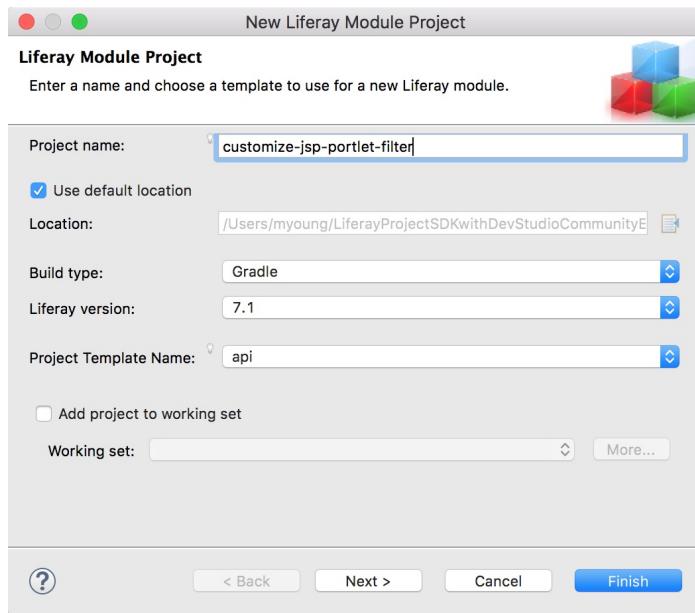
- ① Create a Liferay Module project using the API template
- ② Resolve dependencies
- ③ Create a PortletFilter component implementing the RenderFilter interface
- ④ Override the doFilter() method
- ⑤ Create a StringResponseWrapper nested class to capture the Blogs output
- ⑥ Deploy and test

Create a Liferay Module Project Using the API Template

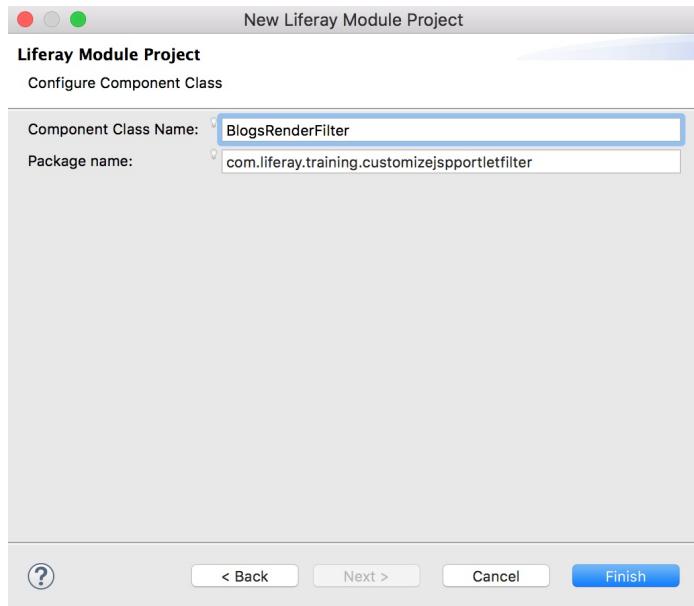
1. [Go to the *training-workspace*.](#)
2. [Right-click](#) on the workspace to open a context menu.
3. [Choose](#) *New → Liferay Module Project*.



4. Type `customize-jsp-portlet-filter` in the *Project Name* field.
5. Choose `api` in the *Project Template Name* field.
6. Click **Next**.



7. Type `BlogsRenderFilter` in the *Component Class Name* field.
8. Type `com.liferay.training.customizejspportletfilter` in the *Package name* field.
9. Click **Finish**.



10. **Expand** the `customize-jsp-portlet-filter` project.
11. **Right-click** on the `com.liferay.training.customizejspportletfilter.api` package in the `src/main/java` folder.
12. **Choose Refactor → Rename**.
13. **Edit** the name of the package to remove `.api`.
 - o It should now be `com.liferay.training.customizejspportletfilter`.
14. **Click** Ok (You may safely ignore the warning → *Continue*).

Resolve Dependencies

Next, we will have to adjust the `bnd.bnd` and `build.gradle` file generated by Liferay Studio. Since we used the `api` template, `build.gradle` will only contain the bare minimum of dependencies to set up an OSGi project. Because we plan to use the `PortletFilter` interface, we have to depend on the `portlet-api` and the `javax.servlet-api` packages. Additionally we need utility classes from the `com.liferay.portal.kernel` package.

1. **Open** the `build.gradle` file and add the required dependencies. Afterwards your `build.gradle` should look as follows:

```
dependencies {
    compileOnly group: "com.liferay.portal", name: "com.liferay.portal.kernel", version: "3.0.0"
    compileOnly group: "javax.portlet", name: "portlet-api", version: "3.0.0"
    compileOnly group: "javax.servlet", name: "javax.servlet-api", version: "3.0.1"
    compileOnly group: "org.osgi", name: "org.osgi.core", version: "6.0.0"
    compileOnly group: "org.osgi", name: "org.osgi.service.component.annotations", version: "1.3.0"
}
```

2. **Save** the file.
3. **Open** the `bnd.bnd` file and replace it with the following.
 - o This step is not necessary, but it's helpful to use a more human-readable `Bundle-Name` in case you need to troubleshoot the bundle in the Gogo Shell, for example.

```
Bundle-Name: Customize JSP Portlet Filter
Bundle-SymbolicName: com.liferay.training.customizejspportletfilter
Bundle-Version: 1.0.0
```

4. **Save** the file.

Create a PortletFilter Component Implementing the RenderFilter Interface

1. Open `BlogsRenderFilter.java`.
2. Implement the `RenderFilter` interface (Hint: type `public class BlogsRenderFilter implements RenderFilter` and use the option `Add unimplemented methods` to fix the indicated error. Afterwards your class should look as follows:

```
package com.liferay.training.customizejspportletfilter;

import java.io.IOException;

import javax.portlet.PortletException;
import javax.portlet.RenderRequest;
import javax.portlet.RenderResponse;
import javax.portlet.filter.FilterChain;
import javax.portlet.filter.FilterConfig;
import javax.portlet.filter.RenderFilter;
```



```
public class BlogsRenderFilter implements RenderFilter {

    @Override
    public void init(FilterConfig filterConfig) throws PortletException {
        // TODO Auto-generated method stub
    }

    @Override
    public void destroy() {
        // TODO Auto-generated method stub
    }

    @Override
    public void doFilter(RenderRequest request, RenderResponse response, FilterChain chain)
        throws IOException, PortletException {
        // TODO Auto-generated method stub
    }
}
```

1. Add following component annotation:

```
...
@Component(
    immediate = true,
    property = {
        "javax.portlet.name=" + PortletKeys.BLOGS
    },
    service = PortletFilter.class
)
public class BlogsRenderFilter implements RenderFilter {
...
```

2. Type `CTRL + Shift + o` to resolve dependencies (when prompted, choose `javax.portlet.filter.PortletFilter`).

Create a StringResponseWrapper Nested Class to Capture the Blogs Output

The `StringResponseWrapper` is responsible for capturing the output of the `Blogs` portlet and preventing it from being written to the actual output stream. This gives us the opportunity to modify the output before it actually gets written to the portlet response.

1. Implement the `StringResponseWrapper` using the following code in the `BlogsRenderFilter` class.

- o The `StringResponseWrapper` is implemented as a private inner class. You can place this class at the end of `BlogsRenderFilter` class:

```
...
private class StringResponseWrapper extends RenderResponseWrapper {

    public StringResponseWrapper(RenderResponse response) {
        super(response);

        _stringWriter = new StringWriter();
        _printWriter = new PrintWriter(_stringWriter);
    }

    public PrintWriter getWriter() throws IOException {
        return _printWriter;
    }

    public String toString() {
        return _stringWriter.toString();
    }

    private PrintWriter _printWriter;
    private StringWriter _stringWriter;
}
```

Override the `doFilter()` Method

In this step, we will be implementing the `doFilter()` method. The code in this section will capture the output of the portlet filter, do a search and replace on the output, and finally write the output to the portlet response. The `BlogsRenderFilter` references the `StringReponseWrapper` implemented in the previous step.

1. Implement the `doFilter()` method using the following code in the `BlogsRenderFilter` class (**Please note:** The string we use as the entry point for our extension (`interestingText`) has **two** blanks between `input` and `class`).

```
@Override
public void doFilter(
    RenderRequest request, RenderResponse response, FilterChain chain)
throws IOException, PortletException {

    RenderResponseWrapper renderResponseWrapper =
        new StringResponseWrapper(response);

    chain.doFilter(request, renderResponseWrapper);

    String text = renderResponseWrapper.toString();

    if (text != null) {
        String interestingText = "<input  class=\"field form-control\"";

        int index = text.lastIndexOf(interestingText);

        if (index >= 0) {
            String newText1 = text.substring(0, index);

            String newText2 =
                "\n<p>StringResponseWrapper captures the output of the Blogs Portlet before " +
```

```
        "it gets written to the output stream. This give us the chance to manipulate" +
        " the output before sending it down the filter chain.</p>\n";
    }

    String newText3 = text.substring(index);

    String newText = newText1 + newText2 + newText3;

    response.getWriter().write(newText);
}

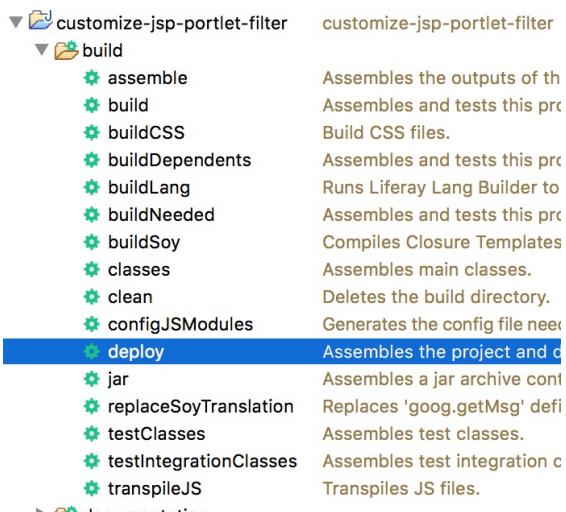
}
```

2. **Press** `CTRL+SHIFT+O` to resolve any missing imports.
 3. **Save** the file.

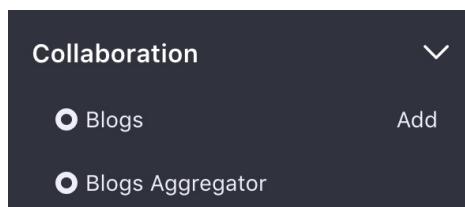
Deploy and Test

Let's deploy and test our customization.

1. **Go to** the *Gradle Tasks* view in *Liferay Developer Studio*.
 2. **Double-click** the `deploy` task from `customize-jsp-portlet-filter/build`.
 - o This will deploy *announcements-jsp-override* to the Liferay Server.



3. Open your browser to <http://localhost:8080> and log in.
 4. Click Add in the top right corner.
 5. Go to Widgets → Collaboration .



6. Click Add next to the *Blogs* widget.
 - You should see that the *Blogs* portlet content has been modified by the *BlogsRenderFilter*.

Make sure the `com.liferay.blogs.api` is active. Remember, we stopped this module in an earlier exercise.

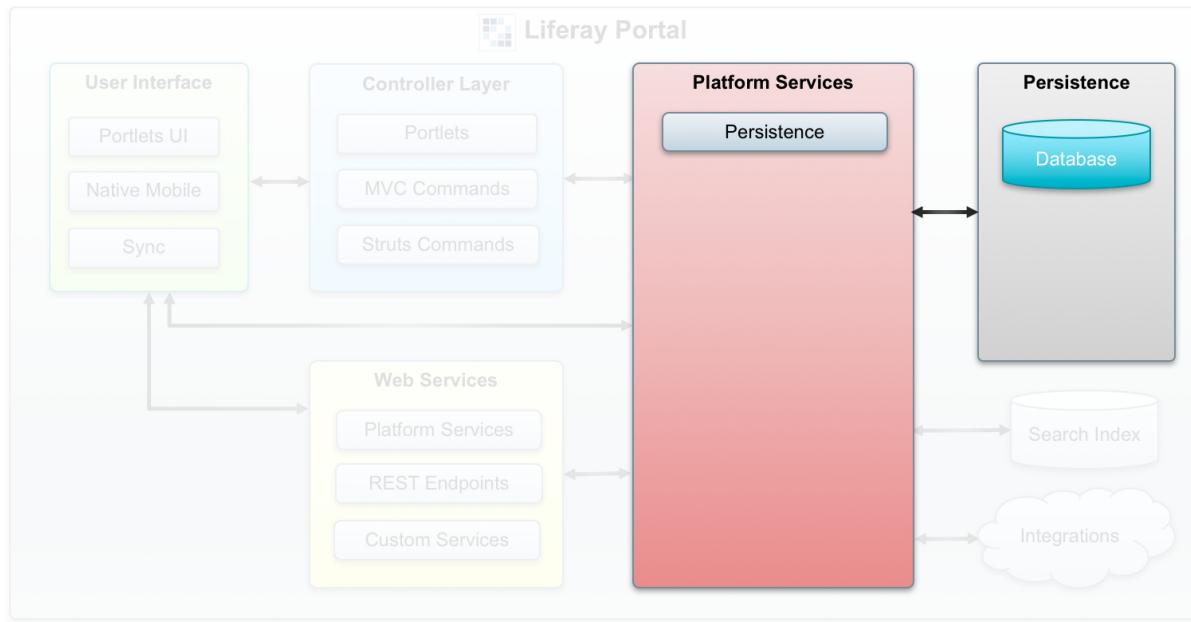
The screenshot shows a Liferay portlet interface. At the top left is the title "BLOGS". To the right are three buttons: "RSS" with a feed icon, "Subscribe" in a light blue box, and a blue "New Entry" button. Below these buttons is a text area containing the following content:

StringResponseWrapper captures the output of the Blogs Portlet before it gets written to the output stream. This give us the chance to manipulate the output before sending it down the filter chain.

Takeaways

Portlet filters can be used to do many different types of pre or post processing for portlets. In this case, you've seen that you can use a RenderFilter to modify the content of a portlet without having to change the code directly. Along with using Module Fragments, this gives you another option to customize the view layer of existing Liferay applications.

Chapter 9: Extend Liferay's Schema



Chapter Objectives

- Learn How to Extend Liferay Data Models and Add Custom Attributes to Entities
- Understand the Expando API Virtual Datamodel

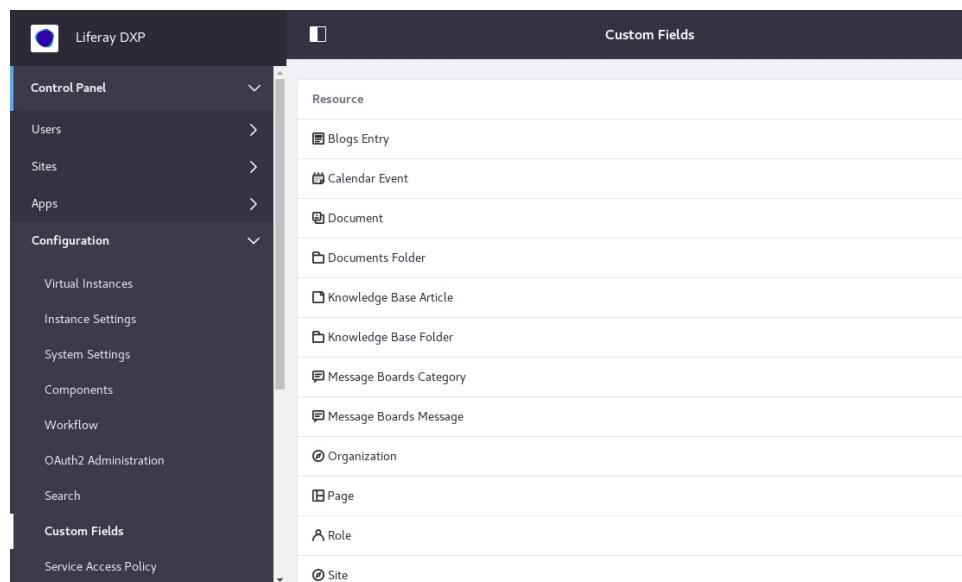
Introducing Custom Fields

As a developer or system administrator, you may sometimes want to add a persistable and manageable attribute to an existing Liferay entity. Such use cases could be, for example:

- Adding an identification number for a user entity
- Adding an author field to the page (layout) entity
- Adding a custom LDAP mapping attribute to the usergroup
- Adding a maximum number of members for a role (in case of programmatical addition)

Custom fields are supported by all the registered portal assets and provide support for adding additional attributes to them. Custom fields are automatically added to the asset editing views.

For creating and managing custom fields, there is a management user interface in the Control Panel and a programmatical API available. The management user interface can be accessed in *Control Panel → Configuration → Custom Fields*.



Custom fields are **typed**. The following types are supported:

- Selection of Integer Values
- Selection of Decimal Values
- Selection of Text Values
- Text Box
- Text Box–Indexed
- Text Field–Secret
- Text Field–Indexed
- Primitives
 - True/False
 - Date
 - Decimal number (64-bit)
 - Group of Decimal numbers (64-bit)
 - Decimal number (32-bit)
 - Group of Decimal numbers (32-bit)
 - Integer (32-bit)

- Group of Integers (32-bit)
- Integer (64-bit)
- Group of Integers (64-bit)
- Decimal Number or Integer (64-bit)
- Group of Decimal numbers or Integer (64-bit)
- Integer (16-bit)
- Group of Integers (16-bit)
- Text
- Group of Text Values
- Localized Text

Setting a custom field to **searchable** means that the value of the field is indexed and searchable. The indexed type can be chosen between a keyword and, for strings, a text type. It should be noticed that when a searchability setting is changed, the indexes must be updated to make the change effective to the existing entities.

Custom fields support permissions:

| Name | Key | Type | Hidden | Searchable |
|-----------------------|-----------------------|------|--------|----------------|
| Identification Number | Identification Number | Text | False | True |
| Googleaccesstoken | googleAccessToken | Text | True | Not Searchable |
| Googlerefreshtoken | googleRefreshToken | Text | True | Not Searchable |

Custom fields can be accessed from FreeMarker templates by calling the expando bridge:

```
<#assign expandoAttribute = user.getExpandoBridge().getAttribute("identification_number") />
```

Expando API

The underlying API for the custom fields is called the Expando API. That's why, when you take a look at Liferay's code, you will notice that custom fields are referred to as Expando fields in the code.

Why and when is a good time to use the programmatical API? A typical use case could be any scenario where there would be any automation in populating the custom field data, like populating a user custom field automatically on login time in an LDAP-integrated system based on the data gotten from LDAP. The Programmatical API also allows you to create virtual tables not related to any model class, extending the features available from the management user interface.

Expando Datamodel

The Expando virtual datamodel has the following entities:

- ExpandoTable
- ExpandoRow
- ExpandoColumn
- ExpandoValue

ExpandoTable represents a virtual table definition for a model class and is bound to it with the **classNameId** field:

```
mysql> describe expandotable;
+-----+-----+-----+-----+-----+
| Field | Type   | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+
| tableId | bigint(20) | NO  | PRI | NULL    |       |
| companyId | bigint(20) | YES | MUL | NULL    |       |
| classNameId | bigint(20) | YES |      | NULL    |       |
| name | varchar(75) | YES |      | NULL    |       |
+-----+-----+-----+-----+-----+
4 rows in set (0,01 sec)
```

ExpandoColumn represents a single column definition in a virtual table. *Name* field is the custom field name:

```
mysql> describe expandoColumn;
+-----+-----+-----+-----+-----+
| Field | Type   | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+
| columnId | bigint(20) | NO  | PRI | NULL    |       |
| companyId | bigint(20) | YES |      | NULL    |       |
| tableId | bigint(20) | YES | MUL | NULL    |       |
| name | varchar(75) | YES |      | NULL    |       |
| type_ | int(11) | YES |      | NULL    |       |
| defaultData | longtext | YES |      | NULL    |       |
| typeSettings | longtext | YES |      | NULL    |       |
+-----+-----+-----+-----+-----+
7 rows in set (0,00 sec)
```

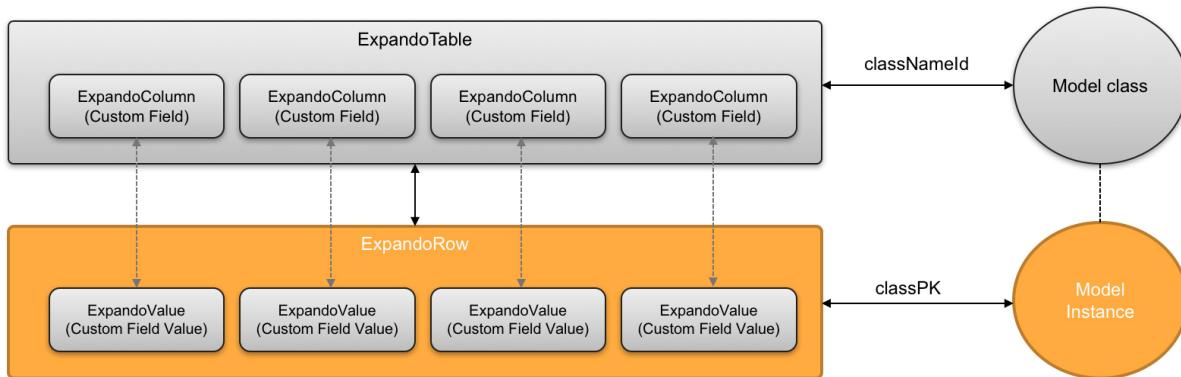
ExpandoRow represents a data row for a model instance and is bound to a model instance with the *classPK* field:

```
mysql> describe expandorow;
+-----+-----+-----+-----+-----+
| Field | Type   | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+
| rowId_ | bigint(20) | NO  | PRI | NULL    |       |
| companyId | bigint(20) | YES |      | NULL    |       |
| modifiedDate | datetime(6) | YES |      | NULL    |       |
| tableId | bigint(20) | YES | MUL | NULL    |       |
| classPK | bigint(20) | YES | MUL | NULL    |       |
+-----+-----+-----+-----+-----+
5 rows in set (0,00 sec)
```

ExpandoValue represents a single custom field value for an asset instance and is bound to that model instance with the *classPK* field:

```
mysql> describe expandoValue;
+-----+-----+-----+-----+-----+
| Field | Type   | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+
| valueId | bigint(20) | NO  | PRI | NULL    |
| companyId | bigint(20) | YES | YES | NULL    |
| tableId | bigint(20) | YES | YES | MUL    |
| columnId | bigint(20) | YES | YES | MUL    |
| rowId_ | bigint(20) | YES | YES | MUL    |
| classNameId | bigint(20) | YES | YES | MUL    |
| classPK | bigint(20) | YES | YES | NULL    |
| data_ | longtext | YES |      | NULL    |
+-----+-----+-----+-----+-----+
8 rows in set (0.01 sec)
```

Putting all this together into a logical view, the Expando model architecture could be presented as in the diagram below:



Using Expando Services

Below is a list of core Expando API services by the virtual datamodel entity:

- **ExpandoTables:**
 - ExpandoTableLocalService
 - ExpandoTableService
- **ExpandoColumns:**
 - ExpandoColumnLocalService
 - ExpandoColumnService
- **ExpandoRows:**
 - ExpandoRowLocalService
 - ExpandoRowService
- **ExpandoValues:**
 - ExpandoValueLocalService
 - ExpandoValueService

As always with Liferay's core services, the local service variant is for privileged access without access control.

Below are two examples of using the Expando services.

Create an Expando table programmatically:

```
protected long getExpandoTableId(long companyId, String className) throws Exception {
    // Try to get the expando table for user. Create if not found.

    ExpandoTable expandoTable = null;

    try {
        expandoTable = _expandoTableLocalService.getDefaultTable(companyId, className);
```

```

    } catch (NoSuchTableException ne) {

        expandoTable = _expandoTableLocalService.addDefaultTable(companyId, className);
    }

    return expandoTable.getTableId();
}

```

Set an Expando field Value:

```

protected void addExpandoValue(HttpServletRequest request, String value)
    throws Exception {

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

    long companyId = themeDisplay.getCompanyId();

    long expandoTableId = getExpandoTableId(companyId, User.class.getName());

    long expandoColumnId = getExpandoColumnId(expandoTableId, ID_COLUMN);

    long classNameId = _portal.getClassNameId(User.class);

    long classPK = themeDisplay.getUser().getPrimaryKey();

    _expandoValueLocalService.addValue(classNameId, expandoTableId, expandoColumnId,
        classPK, value);
}

```

In the user interface, Expando attributes can be accessed by calling the Expando bridge of an entity in the same way as in a FreeMarker template:

```
themeDisplay.getUser().getExpandoBridge().getAttribute("identification_number")
```

All custom attributes can be listed conveniently by using the liferay-ui tag library:

```

<liferay-ui:custom-attribute-list
    className="<%=_User.class.getName() %>"
    classPK="<%=_userId %>"
    label="true"
/>

```

Exercises

Extend Liferay's User Profile Using Expando and Lifecycle Actions

Introduction

In this exercise, we will be adding a LinkedIn profile ID to Liferay's User Profile. We will be using Expando. The implementation class we will be using is a *LifecycleAction* component. This will allow us to execute a piece of code each time the Liferay Server completes its start-up sequence.

