

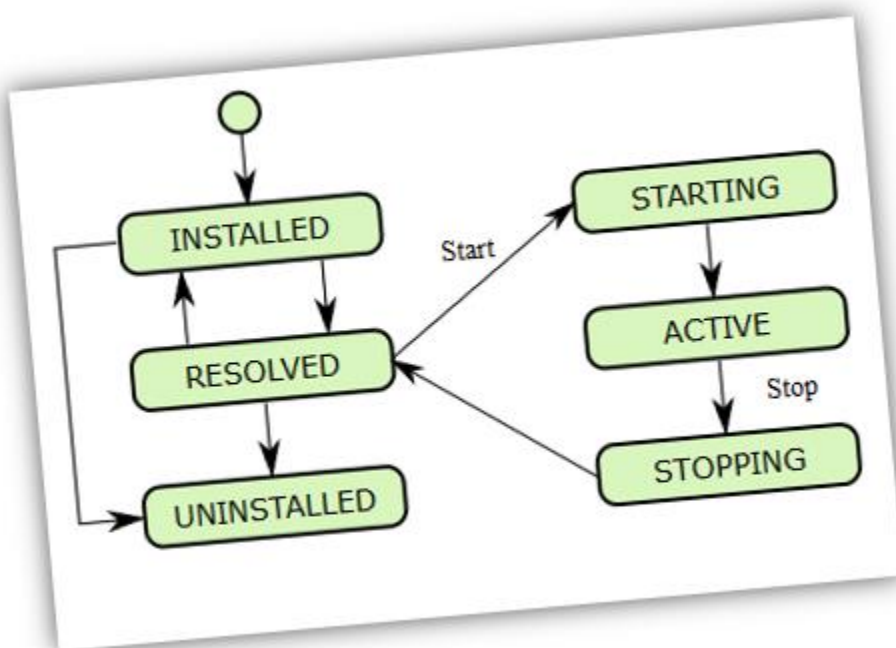
1 Difference between Liferay 6.2 and Liferay Dxp?

	DXP	6.2
BOOT STRAP, AUI Version	Liferay 7 support Bootstrap 3 UI framework. AUI 3	Liferay 6.2 support Bootstrap 2 UI framework. AUI 2
DEPLOYMENT FILE FORMAT	Support war and Jar	Support War
OSGI VS TRADITIONAL	Traditional Application Container	OSGI Container
SEARCH ENGINE ELASTIC VS LUCENE SOLR	Liferay 7 inbuilt search engine Elastic Search. It supports SOLR search as well.	Liferay 6.2 in built search engine Apache Lucene. Liferay 6.2 have SOLR web to enable SOLR search.
SINGLE PAGE APP BUILT IN SUPPORT	Liferay 7 have Single Page Application inbuilt support using Senna.js	Liferay 6.2 does not have inbuilt Single Page Application support but we can integrate SPA framework to achieve it.
DESIGN SUPPORT	Clay and Lexicon design	Do not have
GOGO SHELL ACTIVE DEACTIVE VS INSTALL UNINSTALL	Activate / Deactivate from Gogo shell	Require to remove

1. Liferay Forms
2. Geolocate Content & Documents
3. Image and Media Selector
- 4 Content Sharing Between Sites – child can access parent content
- 5 Image Editor
- 6 Web Content Diffs
- 7 Lexicon
- 8 New Alloy Editor
- 9 Web content folder type specific workflow
- 10 Modularity

<https://www.xtivia.com/media/xtivia-top10liferay-ebook.pdf>

2 State of Bundle



INSTALLED: The bundle has been successfully installed.

RESOLVED: All Java classes that the bundle needs are available. This state indicates that the bundle is either ready to be started or has stopped.

STARTING: The bundle is being started, the **BundleActivator.start** method has been called but the start method has not yet returned. When the bundle has an activation policy, the bundle will remain in the **STARTING** state until the bundle is activated according to its activation policy.

ACTIVE: The bundle has been successfully activated and is running. Its Bundle Activator start method has been called and returned.

STOPPING: The bundle is being stopped. The **BundleActivator.stop** method has been called but the stop method has not yet returned.

3 Steps of Migration from Liferay 6.2 to Liferay DXP

Step 1: Prepare for migration

Sites

- Remove unused sites and its content.

Instances

- Remove instance which is no longer needed. Since they consume highest object in the hierarchy, removing it can optimize upgradation process.

Web Content

- Liferay generates a new version whenever we are updating any web content. You can safely delete all older versions of web content.

Users

- Remove all inactive users and its related data to optimize the process.

Layouts

- Remove unnecessary layouts.

Orphaned data

- Find out orphaned data that are no longer used and remove all those data.

Remove duplicated structure names

- For LR 6.2 you can have different web content structures in your portal, they are likely to have fields with the same names. But DXP requires unique field names of fields for structures. Therefore we need to rename structure to have a unique field name.

Step 2: Liferay Database migration

- **Ready To DB Migrate After Data Cleanup:** - In this step, we will migrate our database from LR 6.2 to LR 7.1. After removing unused objects now our portal is ready to migrate DB from 6.2 to 7.1/DXP.
- **DB Migration Tools:** - Liferay provides a comprehensive database migration tool that will migrate our database using simple steps.
- **Disable Search Index:** - Liferay recommends disabling search indexing before migrating the database.
- **Backup Of DB:** - Before migrating existing DB to LR DXP make sure that you have a backup of it.
- **Enable The Search Index**

Steps

Create a new database for Liferay 7.1/DXP in MySQL server

- e.g. create database **Liferay_dxp_db** character set utf8 Take backup and restore.

Take backup and restoreSpecify database connection properties

Create a file '**portal-setup-wizard.properties**' and specify the below DB connection properties within it:

```
jdbc.default.driverClassName=com.mysql.jdbc.Driverjdbc.default.url=jdbc:mysql://localhost:3306/liferay_dxp_db
```

```
jdbc.default.username=root
```

```
jdbc.default.password=root
```

Note: DO NOT START Liferay DXP server at this point. Otherwise, you will get an error like `java.lang.RuntimeException`

Copy data

Copy document_library (You can find from [Liferay Home]/data/) folder from Liferay 6.2 to Liferay 7.1/DXP, copy images folders into data folder also. This data will be stored in the default store and you can also choose your location, it's up to your choice.

Disable indexer and autoUpgrade

Now, we need to disable indexing since we are going to start the database upgrade process for our new environment. Using this we can disable reindexing content and performance issues to be arised. For this, we have to follow below steps.

- Need to create file in liferay 7.1/DXP :
- **"com.liferay.portal.search.configuration.IndexStatusManagerConfiguration.config"** on this location [Liferay Home]/osgi/configs and add below content
 - `indexReadOnly="true"`
- Then create another file in the same folder :
- **"com.liferay.portal.upgrade.internal.configuration.ReleaseManagerConfiguration.config"** and add below content
 - `autoUpgrade="false"`

Configure the upgrade tool

Unlike previous Liferay versions, here we just need to specify database configuration properties, and database upgrade process will run upon liferay server startup, Liferay 7.x provides a dedicated tool for database upgrade 'portal-tools-db-upgrade-client' in [Liferay Home]/tools folder

Now, we need to configure below files before running the upgrade:

Tomcat configuration in Liferay

```
app-server.properties x
##
## Tomcat
##

dir=../../tomcat-9.0.10
extra.lib.dirs=/bin
global.lib.dir=/lib
portal.dir=/webapps/ROOT
server.detector.server.id=tomcat
```

Liferay portal upgrade database properties

```
*portal-upgrade-database.properties x
##
## MySQL
##

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

Liferay portal upgrade database ext.properties

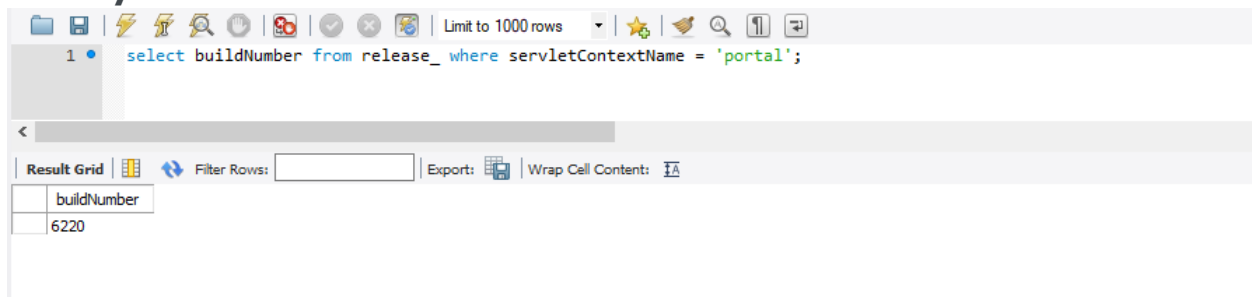
```
portal-upgrade-ext.properties x
liferay.home=../../
module.framework.base.dir=../../osgi
```

Verify buildNumber

We have to check a release buildNumber from DB before running the upgrade tool, it will be incremented after each deployment in Service Builder which is used for generating internal database persistence code.

Using this query you can find current buildNumber:

Liferay 6203 buildNumber



The screenshot shows a database query tool interface. At the top, there's a toolbar with various icons. Below it, a query is entered: `select buildNumber from release_ where servletContextName = 'portal';`. The query is numbered 1. Below the query, there's a "Result Grid" section. It has a "Filter Rows" input field, an "Export" button, and a "Wrap Cell Content" checkbox. The result grid shows a single row with the value 6220 under the column header buildNumber.

buildNumber
6220

Run upgrade tool using below command (Path [Liferay Home]/tools folder) :

For window

Double click on "**db_upgrade.bat**" file

For Ubuntu

`"sh db_upgrade.sh"`

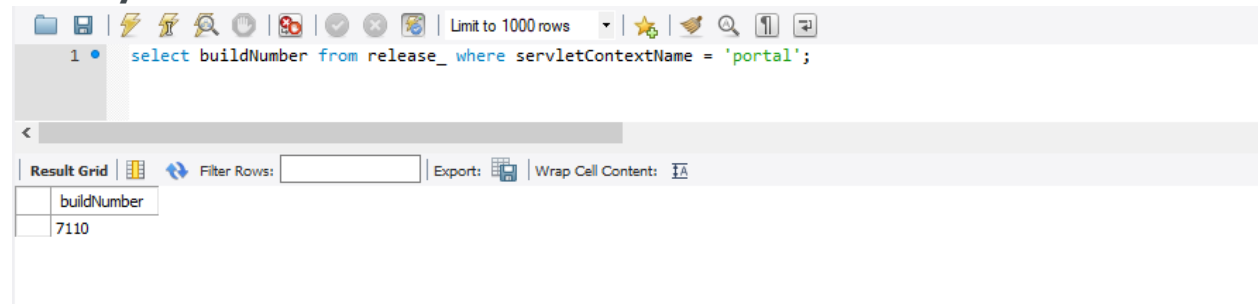
Liferay core upgrade

```
INFO - Starting com.liferay.portal.security.ldap.internal.verify.LDAPPropertiesVerifyProcess#verifyLDAPProperties
2019-04-03 12:15:02.340 INFO [main][LoggingTimer:43] Completed com.liferay.portal.security.ldap.internal.verify.LDAPPropertiesVerifyProcess#verifyLDAPProperties in 1 ms
INFO - Completed com.liferay.portal.security.ldap.internal.verify.LDAPPropertiesVerifyProcess#verifyLDAPProperties in 1 ms
2019-04-03 12:15:02.340 INFO [main][VerifyProcess:80] Completed verification process com.liferay.portal.security.ldap.internal.verify.LDAPPropertiesVerifyProcess in 2ms
INFO - Completed verification process com.liferay.portal.security.ldap.internal.verify.LDAPPropertiesVerifyProcess in 2ms

Completed Liferay core upgrade and verify processes in 115 seconds
Checking to see if all upgrades have completed... your upgrades have failed, have not started, or are still running.
Connecting to Gogo shell...

