Service Builder - A portal-wide (and beyond) data access tool

Liferay service builder is a tool which will generate the service layer to Plugin portlets. Though service builder has not undergone through any major changes in Liferay 7, this section will precisely explain changes. Then, this section covers:

Creating service builder module

Understanding the generated modules and service layer

Using service builder services in Portlets

Creating Service Wrappers to implement the custom logic

Customization of any application is extremely significant for both individual product performance and commercial success of the product. Liferay 7 offers Service wrappers to handle specific customizations. This section covers:

Creation of new wrapper module

Blade Service Template to create new Liferay modules

According to Liferay team itsef, creating Liferay modules in a workspace using Blade CLI is very similar to creating them in a standalone environment. This section will cover step by step guidelines about:

Creation of a module using service template

Using OSGi Services in portlets – The real potential of Liferay 7

The indispensable role of OSGi in Liferay 7 is out rightly clear so far. It is vital to understand optimum usage of OSGi service to make the best out of it. This article covers:

Creating OSGi Service

Consuming OSGi Services

Service Builder - A portal-wide (and beyond) data access tool

Liferay service builder is a tool which will generate the service layer to Plugin portlets. Though service builder has not undergone through any major changes in Liferay 7, this section will precisely explain changes. Then, this section covers:

Creating service builder module

Understanding the generated modules and service layer

Using service builder services in Portlets

Liferay service builder is a tool for developer to quickly develop the business service layer of the system. Based on the entities defined in an XML file, service builder auto-generates model, persistence and service layer code. The generate code provides basic operations like create, read, update, delete and find. This allows developers to save a lot of time and lets them focus on higher level business logic. Apart from this service builder provides inbuilt cache mechanism, dynamic queries, finder method mechanism, auto generated SQL files, remote services etc.

Service builder tool was available prior to Liferay 7 as well. In Liferay 7, service builder has not undergone any major changes apart from few differences mentioned below:

- The business layer is now divided into two modules: *-api and *-service module.
- The modules who want to use the service, needs to access the services using OSGi declarative service injection.
- The service builder modules have module-hbm.xml, module-spring.xml and portlet-model-hints.xml configuration files.

Creating service builder module

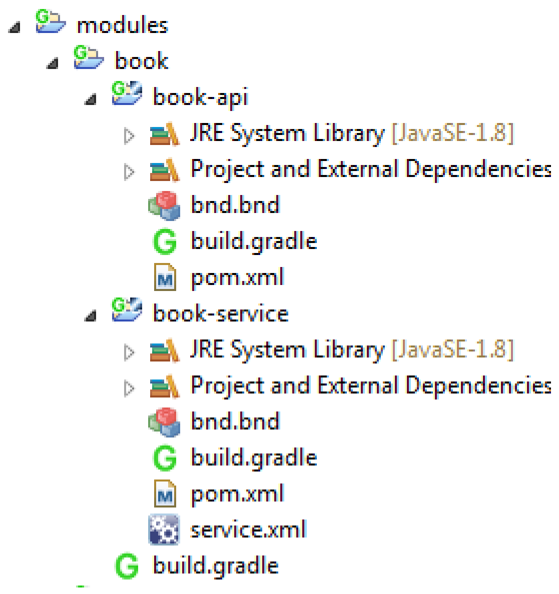
To illustrate the steps, we will create book service.

1) Create new service builder modules.
   The blade command format:
   blade create –t service-builder [-p package-name] project-name

   In our example, we will run the command:
   blade create –t service-builder –p com.azilen.training.book book

2) The above command will create two modules in Liferay workspace: “book-api” and “book-service”. You will find the service.xml file in the book-service module.

This is how the project structure would look like:



```
modules
  book
    book-api
      JRE System Library [JavaSE-1.8]
      Project and External Dependencies
      bnd.bnd
      build.gradle
      pom.xml
    book-service
      JRE System Library [JavaSE-1.8]
      Project and External Dependencies
      bnd.bnd
      build.gradle
      pom.xml
      service.xml
    build.gradle
```