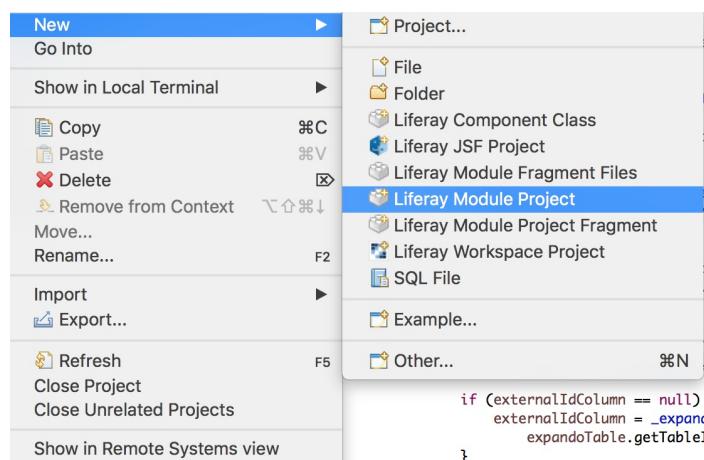
We will be using the *api* project template as a starting point for this module since there doesn't exist a dedicated *startup-action* project template.

Overview

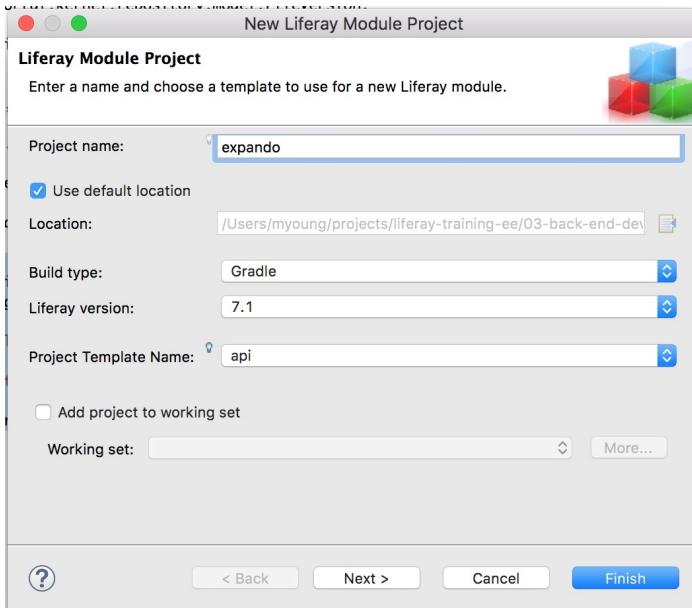
- 1 Create a Liferay Module project using the API template
- 2 Resolve Dependencies
- 3 Create a LifecycleAction component that implements the LifecycleAction interface
- 4 Deploy and test

Create a Liferay Module Project Using the API Template

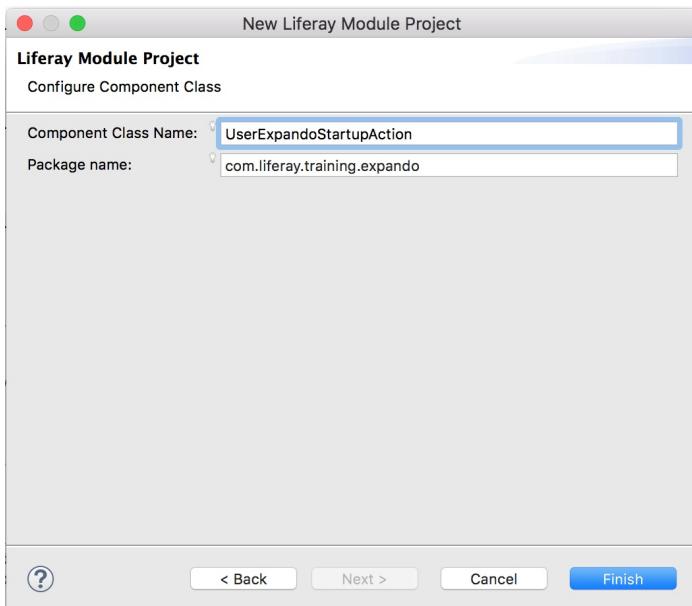
1. [Go to training-workspace](#).
2. [Right-click](#) on the workspace to open a context menu.
3. [Choose New → Liferay Module Project](#).



4. [Type `expando`](#) in the *Project Name* field.
5. [Choose `api`](#) in the *Project Template Name* field.
6. [Click `Next`](#).



7. Type `UserExpandoStartupAction` in the *Component Class Name* field.
8. Type `com.liferay.training.expando` in the *Package name* field.
9. Click *Finish*.



Resolve Dependencies

Next, we will have to adjust the `bnd.bnd` and `build.gradle` file generated by Liferay Studio. Since we used the `api` template, `build.gradle` will only contain the bare minimum of dependencies to set up an OSGi project. Because we plan to use the `LifecycleAction` interface, we have to depend on the `com.liferay.portal.kernel` package. Additionally, we depend on the `org.osgi.service.log` package, since we want to monitor our `LifecycleAction`'s execution.

1. Open the `build.gradle` file and replace it with the following code:

```
dependencies {
    compileOnly group: "com.liferay.portal", name: "com.liferay.portal.kernel", version: "3.0.0"
    compileOnly group: "org.osgi", name: "org.osgi.core", version: "6.0.0"
    compileOnly group: "org.osgi", name: "org.osgi.service.component.annotations", version: "1.3.0"
    compileOnly group: "org.osgi", name: "org.osgi.service.log", version: "1.3.0"
```

```
}
```

2. **Save** the file.
3. **Open** the `bnd.bnd` file and replace it with the following.
 - o This step is not necessary, but it's helpful to use a more human-readable *Bundle-Name* in case you need to troubleshoot the bundle in the Gogo Shell, for example.

```
Bundle-Name: Expando Startup Action
Bundle-SymbolicName: com.liferay.training.expando
Bundle-Version: 1.0.0
```

4. **Save** the file.

Create a UserExpandoStartupAction Component That Implements the LifecycleAction Interface

In this step, we will be implementing a `LifecycleAction` interface. As mentioned in the introduction, a `LifecycleAction` allows you to participate in various parts of Liferay's application lifecycle. For our Expando exercise, we will be waiting for the `application.startup.events` event. The `application.startup.event` is run once for every instance initialization, making this ideal for us to execute code that only needs to be performed once. We specify that we are hooking into the `application.startup.event` with the following component property:

```
property = {"key=application.startup.events"},
```

1. **Expand** the `expando` module project.
2. **Right-click** on the `com.liferay.training.expando.api` package in the `src/main/java` folder.
3. **Choose Refactor → Rename**.
4. **Edit** the name of the package to remove `.api`.
 - o It should now be `com.liferay.training.expando`.
5. **Click Ok** (You may safely ignore the warning → *Continue*).
6. **Open** `UserExpandoStartupAction.java` located in the `com.liferay.training.expando` package.
7. **Implement** the `LifecycleAction` interface (Hint: type `public class UserExpandoStartupAction implements LifecycleAction {` and use the option *Add unimplemented methods* to fix the indicated error). Afterwards your class should look as follows:

```
package com.liferay.training.expando;

import com.liferay.portal.kernel.events.ActionEvent;
import com.liferay.portal.kernel.events.LifecycleAction;
import com.liferay.portal.kernel.events.LifecycleEvent;

public class UserExpandoStartupAction implements LifecycleAction {

    @Override
    public void processLifecycleEvent(LifecycleEvent lifecycleEvent) throws ActionException {
        // TODO Auto-generated method stub
    }
}
```

1. **Add** following component annotation:

```
...
@Component(
    immediate = true,
    property = {"key=application.startup.events"},
    service = LifecycleAction.class
```

```

    )
public class UserExpandoStartupAction implements LifecycleAction {
...

```

2. **Add** references to the Expando and other utility services we'll need for our implementation (below the `processLifecycleEvent` method stub).

```

@Reference
private ExpandoColumnLocalService _expandoColumnLocalService;

@Reference
private ExpandoTableLocalService _expandoTableLocalService;

@Reference
private ExpandoValueLocalService _expandoValueLocalService;

@Reference
private LogService _log;

@Reference
private UserLocalService _userLocalService;

```

3. **Edit** the class to implement the `processLifecycleEvent()` method.

- We check if the table and column exists, and if it doesn't, we'll create them. This will ensure that we only try to create these elements once.

```

@Override
public void processLifecycleEvent(LifecycleEvent lifecycleEvent)
    throws ActionException {

    System.out.println("application.startup.events=" + lifecycleEvent);

    long companyId = CompanyThreadLocal.getCompanyId();

    try {
        ExpandoTable expandoTable = _expandoTableLocalService.getDefaultTable(
            companyId, User.class.getName());

        if (expandoTable == null) {
            expandoTable = _expandoTableLocalService.addDefaultTable(
                companyId, User.class.getName());
        }

        ExpandoColumn externalIdColumn = _expandoColumnLocalService.getColumn(
            expandoTable.getTableId(), "linkedin_profile_id");

        if (externalIdColumn == null) {
            externalIdColumn = _expandoColumnLocalService.addColumn(
                expandoTable.getTableId(), "linkedin_profile_id", ExpandoColumnConstants.STRING);
        }
    }
    catch (PortalException pe) {
        _log.log(LogService.LOG_ERROR, pe.getMessage(), pe);
    }
}

```

4. **Press** `CTRL+SHIFT+O` to resolve any missing imports.

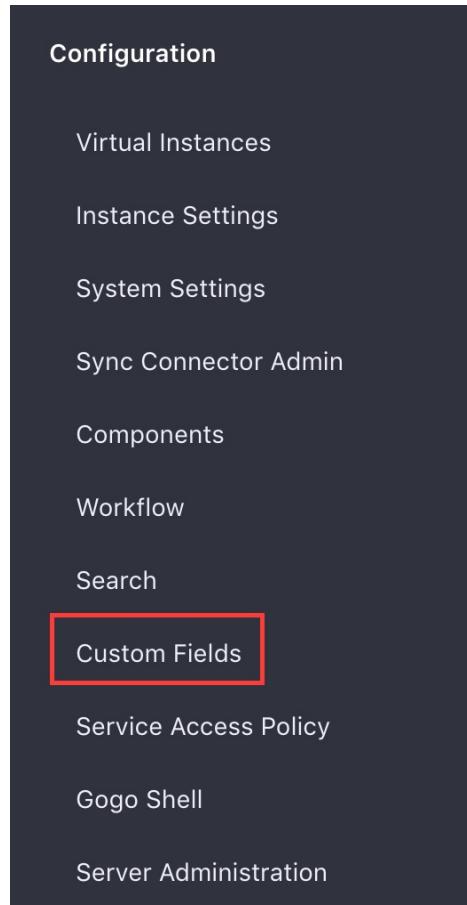
Deploy and Test

Let's deploy and test our customization.

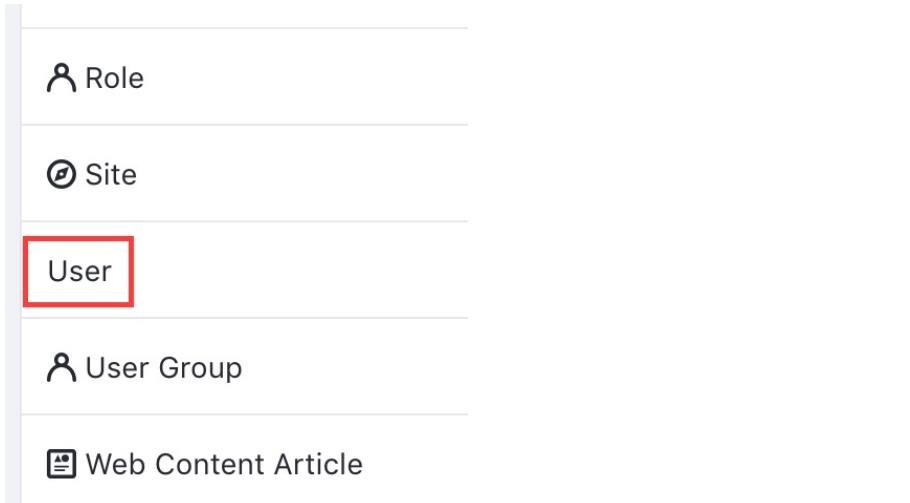
1. **Go to** the *Gradle Tasks* view in *Liferay Developer Studio*.
2. **Double-click** the `deploy` task from `expando/build`.

| ▼  expando | | Expando Startup Action |
|---|--|--------------------------------|
| ▼  build | | |
| assemble | | Assembles the outputs of thi |
| build | | Assembles and tests this pro |
| buildCSS | | Build CSS files. |
| buildDependents | | Assembles and tests this pro |
| buildLang | | Runs Liferay Lang Builder to |
| buildNeeded | | Assembles and tests this pro |
| buildSoy | | Compiles Closure Templates |
| classes | | Assembles main classes. |
| clean | | Deletes the build directory. |
| configJSMODULES | | Generates the config file neec |
| deploy | | Assembles the project and d |
| jar | | Assembles a jar archive cont |
| replaceSoyTranslation | | Replaces 'goog.getMsg' defini |
| testClasses | | Assembles test classes. |
| testIntegrationClasses | | Assembles test integration cl |
| transpileJS | | Transpiles JS files. |

- o This will deploy the `expando` bundle to the Liferay Server.
3. **Restart** your server since the `LifecycleAction` is bound to Liferay's `application.startup.events`.
 4. **Open** your browser to <http://localhost:8080> and log in.
 5. **Click** *Menu* in the top left corner to open the *Product Menu*.
 6. **Go to** `Control Panel → Configuration → Custom Fields`.



7. **Click** *User*.



Here you can verify that the `linkedin_profile_id` appears as one of the user custom fields.

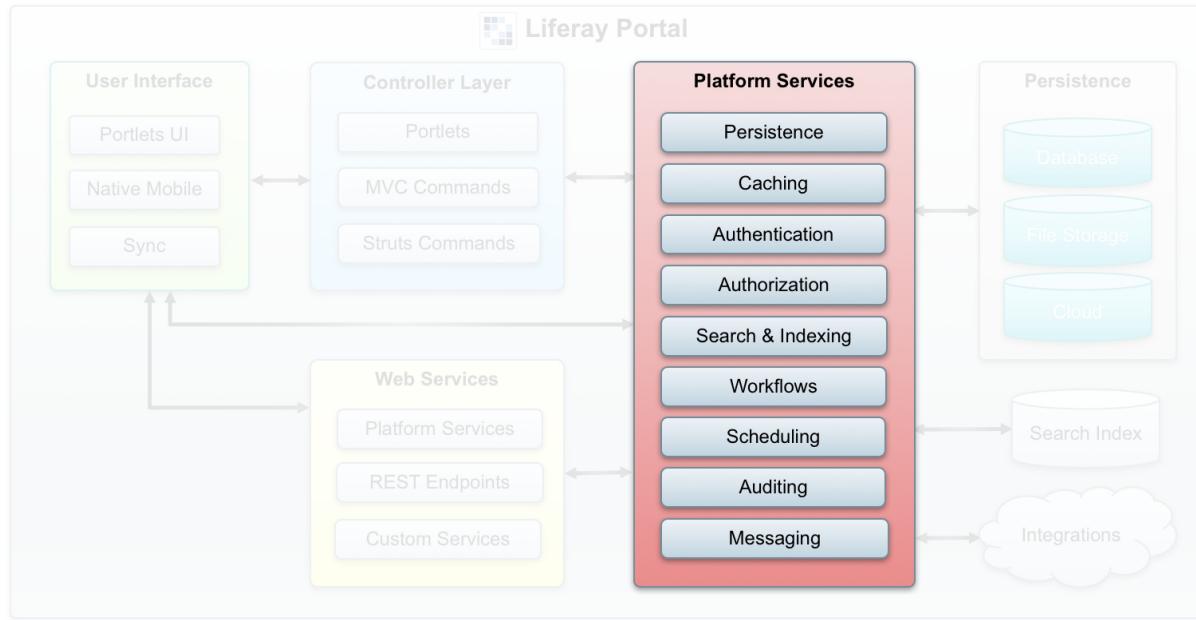
| Name | Key | Type | Hidden | Searchable | |
|--|---------------------|------|--------|----------------|---|
| <input type="checkbox"/> Googleaccesstoken | googleAccessToken | Text | True | Not Searchable | ⋮ |
| <input type="checkbox"/> Googlerefreshtoken | googleRefreshToken | Text | True | Not Searchable | ⋮ |
| <input type="checkbox"/> LinkedIn Profile Id | linkedin_profile_id | Text | False | Not Searchable | ⋮ |

Takeaways

In this exercise, we showed you how to extend Liferay's User object model using Expando. The User object is one of the most commonly extended objects, and, as you've seen here, it's very easy to do with Expando. Although Expando is a great tool to extend or create databases dynamically, they are not without their limitations. If you are doing more than adding simple fields to the database, you may want to consider creating a new database table with the User Id as a foreign key instead.

We also had an introduction to LifecycleAction's, which we will cover in Chapter 12 with much more detail.

Chapter 10: Customize the Service Layer



Chapter Objectives

- Understand the Service Wrapper Concept
- Understand @Reference Annotation Policies
- Learn How to Customize Liferay's Core Services

Introduction

Sometimes you might need to change Liferay's core service behavior. You might, for example, want to add custom logic or add new method signatures or trigger actions on other Liferay services in a transactionally safe way.

In the following, we will discuss two customization scenarios. In the first, we customize a core service with a service wrapper class. In the second, we'll learn how to control and override an injected OSGi service reference in a consuming OSGi component class. Depending on the original implementation, this method allows you to replace a complete service implementation in a platform core class.

Service Wrappers

All Liferay core services have been designed and generated with the Liferay Service Builder pattern, which automatically generates service wrapper APIs for both the remote and local service variants. The wrapper API is a service façade that provides a transactionally safe way to customize the underlying service. In a Service Builder project, this API will be created in the API module:

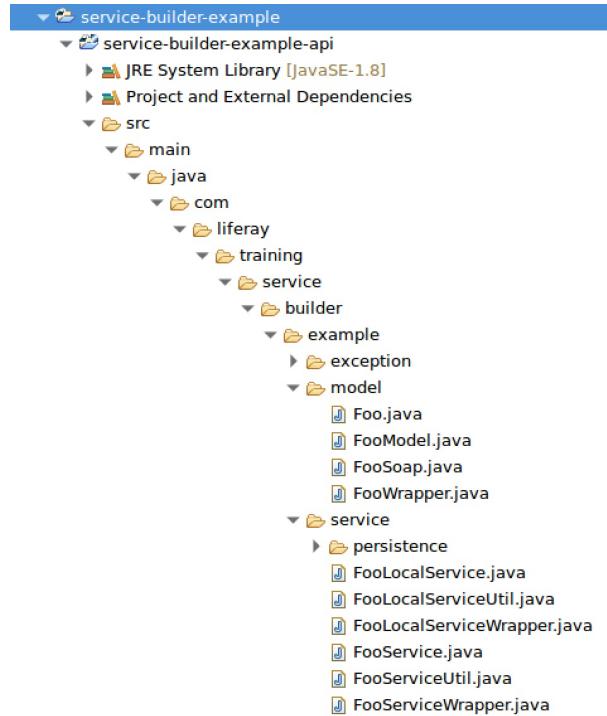


Figure: Service wrappers in the API module.

When we take a look at a platform core application, like the Blogs application, we'll notice that it is having the same design:

[liferay-portal / modules / apps / blogs / blogs-api / src / main / java / com / liferay / blogs / service /](#)

| This branch is 3 commits behind master. | | |
|--|--|----------------------------------|
| | | Pull request |
|  sergiogonzalez and brianchandotcom LPS-80024 Autogenerated | | Latest commit fa6d1ee 7 days ago |
| .. | | |
|  persistence | LPS-80332 Move projects to the apps folder | 8 days ago |
|  BlogsEntryLocalService.java | LPS-80332 Move projects to the apps folder | 8 days ago |
|  BlogsEntryLocalServiceUtil.java | LPS-80332 Move projects to the apps folder | 8 days ago |
|  BlogsEntryLocalServiceWrapper.java | LPS-80332 Move projects to the apps folder | 8 days ago |
|  BlogsEntryService.java | LPS-80024 Autogenerated | 5 days ago |
|  BlogsEntryServiceUtil.java | LPS-80024 Autogenerated | 5 days ago |
|  BlogsEntryServiceWrapper.java | LPS-80024 Autogenerated | 5 days ago |
|  BlogsStatsUserLocalService.java | LPS-80332 Move projects to the apps folder | 8 days ago |
|  BlogsStatsUserLocalServiceUtil.java | LPS-80332 Move projects to the apps folder | 8 days ago |
|  BlogsStatsUserLocalServiceWrapper.java | LPS-80332 Move projects to the apps folder | 8 days ago |

Figure: Blogs application service wrappers

A Service wrapper stub class contains façade methods for all methods of the underlying service class. Below is an excerpt of the `BlogsEntryLocalServiceWrapper` stub. In `addBlogsEntry()`, we see that it is just calling the underlying service:

```
@ProviderType
public class BlogsEntryLocalServiceWrapper implements BlogsEntryLocalService,
    ServiceWrapper<BlogsEntryLocalService> {

    public BlogsEntryLocalServiceWrapper(
        BlogsEntryLocalService blogsEntryLocalService) {
        _blogsEntryLocalService = blogsEntryLocalService;
    }

    ...

    /**
     * Adds the blogs entry to the database. Also notifies the appropriate model listeners.
     *
     * @param blogsEntry the blogs entry
     * @return the blogs entry that was added
     */
    @Override
    public com.liferay.blogs.model.BlogsEntry addBlogsEntry(
        com.liferay.blogs.model.BlogsEntry blogsEntry) {
        return _blogsEntryLocalService.addBlogsEntry(blogsEntry);
    }

    ...
}
```

The important thing is that when you do a service call, in this case, `BlogsEntryLocalService.addBlogsEntry()`, it's actually the service wrapper façade method that gets executed. By default, it just redirects the call to the underlying service. By creating your own service wrapper, you can override this behavior.

Exercises

Customize Liferay UserLocalService with a Service Wrapper

Introduction

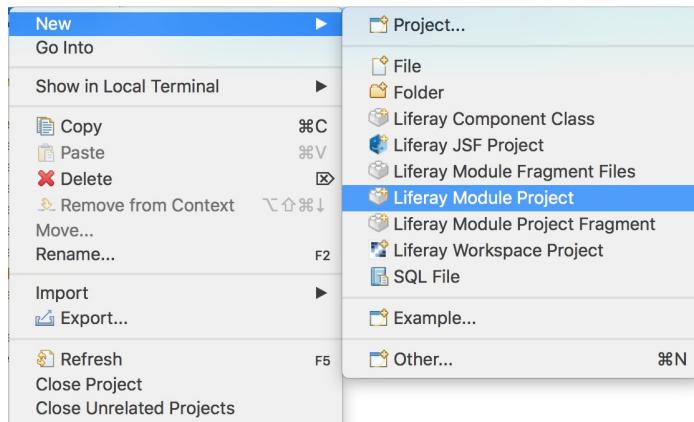
In this exercise, we will be overriding the UserLocalService with a Service Wrapper component. We will override the addUserWithWorkflow() method in such a way that it will only allow users to be added from their work email address.

Overview

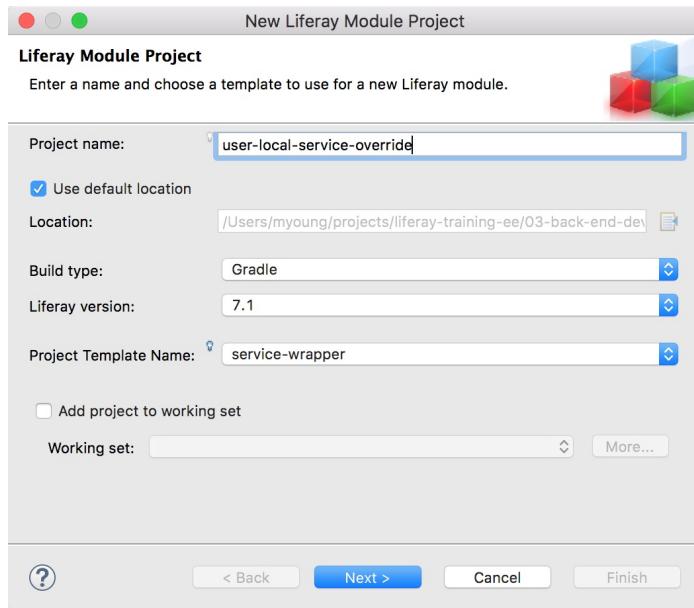
- ① Create a Liferay Module project using the service-wrapper template
- ② Override the addUserWithWorkflow() method
- ③ Deploy and test

Create a Liferay Module Project Using the service-wrapper Template

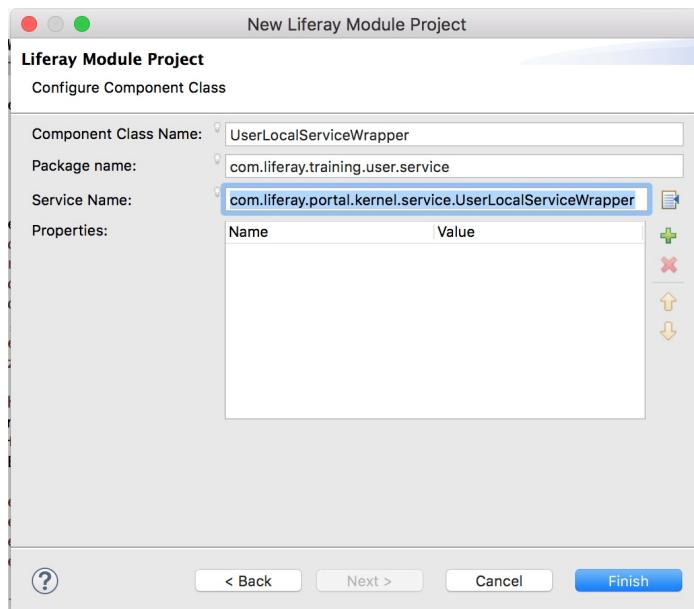
1. [Go to training-workspace](#).
2. [Right-click](#) on the workspace to open the context menu.
3. [Choose New → Liferay Module Project](#).