Type "help" to get available upgrade and verify commands.
Type "help {command}" to get additional information about the command. For example, "help upgrade:list".
```

Liferay 7100 buildNumber



The screenshot shows a database query tool interface. At the top, there is a toolbar with various icons and a text input field containing "Limit to 1000 rows". Below the toolbar, a SQL query is entered in a text area: `select buildNumber from release_ where servletContextName = 'portal';`. Below the query, there is a "Result Grid" section. It includes a "Filter Rows:" input field, an "Export:" button, and a "Wrap Cell Content:" checkbox. The "Result Grid" itself is a table with two columns: "buildNumber" and a single row containing the value "7110".

buildNumber
7110

Enabling indexer

Now, The last and most important step is remaining which can only be taken post-upgrade and to enable indexer. For this, you have to modify the file "**com.liferay.portal.search.configuration.IndexStatusManagerConfiguration.config**" in the folder [Liferay Home]/osgi/configs with the following content

```
indexReadOnly="false"
```

Good job, You're almost at the end. Now let's start Liferay.

Step 3: Code migration

- **Ready For Code migrate:** - Once your database will migrate to DXP you need to update your existing code to make it compatible with DXP. Liferay provides code upgrade tool for it.

- For JSP hook migration it's better to create a hook module from scratch and overwrite the required JSP page. The reason behind that is because Liferay JSPs have changed significantly from Liferay 6.x to 7.x, so using old 6.2 JSP pages for new 7.1 Liferay may break some functionality.

- 

- Theme yo liferay-theme:import
- You can follow LR's documentation to upgrade your existing code to LR DXP
- IV. Code migration
- For code migration, you need to set up a project environment. I use IntelliJ IDEA IDE with Liferay IntelliJ Plugin, Liferay 7.1 bundled with Tomcat, Maven for assembling, and MySQL as a database. All the instructions below are relevant for this stack.

- **IDE SETUP**

- I like this Liferay's liberal position about tools choice. It's up to a developer what tool to choose for generating and managing the project. Liferay's default Gradle management system doesn't suit me perfectly. So I preferred [Maven](#). There are several Maven plugins with artifacts which I can use for module development. A Liferay Workspace works well to create and manage a structure for folds and files built with Maven.
- First, I created a new Liferay project using Liferay Maven Plugin:

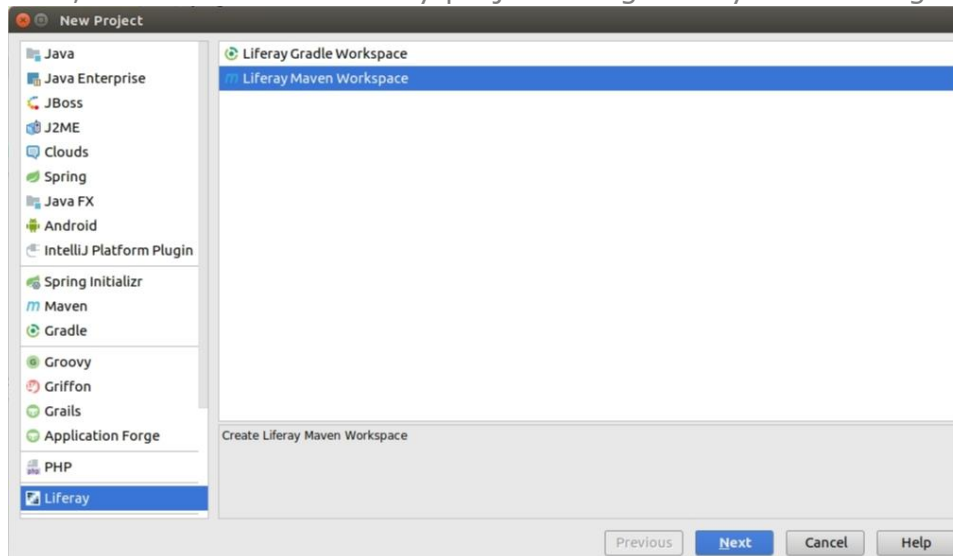


Image 18.

Liferay Maven Plugin

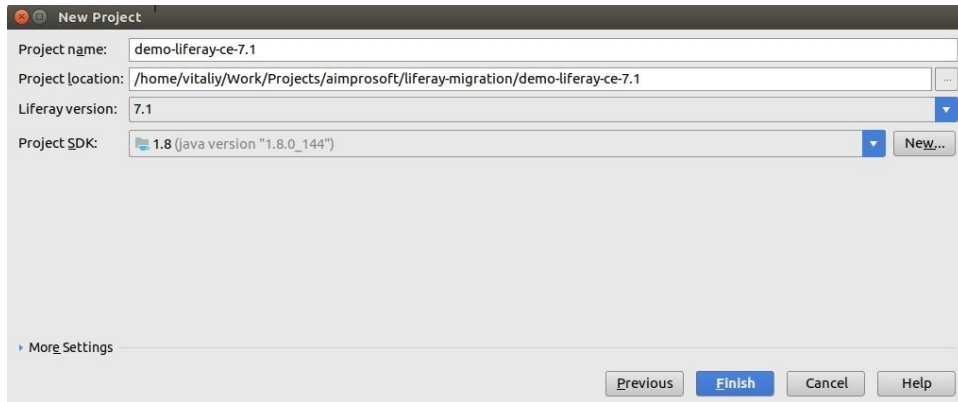


Image 19.

Liferay Maven Plugin_Liferay 7.1

- When you have a new project done, you will see the following structure displayed:

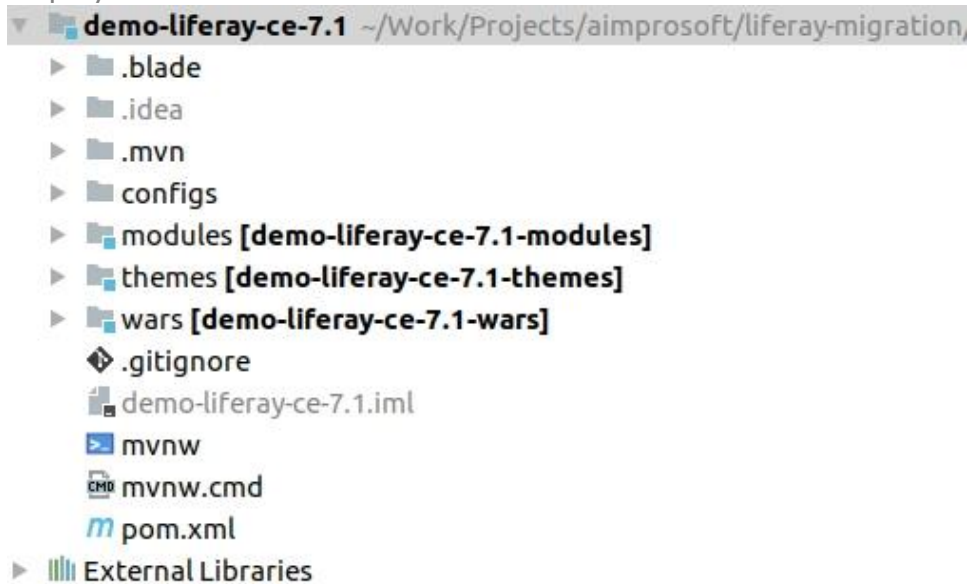


Image 20.

Liferay migration

- Go on with configuring a Liferay server in IDE as I did:

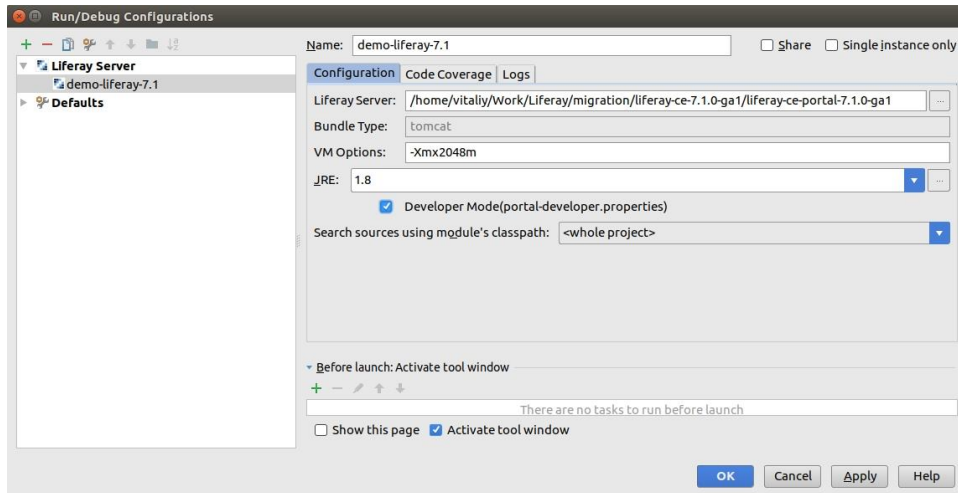


Image 21.

- Liferay server configuration in IDE
- Congrats! Now you can start Liferay 7.1 portal from IDE.
- Liferay theme migration
- Before starting theme migration, we should consider the main points which were changed in Liferay 7.1:
 - Velocity templates are no longer supported in favor of FreeMarker;
 - Dockbar has been replaced with different menus: Product Menu, Control Menu, User Personal Bar.
- Ways to update theme:
- `yo liferay-theme:import` Note: this way works only for themes, created with Ant-base Liferay plugin SDK. Otherwise, such exception will appear:

