Platform Developer

LIFERAY Training

3.1 OSGi Patterns used in Liferay

3.1.1 Whiteboard Pattern

The aim is to improve the Listener pattern. The Listener pattern works on the principle of an event producer and a class that registers to listen to the produced event. Here are some of the disadvantages of Listeners:

- Each Listener is tied (or coupled) directly to the event source.
- Listeners must manage themselves (e.g. add and remove themselves from source).
- It isn't easy to filter Listeners in order to notify selectively based on a set of criteria.

(Figure 3.1, page 103)

The whiteboard pattern severs the tie between the producer and consumer. It introduces the OSGi Service register as a man-in-the-middle entity responsible for keeping the references to the Listeners and providing the Listeners with lists for event producers.

Drawbacks of the whiteboard pattern:

- It may not be clear what interface needs to be implemented to create a new Listener class. This can be easily overcome by providing good documentation.
- The other issue is the dependency on the framework.

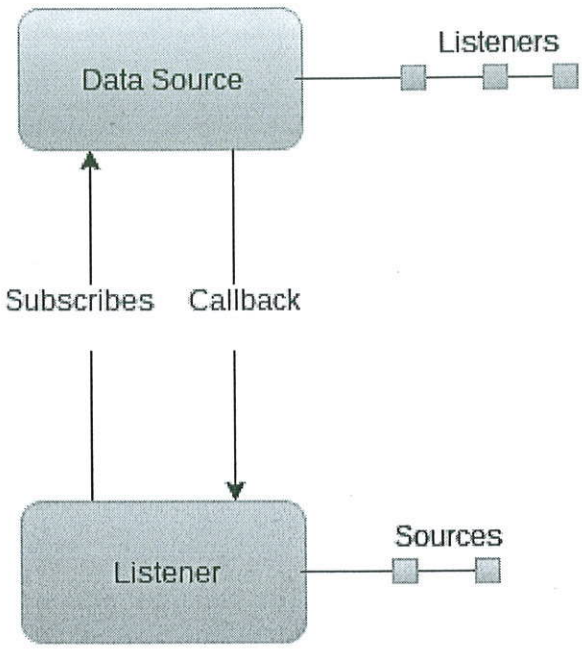
(Figure 3.2, page 104)

Usage Example

Liferay uses the whiteboard pattern in many places; for example, in the message bus implementation. To create an OSGi-based Message Listener, you need to:

- Create a class that implements the MessageListener interface.
- Add the `@Component` annotation with attributes **service = MessageListener.class** and property = {destination.name=}.
- Implement **doReceive(Message message)**.

Figure 3.1:



Listener pattern

- The actual registration is done with the help of the *DefaultMessageBus*.

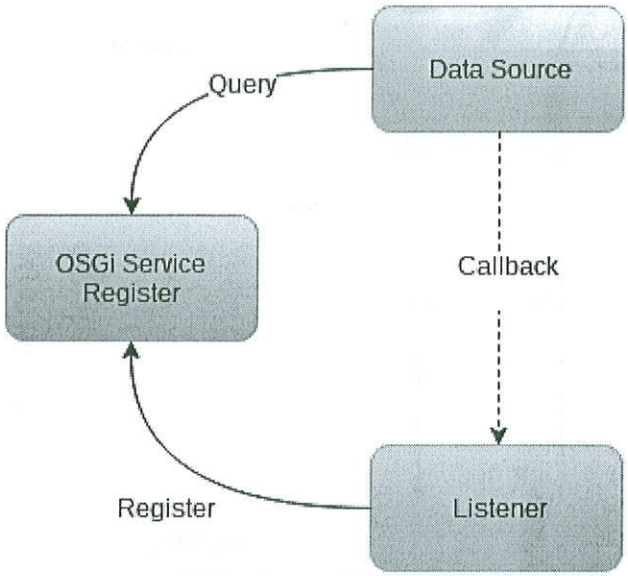
3.1.2 Extender Pattern

The pattern allows for adding extension code to other bundles and helps to reduce boilerplate code.

For example, we can move common code that is required to register Servlets to an extender. The extender then waits till a Servlet bundle gets deployed. When a new Servlet bundle gets activated, the Servlet extender takes it and registers it to the Servlet container.