4. [Type user-local-service-override](#) in the *Project Name* field.
5. [Choose service-wrapper](#) in the *Project Template Name* field.
6. [Click Next](#).

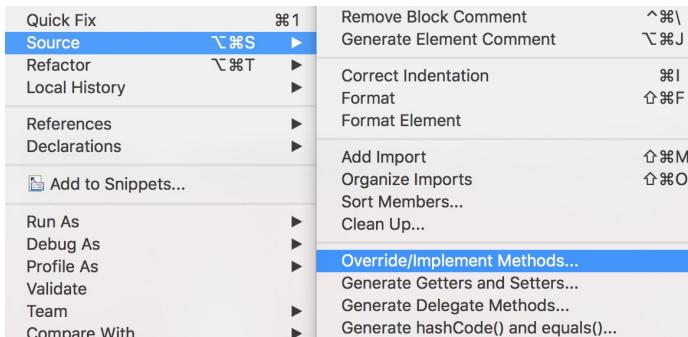


7. Type `UserLocalServiceOverride` in the *Component Class Name* field.
8. Type `com.liferay.training.user.service` in the *Package name* field.
9. Choose `com.liferay.portal.kernel.service.UserLocalServiceWrapper` in the *Service Name* field.
10. Click *Finish*.

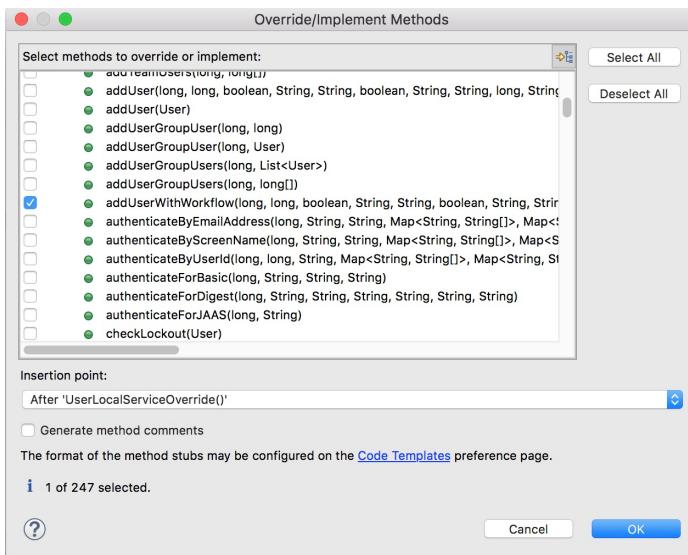


Override the `addUserWithWorkflow()` Method

1. Open the `UserLocalServiceOverride` class in the `com.liferay.training.user.service` package.
2. Right-click on the `UserLocalServiceOverride()` constructor.
3. Choose the *Source → Override/Implement Methods* option.



4. Check the `addUserWithWorkflow()` method.



5. Click OK.

6. Edit the class to implement the `addUserWithWorkflow()` method as follows:

```

@Override
public User addUserWithWorkflow(
    long creatorUserId, long companyId, boolean autoPassword,
    String password1, String password2, boolean autoScreenName,
    String screenName, String emailAddress, long facebookId, String openId,
    Locale locale, String firstName, String middleName, String lastName,
    long prefixId, long suffixId, boolean male, int birthdayMonth,
    int birthdayDay, int birthdayYear, String jobTitle, long[] groupIds,
    long[] organizationIds, long[] roleIds, long[] userGroupIds,
    boolean sendEmail, ServiceContext serviceContext)
throws PortalException {

    if (emailAddress.contains("@gmail.com") ||
        emailAddress.contains("@yahoo.com") ||
        emailAddress.contains("@aol.com") ||
        emailAddress.contains("@hotmail.com")) {

        System.out.println(
            "You must enter a work email address. User will not be added.");

        throw new PortalException("You must enter a work email address.");
    }

    return super.addUserWithWorkflow(
        creatorUserId, companyId, autoPassword, password1, password2,
        autoScreenName, screenName, emailAddress, facebookId, openId,

```

```
        locale, firstName, middleName, lastName, prefixId, suffixId, male,
        birthdayMonth, birthdayDay, birthdayYear, jobTitle, groupIds,
        organizationIds, roleIds, userGroupIds, sendEmail, serviceContext);
}
```

7. **Save** the file.

You'll see that there are a number of personal email domains that will be rejected as the user is added.

Deploy and Test

1. **Go to** the *Gradle Tasks* view in *Liferay Developer Studio*.
2. **Double-click** the `deploy` task from `user-local-service-override/build`.
 - o This will deploy the *User Local Service Override* bundle to the Liferay Server.
3. **Open** your browser to <http://localhost:8080> and log in.
4. **Click** *Menu* in the top left corner.
5. **Go to** *Control Panel* → *Users* → *Users and Organizations*.
6. **Click** the *Add* button.
7. **Create** a user with one of the rejected email addresses in the `addUserWithWorkflow()` method.

8. **Click** *Save* to try and save the new user.

You will see an error message on the page stating that the user was not added. You will also see that the `System.out.println()` statement is in the console view.

```
2018-06-05 22:59:53.323 INFO [Refresh Thread: Equinox Container: 6]
You must enter a work email address. User will not be added.
2018-06-05 23:00:48.686 INFO [fileinstall-/Users/myoung/projects/l]
```

Takeaways

Using the pattern above, you should be able to override any Liferay Service with a service wrapper.

Override OSGi Service References

OSGi Declarative Services allow you to automatically and dynamically inject a service from the OSGi service registry into your component using the `@Reference` annotation. Below is an excerpt of a `BlogsAdminPortlet` component, having multiple service references:

```
public class BlogsAdminPortlet extends BaseBlogsPortlet {

    @Override
    public void render(
        RenderRequest renderRequest, RenderResponse renderResponse)
        throws IOException, PortletException {

        renderRequest.setAttribute(AssetWebKeys.ASSET_HELPER, _assetHelper);

        renderRequest.setAttribute(TrashWebKeys.TRASH_HELPER, _trashHelper);

        super.render(renderRequest, renderResponse);
    }

    @Reference(
        target = "(&(release.bundle.symbolic.name=com.liferay.blogs.web)(release.schema.version=1.2.0))",
        unbind = "-"
    )
    protected void setRelease(Release release) {
    }

    @Reference
    private AssetHelper _assetHelper;

    @Reference
    private TrashHelper _trashHelper;
}
```

<https://github.com/liferay/liferay-portal/blob/7.1.x/modules/apps/blogs/blogs-web/src/main/java/com/liferay/blogs/web/internal/portlet/BlogsAdminPortlet.java>

As an OSGi container allows multiple implementations of a service to coexist, which implementation gets injected? From the perspective of consuming OSGi components, the selection depends on the following factors:

- The service implementation component's `service.ranking` property value
- `@Reference` attributes

service.ranking

The service ranking is an OSGi component property that defines the priority of service components among the components implementing the same service:

```
@Component(
    immediate = true,
    property = {
        "context.id=BladeCustomJspBag",
        "context.name=Test Custom JSP Bag",
        "service.ranking=Integer=100"
    }
)
public class BladeCustomJspBag implements CustomJspBag {
    ...
}
```

The default value for the `service.ranking` property is 0. A service that has a higher number gets a priority over the lower. If there are multiple service implementations available in the OSGi service registry, an implementation that has the highest priority should get injected in the consuming component. In practice, this behavior ultimately depends on the consuming component's `@Reference` annotation policies. With the default values, the first implementation available gets injected and is reluctant to change even if an implementation with higher priority becomes available. This is called a static and reluctant policy. In this scenario, the OSGi bundle startup order defines the implementation.

@Reference Attributes

```

@Reference(
    bind = "setPermissionFilterQueryBuilder",
    cardinality = ReferenceCardinality.MANDATORY,
    policy = ReferencePolicy.DYNAMIC,
    policyOption = ReferencePolicyOption.GREEDY,
    service = PermissionFilterQueryBuilder.class,
    unbind = "removePermissionFilterQueryBuilder"
)
private PermissionFilterQueryBuilder _permissionFilterQueryBuilder;

...

```

Target

Target attributes allow you to filter the service implementation to be injected. A query targets a component's properties and uses LDAP query syntax.

In the example below, the index writer component of the Elasticsearch adapter is asking for a reference to a `SpellCheckIndexWriter` implementation that has a component property `"search.engine.impl=Elasticsearch"`. Without the target attribute, an implementation from some other adapter might get implemented:

```

@Component(
    immediate = true, property = {"search.engine.impl=Elasticsearch"},
    service = IndexWriter.class
)
public class ElasticsearchIndexWriter extends BaseIndexWriter {

    ...

    @Override
    @Reference(target = "(search.engine.impl=Elasticsearch)", unbind = "-")
    public void setSpellCheckIndexWriter(
        SpellCheckIndexWriter spellCheckIndexWriter) {

        super.setSpellCheckIndexWriter(spellCheckIndexWriter);
    }
}

```

Below is the `SpellCheckIndexWriter` that has the matching property:

```

@Component(
    immediate = true, property = {"search.engine.impl=Elasticsearch"},
    service = SpellCheckIndexWriter.class
)
public class ElasticsearchSpellCheckIndexWriter
    extends BaseGenericSpellCheckIndexWriter {

```

Reference Cardinality

Reference cardinality defines whether a reference has to be satisfied for a component to be able to activate:

- **MANDATORY:** the reference must be available and injected before this component will start (default)
- **OPTIONAL:** the reference is not required for the component to start and work
- **MULTIPLE:** the reference is not required, but multiple resources may satisfy the reference and the component will take all of them
- **AT LEAST ONE:** like multiple, but at least one reference is required for the component to start
- When using multiple options, a setter method has to be defined.

Reference Policy

The reference policy defines whether an injected service reference can be replaced dynamically:

- **STATIC:** the reference is required and the component will not be notified of alternative services as they become available (default)
- **DYNAMIC:** the reference is not required and the component accepts new references as they become available

Reference PolicyOption

The policy option provides more granularity for the dynamic injection policy:

- **RELUCTANT:** For a single reference cardinality, new available references will be ignored. For multiple reference cardinality, new reference potentials will be bound.
- **GREEDY:** As new reference potentials become available, the component will bind to them.

Other @Reference Attributes

- **bind:** the name of the bind method for this reference
- **field:** the name of the field for this reference
- **fieldOption:** the field option for this reference:
 - REPLACE: replace the field value with a new value when there are changes to the bound services
 - UPDATE: update the collection referenced by the field when there are changes to the bound services
- **name:** name of this reference
- **scope:** scope for the reference
- **service:** the service class name for this reference
- **unbind:** unbind event method for the reference
- **updated:** updated event method for the reference

Overriding Static and Reluctant References

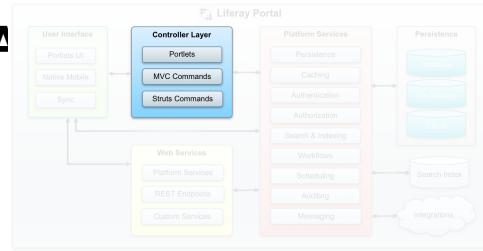
The general steps for overriding static and reluctant service references:

- ① Find the target component and service details
- ② Create a custom service implementation
- ③ Configure the component to use the custom service

Chapter 11: Override Controller A

Chapter Objectives

- Learn the Ways to Override the Portal Controller Layer
- Learn How to Override the Legacy Struts Actions
- Learn How to Override MVC Command Actions



Introduction

The Liferay platform is designed to follow the MVC pattern. In the controller layer, there are two kinds of action frameworks: the Struts actions and MVC actions commands. Although Struts is a legacy framework and it will be gradually replaced, Struts is still being used by some Liferay applications.

Override Struts Actions

Liferay is using the Apache Struts MVC framework version 1.3 for some of the native applications, notably the login portlet. Although it will be replaced by MVC action commands, in some cases, you might still need to override the Struts actions. Applications that still use the Struts actions include:

- **Bookmarks:** Find and Open Bookmark Entries
- **Blogs:** Find Entry
- **Documents and Media:** Find File, Get File, and File Folder
- **Message Boards:** Find Recent Posts, Categories, Threads, Messages, and Edit Discussion
- **Knowledge Base:** Find Articles
- **Login:** Facebook Connect and Google Login
- **Wiki:** Get Page Attachment

Struts Basic Concepts

Let's take a look at the Struts basic concepts:

- Action Servlet
- Action Controller
- Action Form
- View Layer

Action Servlet

The action servlet is a servlet that forwards Struts requests to Struts action controllers. The servlet mapping is defined in `web.xml` and the path to action mapping in `struts-config.xml`.

Action Controller

An action controller is a class that extends the `org.apache.struts.action.Action` and is responsible for performing actions based on the action path. Examples of action paths:

- `/view_entry`
- `/edit_entry`

Action Form (Bean)

The action form represents the model layer. Practically, the action form is a data transfer object that transports model objects from back-end to user interface and vice versa. The action controller takes care of syncing the action form values with the persistence layer.

The user interface action form name to Java bean mapping is defined in the `struts-config.xml` file.

View Layer

The view layer of the Struts framework is implemented with JSP. Process-wise in the back-end, the action controllers first set an `action forward` name. The name for the JSP file mapping is defined in `struts-config.xml` or, if you're using Apache Tiles, in `tiles-defs.xml`.

Struts Example

Creating an action in JSP

```
<portlet:actionURL var="editLayoutURL">
    <portlet:param name="struts_action" value="/layouts_admin/edit_layouts" />
</portlet:actionURL>
```

Mapping an action to the controller in struts-config.xml

```
<action
    path="/portal/edit_layout"
    type="com.liferay.portal.action.EditLayoutAction"
>
```

An action controller

```
public class EditLayoutAction extends Action{

    public ActionForward execute(ActionMapping mapping, ActionForm form,
        HttpServletRequest request, HttpServletResponse response)
        throws Exception {

        MyActionForm actionForm = (MyActionForm) form;
        MyActionForm.setMessage("Hello Struts!");

        return mapping.findForward("success");
    }

    ...
}
```

Action form

```
public class MyActionForm extends ActionForm{

    public String getMessage() {
        return _message;
    }

    public void setMessage(String message) {
        _message = message;
    }

    private String _message;
}
```

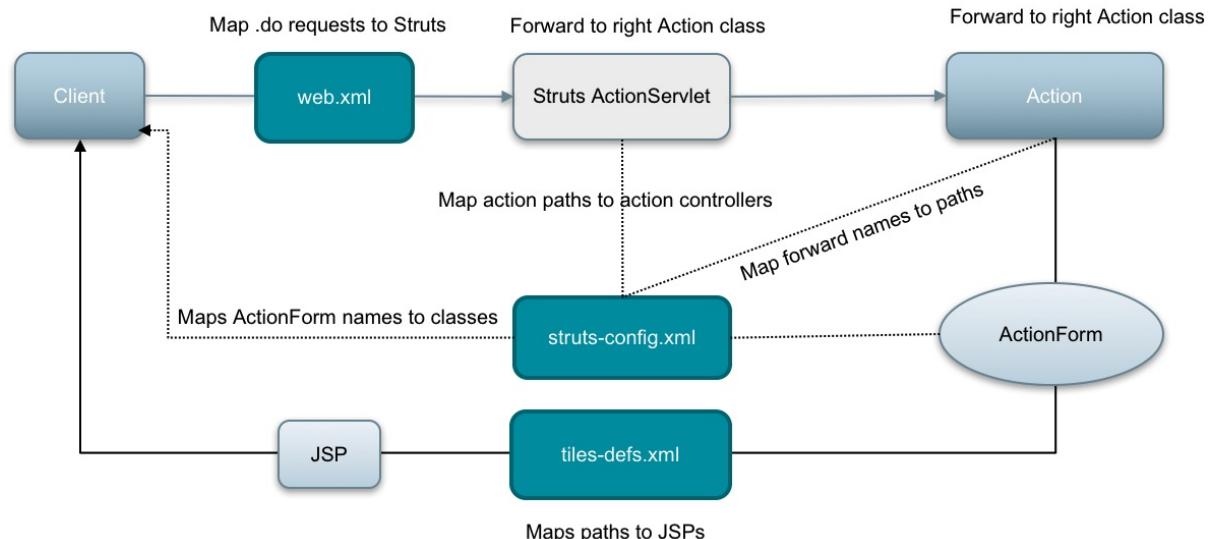
Forward name to JSP mapping in tiles-defs.xml

```
<tiles-definitions>

    <definition name="portlet" path="/common/themes/portlet.jsp">
        <put name="portlet_content" value="" />
        <put name="portlet_decorate" value="true" />
        <put name="portlet_padding" value="true" />
    </definition>
    ...

```

Below is a diagram summarizing the Struts process flow:



Overriding Struts Actions

Overriding Struts actions in the context of Liferay means overriding action controller classes. Generally, the steps for overriding Liferay Struts actions are as follows:

- ① Find the action path you are overriding (WEB-INF/struts- config.xml)
- ② Create a Liferay module using the API template
- ③ Create a new Struts Action component using New Component Class wizard
- ④ Define the Struts action path you are overriding
- ⑤ Override methods

There are two types of actions you can override:

- BaseStrutsActions (portal)
- BaseStrutsPortletAction (portlet applications)

In the exercise, we will override the portal logout action.

Links and Resources

- **Liferay struts-config.xml**
<https://github.com/liferay/liferay-portal/blob/7.1.x/portal-web/docroot/WEB-INF/struts-config.xml>
- **Liferay tiles-config.xml**
<https://github.com/liferay/liferay-portal/blob/7.1.x/portal-web/docroot/WEB-INF/tiles-defs.xml>
- **Override Struts Portal Action Blade Sample**
<https://github.com/liferay/liferay-blade-samples/tree/master/liferay-workspace/extensions/struts-action>
- **Override Struts Portlet Action Blade Sample**
<https://github.com/liferay/liferay-blade-samples/tree/master/liferay-workspace/extensions/struts-portlet-action>

Exercises

Override the Portal Logout Struts Action

Introduction

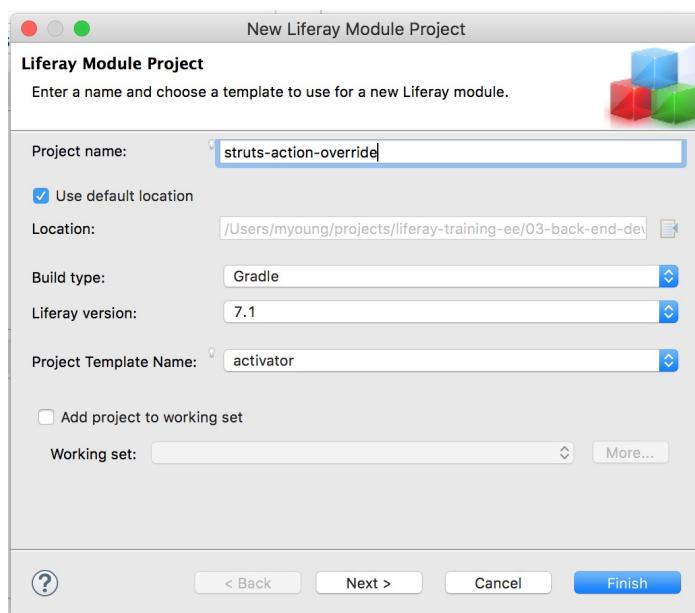
In this exercise, we will be overriding Liferay's LogoutAction struts action. This is one of the more common portal Struts Actions to override, as many organizations would like to redirect the user to a custom location after they logout. This is exactly what we will be doing here.

Overview

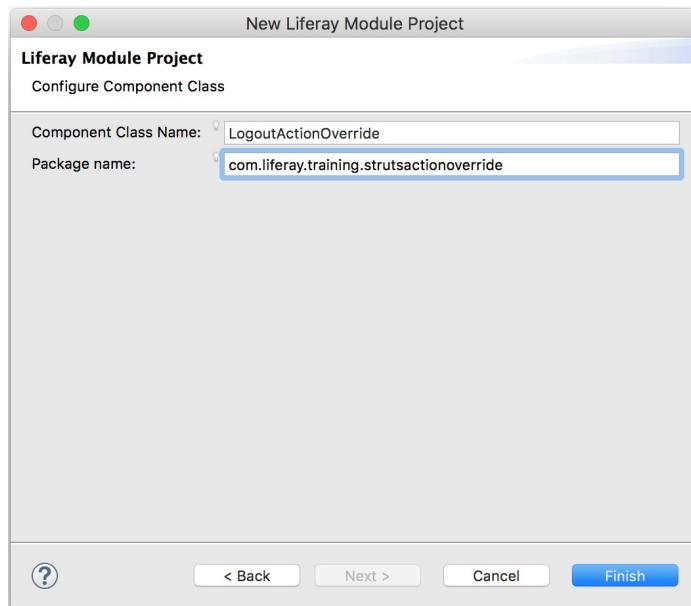
- ① Create a Liferay Module project using the *activator* template
- ② Resolve dependencies
- ③ Create the *LogoutActionOverride* class
- ④ Implement the *execute()* method
- ⑤ Deploy and test

Create a Liferay Module Project Using the Activator Template

1. **Go to** the *training-workspace*.
2. **Right-click** to open a context menu.
3. **Choose** *New → Liferay Module Project*.
4. **Type** *struts-action-override* in the *Project Name* field.
5. **Choose** *activator* in the *Project Template Name* field since there is no dedicated struts-action-override template.
6. **Click** *Next*.



7. Type `LogoutActionOverride` in the *Component Class Name* field.
8. Type `com.liferay.training.strutsactionoverride` in the *Package name* field.
9. Click **Finish**.



10. Expand the `struts-action-override` module project folder.
11. Delete the `LogoutActionOverrideActivator` class that was auto-created in the `com.liferay.training.strutsactionoverride` package.
 - o We will not be using this class in our customization.

Resolve Dependencies

Next, we will have to adjust the `bnd.bnd` and `build.gradle` files generated by Liferay Studio. Since we used the `activator` template, `build.gradle` will only contain the bare minimum of dependencies to set up an OSGi project. Because we intend to implement `com.liferay.portal.kernel.struts.BaseStrutsAction` we have to depend on the `com.liferay.portal.kernel`, `portlet-api`, and `javax.servlet-api` packages. Since we implement an OSGi component, we also have to depend on the `org.osgi.service.component.annotations` package.

1. Open the `build.gradle` file for the project.
2. Add the required dependencies. Afterwards the content of the file should look like the following:

```
dependencies {
    compileOnly group: "com.liferay.portal", name: "com.liferay.portal.kernel", version: "3.0.0"
    compileOnly group: "javax.portlet", name: "portlet-api", version: "3.0.0"
    compileOnly group: "javax.servlet", name: "javax.servlet-api", version: "3.0.1"
    compileOnly group: "org.osgi", name: "osgi.cmpn", version: "6.0.0"
    compileOnly group: "org.osgi", name: "org.osgi.service.component.annotations", version: "1.3.0"
}
```

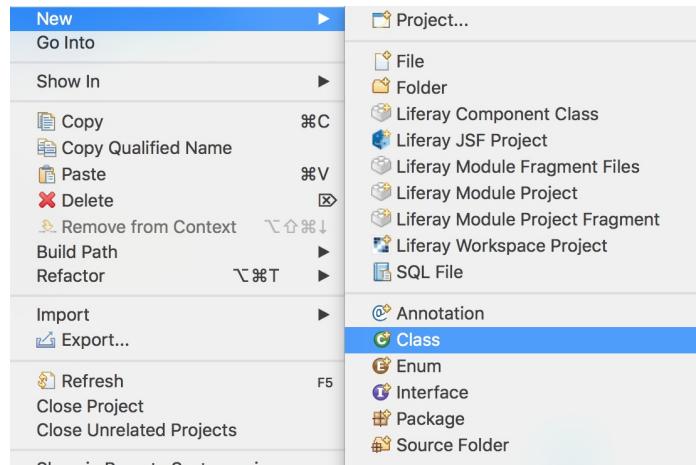
3. Save the file.
4. Open the `bnd.bnd` file.
5. Click on the *Source* tab.
6. Replace the contents of the file with the following:
 - o We are removing the Bundle Activator that we deleted in an earlier step.
 - o We are also renaming the bundle. This step is not necessary, but it's helpful to use a more human-readable `Bundle-Name` in case you need to troubleshoot the bundle in the Gogo Shell, for example.

```
Bundle-Name: Struts Action Override
Bundle-SymbolicName: com.liferay.training.strutsactionoverride
Bundle-Version: 1.0.0
```

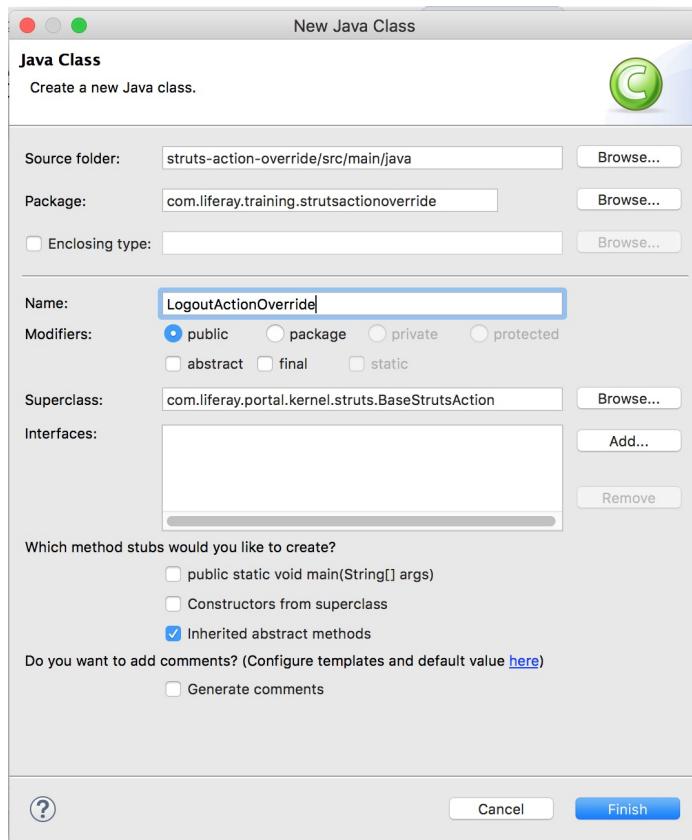
7. **Save** the file.