3) Now, to push our book service example forward, add the following code in the service.xml:

```
<service-builder package-path="com.azilen.training.book">
      <namespace>tr</namespace>

      <entity name="Book" local-service="true" remote-service="true"
            uuid="true">

            <!-- PK fields -->
            <column name="bookId" primary="true" type="long" />

            <!-- Group instance -->
            <column name="groupId" type="long" />

            <!-- Audit fields -->
            <column name="companyId" type="long" />
            <column name="createdBy" type="long" />
            <column name="createDate" type="Date" />
            <column name="modifiedBy" type="long" />
            <column name="modifiedDate" type="Date" />

            <!-- Other fields -->
            <column name="bookName" type="String" />
            <column name="isbn" type="String" />
            <column name="pages" type="int" />
            <column name="authorName" type="String" />
            <column name="publishDate" type="Date" />

            <!-- Order -->
            <order by="asc">
                  <order-column name="bookName" />
            </order>

            <!-- Finder methods -->
            <finder name="BookName" return-type="Collection">
                  <finder-column name="bookName" case-sensitive="false"
                        comparator="LIKE" />
            </finder>
            <finder name="Isbn" return-type="Collection">
                  <finder-column name="isbn" case-sensitive="false"
                        comparator="LIKE" />
            </finder>

      </entity>
</service-builder>
```

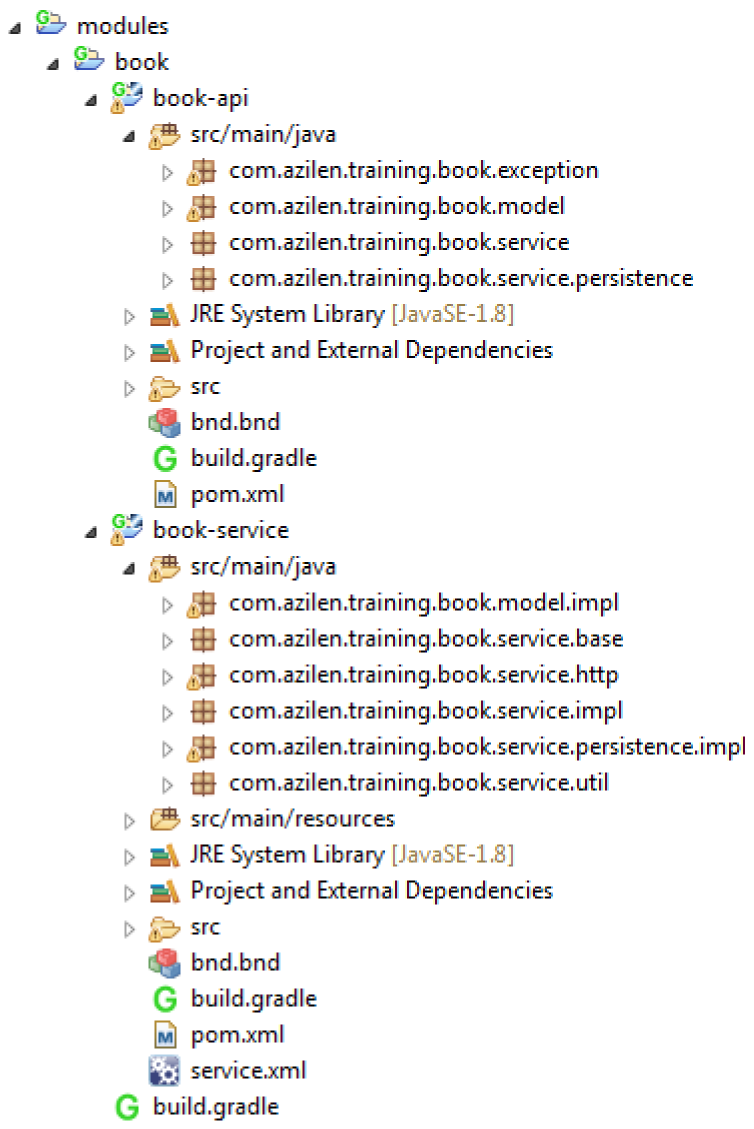
4) Run the following blade command to run the service builder:

   blade gw buildService

5) To deploy the service run “blade deploy” command.

Understanding the generated modules and service layer

After running “blade create” command, two modules will be generated as mentioned above. This is how the project structure will look like after the service builder tool is run:


```
modules
  book
    book-api
      src/main/java
        com.azilen.training.book.exception
        com.azilen.training.book.model
        com.azilen.training.book.service
        com.azilen.training.book.service.persistence
      JRE System Library [JavaSE-1.8]
      Project and External Dependencies
      src
      bnd.bnd
      build.gradle
      pom.xml
    book-service
      src/main/java
        com.azilen.training.book.model.impl
        com.azilen.training.book.service.base
        com.azilen.training.book.service.http
        com.azilen.training.book.service.impl
        com.azilen.training.book.service.persistence.impl
        com.azilen.training.book.service.util
      src/main/resources
      JRE System Library [JavaSE-1.8]
      Project and External Dependencies
      src
      bnd.bnd
      build.gradle
      pom.xml
      service.xml
    build.gradle
```

As you see here, the whole generated API is divided between two modules i.e. book-api and book-service. Since Liferay 7 favours modularity, the OSGi style of module architecture is used for service builder modules. All the interfaces and contract-classes are included in the book-api module. The implementation related classes are taken care by the book-service module.