This pattern is used to register new portlets in Liferay. A portlet class needs to be annotated with `@Component(service = javax.portlet.Portlet.class)`. The *PortletTracker* (extender) is watching for this annotation in bundles and registers the detected portlets into Liferay.

Figure 3.2:



Whiteboard pattern

Another example of an extender is the *Log4jExtender*. It's used to configure logging for a bundle. It is activated when **META-INF/module-log4j.xml** or **META-INF/module-log4j-ext.xml** is detected within the bundle.

(Figure 3.3, page 105)

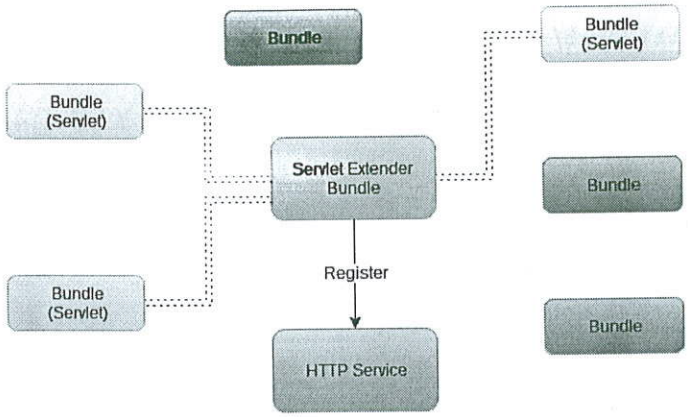
3.2 Integration Points

Modifications in Liferay 7 are different from the classic plugin approach.

3.2.1 Creating Fragments to Override JSPs

- Most of Liferay's JSP files were moved from portal-web to separate OSGi bundles. The moved files cannot be modified by the traditional JSP hook.
- JSPs are included in modules as resources.

Figure 3.3:



Extender pattern

- You can write other bundles to override the existing resources or you can use predefined extension points to add new code fragments.

3.2.2 JSP override fragments

- All resources can be overridden or customized through special modules called fragments.
  - A fragment contains a manifest but doesn't have any components or an activator.

To override a JSP in a deployed app, we need to:

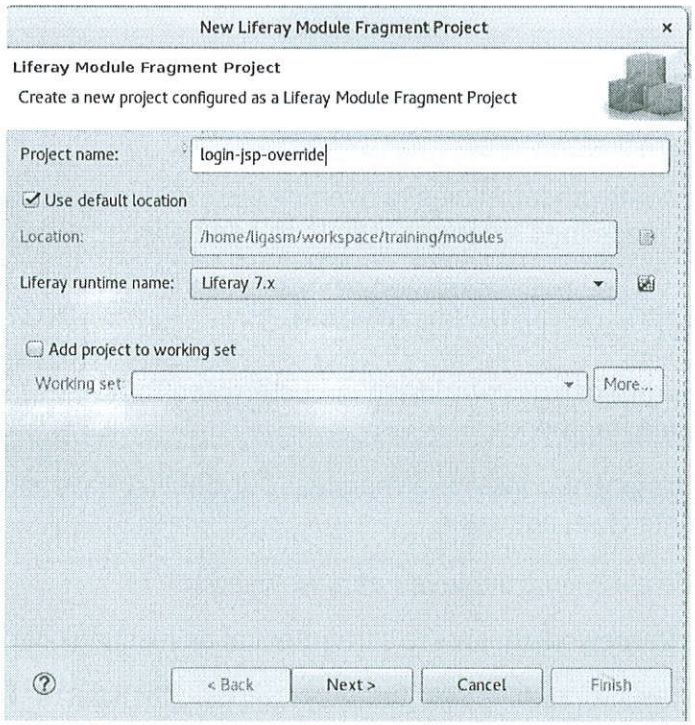
- Tell the framework that we're a fragment of an existing Liferay bundle
  - `Fragment-Host: com.liferay.login.web;bundle-version=1.0.0`
- We need to create the JSP we want to override in a *resource* folder, like `resources/login.jsp`.
- Once we have the resource created, we tell the framework to include it.
  - `-useresource: META-INF/resources=resources/`

When deployed, the module fragment will attach to the host (Fragment-Host), and the JSP provided in the fragment will be picked up by the host module.