Create the LogoutActionOverride Class

1. **Right-click** on the *struts-action-override* project to open a context menu.
2. **Choose** *New → Class*.



3. **Type** *LogoutActionOverride* in the *Name* field.
4. **Type** *com.liferay.portal.kernel.struts.BaseStrutsAction* in the *Superclass* field.
 - o The rest of the fields should be correct by default.
5. **Click** *Finish*.



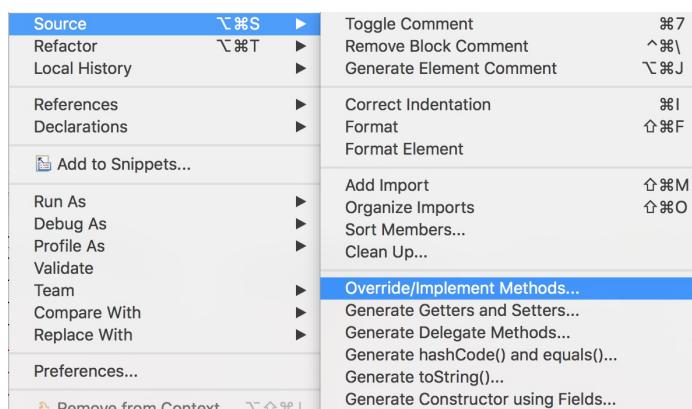
Implement the execute() Method

Some important points to note here:

- To specify which struts path to override: `path=/portal/c/logout`
- `originalStrutsAction` refers to the `LogoutAction` that we are overriding. This is not obvious in the code, so we want to note that here.
- To invoke the original behavior of the `LogoutAction`, which is often important, we call `originalStrutsAction.execute()`.

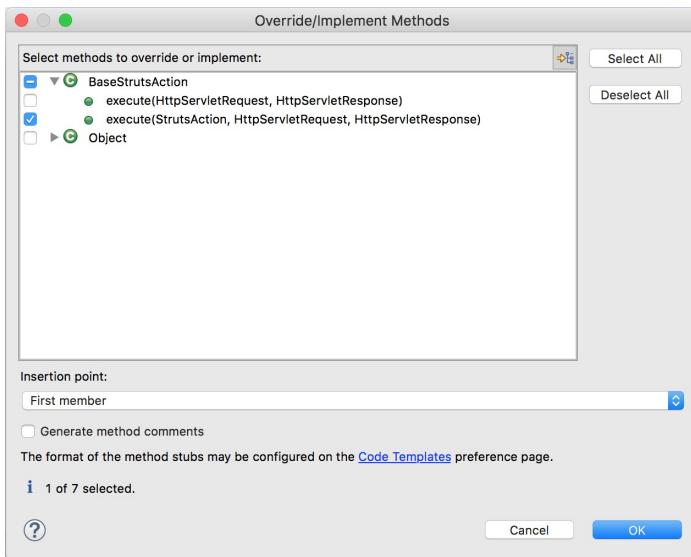
The customization we make here will redirect the user to '<https://university.liferay.com>' after they logout.

1. **Right-click** on the `LogoutActionOverride` class constructor to open a context menu.
2. **Choose** `Source → Override/Implement Methods`.



3. **Choose** the second `execute()` method.

4. Click Ok.



5. Implement the execute() method as specified below:

```
@Override
public String execute(
    StrutsAction originalStrutsAction, HttpServletRequest request,
    HttpServletResponse response)
throws Exception {

    System.out.println("Overriding the Portal Logout Action!");

    originalStrutsAction.execute(request, response);

    response.sendRedirect("https://university.liferay.com");

    return null;
}
```

6. Implement the @Component annotation from the following:

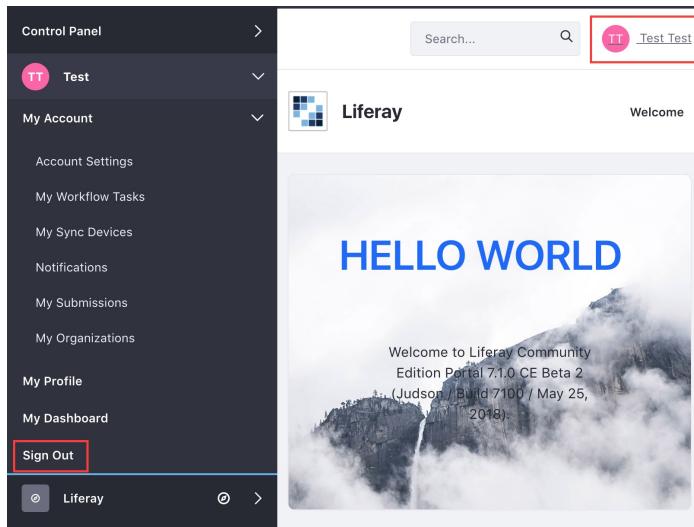
```
@Component(
    immediate = true,
    property = {
        "path=/portal/logout"
    },
    service = StrutsAction.class)
```

7. Press `CTRL+SHIFT+O` to resolve any missing imports.

8. Save the file.

Deploy and Test

1. **Go to the Gradle Tasks view in Liferay Developer Studio.**
2. **Double-click** the `deploy` task from `struts-action-override/build`.
 - o This will deploy `struts-action-override` to the Liferay Server.
3. **Open** your browser to <http://localhost:8080> and log in.
4. **Logout** of the platform from the *Menu*.



You should have been redirected to <https://university.liferay.com>.

The screenshot shows the Liferay University homepage. At the top, there's a navigation bar with the Liferay logo, a search bar, and a dropdown menu. Below the navigation, a large banner says 'Get Started with Liferay Essentials' with a 'Get a Liferay University Passport' button. To the right of the banner is a 3D illustration of a laptop displaying a dashboard with various data visualizations. Below the banner, there are three lesson cards: 'Gain Insight Into the Customer Journey - Liferay Analytics Cloud', 'Fully Customize Your Search Experience', and 'OSGI Basics'. Each card has a small icon, the word 'LESSON' above the title, and a 'Complete' badge in the top right corner. A 'Help' button is located at the bottom left of the card area.

Takeaways

We've shown you here how to modify the behavior of Liferay's Logout Action using a StrutsAction override. Using what you've learned here, you can modify other parts of Liferay that are still implemented in Struts.

Override MVC Commands

All Liferay platform applications are portlets. MVC commands are portlet lifecycle handlers that implement the `com.liferay.portal.kernel.portlet.bridges.mvc.MVCCmd` interface. They are used to break up the controller layer into smaller and more manageable code entities. There are three types of MVC commands that correspond to portlet lifecycle phases:

- MVC Action Commands
- MVC Render Commands
- MVC Resource Commands

MVC Action Commands

MVC action commands are OSGi components that handle a portlet's **action phase** requests. They are typically used to process form submissions and trigger a service action on the model layer. Here are a few example use cases:

- Add, update, or delete an item
- Subscribe to content
- Upload a document

In the example below, an MVC action command is used to handle the small image upload action in the Blogs portlet:

`edit_entry.jsp`

```
<portlet:actionURL name="/blogs/upload_small_image" var="uploadSmallImageURL" />
...
<div class="lfr-blogs-small-image-selector">

<%
String smallImageSelectedItemEventName = liferayPortletResponse.getNamespace()
    + "smallImageSelectedItem";
%>

<liferay-item-selector:image-selector
    fileEntryId="<% smallImageFileEntryId %>"
    itemSelectorEventName="<% smallImageSelectedItemEventName %>"
    itemSelectorURL="<% blogsItemSelectorHelper.getItemSelectorURL(requestBackedPortletURLFactory, the
meDisplay, smallImageSelectedItemEventName) %>"
    maxFileSize="<% PropsValues.BLOGS_IMAGE_MAX_SIZE %>"
    paramName="smallImageFileEntry"
    uploadURL="<% uploadSmallImageURL %>"
    validExtensions='<% StringUtil.merge(imageExtensions, ", ") %>'>
</liferay-item-selector:image-selector>
</div>
```

Source: https://github.com/liferay/liferay-portal/blob/7.1.0-ga1/modules/apps/blogs/blogs-web/src/main/resources/META-INF/resources/blogs/edit_entry.jsp

`UploadSmallImageMVCActionCommand`

```
@Component(
    immediate = true,
    property = {
        "javax.portlet.name=" + BlogsPortletKeys.BLOGS,
        "javax.portlet.name=" + BlogsPortletKeys.BLOGS_ADMIN,
        "mvc.command.name=/blogs/upload_small_image"
    },
)
```

```

        )
public class UploadSmallImageMVCActionCommand extends BaseMVCActionCommand {

    @Override
    protected void doProcessAction(
        ActionRequest actionRequest, ActionResponse actionResponse) throws Exception {
        _uploadHandler.upload(
            _tempImageBlogsUploadFileEntryHandler,
            _imageBlogsUploadResponseHandler, actionRequest, actionResponse);
    }
    ...
}

```

Source: <https://github.com/liferay/liferay-portal/blob/7.1.0-ga1/modules/apps/blogs/blogs-web/src/main/java/com/liferay/blogs/web/internal/portlet/action/UploadSmallImageMVCActionCommand.java>

MVC Render Commands

MVC render commands are OSGi components that handle a portlet's **render phase** requests. When using JSPs in the front-end, the render() method returns the path to the JSP file. A few example use cases:

- Get a list of assets to show in the user interface
- Fetch an asset entry to show in the user interface

Below is an example of an MVC render command in the Blogs application that handles showing an entry:

view.jsp

```

...
PortletURL portletURL = renderResponse.createRenderURL();
portletURL.setParameter("mvcRenderCommandName", "/blogs/view");
...

```

Source: <https://github.com/liferay/liferay-portal/blob/7.1.x/modules/apps/blogs/blogs-web/src/main/resources/META-INF/resources/blogs/view.jsp>

ViewEntryMVCRenderCommand

```

@Component(
    immediate = true,
    property = {
        "javax.portlet.name=" + BlogsPortletKeys.BLOGS,
        "javax.portlet.name=" + BlogsPortletKeys.BLOGS_ADMIN,
        "javax.portlet.name=" + BlogsPortletKeys.BLOGSAGGREGATOR,
        "mvc.command.name=/blogs/view_entry"
    },
    service = MVCRenderCommand.class
)
public class ViewEntryMVCRenderCommand implements MVCRenderCommand {

    @Override
    public String render(
        RenderRequest renderRequest, RenderResponse renderResponse)
        throws PortletException {

        long assetCategoryId = ParamUtil.getLong(renderRequest, "categoryId");
        String assetCategoryName = ParamUtil.getString(renderRequest, "tag");

        if ((assetCategoryId > 0) || Validator.isNotNull(assetCategoryName)) {
            return "/blogs/view.jsp";
        }
    }
}

```

```

        }

    try {
        BlogsEntry entry = ActionUtil.getEntry(renderRequest);
        ...
    }
}

```

Source: <https://github.com/liferay/liferay-portal/blob/7.1.x/modules/apps/blogs/blogs-web/src/main/java/com/liferay/blogs/web/internal/portlet/action/ViewEntryMVCRenderCommand.java>

MVC Resource Commands

MVC resource commands are OSGi components that handle a portlet's **resource serving phase** requests. The resource service phase doesn't trigger the render phase and page refresh, so resource commands are used, for example, for:

- Autocompletion
- Fetching an item into the user interface with an AJAX call
- Captcha checking
- Updating a list without page refresh

The example from a Liferay Microblogs application below shows how, in the editing view, there's an AJAX call doing the auto completion for user mentions:

edit_microblogs_entry.jsp

```

var createAutocomplete = function(contentTextarea) {
    AUI.$.ajax(
        '<liferay-portlet:resourceURL id="/microblogs/autocomplete_user_mentions" />',
        {
            data: {
                userId: <%= user.getUserId() %>;
            },
            success: function(responseData) {
                autocompleteDiv = new A.AutoComplete(
                    {
                        inputNode: contentTextarea,
                        maxResults: 5,
                        on: {
                            clear: function() {
                                var highlighterContent = A.one('#<portlet:namespace />highlighterContent<%' +
formId %>');
                                highlighterContent.html('');
                            },
                            ...
                        }
                    ).render();
            }
        );
    );
};

```

Source: https://github.com/liferay/liferay-portal/blob/7.1.x/modules/apps/microblogs/microblogs-web/src/main/resources/META-INF/resources/microblogs/edit_microblogs_entry.jsp

AutocompleteUserMentionsMVCResourceCommand

```

@Component(
    immediate = true,
    property = {
        "javax.portlet.name=" + MicroblogsPortletKeys.MICROBLOGS,
        ...
    }
)

```

```

        "mvc.command.name=/microblogs/autocomplete_user_mentions"
    },
    service = MVCResourceCommand.class
)
public class AutocompleteUserMentionsMVCResourceCommand
extends BaseMVCResourceCommand {

    @Override
    public void doServeResource(
        ResourceRequest resourceRequest, ResourceResponse resourceResponse)
    throws PortletException {

        try {
            HttpServletRequest request = _portal.getOriginalServletRequest(
                _portal.getHttpServletRequest(resourceRequest));

            long userId = ParamUtil.getLong(request, "userId");
            ...
        }
    }
}

```

Source: <https://github.com/liferay/liferay-portal/blob/master/modules/apps/microblogs/microblogs-web/src/main/java/com/liferay/microblogs/web/internal/portlet/action/AutocompleteUserMentionsMVCResourceCommand.java>

Overriding MVC Action Commands

Generally, the steps for overriding MVC action commands are as follows:

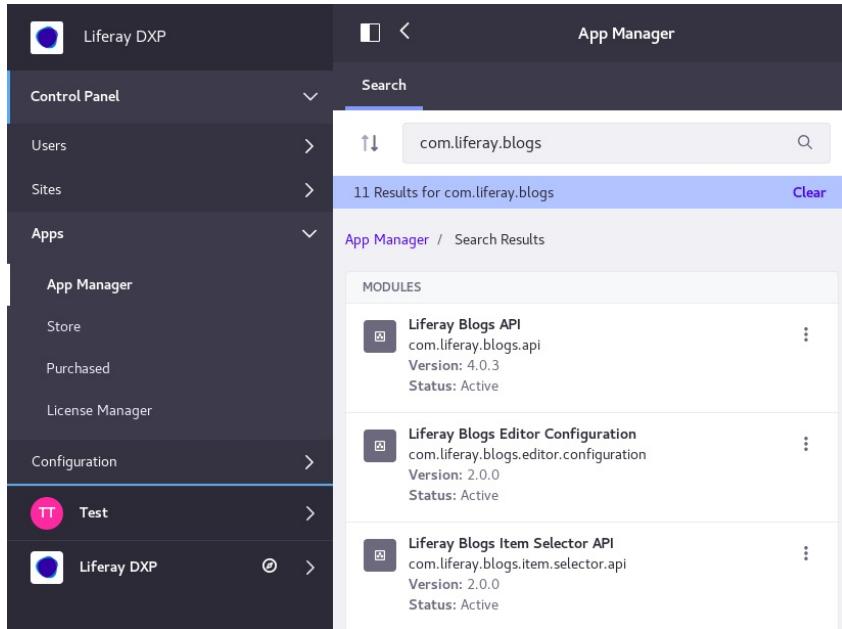
- ① Locate the portlet bundle.
- ② Find the *command* and *mvc.command* name to override.
- ③ Create an overriding OSGi MVC command service component with higher service priority.
- ④ Implement the override by implementing the action command interface or by extending the *BaseMVCActionCommand* superclass.
- ⑤ Deploy.

Example: Overriding the Blogs Admin Portlet Edit Entry Action Command

In the example below, we will customize the MVC action command responsible for editing blog entries. We'll add a notification when somebody is trying to delete an entry from the Control Panel. After the notification, the original action is executed.

Step 1 - Locate the Blogs Web Bundle Name

Use the App Manager to find the web bundle for blogs:



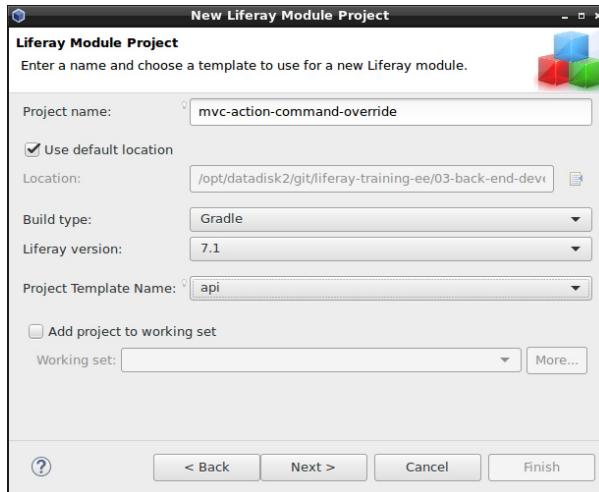
Step 2 - Find the Command and mvc.command Name to Override

[liferay-portal / modules / apps / blogs / blogs-web / src / main / java / com / liferay / blogs / web / internal / portlet / action /](#)

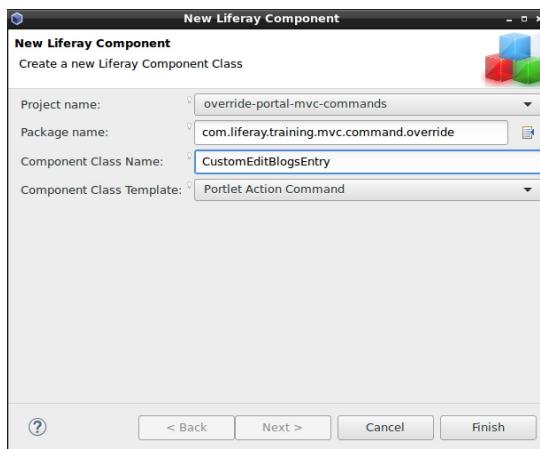
| petershin and brianchandotcom LPS-80332 Move projects to the apps folder | | |
|--|--|------------|
| Latest commit ed74f48 11 days ago | | |
| .. | | |
| ActionUtil.java | LPS-80332 Move projects to the apps folder | 8 days ago |
| BlogsAdminConfigurationAction.java | LPS-80332 Move projects to the apps folder | 8 days ago |
| BlogsAdminViewMVCRenderCommand.java | LPS-80332 Move projects to the apps folder | 8 days ago |
| BlogsAggregatorConfigurationAction.java | LPS-80332 Move projects to the apps folder | 8 days ago |
| BlogsAggregatorViewMVCRenderCommand.j... | LPS-80332 Move projects to the apps folder | 8 days ago |
| BlogsConfigurationAction.java | LPS-80332 Move projects to the apps folder | 8 days ago |
| BlogsFindEntryHelper.java | LPS-80332 Move projects to the apps folder | 8 days ago |
| BlogsViewMVCRenderCommand.java | LPS-80332 Move projects to the apps folder | 8 days ago |
| BlogsViewNotPublishedEntriesMVCRenderC... | LPS-80332 Move projects to the apps folder | 8 days ago |
| EditEntryMVCActionCommand.java | LPS-80332 Move projects to the apps folder | 8 days ago |
| EditEntryMVCRenderCommand.java | LPS-80332 Move projects to the apps folder | 8 days ago |
| EditImageMVCActionCommand.java | LPS-80332 Move projects to the apps folder | 8 days ago |
| FindEntryAction.java | LPS-80332 Move projects to the apps folder | 8 days ago |
| RSSAction.java | LPS-80332 Move projects to the apps folder | 8 days ago |
| TrackbackMVCActionCommand.java | LPS-80332 Move projects to the apps folder | 8 days ago |
| UploadCoverImageMVCActionCommand.java | LPS-80332 Move projects to the apps folder | 8 days ago |
| UploadImageMVCActionCommand.java | LPS-80332 Move projects to the apps folder | 8 days ago |
| UploadSmallImageMVCActionCommand.java | LPS-80332 Move projects to the apps folder | 8 days ago |
| UploadTempImageMVCActionCommand.java | LPS-80332 Move projects to the apps folder | 8 days ago |
| ViewEntryMVCRenderCommand.java | LPS-80332 Move projects to the apps folder | 8 days ago |

Step 3 - Create an Overriding OSGi MVC Command Service Component with a Higher Service Priority

First, create a Module using the API template:



Then, create an MVC action command using the new component wizard:



Step 4 - Implement the Override

```

@Component(
    immediate=true,
    property = {
        "javax.portlet.name=" + BlogsPortletKeys.BLOGS_ADMIN,
        "mvc.command.name=/blogs/edit_entry",
        "service.ranking=Integer=100"
    },
    service = MVCActionCommand.class
)
public class CustomEditBlogsEntryMVCActionCommand extends BaseMVCActionCommand {

    @Activate
    @Modified
    protected void activate(Map<String, Object> properties) {
        System.out.println("Module activated.");
    }

    @Override
    protected void doProcessAction
        (ActionRequest actionRequest, ActionResponse actionResponse)
        throws Exception {

        String cmd = ParamUtil.getString(actionRequest, Constants.CMD);

        if (cmd.equals(Constants.DELETE)) {
            System.out.println("Deleting a Blog Entry");
        }
    }
}

```

```
        }

        mvcActionCommand.processAction(actionRequest, actionResponse);
    }

    @Reference(
        target = "(component.name=com.liferay.blogs.web.internal.portlet.action.EditEntryMVCActionCommand)"
)
protected MVCActionCommand mvcActionCommand;
}
```

Exercises

Override the Documents and Media MVC Action Command

Introduction

In this exercise, we will be overriding the Documents and Media EditFolderMVCActionCommand class. The EditFolderMVCActionCommand is invoked whenever a folder is added, edited, or deleted.

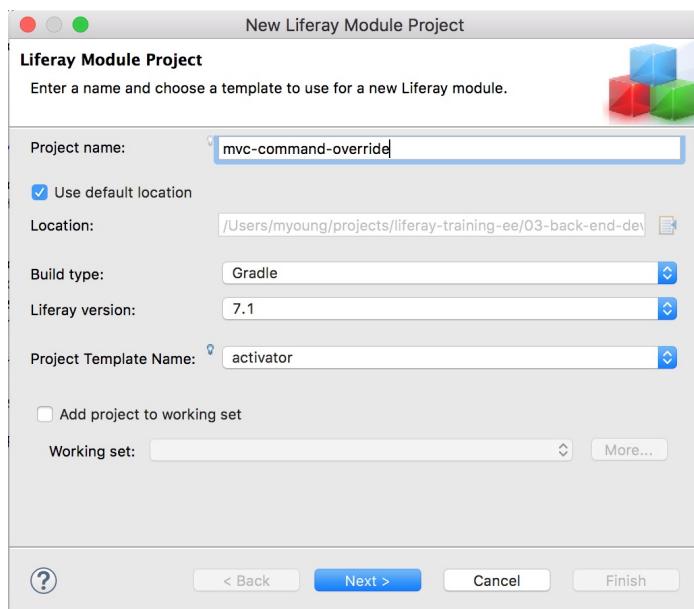
We will be using the `activator` project template as a starting point for this module since there doesn't exist a dedicated `mvc-command-override` project template.