In Liferay 6.2, the service builder tool used to create the interfacing and implementation classes in the same plugin and wrap all the api related interface & classes into a service-jar. In Liferay 7, service builder tool creates two modules and if any other module wants to use the services, it needs to refer the *-api module. The implementation service objects will be injected by the OSGi runtime.

The familiar service.xml file will be generated in the *-service module. The format of the service.xml file has not changed from the 6.2 version (so, all your previous knowledge of service builder still applies here). Apart from the format of service.xml file, the class hierarchy of generated classes is not changed from Liferay 6.2. Followings are some other concepts that remain unchanged in Liferay 7.

- Custom SQL
- Dynamic Query

- Finder methods and finder classes
- Adding new methods into EntityImpl, EntityServiceImpl and EntityLocalServiceImpl classes
- Model Hints

Using service builder services in Portlets

Now that our custom service is created and deployed, it's time to make use of that service. We will step by step see how we can utilize the book service in a portlet.

1) Create a portlet.
   As we have already seen how to create a portlet, run the following command to create an MVC portlet :
   blade create –t mvc-portlet –p com.azilen.training.book.web –c Book book-web
2) For injecting the service object in our portlet, @Reference annotation is used.
   Now open the BookPortlet.java and paste the following code.

```java
package com.azilen.training.book.web.portlet;

import java.io.IOException;
import java.util.List;

import javax.portlet.Portlet;
import javax.portlet.PortletException;
import javax.portlet.RenderRequest;
import javax.portlet.RenderResponse;

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

import com.azilen.training.book.model.Book;
import com.azilen.training.book.service.BookLocalService;
import com.liferay.portal.kernel.portlet.bridges.mvc.MVCPortlet;

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

	private BookLocalService bookLocalService;

	@Reference
	public void setBookLocalService(BookLocalService bookLocalService) {
		this.bookLocalService = bookLocalService;
	}

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

		List<Book> bookList = bookLocalService.getBooks(-1, -1);

		renderRequest.setAttribute("bookList", bookList);

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

In this portlet, we are getting the list of all books and passing that list to the jsp file where we can display the same in desired format.
As you must have noted, we have used @Reference annotation to bind the service exposed by the api module.

Creating Service Wrappers to implement the custom logic

Customization of any application is extremely significant for both individual product performance and commercial success of the product. Liferay 7 offers Service wrappers to handle specific customizations. This section covers:

Creation of new wrapper module

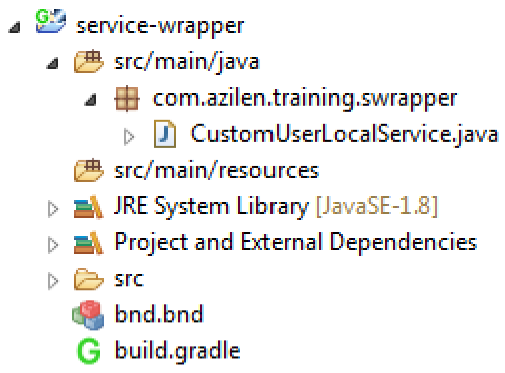
In the “service builder” section, we saw how to create custom entities and create service layer for those custom entities. Liferay’s own service layer is also built using service builder tool. What if we want to customize any service method from Liferay’s services? Well, “Service Wrapper” is the way for you.

In Liferay 6.2, we used to create **service wrapper hooks** to override Liferay’s inbuilt (OOTB) services. In Liferay 7, as OSGi module approach is used, we can create service wrapper modules to implement custom logic over OOTB services.

Let’s create a service wrapper module that overrides method of UserLocalService.

1) Create new service wrapper module.
   The blade command format:
   blade create –t service-wrapper [-p package-name] [-c class-name] [-s service-wrapper-class-name] module-name

   For our example, run following command:
   blade create –t service-wrapper –p com.azilen.training. swrapper –c CustomUserLocalService –s com.liferay.portal.kernel.service.UserLocalServiceWrapper service-wrapper
2) A new module, named service-wrapper, will be created in the “modules” directory of the Liferay workspace. The project structure of the service-wrapper module will be :



```
service-wrapper
  src/main/java
    com.azilen.training.swrapper
      CustomUserLocalService.java
  src/main/resources
  JRE System Library [JavaSE-1.8]
  Project and External Dependencies
  src
  bnd.bnd
  build.gradle
```

3) Now, open the CustomUserLocalService class and write following code.