Hands-On Exercise

1. Make sure you have Liferay 7 configured in the IDE.
2. If you don't have a `Liferay workspace`, **create** one.
3. **Add** a `New Liferay Module Fragment Project`.
4. **Provide** the project name: `login-jsp-override`. *(Figure 3.4, page 106)*

Figure 3.4:



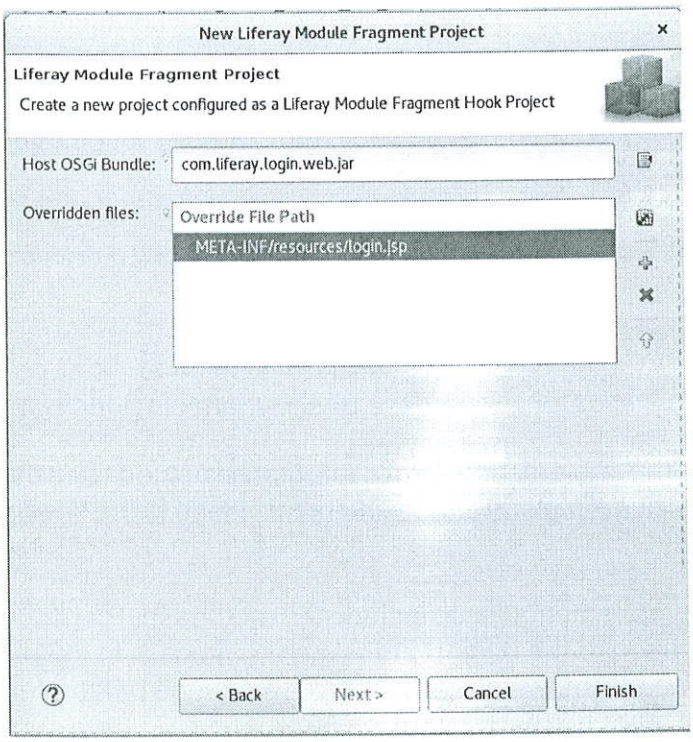
5. Next **Select** `com.liferay.login.web.jar` as the `Host OSGi Bundle`. You can use the Browse button to easily find a bundle.
6. **Click** on the + sign to add files to be overridden. Browse for **META-INF/resources/login.jsp**.
7. **Click** Finish.
8. **Add** welcome text below the init.jsp include.

9. **Check** the `bnd.bnd` to make sure that it contains the host fragment directive. You should this `Fragment-Host: com.liferay.login.web;bundle-version=1.0.0`.

10. **Deploy** the bundle.

(Figure 3.5, page 107)

Figure 3.5:



When you deploy the bundle, you should see the welcome message.

- If the message is not present, go to Gogo console and check if the bundle is resolved.
- If not, look at the MANIFEST file in the bundle (build/libs sub-folder). If it contains `Import-Package: javax.servlet,javax.servlet.http`, remove this line and redeploy the bundle.
- It might be also necessary to refresh the host bundle so that it picks up the changes.

3.2.3 Dynamic include

- Liferay contains predefined extension points in the JSP that can be used to easily add additional functionality to various places.
- This approach consists of two parts.
  - The extension placeholder which is defined by the `liferay-util:dynamic-include` tag.
  - The extension provider which is an OSGi component injects its code into the placeholder.

If you would like to add some code, for example, to the bottom of the page, you can use this extension point.

```
<liferay-util:dynamic-include key="/html/common/themes/bottom.jsp#pre" />
```

To inject code to this place, you need to:

1. Create a class that extends the **BaseDynamicInclude** class.
2. Provide the implementation of the **register** method. This calls the DynamicIncludeRegistry and registers this class to the given extension point key.
3. Provide implementation to **include** the method that can provide some modification to the original JSP. You have access to the request and response object, which can be used to change parameters or write into the output stream."
4. To make this class visible to OSGi, you need to annotate it as a `@Component` and define the service type as DynamicInclude.

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

This approach can be also used in your custom code to provide a clean separation between the original code and your modifications.