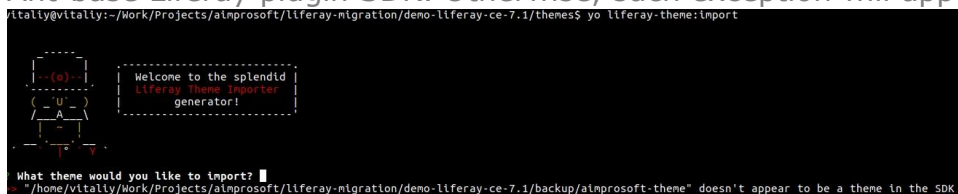
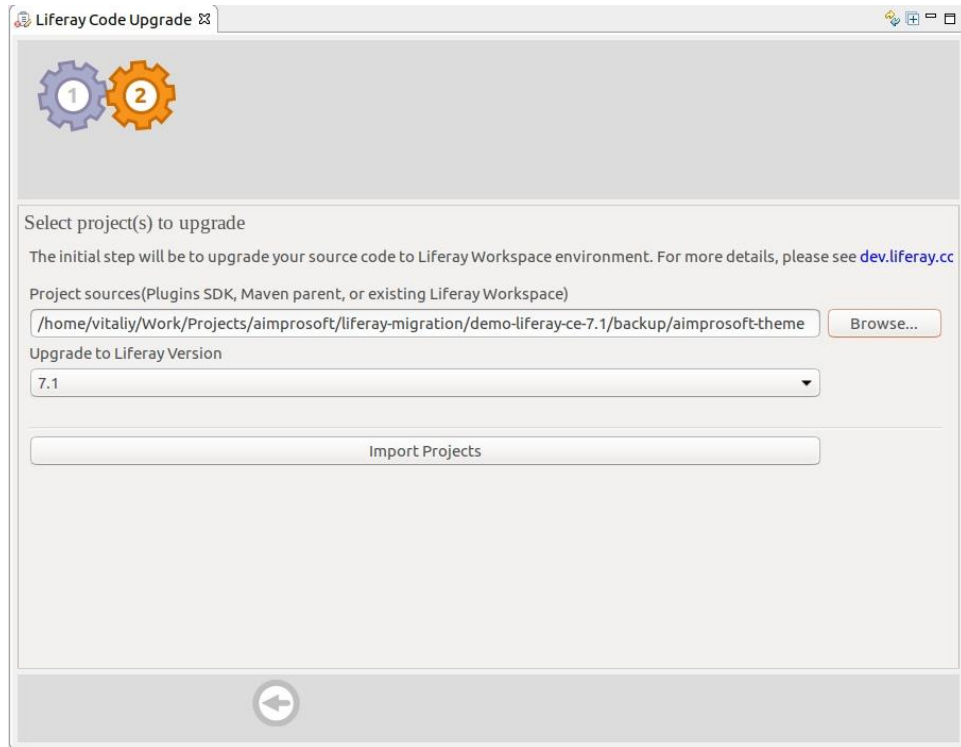


Image 22.

Liferay Theme Importer

- Run Gulp upgrade task to upgrade the theme:
- `yo liferay-theme:import`
- **Note:** this way works only for themes, created with Ant-base Liferay plugin SDK.
- Otherwise, such exception will appear:
- If you don't like a dry sense of humor by Darth Vader, just follow my tips.
 - Use Liferay IDE's Code Upgrade Tool:



Image

23. Liferay code upgrade

- While upgrade tool automates the migrations process, there are still restrictions for theme structure.
- Manual theme creation and code transfer
- Although it's not an automatic process and needs some efforts. Sometimes it's easier to [create a theme from scratch in 7.1](#) and apply appropriate styling from 6.2 given changes in Liferay between those versions.
- Well, let's try to create a new theme using Liferay plugin:

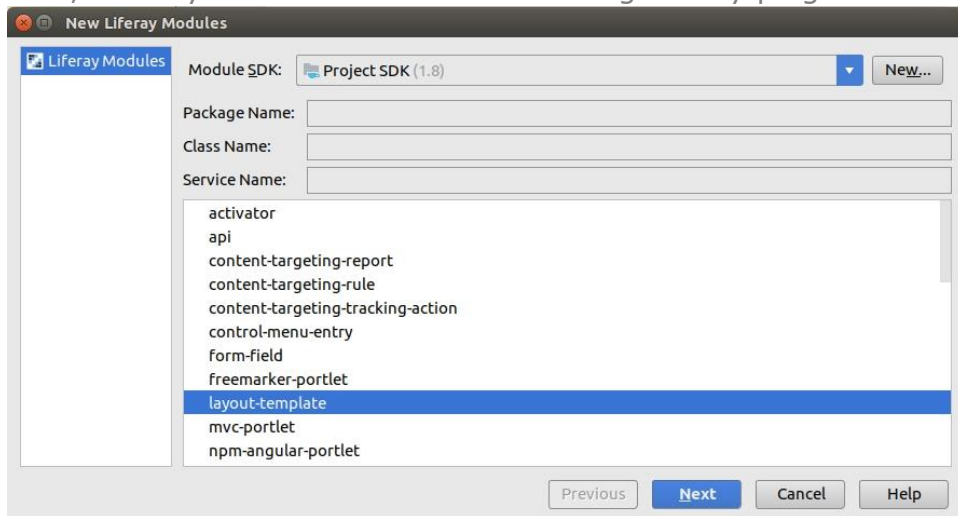


Image 24.

Liferay layout migration

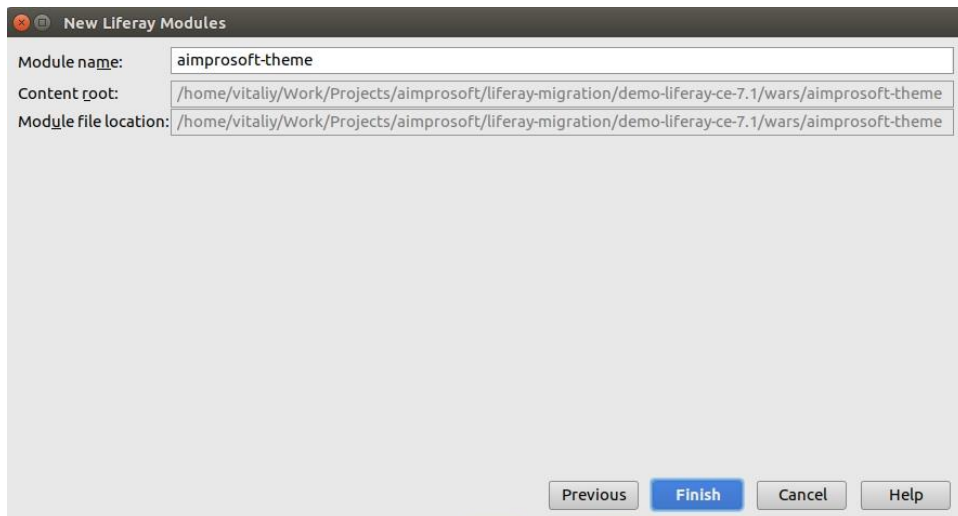


Image 25.

Liferay theme creation_Aimprosoft

- Score! Now you've got an empty, but working theme created. After this, we can apply changes from 6.2 theme taking into account changes in 7.1.
- As in version 6.2, we can also use a classic theme as an example and overwrite required sections (FTL templates, CSS files, etc.).
- Liferay portlet migration
- To migrate portlet code, I used the Code Upgrade Tool in Liferay Developer Studio (Project -> Liferay Code Upgrade Tool menu). After importing project we can click "Find breaking changes" button, and a list of code problems will appear:

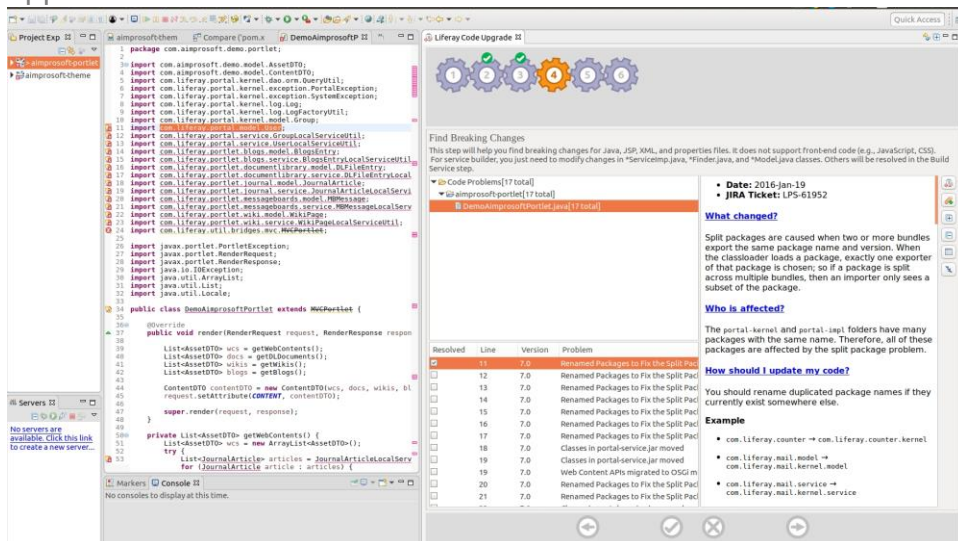


Image 26.

Liferay portlet migration

- Each of them contains API changes explanation, which affects custom code, reasons for it, links to Liferay JIRA tickets, examples and recommendations to fix code issues. Some of code problems can be fixed automatically using Liferay IDE, other ones need manual actions.

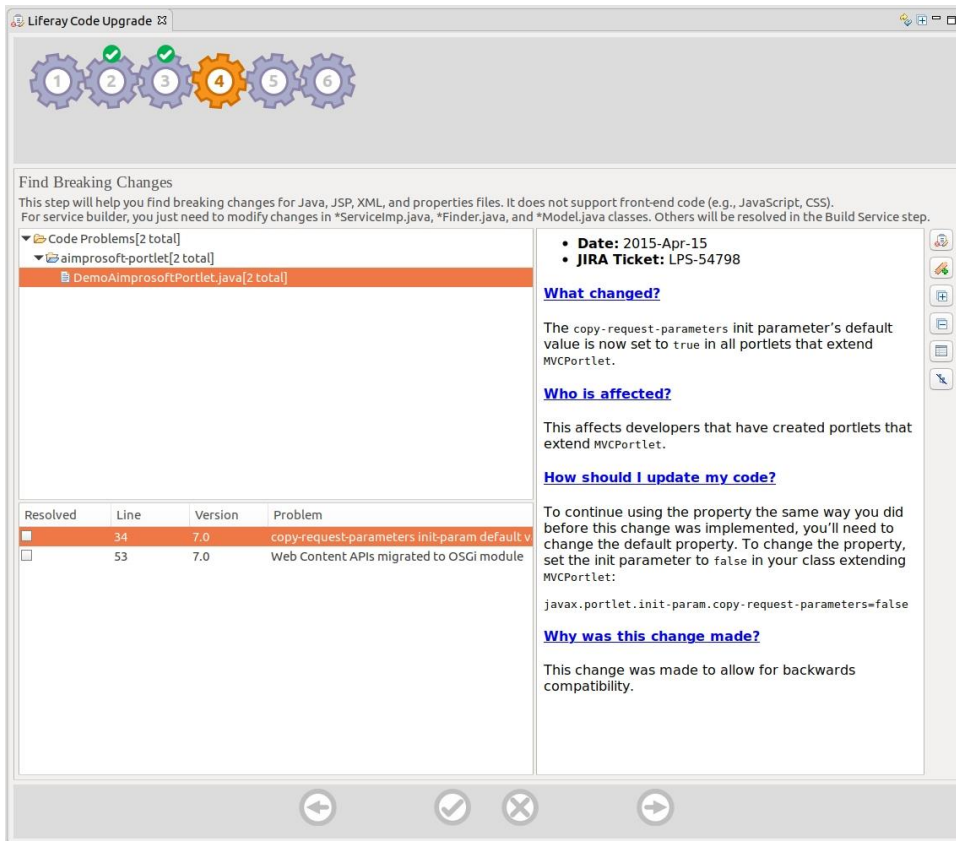


Image 27.

- How to fix Liferay code migration issues
- After fixing all the issues, you can try to build the portlet for Liferay 7.1. If you work in IntelliJ IDEA, you can copy migrated code to it from Liferay IDE.
- Liferay hook migration
- For hook migration (especially for JSP hooks) it's better to create a hook module from scratch and overwrite the required JSP page. As Liferay's JSP pages have changed significantly from Liferay 6.x to 7.x, using old 6.2 JSP pages for new 7.1 Liferay may break some functionality.
- So, you need to create a hook module, find appropriate JSP, and overwrite it in hook, copying changes from 6.2.
- To find what exactly was changed in hook, you can compare 2 JSP files in your Liferay 6.2 instance, xxx.jsp (hooked one) and xxx.portal.jsp (original one). In my example:



Image 28.

Liferay hook migrations

- Then just apply these changes to your Liferay 7.1 page and deploy the hook.
- Liferay layout migration
- For layout migration we can create a new layout with Liferay IntelliJ plugin:

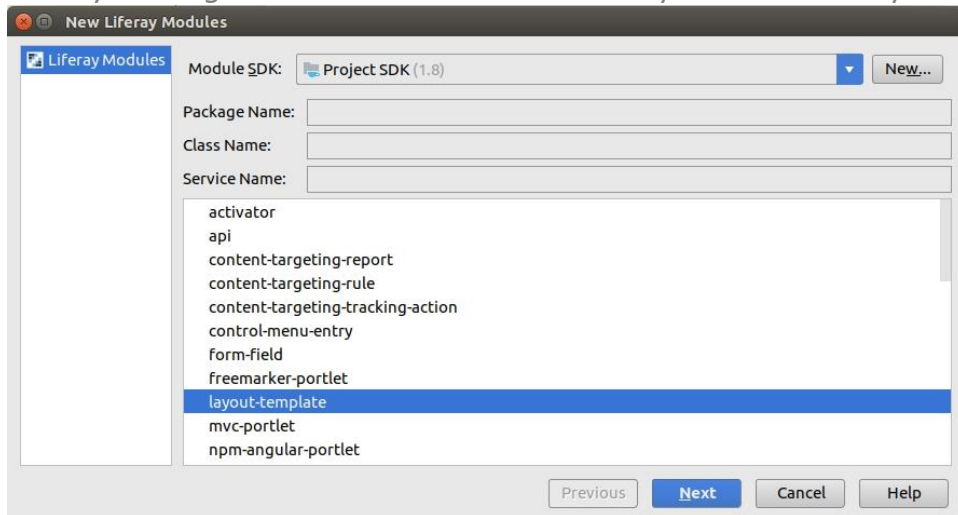


Image 29.

Liferay layout migration

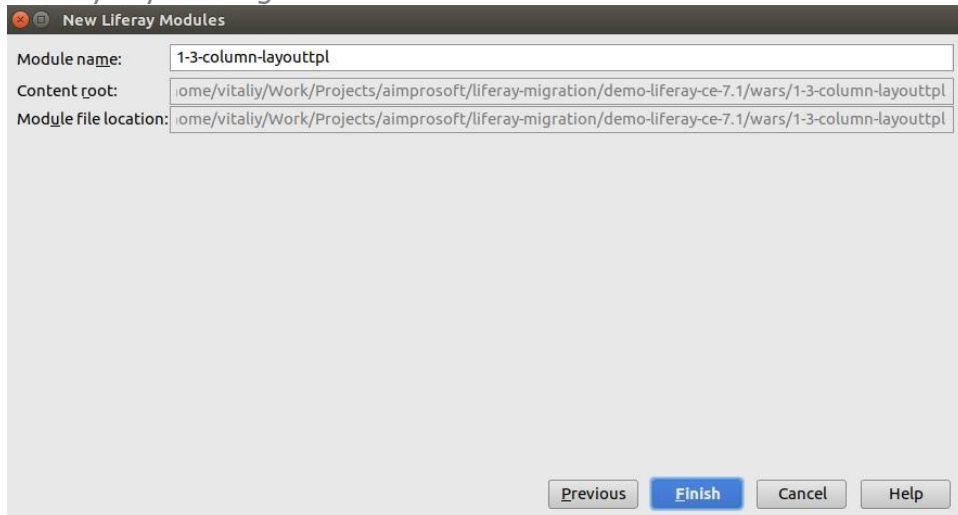


Image 30.

Liferay column layout migration

- After that we can adjust FTL template according to the 6.2 code, for example:



Image 31.

Liferay FTL template

- Now you are free to change an icon for layout, deploy the layout, and use it.
-

Step 4: Change in Deployment Process

- As we know Liferay DXP adopted OSGi framework container which is based on modules based application. Due to this reason deployment process in LR has been changed.