Overview

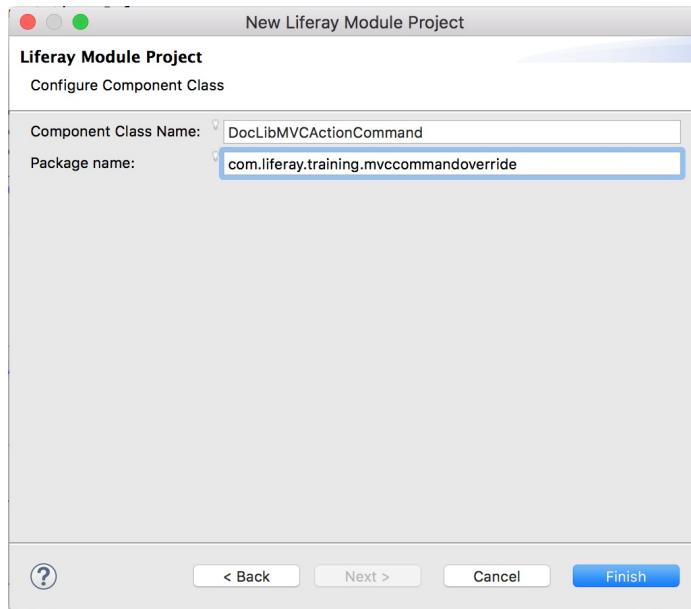
- ① Create a Liferay Module project using the `activator` template
- ② Resolve dependencies
- ③ Create the `DocLibMVCActionCommand` class
- ④ Implement the `doProcessAction()` method
- ⑤ Deploy and test

Create a Liferay Module Project Using the Activator Template

1. [Go to training-workspace](#).
2. [Right-click](#) to open a context menu.
3. [Choose New → Liferay Module Project](#).
4. [Type `mvc-command-override`](#) in the *Project Name* field.
5. [Choose activator](#) in the *Project Template Name* field.
6. [Click Next](#).



7. Type `DocLibMVCActionCommand` in the *Component Class Name* field.
8. Type `com.liferay.training.mvccommandoverride` in the *Package name* field.
9. Click **Finish**.



10. Expand the `mvc-command-override` project.
11. Delete the `DocLibMVCActionCommandActivator` class that was auto-created in the `com.liferay.training.mvccommandoverride` package.
 - o We will not be using this class in our customization.

Resolve Dependencies

Next, we will have to adjust the `bnd.bnd` and `build.gradle` files generated by Liferay Studio. Since we used the `activator` template, `build.gradle` will only contain the bare minimum of dependencies to set up an OSGi project. Because we intend to override a `MVCActionCommand` we have to depend on the `com.liferay.portal.kernel`, `portlet-api`, and `javax.servlet-api` packages. Additionally, we depend on the `com.liferay.document.library.api` package, because we interface with the Document Library. Since we implement an OSGi component, we also have to depend on the `org.osgi.service.component.annotations` package.

1. Open the `build.gradle` file.
2. Add the required dependencies. Afterwards the content of the file should look like as follows:

```
dependencies {
    compileOnly group: "com.liferay", name: "com.liferay.document.library.api", version: "3.3.0"
    compileOnly group: "com.liferay.portal", name: "com.liferay.portal.kernel", version: "3.0.0"
    compileOnly group: "javax.portlet", name: "portlet-api", version: "3.0.0"
    compileOnly group: "javax.servlet", name: "javax.servlet-api", version: "3.0.1"
    compileOnly group: "org.osgi", name: "org.osgi.core", version: "6.0.0"
    compileOnly group: "org.osgi", name: "org.osgi.service.component.annotations", version:"1.3.0"
}
```

3. Save the file.
4. Open the `bnd.bnd` file.
5. Click on the *Source* tab.
6. Replace the contents with the following:
 - o We are removing the Bundle-Activator that we deleted in an earlier step.
 - o We are also renaming the bundle. This step is not necessary, but it's helpful to use a more human-readable

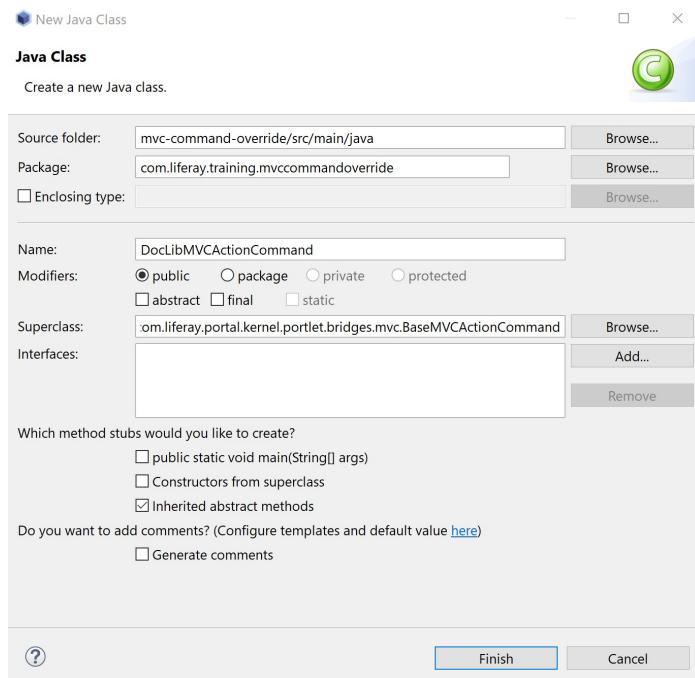
Bundle-Name in case you need to troubleshoot the bundle in the Gogo Shell, for example.

```
Bundle-Name: MVC Command Override
Bundle-SymbolicName: com.liferay.training.mvccommandoverride
Bundle-Version: 1.0.0
```

7. **Save** the file.

Create the DocLibMVCActionCommand Class

1. **Right-click** on the mvc-command-override project to open a context menu.
2. **Choose** New → Class.
3. **Type** DocLibMVCActionCommand in the Name field.
4. **Type** to add com.liferay.portal.kernel.portlet.bridges.mvc.BaseMVCActionCommand as the Superclass.
 - o The rest of the fields should be correct by default.
5. **Click** Finish.



Implement the doProcessAction() Method

Some important points to note here:

- To be sure we override the correct MVC command, we need to copy the `javax.portlet.name` and `mvc.command_name` properties from the command that we are overriding. This is found in Liferay's source code.
- We also need to set the `service.ranking` property to be sure that this Component replaces the original.
- `mvcActionCommand` is a reference to the original `EditFolderMVCActionCommand`.
- To invoke the original behavior of the `EditFolderMVCActionCommand`, which is often important, we call `mvcActionCommand.processAction()`

The customization we make here will print the `CMD` portlet parameter to the console. This is helpful information to have when you are figuring out where you want to override behavior.

1. **Add** a reference to the original `MVCActionCommand`.
 - o The `target` property allows us to select which command should be injected. `component.name` should always be the fully qualified class name of the original MVC Command to be overridden.

```

@Reference(
    target = "(component.name=com.liferay.document.library.web.internal.portlet.action.EditFolderMVC
ActionCommand)"
)
private MVCActionCommand _mvcActionCommand;

```

2. Implement the `doProcessAction()` method as specified below:

```

@Override
public void doProcessAction(ActionRequest actionRequest, ActionResponse actionResponse)
    throws PortletException {

    String cmd = ParamUtil.getString(actionRequest, Constants.CMD);

    System.out.println("CMD=" + cmd);

    _mvcActionCommand.processAction(actionRequest, actionResponse);
}

```

Note: Use `com.liferay.portal.kernel.util.Constants`.

3. Implement the `@Component` annotation using the following:

```

@Component(
    immediate=true,
    property = {
        "javax.portlet.name=" + DLPortletKeys.DOCUMENT_LIBRARY,
        "javax.portlet.name=" + DLPortletKeys.DOCUMENT_LIBRARY_ADMIN,
        "javax.portlet.name=" + DLPortletKeys.MEDIA_GALLERY_DISPLAY,
        "mvc.command.name=/document_library/edit_folder",
        "service.ranking:Integer=100"
    },
    service = MVCActionCommand.class
)

```

4. Press `CTRL+SHIFT+O` to resolve any missing imports.

5. Save the file.

Deploy and Test

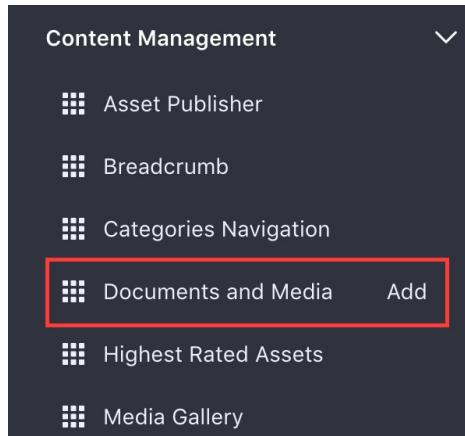
1. Go to the *Gradle Tasks* view in *Liferay Developer Studio*.
2. Double-click the `deploy` task from `mvc-command-override/build`.
 - o This will deploy `mvc-command-override` to the Liferay Server.

| | | |
|----|------------------------|-------------------------------|
| ▼ | 📄 mvc-command-override | mvc-command-override |
| ▼ | 📦 build | |
| ⚙️ | assemble | Assembles the outputs of th |
| ⚙️ | build | Assembles and tests this pr |
| ⚙️ | buildCSS | Build CSS files. |
| ⚙️ | buildDependents | Assembles and tests this pr |
| ⚙️ | buildLang | Runs Liferay Lang Builder to |
| ⚙️ | buildNeeded | Assembles and tests this pr |
| ⚙️ | buildSoy | Compiles Closure Templates |
| ⚙️ | classes | Assembles main classes. |
| ⚙️ | clean | Deletes the build directory. |
| ⚙️ | configJSModules | Generates the config file nee |
| ⚙️ | deploy | Assembles the project and d |

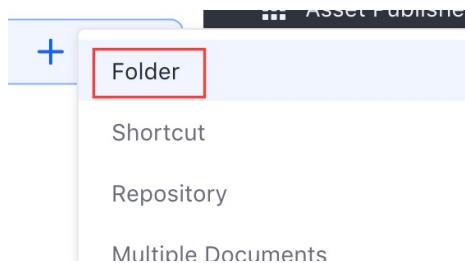
3. Open your browser to `http://localhost:8080` and log in.

4. Click the *Add* button in the top right corner.

5. Go to `Widgets → Content Management`.



6. Click `Add` next to the `Documents and Media` widget.
7. Click `Add` in the top bar of the `Documents and Media` application.
8. Choose `Folder`.



9. Type the name of a folder (for example, `images`), in the `Name` field.
10. Click `Save`.

A screenshot of a "New Folder" creation form. The "Name" field is filled with "images". The "Description" and "Permissions" fields are empty. At the bottom are "Save" and "Cancel" buttons.

| | |
|-------------|---------------|
| Name * | images |
| Description | |
| Permissions | |
| Save | Cancel |

You should see `CMD=add` in your console.

```
2018-06-09 01:22:56.784 INFO [Refresh Thread: Equinox Container: 303dd68b-716b-0f  
Module Activated.  
CMD=add]
```

Takeaways

MVC Commands are the foundation in much of Liferay's application code. After running through this exercise, you should be able to use the pattern here to customize other Liferay applications that are implemented using MVCPortlet and MVC Commands. The same pattern can be used to extend and override MVCRenderCommand, MVCActionCommand, and MVCResourceCommand.

Chapter 12: Catch Portal Events

Chapter Objectives

- Understand Portal Lifecycle and Model Events
- Learn How to Catch and Intercept the Events to Add Custom Logic
- Understand the Use Cases for Leveraging Portal Events



Catch Portal Lifecycle Events

Portal lifecycle events allow you to execute actions on:

- Portal startup and shutdown
- Servlet service call
- Login and logout
- Layout (page) update

Lifecycle event listeners are OSGi components that implement the `com.liferay.portal.kernel.events.LifecycleAction` interface. The event to catch is defined with the components `key` property and the execution order of the listener with the `service.ranking` property. The higher the number, the higher the priority in the execution order (default 0).

Below is an example of a global startup event listener. The `processLifecycleEvent()` gets executed once the portal starts and the component gets activated:

```

@Component(
    immediate = true,
    property = {
        "key=global.startup.events",
        "service.ranking=Integer=100"
    },
    service = LifecycleAction.class
)
public class PortalStartupListener implements LifecycleAction {

    @Override
    public void processLifecycleEvent(LifecycleEvent lifecycleEvent)
        throws ActionException {

        try {
            MailMessage message = new MailMessage();

            message.setBody("The portal was rebooted.");
            message.setTo(getToAddress());

            _mailService.sendEmail(message);

        }
        ...
    }
}

```

Startup / Shutdown Events

Startup events are called once when either the portal (global prefix) or a portal instance (application prefix) starts up or shuts down.

| Key property | Description |
|-----------------------------|--|
| global.startup.events | Run once when the portal initializes |
| global.shutdown.events | Run once when the portal shuts down |
| application.startup.events | Run once for every portal instance that is initialized |
| application.shutdown.events | Run once for every instance that is shut down |

Service Action Events

Service action events are processed before or after a request is processed. Note that service action events are executed on every request, including those for the static resources.

| Key property | Description |
|--------------------------------|--------------------------------|
| servlet.service.events.pre | Occurs before a service action |
| servlet.session.create.events | Occurs on session create |
| servlet.service.events.post | Occurs after a service action |
| servlet.service.destroy.events | Occurs on session destroy |

Login Events

Login events allow you to catch an event pre and post login and logout:

| Key property | Description |
|--------------------|--------------------|
| login.events.pre | Occurs pre-login |
| login.events.post | Occurs post-login |
| logout.events.pre | Occurs pre-logout |
| logout.events.post | Occurs post-logout |

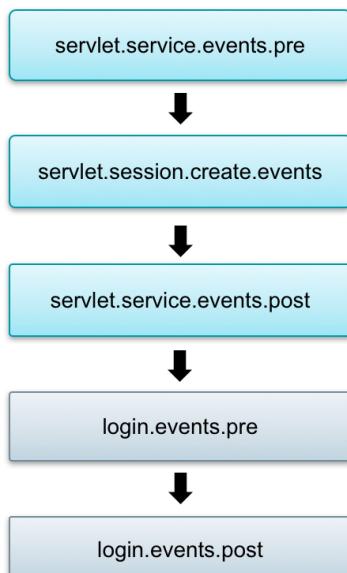
Layout Events

Layout events are triggered when a layout (page) is updated or deleted.

| Key property | Description |
|------------------------------------|-------------------------|
| layout.configuration.action.update | Occurs on layout update |
| layout.configuration.action.delete | Occurs on layout delete |

Event Sequence

Below is an example of the sequence in which the lifecycle event listeners are being executed on login:



For the list of default lifecycle listeners, see the portal.properties at <https://github.com/liferay/liferay-portal/blob/7.1.x/portal-impl/src/portal.properties>.

Steps for Creating a Portal Lifecycle Event Action

- ① Create a LifecycleAction service component implementing `LifecycleAction` interface
- ② Define the event with `key` component property
- ③ If multiple listeners, define the execution order with the `service.ranking` property
- ④ Implement the `processLifecycleEvent()` method

Module Lifecycle Events

OSGi components are activated by default when all references have been satisfied. Using a reference for the `ModuleServiceLifecycle` interface in a component allows you to define an event when a component should be activated and execute actions on the activation. The following events are supported:

- DATABASE_INITIALIZED
- PORTAL_INITIALIZED
- SPRING_INITIALIZED

Below is an example of a component with a `ModuleServiceLifecycle` reference, which gets injected on PORTAL_INITIALIZED, triggering the component activation:

```
package com.liferay.training.events.lifecycle.module;

import com.liferay.portal.kernel.module.framework.ModuleServiceLifecycle;

import java.util.Map;

import org.osgi.service.component.annotations.Activate;
import org.osgi.service.component.annotations.Component;
import org.osgi.service.component.annotations.Reference;

@Component
public class TrainingImpl implements TrainingApi {

    @Activate
    protected void activate(Map<Object, Object> properties) {

        // Components activation method will be run
        // when all the referenced services are initialized
        // Thus, this code block gets executed after PORTAL_INITIALIZED

    }

    @Reference(target = ModuleServiceLifecycle.PORTAL_INITIALIZED)
    private ModuleServiceLifecycle _portalInitialized;
}
```

Links and References

- Waiting on Module Lifecycle Events

https://dev.liferay.com/develop/tutorials/-/knowledge_base/7-1/waiting-on-lifecycle-events

Exercises

Creating a Post Login Event Listener

Introduction

This exercise will go through how to listen for specific events that occur in Liferay, in this case, the `login.events.post`. When listening to an event, we are able to execute our code either before or after the event is finished. In our example, we'll listen to the login event, and after the User has successfully logged in, we will send an email notification.

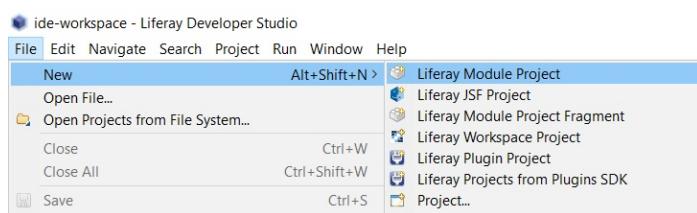
Since we want to send an email, we have to add a dependency on the `mail` package to our `build.gradle` dependencies.

Overview

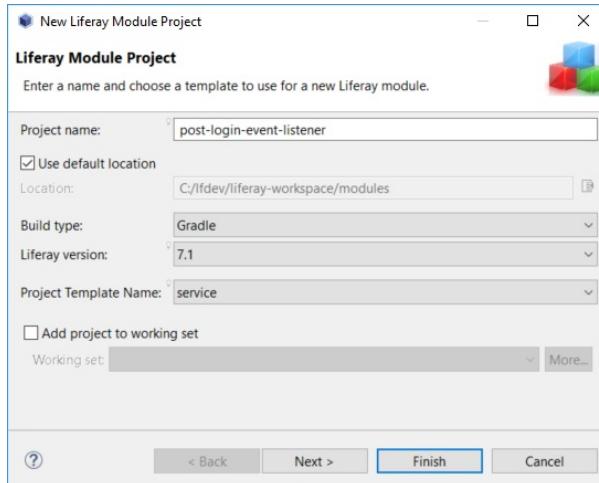
- ① Create a Liferay Module Project
- ② Create and Configure the Component
- ③ Resolve dependencies
- ④ Add required service references
- ⑤ Implement `processLifecycleEvent()`
- ⑥ Deploy and test

Create a Liferay Module Project

1. Click `File → New → Liferay Module Project`.

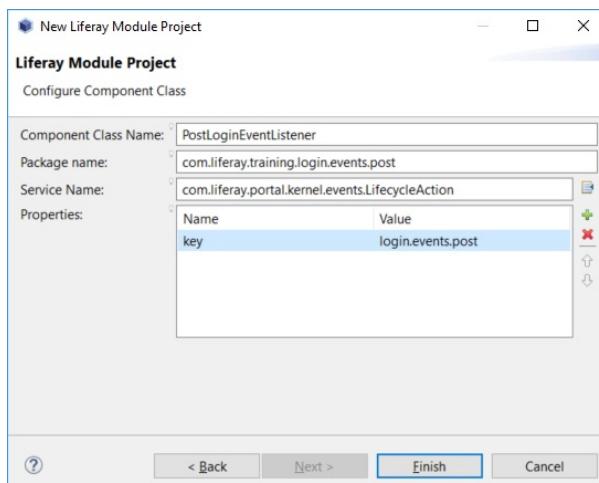


2. Type `post-login-event-listener` in the *Project name*.
3. Choose 7.1 for the Liferay version if it isn't already selected.
4. Choose the service template.
5. Click `Next`.



Create and Configure the Component

1. **Type** `PostLoginEventListener` in the *Component Class Name*.
2. **Type** `com.liferay.training.login.events.post` for the *Package Name*.
3. **Click** on the browse button next to the *Service Name* field.
4. **Type** `*.LifecycleAction` in the *Select Service Name*.
5. **Choose** `com.liferay.portal.kernel.events.LifecycleAction`.
6. **Click** the green plus button to add a property
7. **Type** `key` for the *Name*.
8. **Type** `login.events.post` for the *Value*.
9. **Press** the Enter/Return Key to confirm your property.
10. **Click** *Finish*.



Resolve Dependencies

1. **Double-click** on `build.gradle` for your post-login-event-listener project.
2. **Add** the following dependency in `build.gradle`.

```
compileOnly group: 'javax.mail', name: 'mail', version: '1.4'
```

3. **Save** the file.

Implement `processLifecycleEvent`

1. **Expand** the `post-login-event-listener` project.
2. **Expand** `src/main/java`.
3. **Expand** the `com.liferay.training.login.events.post` package.
4. **Double-click** the `PostLoginEventListener.java` class.
5. **Move** your mouse cursor to hover over `PostLoginEventListener`.
6. **Click** on *Add unimplemented methods* from the quick fix pop-up.

```

10 @Component(
11     immediate = true,
12     property = {"key=login.events.post"
13         // TODO enter required service properties
14     },
15     service = LifecycleAction.class
16 )
17 public class PostLoginEventListener implements LifecycleAction {
18     // The type PostLoginEventListener must implement the inherited abstract method
19     // LifecycleAction.processLifecycleEvent(LifecycleEvent)
20     // 2 quick fixes available:
21     // Add unimplemented methods
22     // Make type 'PostLoginEventListener' abstract
23 }

```

7. **Add** the following code to implement `processLifecycleEvent()`

```

try {
    User user = _userService.getCurrentUser();

    MailMessage message = new MailMessage();

    message.setSubject("Security Alert");
    message.setBody("Liferay has detected that you logged in at " + user.getLastLoginDate());

    InternetAddress toAddress = new InternetAddress(user.getEmailAddress());
    InternetAddress fromAddress = new InternetAddress("do-not-reply@liferay.com");

    message.setTo(toAddress);
    message.setFrom(fromAddress);

    _mailService.sendEmail(message);

} catch (PortalException e) {
    _log.error(e, e);
} catch (AddressException e) {

    _log.error(e, e);
}

```

Add Required Service References

1. **Add** the following to the bottom of the `PostLoginEventListener` class to initialize and retrieve a reference of the service objects.

```

private static final Log _log = LogFactoryUtil.getLog(PostLoginEventListener.class);

@Reference
protected UserService _userService;

@Reference
protected MailService _mailService;

```

2. **Press** `CTRL+SHIFT+O` to resolve any missing imports.
3. **Save** the file.

Deploy and Test

1. **Go to** the `solutions/solutions-chapter-12/exercise-files` folder provided in the *training-workspace*.
2. **Run** the `fakeSMTP-[version].jar`.
3. **Click Start Server** to start the server using the default port `25`.
4. **Run** the `deploy` task for the *post-login-event-listener*, found in the `post-login-event-listener/build` folder in the *Gradle Tasks* menu.
5. **Go to** <http://localhost:8080/> in your browser.
6. **Sign in** with any User.
7. **Check** the FakeSMTP for an email.

Takeaways

Although a simple example, listening for other events is just as easy. The only difference between this event and the others is the value of the key in the property element of the component. The rest is all in the implementation of the `processLifecycleEvent()`.

Intercept Events with Model Listeners

Model listeners listen to persistence events on the models, and their associations allow you to create actions on them. They are available for both core model entities as well as custom, Service Builder-generated entities.

Model listeners are OSGi components that implement the `com.liferay.portal.kernel.model.ModelListener` interface and extend the `com.liferay.portal.kernel.model.BaseModelListener`.

Events available are:

- `onBeforeAddAssociation()`
- `onBeforeCreate()`
- `onBeforeRemove()`
- `onBeforeRemoveAssociation() onBeforeUpdate()`
- `onAfterAddAssociation()`
- `onAfterCreate()`
- `onAfterRemove()`
- `onAfterRemoveAssociation() onAfterUpdate()`

The association-named events are triggered on model associations. For example, a user joining a group triggers `onBeforeAddAssociation()` before and `onAfterAddAssociation()` after the joining.

You can use model listeners for implementation. They're easy and fast to implement, but there are also some restrictions. First, model listeners are called before a database transaction is complete. This means that you might end up triggering something in the model listener and the transaction could still fail. Of the same reason, modifying an entity in the model listener could produce an inconsistent model state.

Another restriction is that the order in which model listeners are invoked, can not be controlled. That's why model listeners should always be independent of each other.

Being aware of these restrictions model listeners are usually good for example for:

- Notification of model persistence events
- Clearing a custom cache on entity update

Below is an example of a model listener:

```

@Component(
    immediate = true,
    service = ModelListener.class
)
public class UserModelListener extends BaseModelListener<User> {

    @Override
    public void onAfterAddAssociation(
        Object classPK, String associationClassName, Object associationClassPK)
        throws ModelListenerException {

        try {

            User user =
                _userService.getUserById(Long.parseLong(classPK.toString()));

            if (associationClassName.equalsIgnoreCase(
                "com.liferay.portal.kernel.model.Role")) {

                Role role = _roleService.getRole(
                    Long.parseLong(associationClassPK.toString()));
            }
        }
    }
}

```

```
MailMessage message = new MailMessage();

StringBundler sb = new StringBundler();
sb.append("User ").append(user.getScreenName()).append(
    " was added to role").append(role.getName());

message.setBody(sb.toString());

...
```

Generally speaking, the steps for creating a model listener are as follows:

- ① Create a new Liferay module using the *api* template
- ② Create a model listener component using the Liferay Component Class wizard
- ③ Create a component extending the `BaseModelListener` class
- ④ Implement the event handler method(s)

Exercises

Creating a User Model Listener

Introduction

This exercise highlights one of the many different Events we are able to listen for when an action on a Model Entity is executed in the persistence layer. There are a few use cases for Model Listeners such as an Audit Listener, Cache Clearing Listener, and Validation Listener. The use case we are implementing is the fourth use case, the Entity Update Listener.