3.2.4 Language Properties

- You can provide custom translation through a Resource Bundle component.
- You need to create a class that extends **ResourceBundle**.
- Then override methods:
  - getKeys
  - handleGetObject
  - handleKeySet
- The class needs to be annotated with the *Component* annotation:

```
@Component(
immediate = true,
property = { "language.id=en_US" },
service = ResourceBundle.class
)
```

3.3 Overriding Services Using OSGi

3.3.1 Writing Custom Service Implementations

- Services built with Liferay's ServiceBuilder can be elegantly wrapped.
- Instead of needing to re-implement an entire service, Liferay provides a service wrapper you can override.
- To implement a custom service wrapper, you simply need to create a Java object that extends the service wrapper for your service.
- From there, you can override the implementation of any of the service methods.

3.3.2 Installing Custom Services

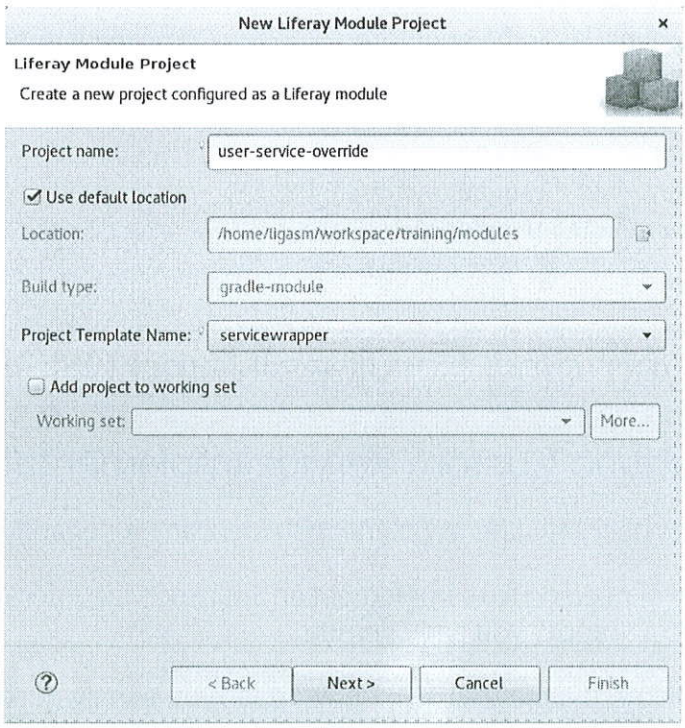
Once you have created the appropriate service wrapper, using it is a simple matter of making a component.

The service wrapper is a simple component that only requires one attribute to be set correctly - `@Component (service = ServiceWrapper.class)` Once the module is deployed, the component will take the place of the original service implementation.

Hands-On Exercise

1. Add `New Liferay Module Project`.
2. Set project name to **user-service-override**. *(Figure 3.6, page 110)*

Figure 3.6:



3. Select `servicewrapper` in `Project Template Name`.
4. Next, **Name** the new class to be created as `OverrideUserLocalServi ce`.
5. **Provide** a package `com.liferay.bootcamp`.
6. **Use** the browse button to **set** the `Service Name` to `com.liferay.port al.kernel.serv UserLocalServiceWrapper`.

7. **Override** the `authenticateByScreenName` and `authenticateByEmailAddress` methods. Add logging into them and call the super class methods.

8. **Deploy** the bundle.

Go to the Platform and try to log in. You should see a logging message in the log.

3.4 Struts Actions

- Struts actions are used when actions are invoked through the platform UI, like submitting a form.
- They handle the action that was invoked and extract the data from the request and then, if the data is being persisted, they invoke the service layer.
- Struts action shouldn't contain the business logic of the application, only logic related to processing the request that invoked the action.
- For example, you could use Struts actions to validate fields on a form that are necessary for validation, but aren't going into the service (like entering an email address a second time for verification).
- Liferay 7 moves away from using Struts but there are still few places where Struts actions are used.
- Struts actions can be easily implemented through components.
- Create a basic component, specifying that the service it provides is `StrutsAction.class`.
- Configure a property on the component to point to the desired Struts path:

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

- Implement a Struts action object, extending the `BaseStrutsAction.class`.
- To gain access to the servlet context, you can reference it like any standard framework service:

```
@Reference (target="(osgi.web.symbolicname=test.strutsaction)")
protected void setServletContext(ServletContext servletContext) {
...
}
```

Hands-On Exercise

1. **Modify** the *LogoutAction* that it will log every User log out.
2. **Create** `New Liferay Module Project`.
    - Liferay IDE does not currently offer an option of an empty project, so we're going to reuse the activator bundle and change it according to our needs.
3. **Set** project name to *custom-logout-action*.
4. **Select activator** in **Project Template Name** and **click** Finish.
5. **Rename** the existing class to **CustomLogoutAction** that will extend **BaseStrutsAction**.
6. **Fix** dependencies in the *build.gradle* file.

```
    compile 'org.osgi:org.osgi.core:6.0.0'
compile 'com.liferay.portal:portal-service:7.0.0-SNAPSHOT'
    compile 'org.osgi:org.osgi.compendium:5.0.0'
    compile 'javax.servlet:servlet-api:2.5'
```

7. **Override** the execute method so that we can add our modifications.
8. We are going to use the execute method that has the original Struts action object in the signature. We can use this object to call the original action.
9. We are only going to add a log message into the body of the method and call the original action.

```
@Override
public String execute(StrutsAction originalStrutsAction,
HttpServletRequest request, HttpServletResponse response)
        throws Exception {
    _log.error("Custom Logout Action");

   return originalStrutsAction.execute(originalStrutsAction,
    request, response);
}

private static final Log _log = LogFactoryUtil.getLog
(CustomLogoutAction.class);
```

10. The class needs to be annotated with the *Component* annotation. The *path* property defines the Struts path that we are registering to.

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

11. **Clean** the bnd.bnd file. **Leave** only the **Bundle-SymbolicName** and `Bundle-Version` properties.
12. **Deploy** the bundle.
13. **Login** in into the platform and **log out**. You should see a log message from your custom code.

3.4.1 Servlet Filters

- A filter is an object that performs filtering tasks on either the request to a resource (a servlet or static content), the response from a resource, or both.
- Servlet filters preprocess requests before they reach a servlet and/or postprocess the response leaving a servlet.
- Filters can:

  - Intercept a servlet's invocation before the servlet is called
  - Examine a request before a servlet is called
  - Modify the request headers and request data by providing a customized version of the request object that wraps the real request
  - Modify the response headers and response data by providing a customized version of the response object that wraps the real response
  - Intercept a servlet's response after the servlet is called

- A servlet filter can be created in an OSGi environment as a component.
- Create a java class that extends the `PortalBaseFilter`.
- Annotate the class with the component annotation and set service as `javax.servlet.Filter`.
- You can define the standard servlet values in the annotation properties.
  - url pattern
  - context name
  - filter name

```
@Component(
    immediate = true,
    property = {
     "servlet-context-name=", "servlet-filter-name=Test Filter",
        "url-pattern=/*"
    },
    service = Filter.class
)
```

- To specify the position in the filter chain, you can use the **after-filter** and **before-filter** properties.
- The filter is applied to the ROOT context.

Hands-On Exercise

1. **Create** a New Liferay Module Project.
2. **Set** project name to **header-filter**.
3. **Select activator** in **Project Template Name**.
4. **Click** Finish.
5. **Delete** the activator class and **create** a new **HeaderFilter** class that extends the **PortalBaseFilter**.
6. **Override** the **processFilter()** method so that it will log all headers from the HttpServletRequest.
7. **Annotate** the class with the *Component* annotation and provide the necessary properties as explained previously.
8. **Clean** the bnd.bnd file. Leave only the **Bundle-SymbolicName** and **Bundle-Version** properties.
9. **Deploy** the bundle.

When you now access Liferay, you should see log messages for every access showing the headers.

3.5 Auth Pipeline

- Liferay has an Authentication Pipeline that allows various authenticators to run in a queue.
- You can define classes that will run before or after Liferay authentication.
  - auth.pipeline.pre
  - auth.pipeline.port
- The classes in the pipeline need to answer with *Authenticator* constants (`SUCCESS, FAILURE, DNE, SKIP_LIFERAY_CHECK`).
- All classes in the pipeline need to return SUCCESS for a successful authentication.