- You can still use WARs in liferay but it is not recommended as you lose access to any services which are deployed in OSGI container. If you deployed WAR to OSGI container LR will convert it to WAB which is web archive bundles. This will be recommended for deploying legacy applications.
- It's better to use bundles which are nothing but JARs with OSGI metadata.

Step 5: Post-migration issues

- Once code and database will be migrated to LR DXP, please make sure to regenerate the search index after the upgrade.
- Here I've added some of the common post migration issues which generally happen.

You must first upgrade to Liferay Portal 7100

- This exception will be thrown when you started LR 7.x with 6.x database. To resolve this issue you need to re-run the database upgrade tool before starting LR DXP.

PwdEncryptorException: invalid keyLength value

- You will face this kind of exception when an incorrect password encryption algorithm. To resolve this error you need to add below property in the portal.properties file.
- `passwords.encryption.algorithm.legacy=SHA`

Step 6: Search Engine

- From Liferay DXP, they have integrated elasticsearch as an embedded search engine. For production use, Liferay only supports Elasticsearch when it runs in a separate JVM. Liferay recommends that elasticsearch servers should be provisioned with a minimum of 8 CPU cores and 16GB of memory.
- If you need to use Solr, it's also supported in Liferay Portal. Basically, you will require a separate search server to maintain the whole portal to work smoothly.

I hope that this will help you to migrate your portal from 6.2 to DXP. If you still have any queries we provide service of the Liferay portal upgrade from older versions to newer versions of Liferay DXP.

Types of Hook

explain you about JSP hook let me summarize list of hooks Liferay 7/DXP offers,

- Struts Action Hook
- [Filter Hook](#)
- JSP Hook
- Language properties Hook
- Model Listener hook
- Service Wrapper Hook

JSP HOOK

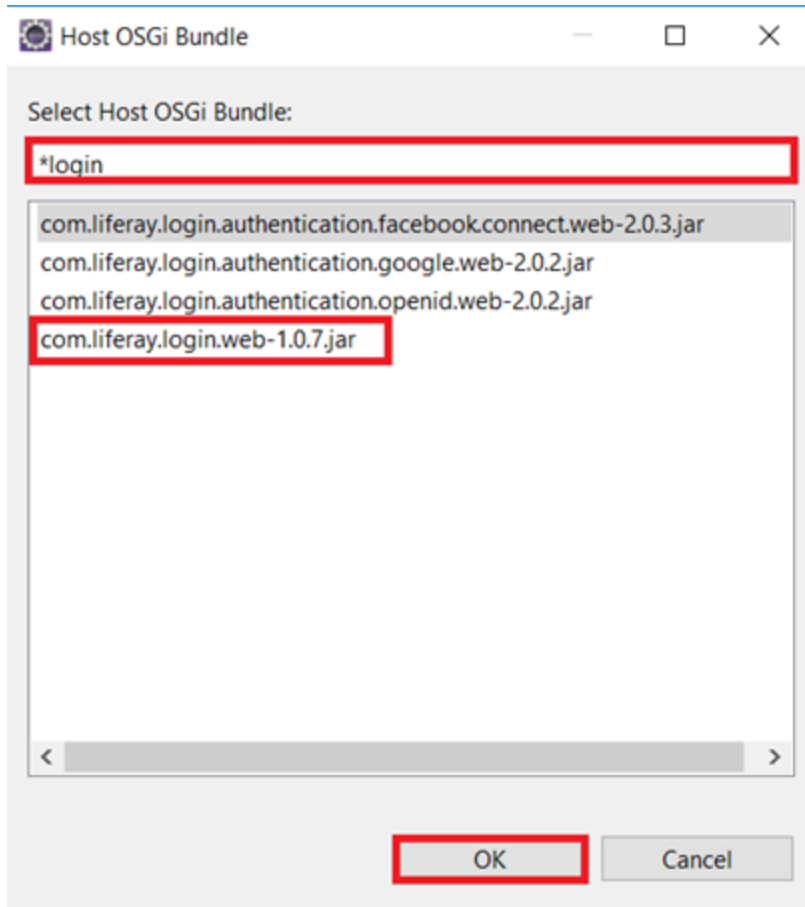
Open File menu → Click on New -> select Liferay Module Fragment Project

- Provide project name.

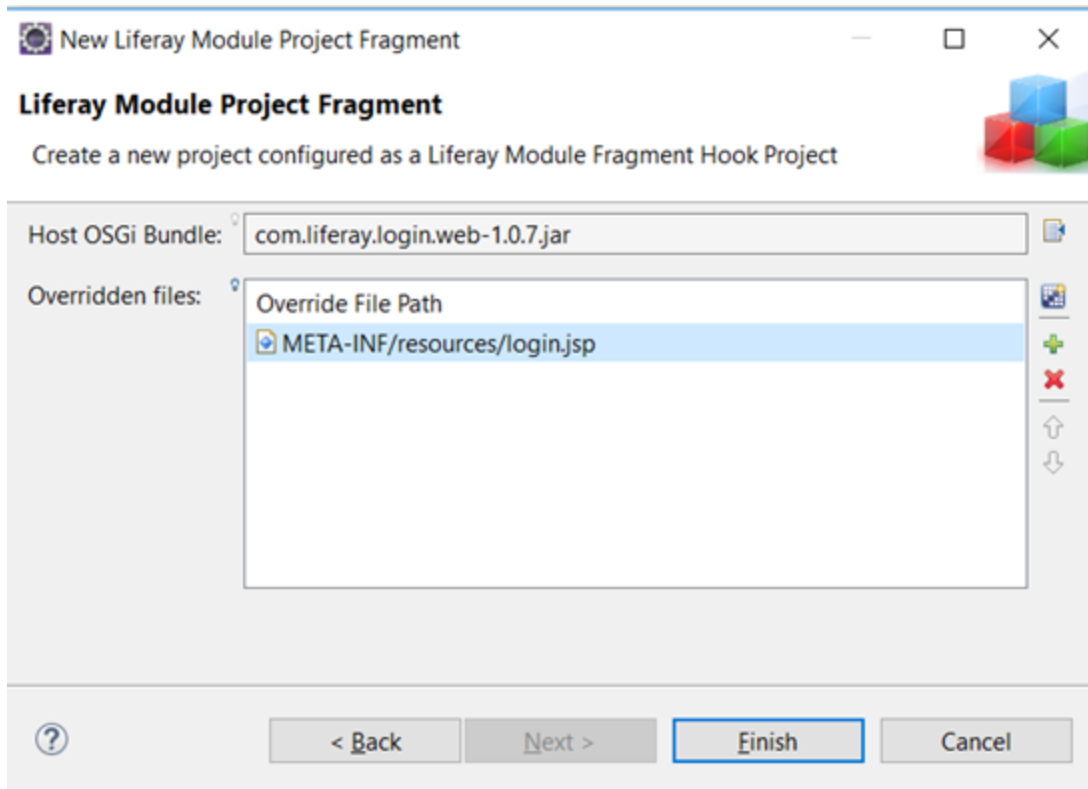
Select OSGI Bundle:

in this post I am extending login.jsp so you need to select relevant OSGI bundle.

You can search OSGI bundle using wild card search from Liferay IDE as shown in following screenshot,

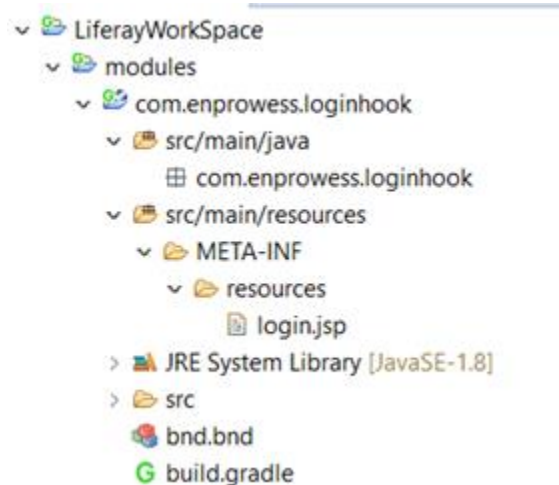


Select **com.liferay.login.web-1.0.7.jar** OSGi bundle for this example and click on 'OK'.



Select JSP:

select JSP you want to override and click on finish. Here I selected login.jsp. see following screenshot,



After you click on finish, you successfully created Liferay module. You should see folder structure as follow,

Edit JSP:

make the change you want in JSP and save it.

Filter Hook

Open File menu → click New → select Liferay Module Project. Provide project name as 'FilterHook' and click on 'Next'.

New Liferay Module Project

Liferay Module Project
Enter a name and choose a template to use for a new Liferay module.

Project name:

☒ Use default location

Location:

Build type:

Project Template Name:

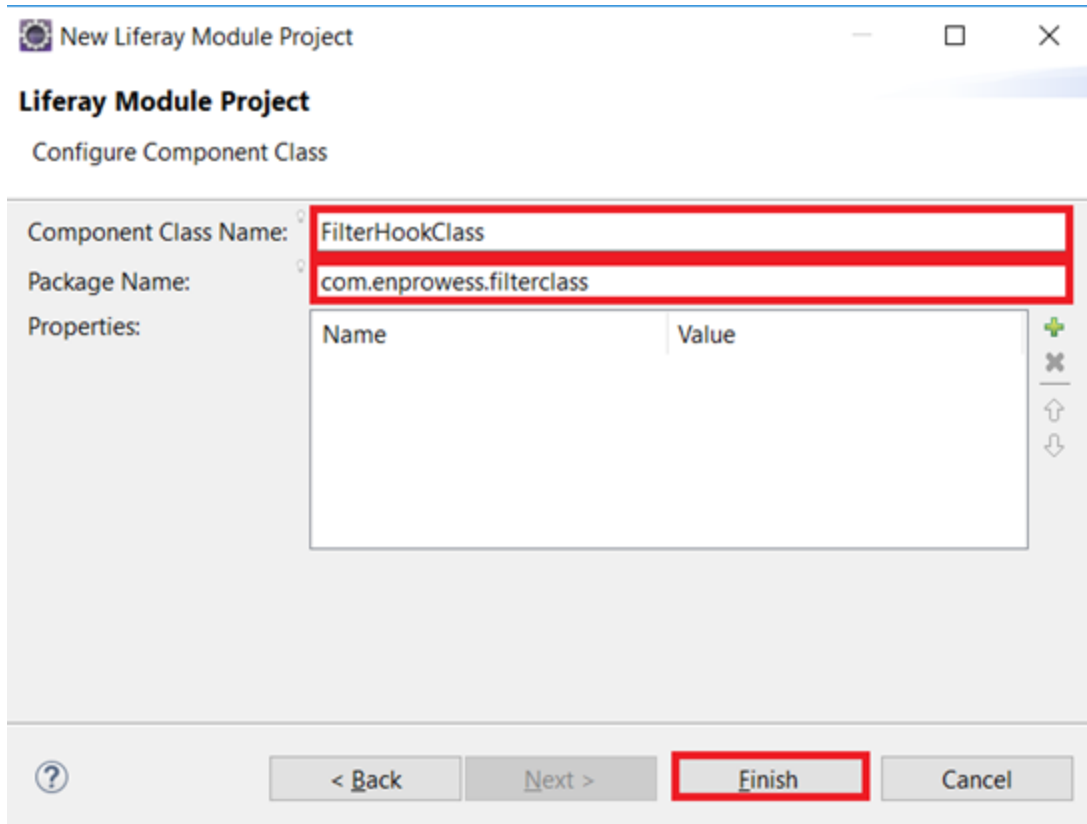
☐ Add project to working set

Working set:

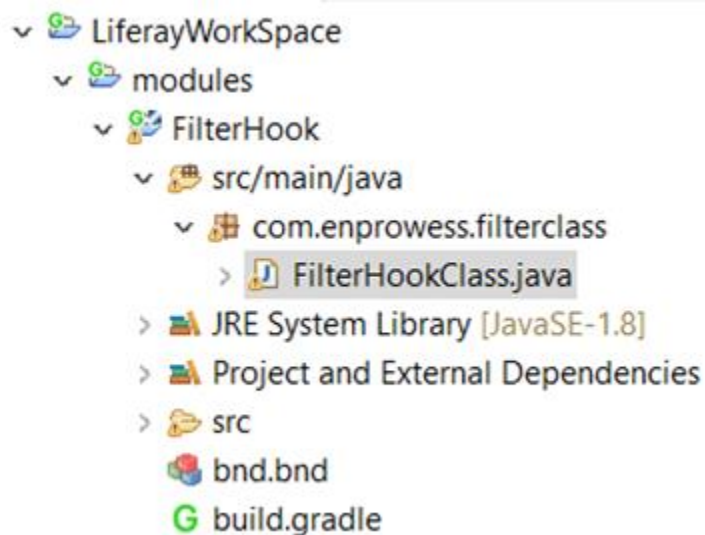
1.

Configure hook properties:

Provide component name and package path for the filter class and click on Finish.



Now class FilterHook with package com.enprowess.filterclass is created in workspace module as follow,



2.

Implement Component:

In class, you need to add following property inside @Component.