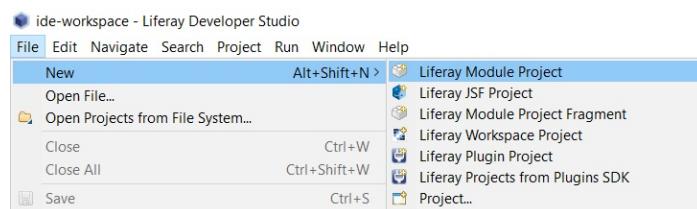
Since there is no dedicated model-listener template, we use the *api* template as the starting point for our project.

Overview

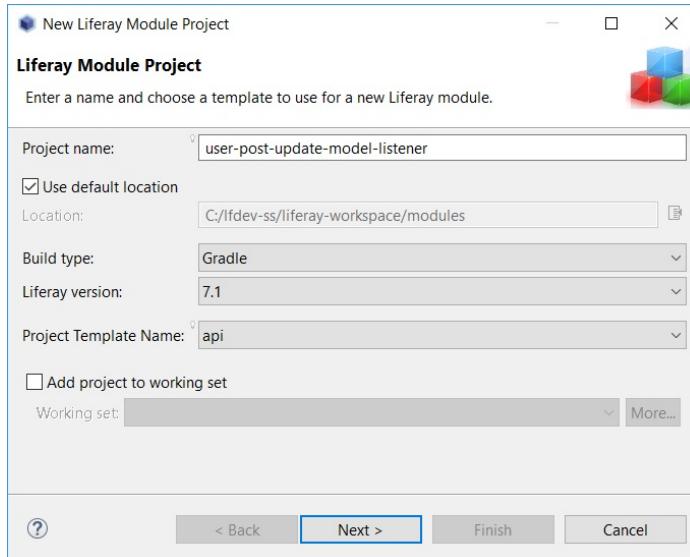
- ① Create a Liferay Module
- ② Create and configure the component
- ③ Resolve dependencies
- ④ Add service references
- ⑤ Override the method(s) for a specific Model Persistence Event
- ⑥ Deploy and test

Create a Liferay Module

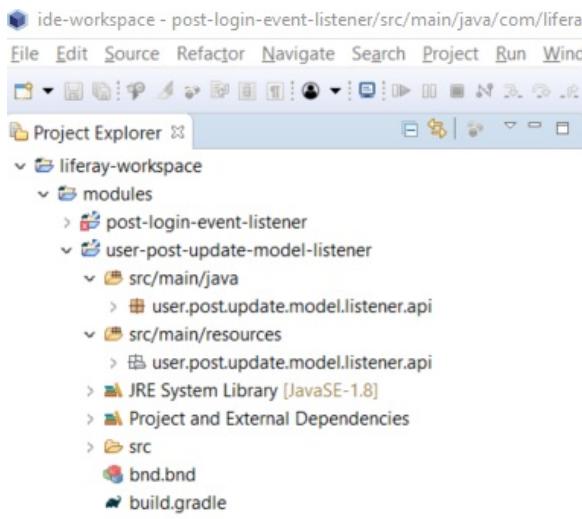
1. Click *File* → *New* → *Liferay Module Project*.



2. Type *user-post-update-model-listener* in the *Project name*.
3. Check that Gradle and 7.1 are selected for the *Build* type and *Liferay* version.
4. Choose the *api* template.
5. Click *Finish*, leaving the default values.



6. **Expand** the `src/main/java` folder for the `user-post-update-model-listener` project.
7. **Delete** the `user.post.update.model.listener.api` package.
8. **Expand** `src/main/resources`.
9. **Delete** the `user.post.update.model.listener.api` package.

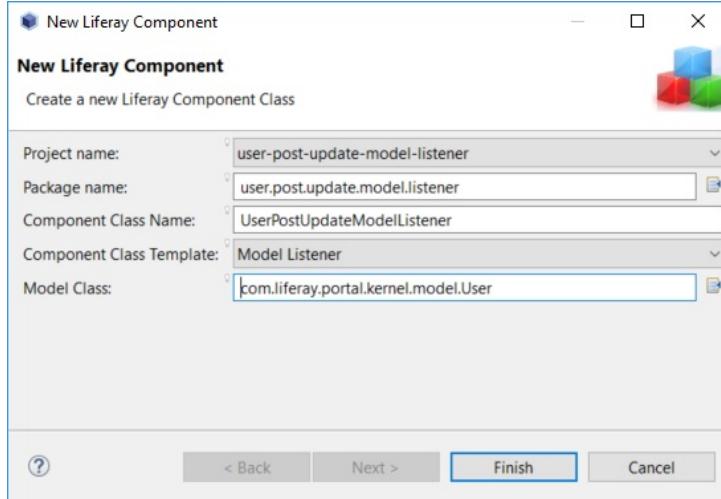


10. **Double-click** `bnd.bnd`.
11. **Click** the `Source` tab.
12. **Delete** `user.post.update.model.listener.api` as an exported package.
13. **Save** the file.

Create and Configure the Component

1. **Right-click** the `user-post-update-model-listener` project.
2. **Choose** `New → Liferay Component Class`.
3. **Type** `user.post.update.model.listener` for the `Package name`.
4. **Type** `UserPostUpdateModelListener` for the `Component Class Name`.
5. **Select** Model Listener for the `Component Class Template`.
6. **Click** on the browse button that corresponds with the `Model Class` field.
7. **Type** `*.User`.

8. Choose `com.liferay.portal.kernel.model.User`.
9. Click **Finish**.



Resolve Dependencies

1. Double-click on `build.gradle`,
2. Add the dependencies for the `com.liferay.portal.kernel` and `mail` packages - afterwards the contents of `build.gradle` should look as follows:

```
dependencies {
    compileOnly group: "com.liferay.portal", name:"com.liferay.portal.kernel", version:"3.0.0"
    compileOnly group: 'javax.mail', name: 'mail', version: '1.4'
    compileOnly group: "org.osgi", name: "org.osgi.core", version: "6.0.0"
    compileOnly group: "org.osgi", name: "org.osgi.service.component.annotations", version: "1.3.0"
}
```

3. Save the file.
4. Refresh the `user-post-update-model-listener` project using the Gradle refresh to pick up your newly-added dependency.

Override the Method for a Specific Persistence Event

1. Expand the `user.post.update.model.listener` package in the project.
2. Double-click the `UserPostUpdateModelListener.java` class.
3. Delete the `onBeforeCreate()` method.
4. Add the following code to the `UserPostUpdateModelListener` class:

```
@Override
public void onAfterUpdate(User model) throws ModelListenerException {
    try {
        MailMessage message = new MailMessage();

        message.setSubject("Security Alert: Account Settings");
        message.setBody("Liferay has detected that your account settings has been changed.");

        InternetAddress toAddress = new InternetAddress(model.getEmailAddress());
        InternetAddress fromAddress = new InternetAddress("do-not-reply@liferay.com");

        message.setTo(toAddress);
        message.setFrom(fromAddress);

        _mailService.sendEmail(message);
    }
}
```

```
        } catch (AddressException e) {
            _log.error(e, e);
        }
    }
```

5. Press **CTRL+SHIFT+O** to resolve any missing imports.

Add Service References

1. Add the following code at the bottom of the class.

```
private static final Log _log = LogFactoryUtil.getLog(UserPostUpdateModelListener.class);

@Reference
protected MailService _mailService;
```

2. Press **CTRL+SHIFT+O** to resolve any missing imports.
3. Save the file.

Deploy and Test

1. Start the FakeSMTP.jar if needed.
2. Run the `deploy` task for the `user-post-update-model-listener`, found in the `user-post-update-model-listener/build` folder in the *Gradle Tasks* menu.
3. Sign in as any User.
4. Open the Menu.
5. Go to User Panel → My Account → Account Settings.
6. Edit any setting for the User.
7. Click Save.
8. Check the FakeSMTP server for an email.

Takeaways

Creating a Model Listener allows you to do as the name tells us, listen to Model Entities and react to changes that occur to them. Implementing a Model Listener boils down to which model entity you want to listen to and what action you want to react to once it's performed on the model entity.

Chapter 13: Leverage the Liferay Message Bus

Chapter Objectives

- Learn the Concepts of Liferay Message Bus (LMB)
- Learn the Use Cases for Liferay Message Bus in Your Custom Applications
- Understand How the Liferay Message Bus is Used Internally



Introducing the Liferay Message Bus

Liferay Message Bus (LMB) is a service-level API for exchanging messages inside Liferay. It is similar to JMS (Java Message Service) but has a smaller feature set, with support for synchronous and asynchronous messaging also in the cluster but lacking, for example, transactional and reliable delivery (acknowledgements).

What does Liferay use the message bus for? The message bus is used for many background and asynchronous processes like:

- Auditing
- Search engine integration (like sending search index write events)
- Running asynchronous background processes
- Running scheduler tasks
- Running cluster operations like cache replication
- Document library processing
- Sending subscription emails
- Monitoring
- Running liferay/hot_deploy

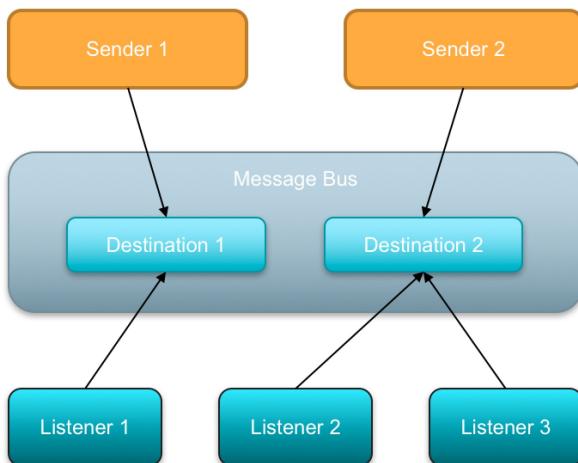
The Message Bus API is available for custom applications as well and is the recommended approach for any time-consuming operation that should not block request processing like, for example:

- Sending bulk emails
- Running scheduled tasks
- File processing
- Cluster communication
- Auditing
- Sending payload to an integrated system

Message Bus Components

The message bus has three main components:

- **Destinations:** Addresses or endpoints to which listeners register to receive messages
- **Senders:** Invoke the Message Bus to send messages to destinations
- **Listeners:** Receive messages sent to their registered destinations



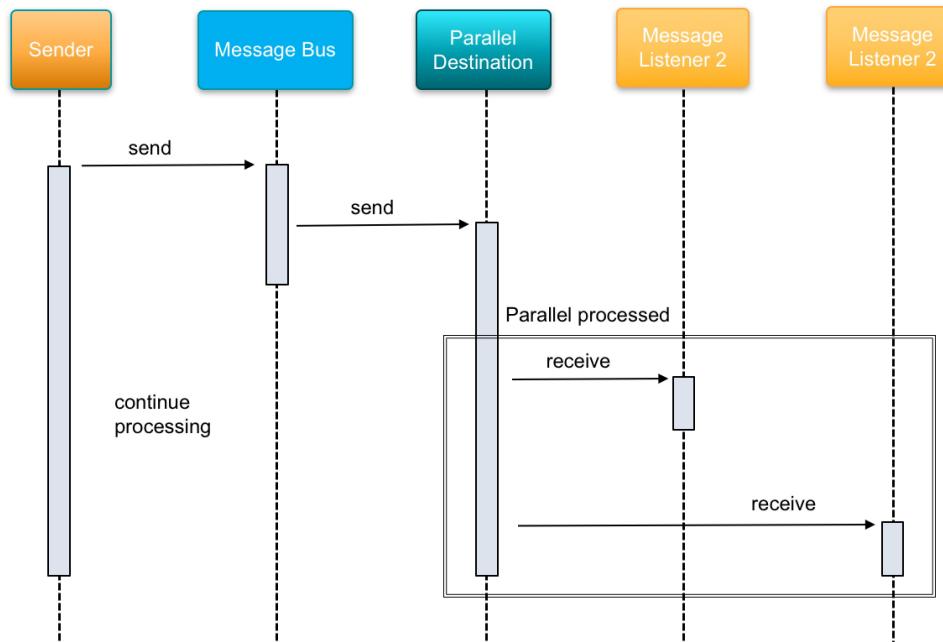
Destinations

A destination is a named endpoint for sending and receiving messages. By allowing any sender or listener to register, it provides a loose coupling between them.

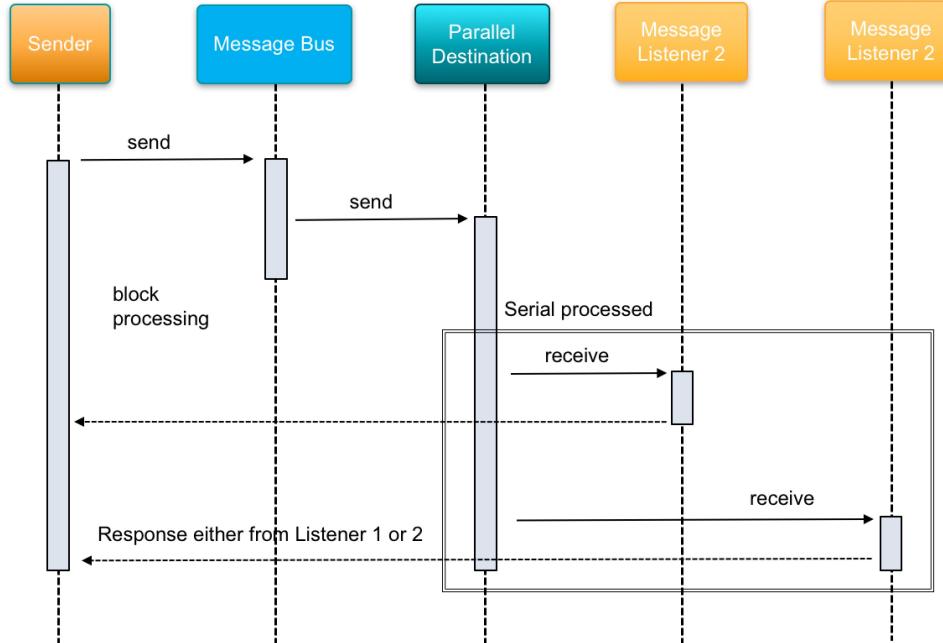
There are three destination types:

- **Parallel:**
 - Received messages are queued
 - One worker thread per message per message listener
- **Serial:**
 - Received messages are queued
 - One worker thread per message
- **Synchronous:**
 - No queue
 - In this case the thread sending the message delivers the message to all message listeners

The diagram below illustrates the processing flow when using asynchronous messaging with a parallel destination:



The diagram below illustrates the processing flow when using synchronous messaging with a serial destination:



Below is an example of a component doing a destination registration and configuration on component activation:

```

@Component(
    immediate = true
)
public class MessageBusDestinationRegistrar {

    @Activate
    protected void activate(ComponentContext componentContext) {
        _bundleContext = componentContext.getBundleContext();

        _log.info(
            "Registering message bus listener for " +
            TrainingDestinationNames.TRAINING_DESTINATION);

        register(
            DestinationConfiguration.DESTINATION_TYPE_PARALLEL,
            TrainingDestinationNames.TRAINING_DESTINATION, null, true);
    }

    /**
     * Register the endpoint.
     *
     * @param destinationType
     * @param destinationName
     * @param destinationPropertyName
     * @param destinationPropertyValue
     */
    protected void register(
        String destinationType, String destinationName,
        String destinationPropertyName, Object destinationPropertyValue) {
        DestinationConfiguration destinationConfiguration =
            new DestinationConfiguration(destinationType, destinationName);

        destinationConfiguration.setMaximumQueueSize(5);
        destinationConfiguration.setRejectedExecutionHandler(
            new CallerRunsPolicy() {

                @Override
                public void rejectedExecution(

```

```

        Runnable runnable, ThreadPoolExecutor threadPoolExecutor) {

    if (_log.isWarnEnabled()) {
        _log.warn(
            "The current thread will handle the request " +
            "because the graph walker's task queue is at " +
            "its maximum capacity");
    }

    super.rejectedExecution(runnable, threadPoolExecutor);
}

});

Destination destination =
    _destinationFactory.createDestination(destinationConfiguration);

Dictionary<String, Object> properties = new HashMapDictionary<>();

properties.put(
    "destination.name", destinationConfiguration.getDestinationName());

if (destinationPropertyName != null) {
    properties.put(destinationPropertyName, destinationPropertyValue);
}

ServiceRegistration<Destination> serviceRegistration =
    _bundleContext.registerService(
        Destination.class, destination, properties);

_serviceRegistrations.put(destination.getName(), serviceRegistration);
}

@Override
protected void deactivate() {

    for (ServiceRegistration<Destination> serviceRegistration : _serviceRegistrations.values()) {

        Destination destination =
            _bundleContext.getService(serviceRegistration.getReference());

        serviceRegistration.unregister();

        destination.destroy();
    }

    _serviceRegistrations.clear();
}

@Modified
protected void modified(ComponentContext componentContext) {

    deactivate();

    activate(componentContext);
}

private static final Log _log =
    LogFactoryUtil.getLog(MessageBusDestinationRegistrar.class);

private volatile BundleContext _bundleContext;

@Reference
private DestinationFactory _destinationFactory;

private final Map<String, ServiceRegistration<Destination>> _serviceRegistrations =
    new HashMap<>();

}

```

Senders

Sending a message can be done from any class. Messages can be sent:

- **Directly** to the message bus
- **Asynchronously** using a SingleDestinationMessageSender
- **Synchronously** using a SynchronousMessageSender

Below is an example of a method sending a message **directly** to the message bus:

```
protected void sendDirectMessage(String messageText) {

    Message message = new Message();

    message.setDestinationName(
        TrainingDestinationNames.TRAINING_DESTINATION);
    message.setPayload(messageText);
    message.setResponseDestinationName(
        TrainingDestinationNames.TRAINING_RESPONSE_DESTINATION);
    message.setResponseId("abcd");

    _messageBus.sendMessage(message.getDestinationName(), message);
}

@Reference
private MessageBus _messageBus;
```

In **asynchronous** sending, after the message is sent (in a different thread), the sender is free to continue processing. By setting messages to a response destination, a callback can be provided. Below is an example of sending an asynchronous message:

```
protected void sendAsyncMessage(String messageText) {

    Message message = new Message();

    message.setDestinationName(
        TrainingDestinationNames.TRAINING_DESTINATION);
    message.setPayload(messageText);
    message.setResponseDestinationName(
        TrainingDestinationNames.TRAINING_RESPONSE_DESTINATION);
    message.setResponseId("abcd");

    _log.info("Sending async message: " + messageText);

    SingleDestinationMessageSender messageSender =
        _messageSenderFactory.createSingleDestinationMessageSender(
            TrainingDestinationNames.TRAINING_DESTINATION);

    messageSender.send(message);
}
```

Synchronous Sending blocks the thread until it receives a response or the response times out. Two operation modes are available:

- **DEFAULT:** Delivers the message in a separate thread with timeout
- **DIRECT:** Delivers the message in the same thread of execution and blocks until it receives a response

Generally, as synchronous messaging can block threads, it should be used only if the delivery order has to be guaranteed.

Below is an example of sending a message synchronously:

```

protected void sendSyncMessage(String messageText)
    throws Exception {

    Message message = new Message();

    message.setPayload(messageText);
    message.setResponseDestinationName(
        TrainingDestinationNames.TRAINING_RESPONSE_DESTINATION);
    message.setResponseId("abcd");

    SingleDestinationSynchronousMessageSender messageSender =
        _messageSenderFactory.createSingleDestinationSynchronousMessageSender(
            TrainingDestinationNames.TRAINING_DESTINATION,
            SynchronousMessageSender.Mode.DIRECT);

    Object response = messageSender.send(message, 10000);
}

```

Listeners

In order to receive messages, a listener has to register to a destination. Registration to an endpoint can be done in three ways:

- **Automatic Registration as a Component:** Publish the listener to the OSGi registry as a Declarative Services Component that specifies a destination. Message Bus automatically wires the listener to the destination.
- **Registering via MessageBus:** Obtain a reference to the Message Bus and use it directly to register the listener to a destination.
- **Registering directly to a Destination:** Obtain a reference to a specific destination and use it directly to register the listener with that destination.

Below is an example of automatic registration as a component:

```

@Component(
    immediate = true,
    property = {
        "destination.name=" + TrainingDestinationNames.TRAINING_DESTINATION
    },
    service = MessageListener.class
)
public class MessageBusListener implements MessageListener {

    @Override
    public void receive(Message message)
        throws MessageListenerException {

        String payload = (String) message.getPayload();

        String responseDestinationName = message.getResponseDestinationName();

        if (Validator.isNotNull(responseDestinationName)) {

            String responsePayload = "Response to " + payload;

            Message responseMessage = new Message();

            responseMessage.setDestinationName(responseDestinationName);
            responseMessage.setPayload(responsePayload);
            responseMessage.setResponseId(message.getResponseId());

            _messageBus.sendMessage(
                message.getResponseDestinationName(), responseMessage);
        }
    }
}

```

In order to receive cluster messages, a ClusterBridgeMessageListener service component has to be registered in the destination. Below is an example:

```

@Component(
    immediate = true,
    service = MessageBusClusterListener.class
)
public class MessageBusClusterListener {

    @Activate
    protected void activate() {

        _clusterBridgeMessageListener = new ClusterBridgeMessageListener();
        _destination.register(_clusterBridgeMessageListener);
    }

    @Deactivate
    protected void deactivate() {

        _destination.unregister(_clusterBridgeMessageListener );
    }

    @Reference(target = "(destination.name=" + TrainingDestinationNames.TRAINING_DESTINATION + ")")
    private Destination _destination;

    private MessageListener _clusterBridgeMessageListener;
}

```

Message Bus and Service Builder

Liferay Service Builder can leverage the message bus with two annotations: `@Async` and `@Clusterable`.

`@Async`

If any public service method is annotated with `@Async`, then the message bus calls to the method will result in the method being converted to an asynchronous method call. This allows you to implement fire and forget capabilities in your services. This option is especially useful for features like notifications.

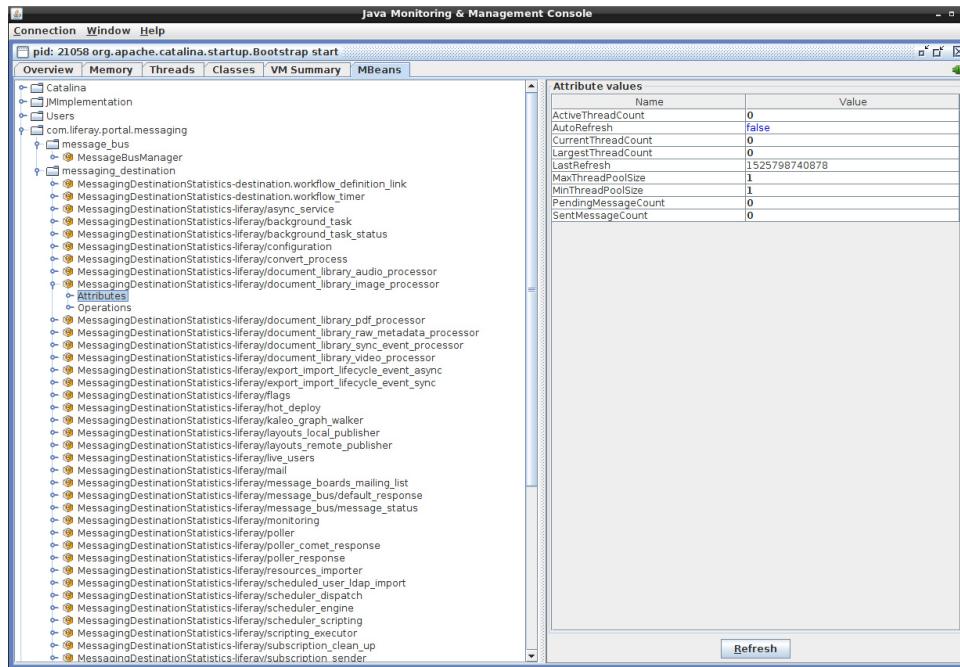
`@Clusterable`

Any service method annotated with `@Clusterable` will be invoked across the cluster. The `@Clusterable` annotation has two attributes:

- **onMaster**: if set to true, it will only execute the request if the current portal JVM is holding the cluster wide “master” token
- **acceptor**: specifies a custom ClusterInvokeAcceptor to determine whether a given portal JVM should accept and execute the request

Message Bus Statistics

Message bus destinations are available as MBeans and can be monitored with any JMX tool like JConsole:



Exercises

Create a Documents and Media Message Bus Listener

Introduction

In this exercise, we will be looking at how we can customize image preview generation in the *Documents and Media* application.

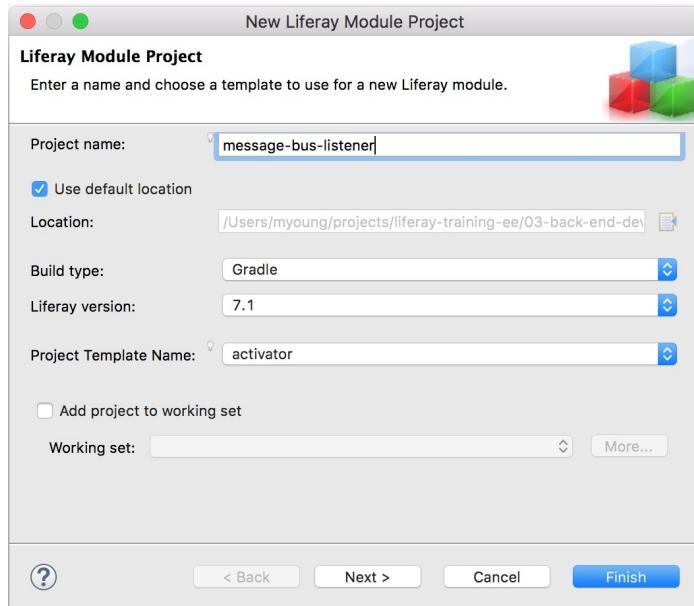
When a file is added to the *Documents and Media* application, you'll see that there is an image preview generated for that file. The *Liferay Message Bus* is used to kick off an asynchronous process to generate these images. Each file type has an image preview generator for its particular type. We are going to create a message bus listener to extend the PDF image generation process. We'll be using the *Automatic Registration as a Component* method described earlier in this chapter.