- In the pre-authentication pipeline, if you want to skip password checking by the internal portal authentication, the authenticator should return `SKIP_LIFERAY_CHECK`.
- To create a pre or post authenticator, you need to implement the *Authenticator* interface.
- Implement the interface methods that do the authentication check:
  - by screen name
  - by email address
  - by user id
- Add the component annotation to the class and set the key property to pre or post auth pipeline.

```
@Component(
immediate = true, property = {"key=auth.pipeline.pre"},
service = Authenticator.class
)
public class BootcampAuthenticatorPre implements Authenticator
```

Hands-On Exercise

Create an Automatic Authenticator that will log in a User without checking the password.

1. Create *New Liferay Module Project.*
2. Set project name to `auto-login-authenticator`.
3. Select *activator* in `Project Template Name`.
4. Click Finish.
5. Implement all methods defined by the interface. Make them return `Authenticator.SKIP_LIFERAY_CHECK`.
6. Add the `Component` annotation and set the *key* property to `auth.pipeline.pre`.
7. Fix the project dependencies. Check to see if the `build.gradle` file contains these dependencies:
   - compile `org.osgi:org.osgi.core:6.0.0`

- compile `com.liferay.portal:com.liferay.portal.kernel:2.0.0`
- compile `org.osgi:org.osgi.service.component.annotations:1.3.0`

8. **Clean** the `bnd.bnd` file. Leave only the *Bundle-SymbolicName* and *Bundle-Version* properties.
9. **Deploy** the bundle.

Try to log in with a random password. If you are not able to log in with a random password and you don't see any log messages in the log, check the Gogo console.

1. **Check** to see if the bundle is Active. If not, try to start it.
2. **Check** for missing dependencies. Try to find the missing dependencies with the **p** Gogo shell command.
3. **Try** to search for the missing packages with the `b 0 | grep package.name` command.
4. **Fix** the `bnd.bnd` file accordingly.
5. **Redeploy** the bundle.
6. **Check** the functionality.

3.5.1 Framework Events

- You can create an action that will be run before or after a certain framework event.
- Supported extension points are:
  - `servlet.service.events.pre` and `servlet.service.events.post` are executed before and after Struts processing.
  - `login.events.pre` and `login.events.post` are executed before and after the login attempt.
  - `logout.events.pre` and `logout.events.post` are executed before and after logout.
- To participate in a framework event, you need to create a class that implements the *LifecycleAction* interface.

- You need to provide an implementation for the *processLifecycleEvent* method
- The class needs to be annotated by the `@Component` annotation and the key property set to the appropriate value.

```
@Component(
immediate = true, property = {"key=login.events.pre"},
service = LifecycleAction.class
)
public class LoginPreAction implements LifecycleAction
```

3.5.2 Auth Failure

- You can detect and react to failed authentication attempts.
- These classes will run when a User has a failed login or when a User has reached the maximum number of failed logins.
- To create an AuthFailure detector class, you need to implement the AuthFailure interface and implement its methods.
- Annotate the class with the *Component* annotation.

```
@Component(
immediate = true, property = {"key=auth.failure"},
service = AuthFailure.class
)
public class LogAuthFailure implements AuthFailure {
```

3.6 Friendly URL

3.6.1 What are Friendly URLs?

- URLs generated by the platform can be complex:
  - Example: `http://www.liferay.com/web/nathan.cavanaugh/blog?p_p_id=33&p_p_lifecycle=0&p_p_state=normal&p_p_mode=view&_33_struts_action=%2Fblogs%2Fview`
- In most cases, the user is oblivious to these URLs, so the added complexity is no issue.

- Human-readable URLs are necessary, however, if you want memorable pages or references to important items (like products and office information).
- Additionally, making pages SEO-friendly requires descriptive URLs.
- Friendly URLs let you route important information to the platform while making URLs readable, creating a URL like this:

  .../web/nathan.cavanaugh/blog/-/blogs/

- In this case, the extra-long URL was obfuscating what was actually being done.
- Liferay's friendly URL mapper can remove the extra cruft from a URL and provide the user with the relevant information in the address.