```
@Component(immediate = true, property = { "servlet-context-name=", "servlet-filter-name=Custom Filter",  
    "url-pattern=/web/guest/home" // Your URL pattern  
}, service = Filter.class)
```

3.

Implement code:

Please note that your filter class will extend BaseFilter class. You will need to override processFilter() method which will take parameters as HttpServletRequest request, HttpServletResponse response, FilterChain filterChain. Please see following piece of code.

```
public class FilterHookClass extends BaseFilter {  
  
    private static final Log _Log = LogFactoryUtil.getLog(FilterHookClass.class);  
  
    @Override  
    protected Log getLog() {  
        // TODO Auto-generated method stub  
        return _Log;  
    }  
  
    @Override  
    protected void processFilter(HttpServletRequest request, HttpServletResponse response, FilterChain filterChain)  
        throws Exception {  
  
        _Log.info("Filter is invoked at /web/guest/home url pattern match");  
        super.processFilter(request, response, filterChain);  
    }  
}
```

Now, you have filter hook ready to deploy in Liferay.

Struts Action Hook

Click on File menu → Click New → Select Liferay Module Project and provide project name as StrutsActionHook and click next. See following screenshot for your reference,

New Liferay Module Project

Liferay Module Project

Enter a name and choose a template to use for a new Liferay module.

Project name:

☒ Use default location

Location:

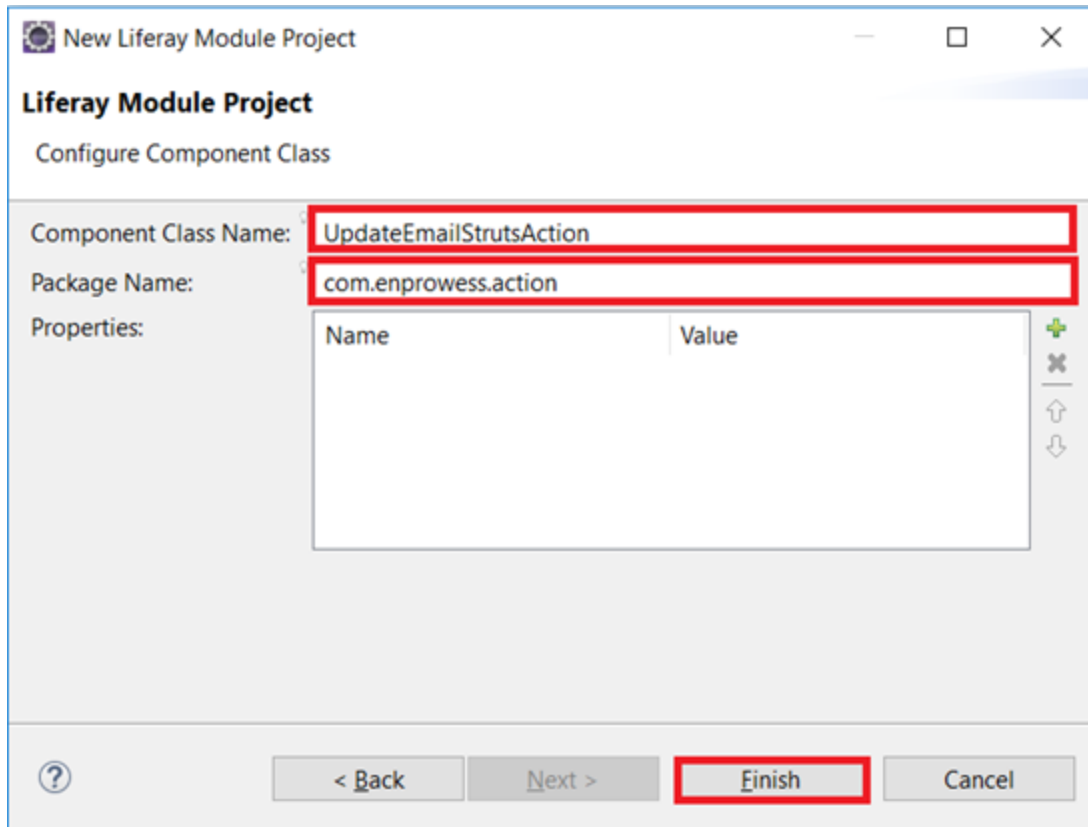
Build type:

Project Template Name:

☐ Add project to working set

Working set: More...

Provide appropriate class name and package name for the class. For this example, I am providing class name as UpdateEmailStrutsAction.



After you finish the wizard, you should see project structure in eclipse as follow,

- ▼ StrutsActionHook
 - ▼ src/main/java
 - ▼ com.enprowess.action
 - > UpdateEmailStrutsAction.java
 - > src/main/resources
 - > JRE System Library [JavaSE-1.8]
 - > Project and External Dependencies
 - > src
 - bnd.bnd
 - build.gradle

4.

Implement Component

:

You successfully created project structure. Now it's time to implement business logic to update email address. Implement @Component as follow in class,

```
@Component(  
    immediate=true,  
    property={  
        "path=/portal/update_email_address"  
    },  
    service = StrutsAction.class  
)
```

5.

Override execute method

:

Ensure that your class extends BaseStrutsAction class. To implement business logic override, execute() method as follow,

```
public class UpdateEmailStrutsAction extends BaseStrutsAction {  
  
    private static final Log _log = LogFactoryUtil.getLog(UpdateEmailStrutsAction.class);  
  
    @Override  
    public String execute(StrutsAction originalStrutsAction, HttpServletRequest request, HttpServletResponse response)  
        throws Exception {  
        _log.info("Calling Struts action at update email address");  
        // you logic goes here  
        return super.execute(originalStrutsAction, request, response);  
    }  
}
```

Congratulation! You successfully implemented struts hook. Now let's quickly understand what you exactly implemented.

In @Component, you configured struts action i.e. `"/portal/update_email_address"` to instruct container that when user trigger action to update email, please execute business logic mentioned in my class.

In case you want to override different action then mentioned in this example, You can see `'struts-config.xml'` file to know all out-of-the-box struts actions. You can find this file on following path in your IDE.

tomcat server → liferay-ce-portal-7.0-ga3 → tomcat-8.0.32 → webapps → ROOT → WEB-INF → struts-config.xml.

- In Liferay Plugin SDK development, we use HOOK / EXT to customize liferay functionalities.
- In Liferay DXP, technically there is no HOOK/EXT concepts any more, but Liferay leverages OSGI platform to modify the core services, filters, indexes, JSP's, Language properties

Liferay 7 Hooks Tutorials

- In Liferay Plugin SDK development, we use HOOK / EXT to customize liferay functionalities.
- In Liferay DXP, technically there is no HOOK/EXT concepts any more, but Liferay leverages OSGI platform to modify the core services, filters, indexes, JSP's, Language properties.
- As you all know, OSGI uses Service Registry modal that lookup service implementation at run time to identify the implementation.
- Liferay takes OSGI framework as advantage to customize liferay functionality
 - OSGI Service Registry – OSGI uses Service Registry concepts and provides service implementation at runtime.
 - Extender pattern will add extension code to already existing bundles via service modal. Liferay's Portlet Container and when a OSGI component has (service = Portlet.class) then it will add to the Portlet Container bundle). In the same way, ServiceWrapper, Filter, Struts Actions, PortalActions, Modal Listeners, Indexer will be customized.
 - Fragment Host:
 - A fragment bundle will be created to modify the contents in core module (called as host bundle).
 - The resources (JSP pages in resource folder) defined in fragment host are replaced with resources in Host bundle.
 - Let's take example:
 - Liferay login-web bundle – A Host bundle contains login.jsp, forgot_password.jsp in resource folder
 - login-hook-web bundle: it is fragment host bundle where resource defined in this bundle are replaced with login-web bundle. We need to add fragment host entry in MANIFEST.MF file to override the Host bundle resources.

Liferay follows same kind of pattern to override the core functionalities such as services, modal listeners, filters, actions like below:

1. Create Component class

2. Add the service class name property in @Component annotation
 - a. **@Component(property = {for language,filters }, service = {BaseClass.class})**
3. Component class is required to extend/implement corresponding class/interface.
We will see complete tutorials in the following sections.

Liferay 7 JSP HOOK:

Liferay Uses OSGI fragment concepts to override liferay module JSP's.

- Create Module
- Add Fragment-Host:{bundle-name}
 - a. Manifest-Version: 1.0Bnd-LastModified: 1483624888998Bundle-ManifestVersion: 2Bundle-Name: org.javasavvy.demo.hooksBundle-SymbolicName: org.javasavvy.demo.hooksBundle-Version: 1.0.0Created-By: 1.8.0_101 (Oracle Corporation)Fragment-Host: com.liferay.login.web;bundle-version="1.1.3"
- copy the JSP page from Host bundle(Actual bundle) to Fragmented Bundle
- [Click here to access for complete tutorial: Liferay 7 JSP Hook tutorial](#)

Liferay 7 Language Properties Hook Configuration:

- Create Component class
- add the below configuration in to @Component annotation
 - b. **@Component(property = { "language.id=en_US" }, service = ResourceBundle.class)**
- Class Need to extend ResourceBundle
- [Click here to access Liferay 7 Language Hook.](#)

Liferay 7 Services Hook Configuration:

- Create component class
- Add the below annotation config in @Component annotations
 - **@Component(immediate=true, service = ServiceWrapper.class)**
- Class need to extend the {entity}ServiceWrapper.java (UserServiceWrapper,JournalArticalServiceWrapper) and override the required methods
- [Click here to access Liferay 7 Service Customization Hook tutorial](#)

Liferay 7 Struts action Hook Configuration:

- Create component class
- Add the below annotation config in @Component annotations
 - **@Component(immediate=true, property={ "path={struts-action-path}" }, service = StrutsAction.class)**
- Class need to extend the BaseStrutsAction class and override the execute method
- [Click here to access Liferay 7 Struts Action tutorial](#)

Liferay 7 Action Commands Hook Configuration:

- Create component class

- Add the below annotation config in @Component annotations
 - **@Component(immediate = true, property = {
"javax.portlet.name=com_liferay_login_web_portlet_LoginPortlet",
"mvc.command.name=/login/login", "service.ranking:Integer=100" }, service =
MVCActionCommand.class)**
- service.ranking:Integer is mandatory to override action command
- Class need to extend the class and override the doProcessAction or ProcessAction methods
- [Click here to access Liferay 7 Action command Tutorial](#)

Liferay 7 Modal Listener Hook Configuration:

- Create component class
- Add the below annotation config in @Component annotations
 - **@Component(immediate = true, service = ModelListener.class)**
- Class need to extend the class and override the required method
- [Click here to access Liferay 7 Modal Listener Hook tutorial](#)

Liferay 7 Filter Hook Configuration:

- Create component class
- Add the below annotation config in @Component annotations and update the url path in "url-pattern":
 - **@Component(immediate = true, property = { "servlet-context-name=", "servlet-filter-name=Custom Filter", "url-pattern=/*" }, service = Filter.class)**
- Class need to extend the BaseFilter class and override the required method: processFilter()

4 Customize search liferay

Customizing Liferay Search

There are several extension points available for users to customize. The most obvious is the ability to add a new search engine adapter.

Adding a new Search Engine Adapter

To add a new search engine adapter, developers must create the following components and publish them to Liferay's OSGi registry:

4. Implement a new `IndexSearcher` that should convert the Liferay Search objects to the underlying search engine's dialects:
 - `QueryTranslator`: Translates Liferay Queries to the native search engine's queries.
 - `FilterTranslator`: Translates Liferay filters into native search engine's filters.
 - `GroupByTranslator`: Translates the GroupBy aggregation to the search engine's group by top hits aggregation.
 - `StatsTranslator`: Translates Stats request to the appropriate search engine's statistics aggregation.
 - `SuggesterTranslator`: Translates suggestion requests to the appropriate search engine's suggester API.
5. Implement a new `IndexWriter` that should
 - Convert the Liferay Document to a format understood by the underlying search engine's Document Format.
 - Use the search engine's API to update, add, and delete documents.

6. Implement a new `SearchEngineConfigurator` that should extend `AbstractSearchEngineConfigurator` to perform proper wiring.
7. Implement a new `SearchEngine` that
 - Should extend from `BaseSearchEngine` to perform any search engine specific initialization.
 - Should be published to Liferay's OSGi registry along with the property `search.engine.id=[searchEngineId]`.

Customizing IndexerRequestBufferOverflowHandler

`IndexerRequestBufferOverflowHandler` controls how the search infrastructure handles situations where buffered indexer requests has exceeded the configured maximum buffer size.

To customize, implement an `IndexerRequestBufferOverflowHandler` and publish it to Liferay's OSGi registry.

Customizing HitsProcessors

`com.liferay.portal.kernel.hits.HitsProcessor` objects are held in a `com.liferay.portal.kernel.hits.HitsProcessorRegistry`. To add a new `HitsProcessor`, simply implement the interface and publish to the OSGi registry with the property `sort.order`.

5 Indexer

```
@Component( immediate = true, property = {
"indexer.class.name=com.liferay.portal.kernel.model.User",
"indexer.class.name=com.liferay.portal.kernel.model.UserGroup" }, service =
IndexerPostProcessor.class)
```

6 Action Hook

7 Service Hook

8 Model Listener

- Model Listener is used to listen to Liferay Model Events. We can perform custom action on Before/After the persistence event of Model objects using Model Listener.
- Model Listeners implement **ModelListener** interface. They are useful in terms of processing, custom login on Model objects while they are

added/updated/removed. For ex, if we want to perform any specific action before User is created then we can implement the Model Listener for User model and perform actions on **onBeforeCreate** method.

- This would be invoked before persistence layer's create method.

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

```
public class MyCustomRoleModelListenerPortlet extends BaseModelListener<Role> {
```

```
@Override
```

```
public void onAfterCreate(Role model) throws ModelListenerException {
```

```
@Override
```

```
public void onBeforeCreate(Role model) throws ModelListenerException {
```


9 Service Builder

10 Portlet Filter

Portlet Filter is used to intercept and manipulate the request and response before it is delivered to the portlet at given life cycle such as action, render and resource

What Is Portlet Filter?

Portlet Filter is used to intercept and manipulate the request and response before it is delivered to the portlet at given life cycle such as action, render and resource

Types Of Filters For A Liferay Portlet

Action Filter

Render Filter

Resource Filter

Base Filter

Which Filter To Use?

Action Filter : To Intercept only The Action Requests

Render Filter : To Intercept only The Render Requests

Resource Filter : To Intercept only The Resource Requests

Base Filter : To Intercept all the http request comes to your portlet

Steps To Create Filter

- Create A Portlet Filter Component Class
You can create Portlet Filter Component Class Using Eclipse, by right clicking on you portlet module project and select select "New Component Class" and Select "Portlet Filter" Component Template, I prefer to create a class manually for now, I Am creating a Class named "MyRenderFilter"
- **Step 3** : Component Declaration
Declare the Class as an OSGI Component Using the Below Code Snippet