```
package com.azilen.training.swrapper;

import java.util.Map;

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

import com.liferay.portal.kernel.exception.PortalException;
import com.liferay.portal.kernel.log.Log;
import com.liferay.portal.kernel.log.LogFactoryUtil;
import com.liferay.portal.kernel.service.ServiceWrapper;
import com.liferay.portal.kernel.service.UserLocalServiceWrapper;

@Component(
	immediate = true,
	property = {
	},
	service = ServiceWrapper.class
)
public class CustomUserLocalService extends UserLocalServiceWrapper {

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

	public CustomUserLocalService() {
		super(null);
	}

	@Override
	public int authenticateByEmailAddress(long companyId, String emailAddress,
		String password, Map<String, String[]> headerMap, Map<String,
		String[]> parameterMap, Map<String, Object> resultsMap)
			throws PortalException {

		// Custom logic
		_log.info("Custom Implementation of authenticateByEmailAddress()");

		// Calling Liferay's default implementation of service
		return super.authenticateByEmailAddress(companyId, emailAddress,
			password, headerMap, parameterMap, resultsMap);
	}

}
```

4) Here, we have overridden the authenticateByEmailAddress() method. We are just adding an informative message before calling the default/super implementation of the method. Note that in the @Component annotation, the type of “service” is ServiceWrapper.class.
5) Now deploy your module to the Liferay instance by this command :
   blade deploy
6) When you sign in to the portal using email address, you will see the message in the server log.
7) Similarly, using service wrapper modules, you can override any methods of Liferay’s OOTB services from any service class.

Blade Service Template to create new Liferay modules

According to Liferay team itsef, creating Liferay modules in a workspace using Blade CLI is very similar to creating them in a standalone environment. This section will cover step by step guidelines about:

Creation of a module using service template

Blade tool provides number of templates to help developers create different kinds of modules. One of the templates provided by blade tools is “service” template. This template is used allows to create a Liferay service as a Liferay module. In other words, service template helps to create a DS component module.

com.liferay.portal.kernel.events.LifecycleAction is one of the extension points that can be leveraged by service template. In Liferay 7, LifecycleAction interface replaces all legacy lifecycle events API such as com.liferay.portal.kernel.events.Action, com.liferay.portal.kernel.events.SessionAction, and com.liferay.portal.kernel.events.SimpleAction. OSGi service property “key” identifies a particular lifecycle event.

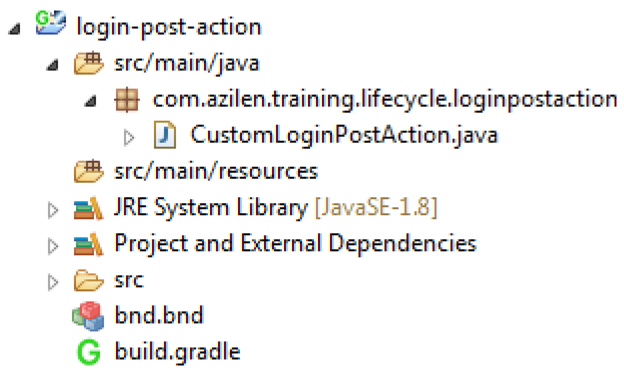
Let’s understand this by taking an example of module which specifies login post action. We will create a post login action which redirects user to his/her private page.

Follow the steps below:

1) Create a new module using “service” template :
   The blade command format:
   blade create –t service [-p package-name] [-c class-name] [-s service-class] module-name

   For our example, run the following command:
   blade create –t service –p com.azilen.training.lifecycle.loginpostaction –c CustomLoginPostAction –s com.liferay.portal.kernel.events.LifecycleAction login-post-action

2) A new module, named “login-post-action”, will be created in “modules” directory of the Liferay workspace. The project structure will be :



```
login-post-action
    src/main/java
        com.azilen.training.lifecycle.loginpostaction
            CustomLoginPostAction.java
    src/main/resources
    JRE System Library [JavaSE-1.8]
    Project and External Dependencies
    src
    bnd.bnd
    build.gradle
```

3) Now, open the CustomLoginPostAction class and paste the following code.