3.6.2 How do Frendly URLs Work?

- Friendly URLs work based on routes, which define a pattern in the URL to match: /[urlTitle]/[p_p_state]
- The developer then maps this route to a set of parameters, both explicit and implicit.
- The resulting mapped URL is then used by the platform to process the request.
- All of the traditional parameters in a Portlet URL are available:
  - p_p_id
  - p_p_lifecyle
  - p_p_state
  - p_p_mode

3.6.3 Friendly URL Mapping

- Liferay 7 uses the same xml file to define the routes that were used in Liferay 6.2.
- You need to create an OSGi component that points to the xml for the platform to recognize it.

- The component needs to extend the `DefaultFriendlyURLMapper` and implement the **getMapping** method.
  - The return mapping serves as the Friendly URL prefix.
- The created class needs to be annotated by a `@Component` annotation and 2 properties needs to be set.
  - `com.liferay.portlet.friendly-url-routes` points to the routes.xml file.
  - `javax.portlet.name` points to the portlet that uses these friendly URLs.

```
@Component(
    property = {
        "com.liferay.portlet.friendly-url-routes=
        META-INF/friendly-url-routes/routes.xml",
        "javax.portlet.name=com_liferay_bootcamp_TestPortlet"
    },
    service = FriendlyURLMapper.class)
```

3.6.4 Model Listeners

- Model Listeners allow you to listen for specific events during the life-cycle of the model.
- Model Listeners allow you to execute some custom code before or after *add, update, associate, and delete* actions.
- They have essentially the same use cases as Service Wrappers, but are not as powerful.
- Service Wrappers wrap not only the add, update, associate, and delete methods, but also any method in your services, giving you the flexibility not only to add code but also to overwrite existing logic.
- Model Listeners can work similarly in Liferay 7.
- You need to create a class and extend the BaseModelListener with the desired model type.
- Depending on your needs, you will override the desired base methods to provide additional functionality before and/or after create, update, delete operations.

- To make these Listeners recognizable by Liferay 7, they need to be annotated with the `@Component` annotation.

```
@Component(
immediate = true,
service = ModelListener.class
)
public class CustomLayoutListener extends
BaseModelListener<Layout> {
```

3.6.5 Portlet Filters

- Portlet filters are Java components that allow on-the-fly transformations of information in both the request to and the response from a portlet.
- They allow chaining reusable functionality before or after any phase of the portlet lifecycle: *processAction() processEvent() serveResource() render()*
- Portlet filters are modeled after the filters of the servlet specification.
- Portlet filters become more interesting in Liferay 7 because they allow an easy way to manipulate the data flowing between the Controller class and the presentation framework (JSPs).
- You need to create a new class that implements one of the standard filter interfaces to create a PortletFilter.
  - ActionFilter
  - RenderFilter
  - ResourceFilter
- Implement the **doFilter** method to provide your implementation.
- Don't forget to continue the chain.
- The implemented class needs to be annotated with the `@Component` annotation. The filter always applies to a portlet; therefore, the `javax.portlet.name` property needs to point to the targeted annotation.

```
@Component(
immediate = true,
property = {
   "javax.portlet.name=blade_portlet_filter_ExamplePortlet"
},
service = PortletFilter.class
)
```

Hands-On Exercise

Create a simple generic portlet that will display a "Hello, World" message set in the View method. It will be changed by the filter to a greeting if the User is signed in.

1. Create *New Liferay Module Project.*
2. Set the name to *HelloWorld.*
3. Use the *portlet* Project Template.
4. Click Next.
5. Set the name of the portlet class to *HelloWorldPortlet.*
6. Set the package to *com.liferay.bootcamp.*
7. Click Finish.
8. Go to the `HelloWorldPortlet` class and change the body of the *doView* method to:

```
PrintWriter printWriter = response.getWriter();

String customAttr = (String) request.
getAttribute("CUSTOM_ATTRIBUTE");
if(customAttr == null){
    customAttr = "Hello World";
}

printWriter.print(customAttr);
```