```
import org.osgi.service.component.annotations.Component;import
javax.portlet.filter.PortletFilter;@Component(immediate = true, property = {
"javax.portlet.name="+MyPortletKeys.My, }, service = PortletFilter.class)Note : Make
sure you give correct portlet name and service must be PortletFilter.class
```

Step 4 : Extend The RenderFilter

```
import javax.portlet.filter.RenderFilter;public class MyRenderFilter implements
RenderFilter{}
```

Step 5 : Implement Unimplemented Methods of the Parent Class RenderFilter

```
public void init();public void destroy();public void doFilter();
```

Complete MyRenderFilter.java Code

- package com.liferayystack.filter;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.PortletFilter;import
javax.portlet.filter.RenderFilter;import com.liferay.portal.kernel.log.Log;import
com.liferay.portal.kernel.log.LogFactoryUtil;import
com.liferayystack.constants.MyPortletKeys;import
org.osgi.service.component.annotations.Component;/** * @author Syed Ali
*/@Component(immediate = true, property =
{"javax.portlet.name="+MyPortletKeys.My, },service = PortletFilter.class)public class
MyRenderFilter implements RenderFilter{private static Log _log =
LogFactoryUtil.getLog(MyRenderFilter.class);@Overridepublic void init(FilterConfig
filterConfig) throws PortletException {_log.info("init.....");}@Overridepublic void
destroy() {_log.info("destroy.....");}@Overridepublic void doFilter(RenderRequest
request, RenderResponse response, FilterChain chain) throws IOException,
PortletException {_log.info("RenderDoFilter.....");chain.doFilter(request, response);}}

Each time when you Hit a Render URL doFilter() method the MyRenderFilter will be triggered.

• Portlet Filter For Action Request

- Similarly You can create the filter for all Action Request By extending ActionFilter Class, as shown below

- package com.liferayystack.filter;import java.io.IOException;import
javax.portlet.ActionRequest;import javax.portlet.ActionResponse;import
javax.portlet.PortletException;import javax.portlet.filter.ActionFilter;import
javax.portlet.filter.FilterChain;import javax.portlet.filter.FilterConfig;import
javax.portlet.filter.PortletFilter;import com.liferay.portal.kernel.log.Log;import
com.liferay.portal.kernel.log.LogFactoryUtil;import
com.liferayystack.constants.MyPortletKeys;import
org.osgi.service.component.annotations.Component;/** * @author Syed Ali

```

*/@Component(immediate = true, property =
{"javax.portlet.name="+MyPortletKeys.My, },service = PortletFilter.class)public class
MyActionFilter implements ActionFilter{private static Log _log =
LogFactoryUtil.getLog(MyActionFilter.class);@Overridepublic void init(FilterConfig
filterConfig) throws PortletException {_log.info("init.....");}@Overridepublic void
destroy() {_log.info("destroy.....");}@Overridepublic void doFilter(ActionRequest
request, ActionResponse response, FilterChain chain) throws IOException,
PortletException {_log.info("ActionDoFilter.....");chain.doFilter(request, response);}}

```

-

• Portlet Filter For Resource Request

-

For all Resource request you can extend your class with ResourceFilter Class, as shown below

- package com.liferayystack.filter;import java.io.IOException;import javax.portlet.PortletException;import javax.portlet.ResourceRequest;import javax.portlet.ResourceResponse;import javax.portlet.filter.FilterChain;import javax.portlet.filter.FilterConfig;import javax.portlet.filter.PortletFilter;import javax.portlet.filter.ResourceFilter;import com.liferay.portal.kernel.log.Log;import com.liferay.portal.kernel.log.LogFactoryUtil;import com.liferayystack.constants.MyPortletKeys;import org.osgi.service.component.annotations.Component;/** * @author Syed Ali */@Component(immediate = true, property = {"javax.portlet.name="+MyPortletKeys.My, },service = PortletFilter.class)public class MyResourceFilter implements ResourceFilter{private static Log _log = LogFactoryUtil.getLog(MyResourceFilter.class);@Overridepublic void init(FilterConfig filterConfig) throws PortletException {_log.info("init.....");}@Overridepublic void destroy() {_log.info("destroy.....");}@Overridepublic void doFilter(ResourceRequest request, ResourceResponse response, FilterChain chain) throws IOException, PortletException {_log.info("ResourceDoFilter.....");chain.doFilter(request, response);}}

-

• Servlet Filter In Liferay 7 DXP

- This Filter Intercepts all the requests comes to the portlet, whether it is action, render, resource and it also can be HttpServletRequest , if you want filter all the Http Servlet Request comes to your portlet please use this filter, your filter component just have to extend "BaseFilter" Class, refer the below code

- package com.liferayystack.filter;import javax.servlet.Filter;import javax.servlet.FilterChain;import javax.servlet.http.HttpServletRequest;import javax.servlet.http.HttpServletResponse;import com.liferay.portal.kernel.log.Log;import com.liferay.portal.kernel.log.LogFactoryUtil;import com.liferay.portal.kernel.servlet.BaseFilter;import org.osgi.service.component.annotations.Component;/** * @author Syed Ali

```

*/@Component(immediate = true, property = {"servlet-context-name=", "servlet-filter-
name=http-filter", "url-pattern=/*" }, service = Filter.class)public class MyBaseFilter
extends BaseFilter{private static final Log _log =
LogFactoryUtil.getLog(MyBaseFilter.class);@Overrideprotected void
processFilter(HttpServletRequest request, HttpServletResponse response, FilterChain
filterChain) throws Exception {_log.info("processFilter
BaseFilter.....");filterChain.doFilter(request, response);super.processFilter(request,
response, filterChain);}@Overrideprotected Log getLog() {return _log;}}

```

-

Note : This BaseFilter Will be Triggered Before all the other filters such as ActionFilter, RenderFilter and ResourceFilter.

I have created a portlet which can easily make you understand flow of Portlet Filters and Portlet Requests In detail, please download the portlet and deploy to test, it can give more insight into the portlet Filters In Liferay, when you add this portlet to the page init() and doFilter() method will be triggered.

11 JSR 286 Standard

Java Portlet Specification (JSR 168 vs JSR 286)

Initially portal vendors had their own portlet frameworks but the problem is those portlets are specific to portal Server and cannot be deployed to another server. So their is specification or we can say Standards for portlet development. By using those standards you can write a portlet which can be run on all compliant portlet Containers.

There are basically two JSR standard for portals:-

1) Java portlet specification 1.0 ie JSR-168

JSR-168 released in 2003 and give standards for :-

- Portlet API.
- Portlet Taglib
- Portlet Life Cycle
- Window States
- View Preferences

This specification introduced two phases:

- Render Phase
- Action Phase

2) Java portlet specification 2.0 ie JSR-286

This specification introduced two extra phases so now we have 4 phases:-

- Render Phase
- Action Phase
- Event Phase for IPC
- Resource Serving phase for Ajax calls.

Features in JSR-286:-

- 1)Portlet filter is introduced .
- 2)Annotation support in Generic portlet.
- 3)Generate non mark up content like images, pdf using Resource serving phase.
- 4)Interportlet Communication is easily handled using Events.

12 MVC Command Render override

MVC Commands are used to break up the controller layer of a Liferay MVC application into smaller, more digestible code chunks.

- Sometimes you'll want to override an MVC command, whether it's in a Liferay application or another Liferay MVC application whose source code you don't own.
- Since MVC commands are components registered in the OSGi runtime, you can simply publish your own component, and give it a higher service ranking. Your MVC command will then be invoked instead of the original one.

The logical way of breaking up the controller layer is to do it by portlet phase. The three MVC command classes you can override are

MVCActionCommand: An interface that allows the portlet to process a particular action request.

MVCRenderCommand: An interface that handles the render phase of the portlet.

MVCResourceCommand: An interface that allows the portlet to serve a resource.

Find more information about implementing each of these MVC command classes in the tutorials on Liferay MVC Portlets. Here we're going to focus on overriding the logic contained in existing MVC commands.

- Note: While it's possible to copy the logic from an existing MVC command into your override class, then customize it to your liking,
- it's strongly recommended to decouple the original logic from your override logic.
- Keeping the override logic separate from the original logic will keep the code clean, maintainable, and easy to understand.

To do this, use the `@Reference` method to fetch a reference to the original MVC command component. If there are no additional customizations on the same command, this reference will be the original MVC command.

```
@Reference(
```

```
    target =  
    "(component.name=com.liferay.blogs.web.internal.portlet.action.EditEntryMVCRend  
erCommand)")
```

```
protected MVCRenderCommand mvcRenderCommand;
```

Set the `component.name` target to the MVC command class name. If you use this approach, your extension will continue to work with new versions of the original portlet, because no coupling exists between the original portlet logic and your customization. The command implementation class can change. Make sure to keep your reference updated to the name of the current implementation class.

Note: In Liferay DXP 7.0 GA1, there's a bug that occurs when modules with override MVC commands are removed from the OSGi runtime. Instead of looking for an MVC command with a lower service ranking (the original MVC command in most cases) to replace the removed one, the reference to the command is removed entirely. This bug is documented and fixed [here](#)

Start by learning to override `MVCRenderCommand`. The process will be similar for the other MVC commands.

Overriding `MVCRenderCommand`

You can override `MVCRenderCommand` for any portlet that uses Liferay's MVC framework and publishes an `MVCRenderCommand` component.

For example, Liferay's Blogs application has a class called `EditEntryMVCRenderCommand`, with this component:

```
@Component(
```

```
    immediate = true,
```

```
    property = {
```

```
        "javax.portlet.name=" + BlogsPortletKeys.BLOGS,
```

```
        "javax.portlet.name=" + BlogsPortletKeys.BLOGS_ADMIN,
```

```
        "javax.portlet.name=" + BlogsPortletKeys.BLOGS_AGGREGATOR,
```



```

        "mvc.command.name=/blogs/edit_entry"
    },
    service = MVCRenderCommand.class
)

```

This MVC render command can be invoked from any of the portlets specified by the `javax.portlet.name` parameter, by calling a render URL that names the MVC command.

```

<portlet:renderURL var="addEntryURL">
<portlet:param name="mvcRenderCommandName" value="/blogs/edit_entry" />
<portlet:param name="redirect" value="<%= viewEntriesURL %>" />
</portlet:renderURL>

```

What if you want to override the command, but not for all of the portlets listed in the original component? In your override component, just list the `javax.portlet.name` of the portlets where you want the override to take effect. For example, if you want to override the `/blogs/edit_entry` MVC render command just for the Blogs Admin portlet (the Blogs Application accessed in the site administration section of Liferay), your component could look like this:

```

@Component(
    immediate = true,
    property = {
        "javax.portlet.name=" + BlogsPortletKeys.BLOGS_ADMIN,
        "mvc.command.name=/blogs/edit_entry",
        "service.ranking:Integer=100"
    },
    service = MVCRenderCommand.class
)

```

Note the last property listed, `service.ranking`. It's used to tell the OSGi runtime which service to use, in cases where there are multiple components registering the same service, with the same properties. The higher the integer you specify here, the more weight your component carries. In this case, the override component will be used instead of the original one, since the default value for this property is 0.

After that, it's up to you to do whatever you'd like. You can add logic to the existing render method or redirect to an entirely new JSP.

ADDING LOGIC TO AN EXISTING MVC RENDER COMMAND

Don't copy the existing logic from the MVC render command into your override command class. This unnecessary duplication of code that makes maintenance more difficult. If you want to do something new (like set a request attribute) and then execute the logic in the original MVC render command, obtain a reference to the original command and call its render method like this:

```
@Override
```

```
public String render(RenderRequest renderRequest,  
                    RenderResponse renderResponse) throws PortletException {
```

```
    //Do something here
```

```
    return mvcRenderCommand.render(renderRequest, renderResponse);  
}
```

```
@Reference(target =
```

```
"(component.name=com.liferay.blogs.web.internal.portlet.action.EditEntryMVCRend  
erCommand)")
```

```
    protected MVCRRenderCommand mvcRenderCommand;  
}
```

Sometimes, you might need to redirect the request to an entirely new JSP that you'll place in your command override module.

REDIRECTING TO A NEW JSP

If you want to render an entirely new JSP, the process is different.

The render method of MVCRenderCommand returns the path to a JSP as a String. The JSP must live in the original module, so you cannot simply specify a path to a custom JSP in your override module. You need to make the method skip dispatching to the original JSP altogether, by using the MVC_PATH_VALUE_SKIP_DISPATCH constant from the MVCRenderConstants class. Then you need to initiate your own dispatching process, directing the request to your JSP path. Here's how that might look in practice:

```
public class CustomEditEntryMVCRenderCommand implements
MVCRenderCommand {

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

        System.out.println("Rendering custom_edit_entry.jsp");

        RequestDispatcher requestDispatcher =
            servletContext.getRequestDispatcher("/custom_edit_entry.jsp");

        try {
            HttpServletRequest httpServletRequest =
                PortalUtil.getHttpServletRequest(renderRequest);
            HttpServletResponse httpServletResponse =
                PortalUtil.getHttpServletResponse(renderResponse);

            requestDispatcher.include
                (httpServletRequest, httpServletResponse);
        } catch (Exception e) {
            throw new PortletException
```