Overview

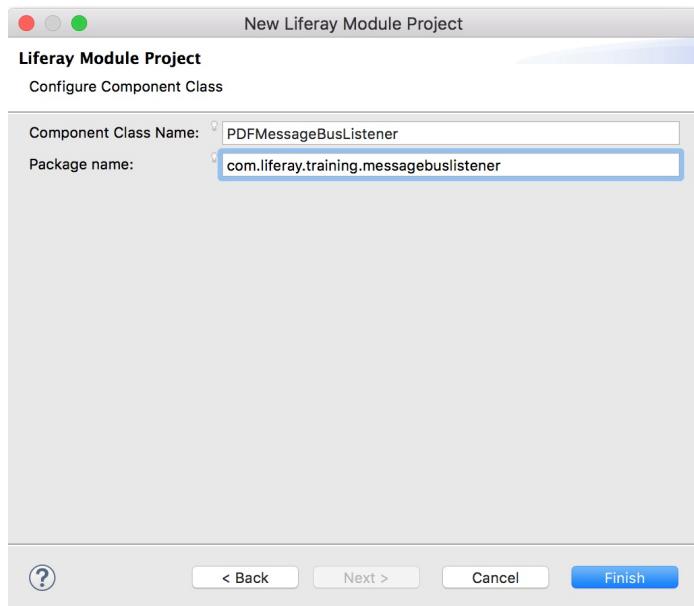
- ① Create a Liferay Module project using the *activator* template
- ② Resolve Dependencies
- ③ Create the `PDFMessageListener` class
- ④ Implement the `receive()` method
- ⑤ Deploy and test

Create a Liferay Module Project Using the Activator Template

1. [Go to *training-workspace*.](#)
2. [Right-click](#) the workspace to open a context menu.
3. [Choose *New* → *Liferay Module Project*.](#)
4. [Type *message-bus-listener* in the *Project Name* field.](#)
5. [Choose *activator* in the *Project Template Name* field.](#)
6. [Click *Next*.](#)



7. Type `PDFMessageBusListener` in the *Component Class Name* field.
8. Type `com.liferay.training.messagebuslistener` in the *Package name* field.
9. Click **Finish**.



10. Expand the new *message-bus-listener* module project.
11. Delete the `PDFMessageBusListnerActivator` class that was auto-created in the `com.liferay.training.messagebuslistener` package.
 - o We will not be using this class in our customization.

Resolve Dependencies

Next, we'll have to adjust the `bnd.bnd` and `build.gradle` files generated by Liferay Developer Studio. Since we used the *activator* template, `build.gradle` will only contain the bare minimum of dependencies to set up an OSGi project. Since we plan to implement a `com.liferay.portal.kernel.messaging.MessageListener` we have to depend on the

`com.liferay.portal.kernel` package. In order to use OSGi annotations, we need a dependency on the `org.osgi.service.component.annotations` package as well.

1. Open the `build.gradle` file.
2. Add the dependencies for the `com.liferay.portal.kernel` and the `org.osgi.service.component.annotations` package. Afterwards the contents of the file should look as follows:

```
dependencies {
    compileOnly group: "com.liferay.portal", name: "com.liferay.portal.kernel", version: "3.0.0"
    compileOnly group: "org.osgi", name: "org.osgi.core", version: "6.0.0"
    compileOnly group: "org.osgi", name: "org.osgi.service.component.annotations", version: "1.3.0"
}
```

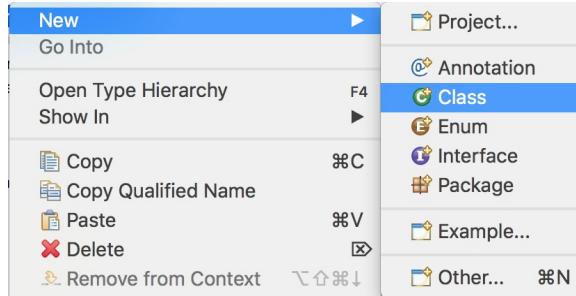
3. Save the file.
4. Open the `bnd.bnd` file.
5. Click on the Source tab.
6. Replace the contents with the following:
 - o We are removing the Bundle-Activator that we deleted in an earlier step.
 - o We are also renaming the bundle. This step is not necessary, but it's helpful to use a more human-readable `Bundle-Name` in case you need to troubleshoot the bundle in the Gogo Shell, for example.

```
Bundle-Name: Message Bus Listener
Bundle-SymbolicName: com.liferay.training.messagebuslistener
Bundle-Version: 1.0.0
```

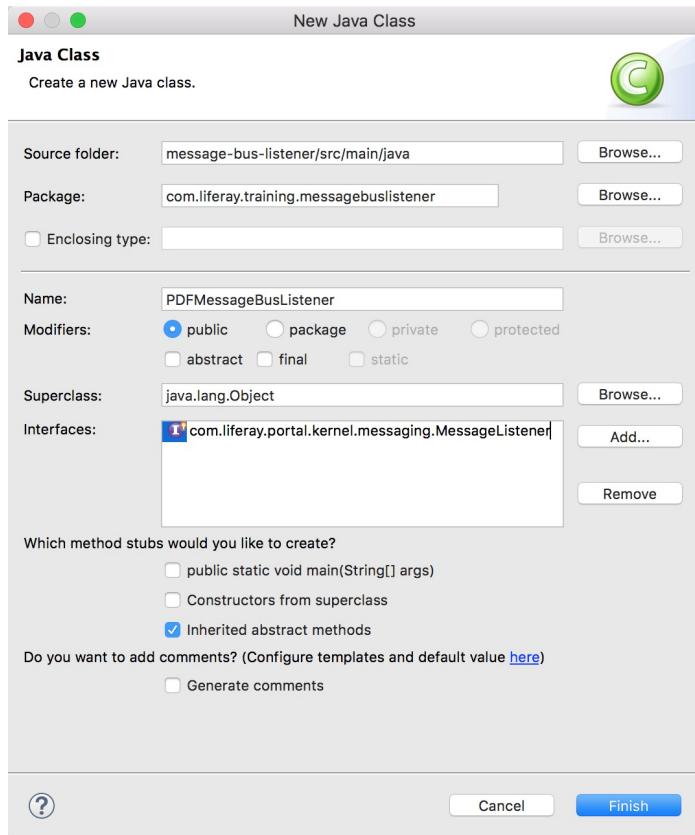
7. Save the file.

Create the PDFMessageListener Class

1. Right-click on the `message-bus-listener` module to open a context menu.
2. Choose `New → Class`.



3. Type `PDFMessageListener` in the Name field.
4. Add `com.liferay.portal.kernel.messaging.MessageListener` as an interface.
 - o The rest of the fields should be correct by default.
5. Click `Finish`.



Implement the receive() Method

We'll be listening on the destination `DestinationNames.DOCUMENT_LIBRARY_PDF_PROCESSOR` for the Documents and Media PDF Processor.

1. Implement the `@Component` annotation using the following:

```
@Component (
    immediate = true,
    property = {
        "destination.name=" + DestinationNames.DOCUMENT_LIBRARY_PDF_PROCESSOR
    },
    service = MessageListener.class
)
```

2. Implement the `receive()` method as specified below:

```
@Override
public void receive(Message message)
throws MessageListenerException {

    Object[] payload = (Object[])message.getPayload();

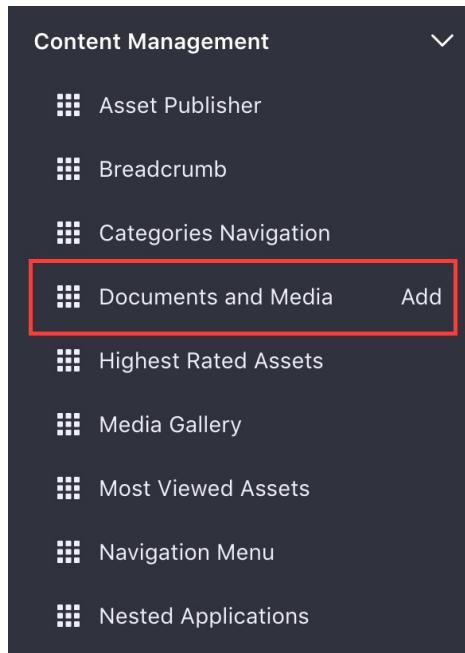
    FileVersion fileVersion = (FileVersion)payload[1];

    System.out.println("Title=" + fileVersion.getTitle());
}
```

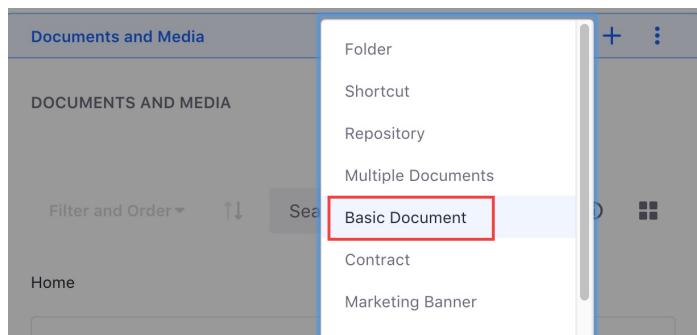
3. Press `CTRL+SHIFT+O` to resolve any missing imports.
4. Save the file.

Deploy and Test

1. **Go to** the *Gradle Tasks* view in *Liferay Developer Studio*.
2. **Double-click** the `deploy` task from `message-bus-listener/build`.
 - o This will deploy *message-bus-listener* to the Liferay Server.
3. **Open** your browser to `http://localhost:8080` and log in.
4. **Click** *Add* in the top right corner.
5. **Go to** *Widgets* → *Content Management*.
6. **Click** *Add* next to the *Documents and Media* widget.



7. **Click** *Add* in the top bar of the *Documents and Media* application.
8. **Choose** *Basic Document*.



9. **Choose** a PDF file to upload.
 - o If you don't have one, there is a `sample-pdf.pdf` file that you can use in solutions folder for this exercise (`solutions/solutions-chapter-13/exercise-files`).
10. **Click** *Publish*.

[New Document](#)

Upload documents no larger than 100MB.

File

Choose File sample-pdf.pdf

Title *

sample-pdf.pdf

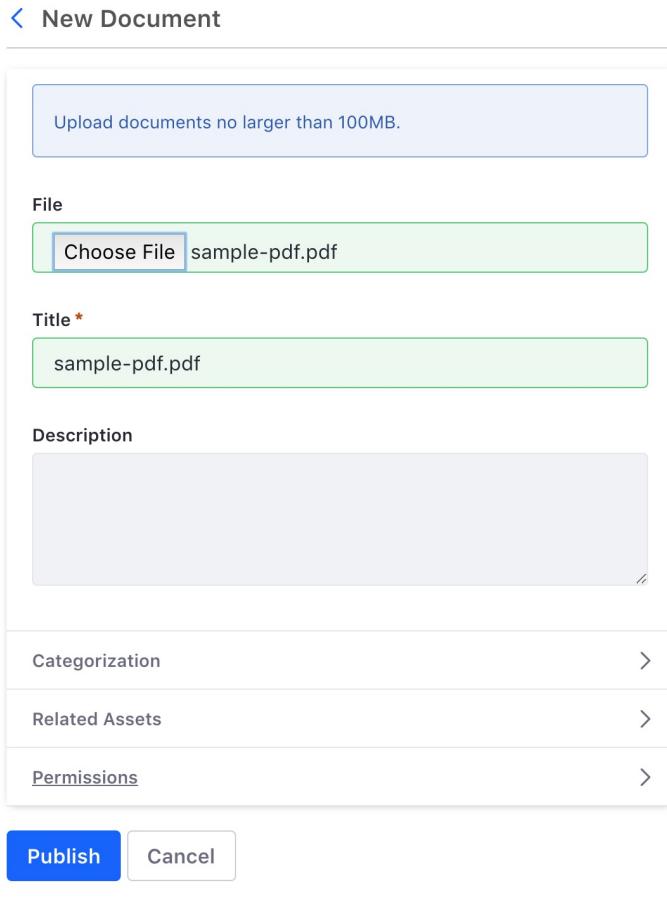
Description

Categorization >

Related Assets >

Permissions >

Publish **Cancel**



You should see `Title=sample-pdf.pdf`, or the name of the file you uploaded in the console.

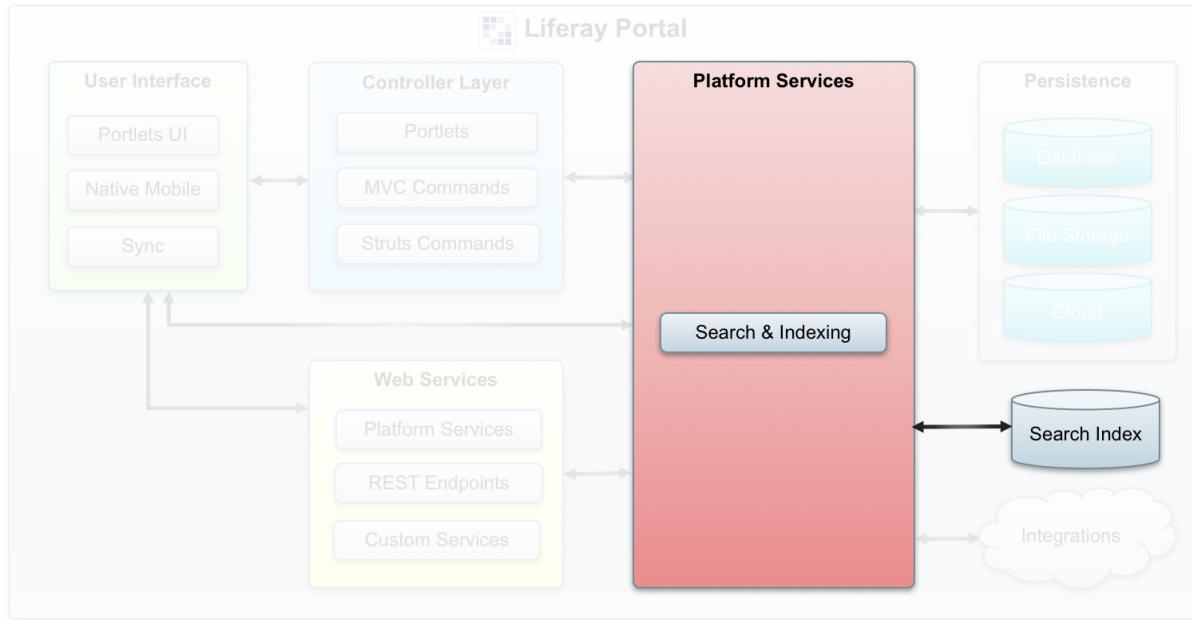
If you don't see the message displayed on your console right away, try restarting the platform.

```
2018-06-11 11:47:16.160 INFO [main][Pr  
2018-06-11 11:47:16.164 INFO [main][Pr  
2018-06-11 11:47:16.166 INFO [main][Pr  
Loading jar:file:/Users/myoung/projects  
Title=sample-pdf.pdf
```

Takeaways

We've shown you how to implement a Message Bus Listener in this exercise to listen for messages generated by Liferay's applications. In addition to implementing a listener, you can create destinations and senders for your own applications. Remember, the message bus is often applied in situations where you do not want a long running process to affect the responsiveness of your application.

Chapter 14: Customize the Portal Search



Chapter Objectives

- Understand General Search Concepts
- Understand the Architecture and Components of the Liferay Portal Search Framework
- Learn How Indexing and Searching Works in Liferay
- Learn How to Approach Portal Search Customization

Introduction

Liferay has an extensive search framework. Customization can be done on all layers from the user interface to the back-end search engine. Because of the pluggable architecture and strong APIs, even writing a completely new custom adapter for a third-party search engine is possible.

In the following sections, we will be focusing on the features and customization of the default Elasticsearch engine. The customization methods for the platform search can generally be divided into the following categories:

- Customizing the **index settings and type mappings** from the Control Panel
- Modifying the front-end by developing a **module fragment for the search portlet**
- Modify the result hits with **Hits Processors**
- Modifying indexing and querying with **Indexer Post Processors**
- Creating a **custom search engine adapter** that implements the IndexSearcher and IndexWriter interfaces

Liferay Search Architecture

Why Do We Need a Search Engine?

Why do we need a separate search engine? Why can't we just use direct database queries? There are several reasons: RDBMS data structures are not optimal for searching and database JOIN clauses can be performance killers. Search engines, on the other hand, use storage algorithms and structures like **inverted indexes**, which are optimized for speed. Search engines also use a fully configurable process called **analysis** that optimizes the data for searchability for different use cases and scenarios. Search engines also give access to querying features like **relevance** and **scoring** and provide support for **advanced query types** and features like fuzzy or proximity searches. In production systems, search engines typically run in a dedicated server, reducing the load from Liferay servers and improving the overall **performance**.

Liferay Search Engine Support

Liferay has a pluggable search engine support. Elasticsearch is the default engine, but an adapter for SOLR is also provided. Additionally, it is possible to develop a custom search engine adapter.

Elasticsearch has two available operation modes:

1. Embedded
2. Remote

When running in embedded mode, Liferay is using the embedded Elasticsearch engine, which runs in the same process as the Liferay platform. In the remote operation mode, Liferay is configured to use an external Elasticsearch server or cluster. Using embedded mode is not recommended in the production systems. Depending on the scenario, the search engine might be intensive in its resource usage and have a negative impact on the Liferay platform's overall performance.

The screenshot shows the Liferay System Settings interface with the 'Search' tab selected. On the left, there is a sidebar with various system scope settings. The main content area is titled 'Elasticsearch 6' and contains the following configuration fields:

- Cluster Name:** LiferayElasticsearchCluster. A note below states: "The name of the cluster to join. The name should match the remote cluster when Operation Mode is set to remote."
- Operation Mode:** A dropdown menu currently set to 'REMOTE'. Other options include 'CHOOSE AN OPTION' and 'EMBEDDED'.
- Index Prefix:** A note below states: "The prefix for the search index name. This value should not be changed under normal conditions. If you change this value, you must also reindex the portal and then manually delete the old index using the Elasticsearch administration console."
- Number of Index Replicas:** A note below states: "The number of replicas for each Liferay index. If unset, no replicas are used. Changes to this value take effect after a full reindex."

Portal Search API

Applications or code using the Liferay platform search always communicate through the portal search API. The underlying search engine-specific adapter is never exposed:

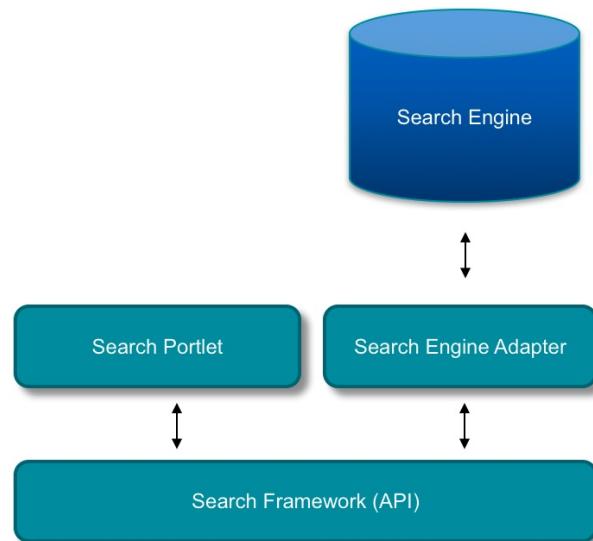


Figure: Portal search overall architecture.

The following diagram illustrates the architecture of the search-related modules:

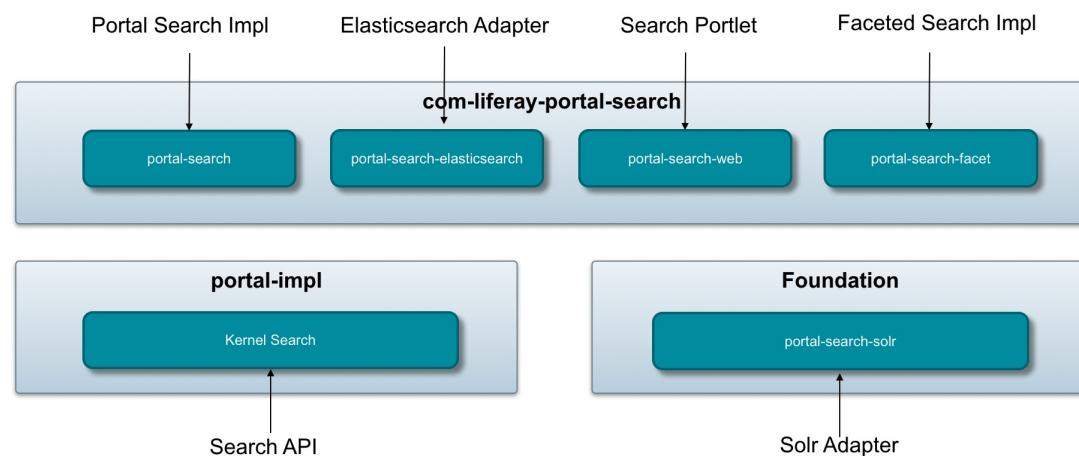


Figure: Portal search modules

Related Source Code

- **com-liferay-portal-search**
<https://github.com/liferay/com-liferay-portal-search/tree/master>
- **portal-impl (search)**
<https://github.com/liferay/liferay-portal/tree/7.1.x/portal-kernel/src/com/liferay/portal/kernel/search>
- **portal-search-elasticsearch6**
<https://github.com/liferay/liferay-portal/tree/7.1.x/modules/apps/portal-search-elasticsearch6>
- **portal-search-solr**
<https://github.com/liferay/liferay-portal/tree/7.1.x/modules/apps/portal-search-solr>

Basic Search Concepts

Many of the platform search customization tasks require an understanding of basic search concepts that are not Liferay-specific but common for search engines in general. Some of the these fundamental concepts are *indexing*, *analysis*, *searching*, and *queries*.

Indexing

Indexing is a process of transforming input data to a search engine document, which is the storage model type for search engines.

In the context of Liferay, the input data is usually a registered portal asset. Every platform asset type has a dedicated indexer that knows how to transform the asset data specific to the type to the index document. This indexer class is extending the `com.liferay.portal.kernel.search.BaseIndexer` class. The `doGetDocument()` method has the main responsibility of the transformation:

```
...
@Component(immediate = true, service = Indexer.class)
public class BlogsEntryIndexer extends BaseIndexer<BlogsEntry> {

    public static final String CLASS_NAME = BlogsEntry.class.getName();

    ...
    @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.DISPLAY_DATE, blogsEntry.getDisplayDate());
        document.addDate(Field.MODIFIED_DATE, blogsEntry.getModifiedDate());
        document.addText(Field.SUBTITLE, blogsEntry.getSubtitle());
        document.addText(Field.TITLE, blogsEntry.getTitle());

        return document;
    }
}
```

Source: <https://github.com/liferay/liferay-portal/blob/7.1.x/modules/apps/blogs/blogs-service/src/main/java/com/liferay/blogs/internal/search/BlogsEntryIndexer.java>

Analysis

Analysis is a field-level process of transforming input data into search engine document field data. This process slightly varies depending on the engine, but in Elasticsearch there are three phases:

1. Character filtering, like removing HTML tags
2. Tokenizing, effectively splitting the field value into individual tokens, sometimes also called words
3. Token filtering by, for example, language-specific analysis, removal of stop words, etc.

Analysis is a fully configurable process. Individual configurations are sometimes called *analyzers* and are defined on an individual field-level, meaning that every single field in the index document can have a different analyzer assigned. Analyzers are configured in the index settings and assigned to the fields in the mapping definitions.

Analyzers are used both in indexing and in querying the index. Notice that the same field analyzers have to be used both at index- and at query-time in order to get predictable results. For example, if a field is indexed using an English language filter, analyzing the search query with a German filter would result in a mismatch in, for example, possessive and plural word forms.

Below is a diagram of the indexing and analysis process in Liferay:

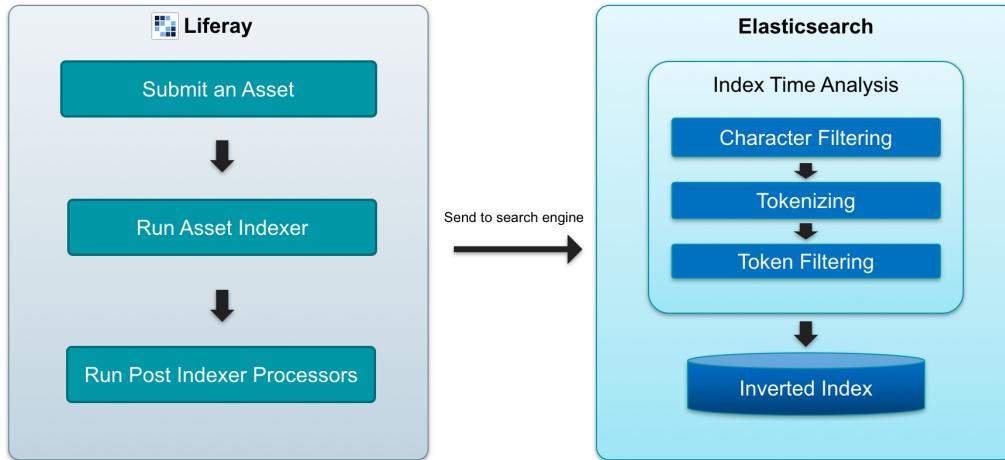


Figure: Indexing

Searching

Searching, in brief, is sending a search query and obtaining results, also called hits, from the search engine.