9. Create a new class GreetingsRenderFilter that implements the RenderFilter.

10. Change the body of the doFilter method to:

```
ThemeDisplay themeDisplay = (ThemeDisplay)request.
getAttribute(
        WebKeys.THEME_DISPLAY);
if(themeDisplay.isSignedIn()){
    User user = themeDisplay.getUser();
    request.setAttribute("CUSTOM_ATTRIBUTE",
     "Greetings "+ user.getFullName());
}
chain.doFilter(request, response);
```

11. Add the component annotation to the filter class.

```
@Component(
    immediate = true,
    property = {
        "javax.portlet.name=com_liferay_
        bootcamp_HelloWorldPortlet"
    },
    service = PortletFilter.class
)
```

12. Build and deploy the bundle.

Go to the platform and place the portlet on the page. If you are logged in, you should see the *Greeting* message. Sign out, and only the *Hello World* message will be displayed.

Indexer post-processor

- The Indexer builds a post processing system on top of an existing indexer.
- The Indexer can be used to modify search summaries, indexes, and queries.
- To create a new post-processor, you need to create a class that implements the IndexerPostProcessor interface.

- The `@Component` annotation is used to register the indexer to the platform. The *indexer.class.name* property defines the model for that we are providing the extension.

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

3.7 REST

- Liferay 7 is introducing a new endpoint for creating REST APIs.
- Standard JAVA WS RS API is used to define an endpoint.
- It is possible to inject OSGi services (e.g., Liferay service layer) into the REST interface class.
- This interface does not provide any security layer.
- To create an endpoint, you need to extend the `javax.ws.rs.core.Application` class and create a standard JAVA RS WS definition of a REST API. Use the `javax.ws.rs.*` annotations.
- You need to add *Component* annotation to the class to make it visible to the platform.

```
Component(
    immediate = true,
    property = {
        "jaxrs.application=true"
    },
    service = Application.class)
```

3.7.1 Custom Portlet initialization

- In a plain OSGi application, the bundle (module) implements a Bundle Activator to start, stop, and register services. We can instead use Declarative Services and annotate a service class. Now we no longer have to use a Bundle Activator to register the service.

- The BundleActivator may come in handy when upgrading old portlet classes to OSGi bundles. You don't need to touch the original Portlet class; you can just add a BundleActivator that would be responsible for the Portlet registration in the platform.

Hands-On Exercise

Use the BundleActivator to deploy a Generic Portlet.

1. **Create** a New Liferay Module Project.
2. **Set** project name to **legacy-portlet.**
3. **Select activator** in **Project Template Name.**
4. **Click** Finish.
5. **Create** a new class in the project called **LegacyPortlet** and **replace** the class content with the snippet.

```
public class LegacyPortlet extends GenericPortlet {

    @Override
    protected void doView(RenderRequest request,
    RenderResponse response)
        throws IOException, PortletException {

        PrintWriter printWriter = response.getWriter();

       printWriter.print("Legacy Portlet - Hello World!");
    }
}
```

6. Add missing dependencies to the *build.gradle* file.

```
compile 'javax.portlet:portlet-api:2.0'
compile 'org.osgi:org.osgi.compendium:5.0.0'
```

7. **Rename** the created activator to **LegacyPortletActivator.**
8. **Fix** the *Bundle-Activator* property in bnd.bnd to point to the Activator class.

9. Add the portlet activation code to the bundle activator.

```
public class LegacyPortletActivator implements BundleActivator {
 public void start(BundleContext bundleContext) throws Exception {
   Dictionary<String, Object> properties = new Hashtable<>();

     properties.put(
      "com.liferay.portlet.display-category", "category.sample");
   properties.put("com.liferay.portlet.instanceable", "true");
   properties.put("javax.portlet.display-name", "Legacy Portlet");
     properties.put(
         "javax.portlet.security-role-ref",
         new String[] {"power-user", "user"});

    _serviceRegistration = bundleContext.registerService(
        Portlet.class, new LegacyPortlet(), properties);
 }

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

 private ServiceRegistration<Portlet> _serviceRegistration;

}
```

10. Build and deploy the bundle.

Go to the platform and try to add the legacy portlet to the page.