```

        ("Unable to include custom_edit_entry.jsp", e);
    }

    return MVCRenderConstants.MVC_PATH_VALUE_SKIP_DISPATCH;
}

@Reference(target = "(osgi.web.symbolicname=com.custom.code.web)")
protected ServletContext servletContext;

}

```

In this approach, there's no reference to the original MVC render command because the original logic isn't reused. Instead, there's a reference to the servlet context of your module, which is needed to use the request dispatcher.

A servlet context is automatically created for portlets. It can be created for other modules by including the following line in your bnd.bnd file:

```
Web-ContextPath: /custom-code-web
```

Once we have the servlet context we just need to dispatch to the specific JSP in our own module.

Overriding MVCActionCommand

You can override MVC action commands using a similar process to the one presented above for MVC render commands. Again, you'll register a new OSGi component with the same properties, but with a higher service ranking. This time the service you're publishing is `MVCActionCommand.class`.

For MVC action command overrides, extend the `BaseMVCActionCommand` class, and the only method you'll need to override is `doProcessAction`, which must return void.

As with MVC render commands, you can add your logic to the original behavior of the action method by getting a reference to the original service, and calling it after

your own logic. Here's an example of an `MVCActionCommand` override that checks whether the delete action is invoked on a blog entry, and prints a message to the log, before continuing with the original processing:

```
@Component(  
    property = {  
        "javax.portlet.name=" + BlogsPortletKeys.BLOGS_ADMIN,  
        "mvc.command.name=/blogs/edit_entry",  
        "service.ranking:Integer=100"  
    },  
    service = MVCActionCommand.class)  
public class CustomBlogsMVCActionCommand extends BaseMVCActionCommand {  
  
    @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;
```

```
    }
```

It's straightforward to override MVC action commands while keeping your code decoupled from the original action methods. You can also override MVC resource commands.

Overriding MVCResourceCommand

There are fewer uses for overriding MVC resource commands, but it can also be done.

The process is similar to the one described for MVCRenderCommand and MVCActionCommand. There's a couple things to keep in mind:

The service to specify in your component is MVCResourceCommand.class

As with overriding MVCRenderCommand, there's no base implementation class to extend. You'll implement the interface yourself.

Keep your code decoupled from the original code by adding your logic to the original MVCResourceCommand's logic by getting a reference to the original and returning a call to its serveResource method:

```
        return mvcResourceCommand.serveResource(resourceRequest,  
        resourceResponse);
```

The following example overrides the behavior of com.liferay.login.web.portlet.action.CaptchaMVCResourceCommand, from the login-web module of the Login portlet. It simply prints a line in the console then executes the original logic: returning the Captcha image for the account creation screen.

```

@Component(
    property = {
        "javax.portlet.name=" + LoginPortletKeys.LOGIN,
        "mvc.command.name=/login/captcha" },
    service = MVCResourceCommand.class)
public class CustomCaptchaMVCResourceCommand implements
MVCResourceCommand {

    @Override
    public boolean serveResource
        (ResourceRequest resourceRequest, ResourceResponse resourceResponse) {

        System.out.println("Serving login captcha image");

        return mvcResourceCommand.serveResource(resourceRequest,
resourceResponse);
    }

    @Reference(target =
"(component.name=com.liferay.login.web.internal.portlet.action.CaptchaMVCResou
rceCommand)")
    protected MVCResourceCommand mvcResourceCommand;

}

```

And that, as they say, is that. Even if you don't own the source code of an application, you can override its MVC commands just by knowing the component class name.

13 How to connect multiple data source liferay?

Liferay 7 External Database ServiceBuilder Integration

ByJavasavvy

FEB 16, 2017 Liferay 7 custom datasource, Liferay 7 External Database ServiceBuilder Integration, Liferay 7 Service Builder External database config, liferay 7JNDI, Liferay DXP external Database, Liferay DXP External database integration, liferay JNDI, Remove term: Liferay 7 external Database Liferay 7 external Database

Liferay 7 External Database ServiceBuilder Integration

In this tutorial, we will see how to configure external database via service builder. If your application wants to communicate to external database to load the data into liferay system. you can follow below approaches

Custom Implementation : you can use hibernate or native JDBC to load the data into portal system.

Service Builder Implementation: Service builder generates you the business layer that includes cache, finder impl to load the data.

In this tutorial, we will see the Service builder approach to connect to external database in liferay.

Create New New Project of Service builderLiferay 7 Custom Database

Give package path like below and leave Component class name value as emptyLiferay 7 External Database

Create Service.xml file like below: At the entity level provide

data-source attribute to custom data source.. let's say "extDataSource"

table as Country

at column level, update db-name value with corresponding column name

```
<service-builder package-path="org.javasavvy.external" >
```

```
<namespace>jay</namespace>
```

```
<entity local-service="true" name="Country" table="country" data-source="extDataSource"
```

```
remote-service="false" uuid="false">
```

```

<column name="countryId" db-name="countryId" primary="true" type="long" />
<column name="countryName" db-name="countryName" type="String" />
</entity>
</service-builder>

```

In this Step, we will create ext-spring.xml file in META-INF/spring folder and you can configure data base connection details in the following approaches:

via Portal-ext Properties:

portlet-ext properties: This is recommended approach to configure Database connection details in portlet-ext properties

```
jdbc.external.url=jdbc:mysql://localhost:3306/external
```

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

```
jdbc.external.username=root
```

```
jdbc.external.password=root
```

ext-spring.xml: In the bean "liferayDataSourceImpl", just set the properties "propertyPrefix" to "jdbc.external."

Maku sure you restart the server after setting properties in portal-ext.properties

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<beans default-destroy-method="destroy" default-init-
method="afterPropertiesSet"
```

```
xmlns="http://www.springframework.org/schema/beans"
```

```
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
xsi:schemaLocation="http://www.springframework.org/schema/beans
```

```
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
```

```
<bean class="com.liferay.portal.dao.jdbc.spring.DataSourceFactoryBean"
id="liferayDataSourceImpl">
```

```
    <property name="propertyPrefix" value="jdbc.external." />
```

```
</bean>
```

```

<bean
class="org.springframework.jdbc.datasource.LazyConnectionDataSourceProxy"
id="liferayDataSource">

<property name="targetDataSource" ref="liferayDataSourceImpl" />

</bean>

```

```

<alias alias="extDataSource" name="liferayDataSource" />

```

```

</beans>

```

through spring configuration: In this, approach, we will provide the database connection details in the bean it self or you can use spring property placeholder bean to load from jdbc.properties file. Do not change the "liferayDataSource" bean Id as it is being using by hibernate session factory and it will give you bean creation error. Liferay is restricted this bean name to liferayDataSource, not sure why they configured like this.

```

<?xml version="1.0" encoding="UTF-8"?>

```

```

<beans default-destroy-method="destroy" default-init-
method="afterPropertiesSet"

```

```

xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

```

```

xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

```

```

<bean class="com.liferay.portal.dao.jdbc.spring.DataSourceFactoryBean"
id="dataSourceBean">

```

```

    <property name="propertyPrefix" value="jdbc." />

```

```

    <property name="properties">

```

```

        <props>

```

```

            <prop key="jdbc.driverClassName">com.mysql.jdbc.Driver</prop>

```

```

            <prop key="jdbc.url">jdbc:mysql://localhost:3306/external</prop>

```

```

            <prop key="jdbc.username">root</prop>

```

```

            <prop key="jdbc.password">root</prop>

```

```

        </props>

```

```
</property>
</bean>
<bean
class="org.springframework.jdbc.datasource.LazyConnectionDataSourceProxy"
id="liferayDataSource">
    <property name="targetDataSource" ref="dataSourceBean" />
</bean>
<alias alias="extDataSource" name="liferayDataSource" />
</beans>
```

Now execute buildService gradle task and deploy the modules. you can use this as gradle dependency in other modules and invoke the CountryLocalServiceUtil.

14 How to create custom workflow?

Creating a Workflow Handler for Guestbooks

Handling Workflow

Step 1 of 2

Each workflow enabled entity needs a `WorkflowHandler`. Create a new package in the `guestbook-service` module called `com.liferay.docs.guestbook.workflow`, then create the `GuestbookWorkflowHandler` class in it. Extend `BaseWorkflowHandler` and pass in `Guestbook` as the type parameter:

```
public class GuestbookWorkflowHandler extends BaseWorkflowHandler<Guestbook> {
```

Make it a Component class:

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

There are three abstract methods to implement: `getClassName`, `getType`, and `updateStatus`.

```

    @Override    public String getClassName() {        return
Guestbook.class.getName();    }

```

getClassName returns the guestbook entity's fully qualified class name (com.liferay.docs.guestbook.model.Guestbook).

```

    @Override    public String getType(Locale locale) {        return
_resourceActions.getModelResource(locale, getClassName());    }

```

getType returns the model resource name (model.resource.com.liferay.docs.guestbook.model.Guestbook). The meat of the workflow handler is in the updateStatus method:

```

    @Override    public Guestbook updateStatus(                int status, Map<String,
Serializable> workflowContext)                throws PortalException {                long userId
= GetterUtil.getLong(
(String)workflowContext.get(WorkflowConstants.CONTEXT_USER_ID));                long
resourcePrimKey = GetterUtil.getLong(                (String)workflowContext.get(
WorkflowConstants.CONTEXT_ENTRY_CLASS_PK));                ServiceContext serviceContext
= (ServiceContext)workflowContext.get(                "serviceContext");
return _guestbookLocalService.updateStatus(                userId, resourcePrimKey,
status, serviceContext);    }

```

When you crafted the service layer's updateStatus method (see the last section for more details), you specified parameters that must be passed to the method. Here you're making sure that those parameters are available to pass to the service call. Get the userId and resourcePrimKey from GetterUtil. Its getLong method takes a String, which you can get from the workflowContext Map using WorkflowConstants for the context user ID and the context entry class PK.

Make sure you inject the ResourceActions service into a private variable at the end of the class, using the @Reference annotation:

```

    @Reference(unbind = "-")    protected void setResourceActions(ResourceActions
resourceActions) {                _resourceActions = resourceActions;    }    private
ResourceActions _resourceActions;

```

Inject a GuestbookLocalService into a private variable using the @Reference annotation.

```

    @Reference(unbind = "-")    protected void setGuestbookLocalService(
GuestbookLocalService guestbookLocalService) {                _guestbookLocalService =

```



```

guestbookLocalService;    )    private GuestbookLocalService
_guestbookLocalService;)


```

Organize imports ([CTRL]+[SHIFT]+O) and save your work.

Now the Guestbook Application updates the database with the necessary status information, interacting with Liferay's workflow classes to make sure each entity is properly handled by Liferay DXP. At this point you can enable workflow for the Guestbook inside Liferay DXP and see how it works. Navigate to *Control Panel* → *Workflow Configuration*. The Guestbook entity appears among Liferay DXP's native entities. Enable the Single Approver Workflow for Guestbooks; then go to the Guestbook Admin portlet and add a new Guestbook. A notification appears next to your user name in the product menu. You receive a notification from the workflow that a task is ready for review. Click it, and you're taken to the My Workflow Tasks portlet, where you can complete the review task.



Figure 1: Click the workflow notification in the Notifications portlet to review the guestbook submitted to the workflow.

To complete the review, click the actions button () from My Workflow Tasks and select *Assign to Me*. Click the actions button again and select *Approve*.

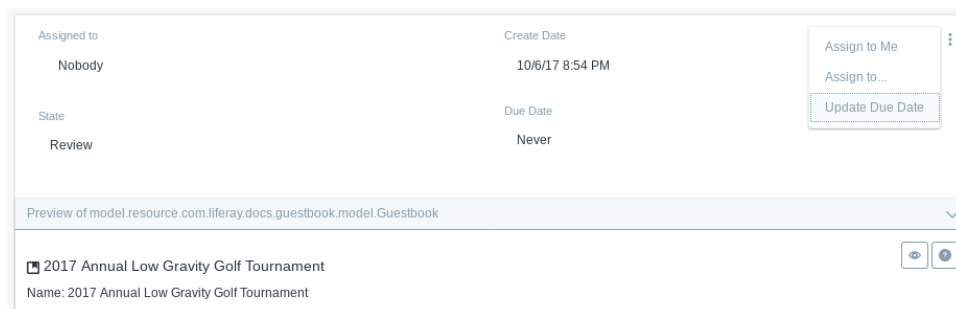


Figure 2: Click the workflow notification in the Notifications portlet to review the guestbook submitted to the workflow.

Right now the workflow process for guestbooks is functional, but the UI isn't adapted for it. You'll write the workflow handler for guestbook entries next, and then update the UI to account for each entity's workflow status.

[« Handling WorkflowCreating a Workflow Handler for Guestbook Entries »](#)

Was this article helpful?

15 How to create custom asset?

How to make custom entity as Asset?