Searching involves regular and filter queries. Queries may also have sub-queries.

A regular **query** asks if an index document field matches the keywords in a way defined in the query (type) and how relevant the document field is to the search terms (score). The query type can be, for example, an exact match, wildcard, or term query. Every query type may have its own properties, but common ones are *operator*, which defines if a query having multiple words has to match all, none, or just one (AND, NOT, OR) and *boosting*, which defines the scoring weight of that specific query in the final, composed query.

A **filter query** is composed the same way as a regular query. Its only difference is that it works in a different context and returns only simple "yes" or "no" to its condition, instead of scoring the relevancy. As filter queries don't impact scoring, they are practically used to limit the result set. For example, a filter query might limit the results only to a certain Liferay site and the regular query would then find any relevant documents belonging to that site and score them against the search phrase.

Although the same result set can be achieved by using regular queries instead of filter queries, it should be remembered that filter queries are much faster.

Below is a diagram that shows the different phases and components when doing a search in Liferay with the standard search portlet:

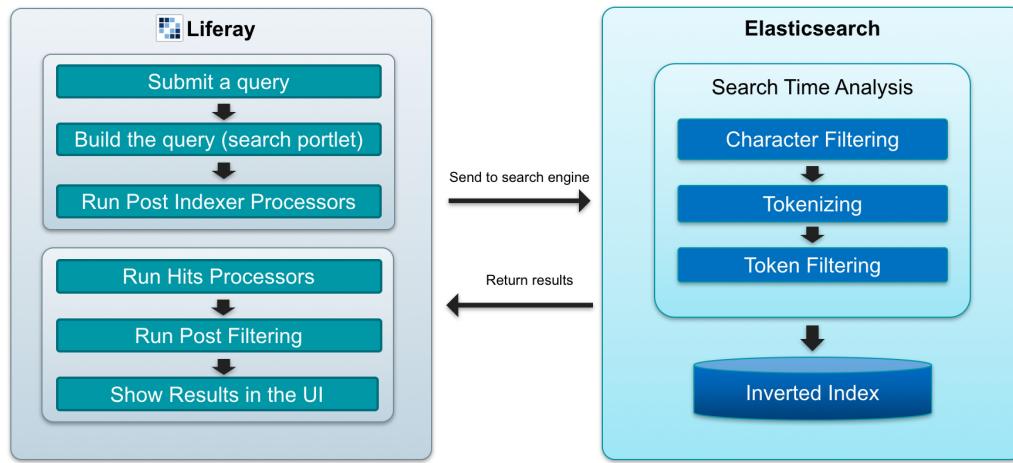


Figure: How Liferay searching works

Links and Resources

- **Elasticsearch Query DSL**

<https://www.elastic.co/guide/en/elasticsearch/reference/6.1/query-dsl.html>

Customize Indexing and Search Results with Processors

Indexer Post Processor

Indexer post processors are components that allow you to modify the way portal assets are being indexed. The index is queried as are the contents of the results, which are returned to the user interface.

From a code perspective, an indexer post processor is an OSGi component that implements the `com.liferay.portal.kernel.search.IndexerPostProcessor` interface. The targeted model (asset) type(s) are defined by a component's `indexer.class.name` property.

There can be multiple indexer post processors registered to a single model type. Put another way, a single indexer post processor can register to multiple model types.

Below is an example of an indexer post processor registered to a BlogsEntry and JournalArticle:

```

@Component(
    immediate = true,
    property = {
        "indexer.class.name=com.liferay.blogs.model.BlogsEntry",
        "indexer.class.name=com.liferay.journal.model.JournalArtice"
    },
    service = IndexerPostProcessor.class
)
public class BlogsIndexerPostProcessor implements IndexerPostProcessor {

    @Override
    public void postProcessSearchQuery(
        BooleanQuery searchQuery, BooleanFilter booleanFilter,
        SearchContext searchContext)
        throws Exception {

        // Query processing code here
        ...
    }
}

```

Hits Processor

The hits processors allows you to perform actions on the hits returned. Example use cases include processing the hits before they are sent to the user interface, spell-checking, and doing an alternative query and indexing successful queries for the autocomplete / keyword suggestion purposes.

Hits processors are OSGi components that implement the `com.liferay.portal.kernel.search.hits.HitsProcessor` interface.

Hits processors can be chained. The processing orders is defined by the components `sort.order` property.

Below is an example of a hits processor:

```

@Component(
    immediate = true,
    property = "sort.order=0",
    service = HitsProcessor.class
)
public class CollatedSpellCheckHitsProcessor implements HitsProcessor {

    @Override
    public boolean process(SearchContext searchContext, Hits hits)
}

```

```

throws SearchException {

    QueryConfig queryConfig = searchContext.getQueryConfig();

    if (!queryConfig.isCollatedSpellCheckResultEnabled()) {
        return true;
    }

    int collatedSpellCheckResultScoresThreshold =
        queryConfig.getCollatedSpellCheckResultScoresThreshold();

    if (hits.getLength() >= collatedSpellCheckResultScoresThreshold) {
        return true;
    }

    String collatedKeywords = IndexSearcherHelperUtil.spellCheckKeywords(
        searchContext);

    if (collatedKeywords.equals(searchContext.getKeywords())) {
        collatedKeywords = StringPool.BLANK;
    }

    hits.setCollatedSpellCheckResult(collatedKeywords);

    return true;
}
}

```

The following hits processors are run by default:

CollatedSpellCheckHitsProcessor

CollatedSpellCheckHitsProcessor performs a spell check if the minimum score for the results is less than a given threshold. The threshold is controlled by the following portal property:

- index.search.collated.spell.check.result.scores.threshold

AlternateKeywordQueryHitsProcessor

When configured, this processor automatically issues an alternate query based on suggested keywords from the CollatedSpellCheckHitsProcessor.

QueryIndexingHitsProcessor

The QueryIndexingHitsProcessor indexes a query if the number of hits has exceeded a configured quantity. This processor is controlled by the following portal properties:

- index.search.query.indexing.enabled
- index.search.query.indexing.threshold

QuerySuggestionHitsProcessor

If query indexing is enabled, this processor allows you to suggest other potential queries that previous searches have yielded more results. The processor is controlled by the following portal properties:

- index.search.query.suggestion.scores.threshold
- index.search.query.suggest.max

Links and Resources

- **Indexer Post Processor API**

<https://github.com/liferay/liferay-portal/blob/master/portal-kernel/src/com/liferay/portal/kernel/search/IndexerPostProcessor.java>

- **Hit Processor API**

<https://github.com/liferay/liferay-portal/blob/7.1.x/portal-kernel/src/com/liferay/portal/kernel/search/hits/HitsProcessor.java>

Exercises

Extend User Search Using an Indexer Post Processor

Introduction

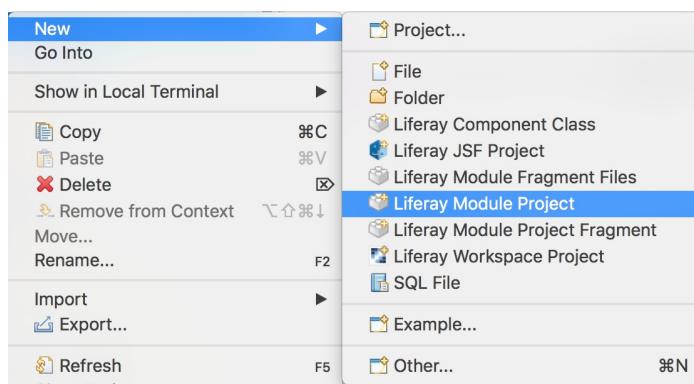
In this exercise, we will extend the capabilities of the user search functionality. Specifically, we are going to enable search by user id and gender. By implementing the `IndexerPostProcessor` interface, we can modify what gets indexed in the `postProcessDocument()` method, and we can modify what can be queried in the `postProcessSearchQuery()` method.

Overview

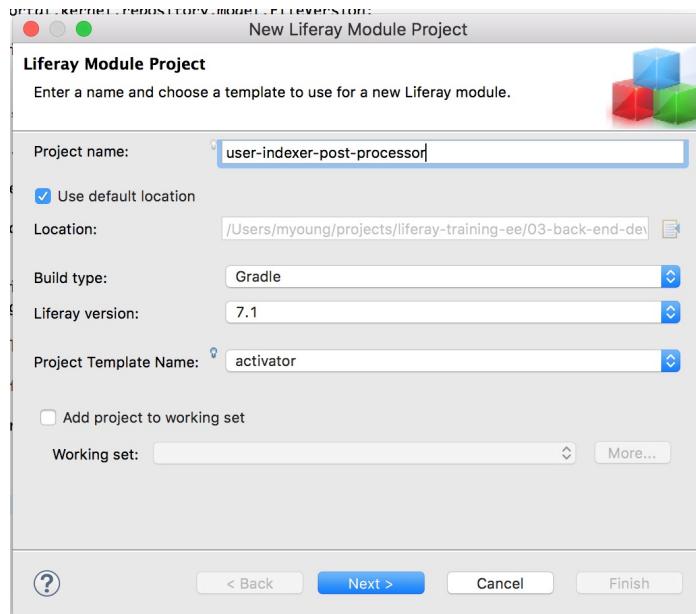
- ① Create a Liferay Module project using the `activator` template
- ② Resolve dependencies
- ③ Create the `UserIndexerPostProcessor` class
- ④ Implement the `postProcessDocument()` method
- ⑤ Implement the `postProcessSearchQuery()` method
- ⑥ Deploy and test

Create a Liferay Module Project Using the Activator Template

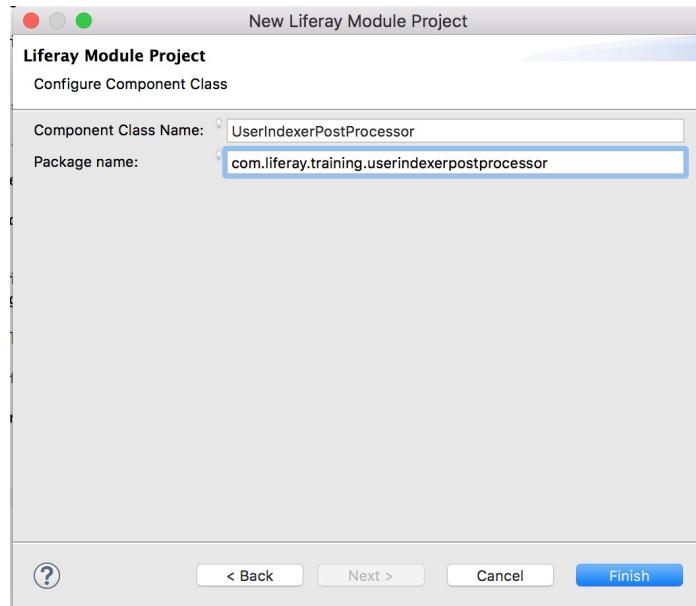
1. [Go to the `training-workspace`.](#)
2. [Right-click](#) on the workspace to open a context menu.
3. [Choose `New → Liferay Module Project`.](#)



4. [Type `user-indexer-post-processor` in the `Project Name` field.](#)
5. [Choose `activator` in the `Project Template Name` field.](#)
6. [Click `Next`.](#)



7. Type `UserIndexerPostProcessor` in the *Component Class Name* field.
8. Type `com.liferay.training.userindexerpostprocessor` in the *Package name* field.
9. Click **Finish**.



10. Expand the `user-indexer-post-processor` project.
11. Delete the `UserIndexerPostProcessorActivator.java` class that was auto-created in the `com.liferay.training.userindexerpostprocessor` package.
 - o We will not be using this class in our customization.

Resolve Dependencies

Next, we will have to adjust the `bnd.bnd` and `build.gradle` files generated by Liferay Studio. Since we used the `activator` template, `build.gradle` will only contain the bare minimum of dependencies to set up an OSGi project. Because we want to implement `com.liferay.portal.kernel.search.IndexerPostProcessor` interface we have to depend on the `com.liferay.portal.kernel`

, `portlet-api`, and `javax.servlet-api` packages. Since we implement an OSGi component, we also have to depend on the `org.osgi.service.component.annotations` package.

We'll need to fix the `bnd.bnd` and `build.gradle` file generated by Liferay Developer Studio. Liferay Developer Studio generated these files using the activator template, so they don't have the dependencies set up for our particular project.

1. **Open** the `build.gradle` file.
2. **Replace** the contents with the following code:

```
dependencies {  
    compileOnly group: "com.liferay.portal", name: "com.liferay.portal.kernel", version: "3.0.0"  
    compileOnly group: "org.osgi", name: "org.osgi.core", version: "6.0.0"  
    compileOnly group: "org.osgi", name: "osgi.cmpn", version: "6.0.0"  
    compileOnly group: "javax.portlet", name: "portlet-api", version: "3.0.0"  
    compileOnly group: "javax.servlet", name: "javax.servlet-api", version: "3.0.1"  
}
```

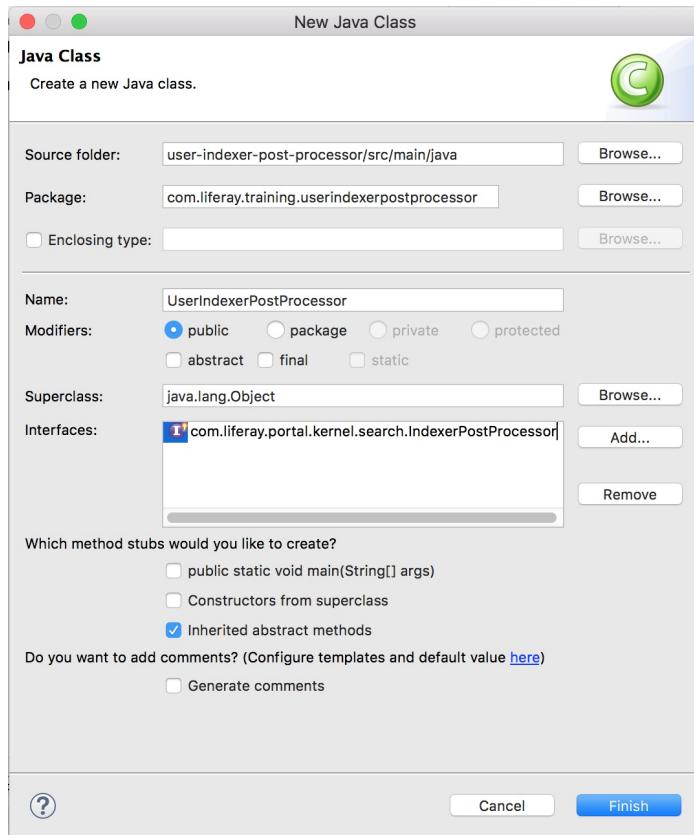
3. **Save** the file.
4. **Open** the `bnd.bnd` file.
5. **Click** on the `Source` tab.
6. **Replace** the contents with the following:
 - o We are removing the Bundle-Activator that we deleted in an earlier step.
 - o We are also renaming the bundle. This step is not necessary, but it's helpful to use a more human-readable `Bundle-Name` in case you need to troubleshoot the bundle in the Gogo Shell, for example.

```
Bundle-Name: User Indexer Post Processor  
Bundle-SymbolicName: com.liferay.training.userindexerpostprocessor  
Bundle-Version: 1.0.0
```

7. **Save** the file.

Create the `UserIndexerPostProcessor` Class

1. **Right-click** on the `user-indexer-post-processor` project to open a context menu.
2. **Choose** `New → Class`.
3. **Type** `UserIndexerPostProcessor` in the `Name` field.
4. **Add** `com.liferay.portal.kernel.search.IndexerPostProcessor` as an interface.
 - o The rest of the fields should be correct by default.
5. **Click** `Finish`.



Implement the postProcessDocument() Method

1. **Implement** the `@Component` annotation using the following code below.

- The `indexer.class.name=com.liferay.portal.kernel.model.User` specifies that we are implementing the `IndexerPostProcessor` for the User model object.

```
@Component(
    immediate = true,
    property = {
        "indexer.class.name=com.liferay.portal.kernel.model.User",
    },
    service = IndexerPostProcessor.class
)
```

2. **Implement** the `postProcessDocument()` method as specified below:

```
@Override
public void postProcessDocument(Document document, Object obj)
    throws Exception {

    System.out.println("postProcessDocument");

    User user = (User)obj;

    int male = 0;

    if (user.isMale()) {
        male = 1;
    }

    document.addNumber("male", male);
}
```

In the `postProcessDocument()` method, we are grabbing the User's gender and adding it to the *document*. The document is the contract to our search API to tell it what to index for the User entity.

Implement the `postProcessSearchQuery()` Method

1. **Implement** the `postProcessSearchQuery()` method as specified below:

```
@Override
public void postProcessSearchQuery(
    BooleanQuery searchQuery, SearchContext searchContext)
throws Exception {

    System.out.println("postProcessSearchQuery-2");

    String keywords = searchContext.getKeywords();

    if (keywords == null) {
        return;
    }

    searchQuery.addTerm("userId", searchContext.getKeywords());

    if (keywords.contains("boys") || keywords.contains("guys") ||
        keywords.contains("men") || keywords.contains("dudes")) {

        searchQuery.addTerm("male", 1);
    }
}
```

2. **Press** `CTRL+SHIFT+O` to resolve any missing imports.

- o Choose the first import option when prompted.

3. **Save** the file.

In the `postProcessSearchQuery()` method, we grab the keyword search entered in the search box, and we enable the user to be searched by user id. After that, we make the search a little more free-form. If someone searches for males, boys, guys, or men, it will search for all male User entities. We did not implement this here, but this could easily be extended to search for female User entities.

Deploy and Test

1. **Go to** the *Gradle Tasks* view in *Liferay Developer Studio*.
2. **Double-click** the `deploy` task from `user-indexer-post-processor/build`.
 - o This will deploy `user-indexer-post-processor` to the Liferay Server.
3. **Open** your browser to `http://localhost:8080` and log in.
4. **Open** up the *Menu*.
5. **Go to** `Control Panel → Users → Users and Organizations`.

A screenshot of a user search interface. At the top, there is a toolbar with a filter dropdown, a search bar containing "Search for.", and a blue "Add" button with a white plus sign, which is highlighted with a red box. Below the toolbar is a table with three columns: "Name", "Screen Name", and "Job Title". A single user entry is visible: "Test Test" in the Name column, "test" in the Screen Name column, and an empty string in the Job Title column. There is also a small grey progress bar below the table.

6. Click Add next to the Search Bar.
7. Create a User.
8. Type the Screen Name, Email Address, First Name, Last Name.
9. Choose male for gender.
10. Click Save.

A screenshot of a user creation form titled "Information". The form is divided into sections: "PERSONAL INFORMATION" and "Names".
In the "PERSONAL INFORMATION" section:

- "Screen Name *": Input field containing "joesmith" (highlighted with a green border).
- "Birthday": Input field containing "01/01/1970".
- "Email Address *": Input field containing "jsmith@test.com" (highlighted with a green border).
- "Gender": A dropdown menu showing "Male".
- "Language": A dropdown menu showing "English (United States)".
- "Job Title": An input field that is currently empty.
- "Prefix": A dropdown menu that is currently empty.

In the "Names" section:

- "First Name *": Input field containing "joseph" (highlighted with a green border).
- "Middle Name": An input field that is currently empty.
- "Last Name *": Input field containing "smith" (highlighted with a blue border).

11. Copy the User Id to your clipboard.

User ID

34120

12. [Click Users and Organizations.](#)
13. [Paste](#) the *User Id* from the earlier step in the search bar.
 - o The user you created should be part of the search result.

The screenshot shows a search interface with the following details:

- Search bar: 34120
- Results count: 1 results for 34120
- Table headers: Name, Screen Name, Job Title, Organizations, User Groups
- Table data:

| Name | Screen Name | Job Title | Organizations | User Groups |
|---------------------------------------|-------------|-----------|---------------|-------------|
| <input type="checkbox"/> joseph smith | joesmith | | | |

14. [Enter guys, boys, men, or dudes](#) into the search bar.
 - o The user you created should be part of the search result.

The screenshot shows a search interface with the following details:

- Search bar: guys
- Results count: 1 results for guys
- Table headers: Name, Screen Name, Job Title, Organizations, User Groups
- Table data:

| Name | Screen Name | Job Title | Organizations | User Groups |
|---------------------------------------|-------------|-----------|---------------|-------------|
| <input type="checkbox"/> joseph smith | joesmith | | | |

Takeaways

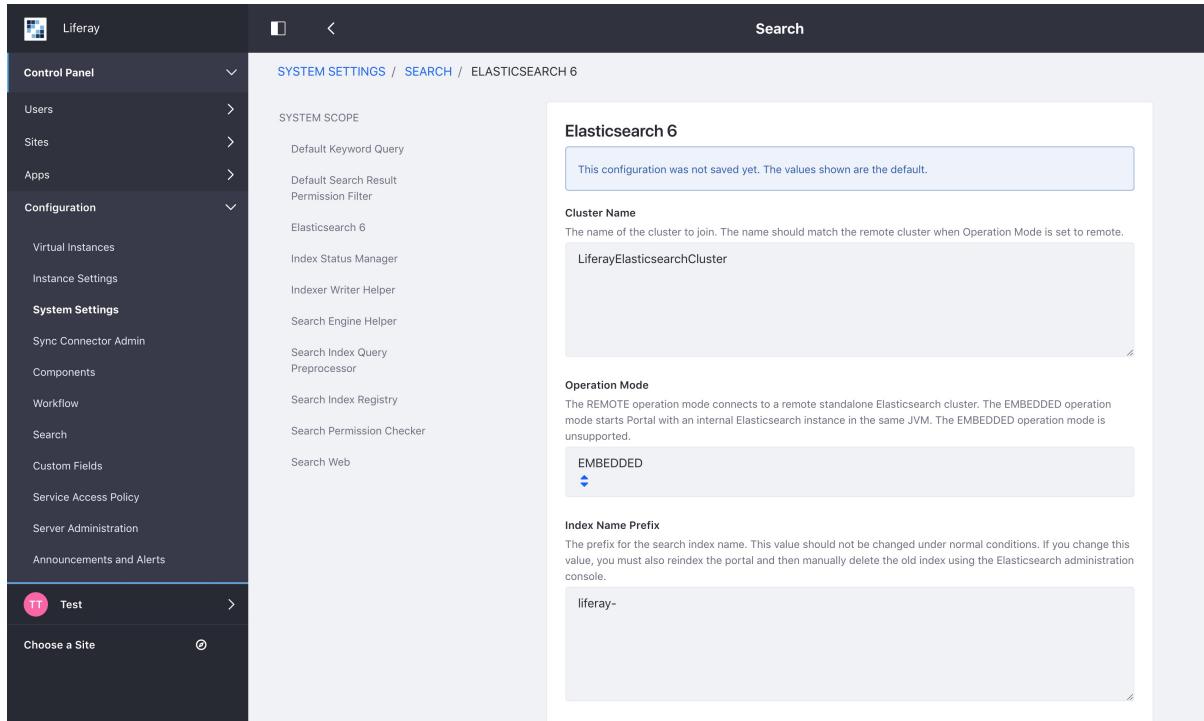
We've shown you how to extend the indexing and querying functionality of our search framework. Using this technique, you should be able to customize the search for any of the built-in Liferay entities. In addition, developers should be able to use this same technique to extend the search functionality of custom applications as well.

Customize Index Settings and Type Mapping

Elasticsearch index settings and type mapping can be modified through the Control Panel → System Settings → Platform → Search → Elasticsearch 6.

The management user interface allows you to:

- Define custom analyzers
- Customize stopword and synonym handling
- Customize type mappings, like:
 - Change index field settings
 - Configure new fields for the index



Links and Resources

- **Customization instructions on dev.liferay.com**
https://dev.liferay.com/de/discover/deployment/-/knowledge_base/7-1/advanced-configuration-of-the-liferay-elasticsearch-adapter
- **Elasticsearch mapping files**
<https://github.com/liferay/liferay-portal/tree/7.1.x/modules/apps/portal-search-elasticsearch6/portal-search-elasticsearch6-impl/src/main/resources/META-INF>

Links and References

- **Blade Samples** <https://github.com/liferay/liferay-blade-samples>
- **IntelliJ Plugin** <https://github.com/liferay/liferay-intellij-plugin>
- **Liferay Community Forums** <https://community.liferay.com/forums>
- **Liferay Customer portal (requires authorization)** <http://customer.liferay.com>
- **Liferay Dev Studio** <https://web.liferay.com/downloads/liferay-projects/liferay-ide>
- **Liferay Portal Compatibility Matrix**
<https://web.liferay.com/services/support/compatibility-matrix>
- **Liferay eLearning Platform** <https://university.liferay.com>
- **Liferay Official Documentation** <https://dev.liferay.com>
- **Liferay Source Code** <https://github.com/liferay>