Define AssetEntry reference in service.xml

```
<reference entity="AssetEntry" package-path="com.liferay.portlet.asset" />
```

```
<reference entity="AssetTag" package-path="com.liferay.portlet.asset" /
```

When adding entry then invoked assetEntryLocalService.updateEntry() method

```
AssetEntry assetEntry = assetEntryLocalService.updateEntry( user.getUserId(),  
serviceContext.getScopeGroupId(),
```

```
new Date(), new Date(), Leave.class.getName(), leave.getLeaveId(),  
leave.getUuid(),0, null, null, true,
```

```
false, new Date(), null,new Date(), null,ContentTypes.TEXT_HTML,  
leave.getLeaveName(), leave.getLeaveName(),
```

```
null, null, null, 0, 0, null);
```

When updating entry then

```
assetEntryLocalService.updateEntry(Leave.class.getName(), leaveId, new  
Date(),null, true, true);
```

To update the Asset visibility then invoke below method:

```
assetEntryLocalService.updateVisible(Leave.class.getName(), leaveId, false);
```

What Next after adding entry into AssetEntry table?

The Entity need to be displayed on the user interface. Liferay provides Asset Publisher API for default rendering of custom entity attributes such as title and description.

Liferay also provides custom AssetRenderer API to display additional attributes and editing entity

High Level Steps:

Create Asset Renderer

Create AssetRenderer Factory OSGI service to create AssetRenderer

Let's use the existing leave application explained in the below and ill make Leave Entity as Asset Renderer

Create the packages "org.javasavvy.leave.asset" and Create classes
LeaveAssetRenderer and LeaveAssetRenderFactory.java
Leave AssetRender Factory
AssetRender class is:

```
package org.javasavvy.leave.asset;
```

```
import java.util.Locale;
```

```
import javax.portlet.PortletRequest;
```

```
import javax.portlet.PortletResponse;
```

```
import javax.servlet.http.HttpServletRequest;
```

```
import javax.servlet.http.HttpServletResponse;
```

```
import org.javasavvy.leave.model.Leave;
```

```
import com.liferay.asset.kernel.model.BaseJSPAssetRenderer;
```

```
import com.liferay.portal.kernel.util.ResourceBundleLoader;
```

```
public class LeaveAssetRenderer extends BaseJSPAssetRenderer<Leave> {
```

```
    private final Leave leave;
```

```
    private final ResourceBundleLoader resourceBundleLoader;
```

```
    public LeaveAssetRenderer(Load leave, ResourceBundleLoader  
resourceBundleLoader) {
```

```
        this.leave = leave;
```

```
        this.resourceBundleLoader = resourceBundleLoader;
```

```
    }
```

```
@Override
```

```
public Leave getAssetObject() {
```

```
    return leave;
```

```
}
```

```
@Override
```

```
public long getGroupId() {  
    return leave.getGroupId();  
}
```

```
@Override
```

```
public long getUserId() {  
    return leave.getUserId();  
}
```

```
@Override
```

```
public String getUserName() {  
    return leave.getUserName();  
}
```

```
@Override
```

```
public String getUuid() {  
    return leave.getUuid();  
}
```

```
@Override
```

```
public String getClassName() {  
    return Leave.class.getName();  
}
```

```
@Override
```

```
public long getClassPK() {  
    return leave.getLeaveId();  
}
```

```
@Override
```

```
public String getSummary(PortletRequest portletRequest, PortletResponse  
portletResponse) {  
    return leave.getLeaveName()+ "by  
"+leave.getUserName()+",from:"+leave.getStartDate();  
}
```

```
@Override  
public int getStatus() {  
    return leave.getStatus();  
}
```

```
@Override  
public String getTitle(Locale locale) {  
    return leave.getLeaveName();  
}
```

```
@Override  
public int getAssetRendererType() {  
    return super.getAssetRendererType();  
}
```

```
@Override  
public String getJspPath(HttpServletRequest request, String template) {  
    return "/leave/leaveAssetInfo.jsp";  
}
```

```
@Override  
public boolean include(HttpServletRequest request, HttpServletResponse response,  
String template) throws Exception {
```

```

        request.setAttribute("leaveEntry", leave);
        return super.include(request, response, template);
    }

```

```

    }

```

LeaveAssetRenderFactory class:

This class need to be OSGI Service type of AssetRenderFactory

The class need below @component

```

@Component(
    immediate = true,
    property = {"javax.portlet.name=org_javasavvy_web_leave_portlet"},
    service = AssetRenderFactory.class
)
package org.javasavvy.leave.asset;

```

```

import javax.servlet.ServletContext;

```

```

import org.javasavvy.leave.model.Leave;
import org.javasavvy.leave.service.LeaveLocalService;
import org.osgi.service.component.annotations.Component;
import org.osgi.service.component.annotations.Reference;

```

```

import com.liferay.asset.kernel.model.AssetRenderer;
import com.liferay.asset.kernel.model.AssetRenderFactory;
import com.liferay.asset.kernel.model.BaseAssetRenderFactory;
import com.liferay.portal.kernel.exception.PortalException;
import com.liferay.portal.kernel.util.ResourceBundleLoader;

```

```

@Component(

```

```

    immediate = true,
    property = {"javax.portlet.name=org_javasavvy_web_leave_portlet"},
    service = AssetRendererFactory.class
)

public class LeaveAssetRenderFactory extends BaseAssetRendererFactory<Leave>
{

    private LeaveLocalService leaveService;
    private ResourceBundleLoader resourceBundleLoader;
    private ServletContext servletContext;

    @Reference(unbind = "-")
    protected void setLeaveService(LearnLocalService leaveService) {
        this.leaveService = leaveService;
    }

    @Reference(unbind = "-")
    public void setResourceBundleLoader(ResourceBundleLoader
resourceBundleLoader) {
        this.resourceBundleLoader = resourceBundleLoader;
    }

    @Reference(unbind = "-")
    public void setServletContext(ServletContext servletContext) {
        this.servletContext = servletContext;
    }

    public LeaveAssetRenderFactory() {
        setClassName( Leave.class.getName());
        setCategorizable(true);
    }
}

```

```

        setLinkable(true);
        setPortletId("org_javasavvy_web_leave_portlet");
        setSearchable(true);
        setSelectable(true);
    }

```

@Override

```

public AssetRenderer<Leave> getAssetRenderer(long classPK, int type) throws
PortalException {
    Leave leave = leaveService.getLeave(classPK);
    LeaveAssetRenderer assetRenders = new LeaveAssetRenderer(leave,
resourceBundleLoader);
    assetRenders.setAssetRendererType(type);
    assetRenders.setServletContext(servletContext);
    return assetRenders;
}

```

@Override

```

public String getType() {
    return "leave";
}

```

@Override

```

public String getClassName() {
    return Leave.class.getName();
}
}

```

16 How to solve dependency problem liferay

17 Self Register page which hook require to create?

19 How to implement action command?

6. Add `javax.portlet.name` in `LeavePortlet` Controller if not there

- a. `@Component(immediate = true, property = { "com.liferay.portlet.display-category=category.sample", "com.liferay.portlet.instanceable=true", "javax.portlet.display-name=Leave Application", "javax.portlet.name=org_javasavvy_web_leave_portlet", "javax.portlet.init-param.template-path=/", "javax.portlet.init-param.view-action=/leave/view", "javax.portlet.init-param.view-template=/leave/view.jsp", "javax.portlet.resource-bundle=content.Language", "javax.portlet.security-role-ref=power-user,user" }, service = Portlet.class)`
`public class LeavePortlet extends MVCPortlet { }`

7. Create `EditLeaveActionCommand` class and make that as OSGI Component with below config:

- a. `@Component(property = { "javax.portlet.name=org_javasavvy_web_leave_portlet", "mvc.command.name=leave_editLeave" }, service = MVCActionCommand.class)`
`public class EditLeaveActionCommand extends BaseMVCActionCommand {`
`private LeaveLocalService leaveService;`
`@Reference(unbind = "-") protected void setLeaveService(LeaveLocalService leaveService) { this.leaveService = leaveService; }`
`@Override protected void doProcessAction(ActionRequest actionRequest, ActionResponse actionResponse) throws Exception {`
`SimpleDateFormat sdf = new SimpleDateFormat("MM/dd/yyyy");`
`String name = ParamUtil.getString(actionRequest, "name");`
`System.out.println("Leave Edit"+name);`
`Date startDate= ParamUtil.getDate(actionRequest, "startDate",sdf);`
`Date endDate = ParamUtil.getDate(actionRequest, "endDate", sdf);`
`ThemeDisplay themeDisplay = (ThemeDisplay) actionRequest.getAttribute(WebKeys.THEME_DISPLAY);`
`long groupId = themeDisplay.getScopeGroupId();`
`Leave leave = leaveService.addLeave(name, themeDisplay.getRealUserId(), groupId, themeDisplay.getCompanyId(), startDate, endDate);`
`}`

8. In JSP, portlet action URL should be like belows. We can use either of below action requests:

- a. `<liferay-portlet:actionURL name="leave_editLeave" var="editLeave">` `<portlet:param name="mvcActionCommand" value="leave_editLeave" />` `</liferay-portlet:actionURL>`
- b. `<liferay-portlet:actionURL name="leave_editLeave" var="editLeave"></liferay-portlet:actionURL>`
`<au:form action="<%= editLeave %>" cssClass="container-fluid-1280" method="post" name="fm">` `<au:form>`

Benefits Of OSGi

Instead of gigantic big application, using OSGi we can separate them as independent modules.

Easy maintenance

Each Module with its own clear responsibility

Restrict and Expose the visibility of classes properly

Advantages of using OSGi:

1. OSGi allows us to create dynamic, live architectures and applications.

2. OSGi provides a framework to develop modular application. We can decompose a complex application into multiple modules. Each modules focuses on specific task. It also motivates us as a developer to build distinctly identifiable JAR files, small sets of code that do a few things in a clearly defined manner.

3. We can install, uninstall, start, and stop various modules of our application dynamically without restarting the container.

4. Our application can have more than one version of a particular module running at the same time.

5. OSGi is lightweight and customizable.

6. OSGi provides extensibility without eroding the system.

7. OSGi manages the complexity of large systems.

8. Starting with bundles, developers are able to encapsulate functional portions of code into single deployable units, with explicitly stated imported and exported packages.

9. In contrast to a regular classloading environment, we as a developer can control explicitly what you expose, share, and how it is versioned.

22 Use of export and import

Importing Packages¹

You often find yourself in a position of needing functionality provided by another module. To access this functionality, you must import packages from other modules into your module's classpath. This requires that those other modules have already [exported](#) their packages containing the functionality you want. The OSGi framework wires the packages to the importing module's classpath. The module JAR's META-INF/MANIFEST.MF file uses the Import-Package OSGi header to import packages.

Import-Package: javax.portlet,com.liferay.portal.kernel.util 

Import packages must sometimes be specified manually, but not always. Conveniently, [Workspace](#)-based module projects automatically detect

required packages and add them to the module manifest's package import list. Import packages must sometimes be specified manually.

There are two different package import scenarios:

- [Automatic Package Imports](#)
- [Manual Package Imports](#)

Read below to explore how package imports are specified in these scenarios.

Automatic Package Imports

[Workspace](#)-based projects from the tutorial examples (see [Module Projects](#)) or created using [Blade CLI](#) or [Liferay Developer Studio](#) use [Bnd](#). A Gradle plugin invokes Bnd, which can then read the Gradle dependencies and resolve the imports. When you build the project's JAR, Bnd detects the packages the module uses, generates a META-INF/MANIFEST.MF file, and assigns the packages to an Import-Package header. In that sense, package import is automatic, because you must only define your dependencies in one place: the build script.

NOTE

Liferay's project templates use [a third-party Gradle plugin](#) to invoke Bnd.

The [Gogo Command Sample](#)'s build.gradle, for example, uses packages from com.liferay.portal.kernel and org.osgi.service.component.annotations. Here's the sample's build.gradle file:

```
dependencies {  
    compileOnly group: "com.liferay.portal", name:  
"com.liferay.portal.kernel" compileOnly g: "org.osgi", name:  
"org.osgi.service.component.annotations"}  

```

And here's the Import-Package header that Bnd generates in sample JAR META-INF/MANIFEST.MF file:

Import-Package: com.liferay.portal.kernel.service;version="[4.3,5)" 

The build file specifies dependencies. Bnd examines the module classpath to import packages the module uses. The examination includes all classes found in the classpath—even those from embedded [third party library JARs](#).

NOTE

For a plugin WAR project, Liferay's [WAB Generator](#) detects packages used in the WAR's JSPs, descriptor files, and classes (in WEB-INF/classes and embedded JARs). Also the WAB Generator searches the web.xml, liferay-web.xml, portlet.xml, liferay-portlet.xml, and liferay-hook.xml descriptor files. It

adds package imports for classes that are neither found in the plugin's `WEB-INF/classes` folder nor in its embedded JARs.

Manual Package Imports

If a module references a class in only the following places, you must manually add a package import.

- Unrecognized descriptor file
- Custom or unrecognized descriptor element or attribute
- Reflection code
- Classloader code

Here's how to manually import the package:

9. Open your module's `bnd.bnd` file.
10. Add the `Import-Package` header.
11. Add the package to the header's package list.

`Import-Package: [... existing package list,][add the package here]` 

NOTE

To manually import a package in a plugin WAR project, add an `Import-Package` header like the one above to the project's `WEB-INF/liferay-plugin-package.properties` file.

23 how to provide multiple version?

24 How to connect multiple db