```
package com.azilen.training.lifecycle.loginpostaction;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpSession;

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

import com.liferay.portal.kernel.events.ActionException;
import com.liferay.portal.kernel.events.LifecycleAction;
import com.liferay.portal.kernel.events.LifecycleEvent;
import com.liferay.portal.kernel.exception.PortalException;
import com.liferay.portal.kernel.log.Log;
import com.liferay.portal.kernel.log.LogFactoryUtil;
import com.liferay.portal.kernel.model.User;
import com.liferay.portal.kernel.struts.LastPath;
import com.liferay.portal.kernel.util.PortalUtil;
import com.liferay.portal.kernel.util.StringPool;
import com.liferay.portal.kernel.util.WebKeys;

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

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

	@Override
	public void processLifecycleEvent(LifecycleEvent lifecycleEvent) throws
		ActionException {

		HttpServletRequest request = lifecycleEvent.getRequest();

		User user = null;
		try {
			user = PortalUtil.getUser(request);
		} catch (PortalException e) {
			_log.error(e);
		}

		if (user != null) {
			LastPath lastPath = new LastPath(StringPool.BLANK, "/user/" +
				user.getScreenName());

			HttpSession session = request.getSession();

			session.setAttribute(WebKeys.LAST_PATH, lastPath);
		}
	}

}
```

- Note that we have provided an OSGi service propery named “key” which identifies the

lifecycle event as login.events.post.
- The OSGi service type is LifecycleAction.class.
- We have implemented the method `processLifecycleEvent(LifecycleEvent lifecycleEvent)`.

4) Now deploy your module to the Liferay instance by this command :
   blade deploy

5) When a user signs in to the portal, he/she will be redirected to his/her private pages.

Apart from this lifecycle event, there are many other such portal lifecycle events supported like:

login.events.post,
logout.events.post,
logout.events.pre,
global.shutdown.events,
global.startup.events,
application.shutdown.events,
application.startup.events

Complete list can be found at:
https://github.com/liferay/liferay-blade-samples#logineventspre

Using OSGi Services in portlets – The real potential of Liferay 7

The indispensable role of OSGi in Liferay 7 is out rightly clear so far. It is vital to understand optimum usage of OSGi service to make the best out of it. This article covers:

Creating OSGi Service

Consuming OSGi Services

We have already seen how to use service builder and create modules containing custom entities along with service layer in Liferay 7. When it comes to services, OSGi can unleash a lot of potential. OSGi platform supports creating and registering dynamic services and those services can be injected in other modules.

For a service to be available, its implementation must be registered in the OSGi runtime. The consumer modules can use @Reference annotation to get the service object injected. Let’s see this in action.

We will create a calculator OSGi service module and another module for the consumer portlet. To keep these two related modules together, we will put them in one directory named “calc”. Firstly, we will create a service module to register and expose the service in OSGi runtime.

Creating OSGi Service

1) Blade tool provides an activator module, by which we will be registering the service with OSGi when the module is being loaded.
   To create a module with activator template, the command format is:
   blade create –t activator [-p package-name] [-c class-name] module-name

   For our example, run the following command in the directory modules/calc:
   blade create –t activator –p com.azilen.training.calc.activator –c CalcServiceActivator calc-service
2) A module named “calc-service” will be created under “modules/calc” directory. The command will auto-generate activator class com.azilen.training.calc.activator.CalcServiceActivator.
3) Now we will add an interface for the service. Create an interface CalcService in package com.azilen.training.calc.service.
   We will keep one method for adding two integers as shown below.

```
package com.azilen.training.calc.service;

public interface CalcService {

	int add(int a, int b);

}
```

4) We will need to add an implementation class for the service method. For that, create a class named CalcServiceImpl implementing CalcService. The class will look like this.

```java
package com.azilen.training.calc.service.impl;

import com.azilen.training.calc.service.CalcService;

public class CalcServiceImpl implements CalcService {

	@Override
	public int add(int a, int b) {
		return a + b;
	}

}
```

5) It's time to modify the activator to register the service.

```java
package com.azilen.training.calc.activator;

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

import com.azilen.training.calc.service.CalcService;
import com.azilen.training.calc.service.impl.CalcServiceImpl;

public class CalcServiceActivator implements BundleActivator {

	private ServiceRegistration<CalcService> reg;

	@Override
	public void start(BundleContext bundleContext) throws Exception {

		reg = bundleContext.registerService(CalcService.class, new
			CalcServiceImpl(), null);
	}

	@Override
	public void stop(BundleContext bundleContext) throws Exception {

		reg.unregister();
	}
}
```

The CalcServiceActivator class implements BundleActivator interface which comes from OSGi API. We need to define two bundle lifecycle methods i.e. start() and stop().
When the OSGi runtime starts the bundle, it will call start method and when the bundle is stopped the OSGi runtime will call the activator's stop method.
To have activator class is not a compulsion for an OSGi module though.

6) In the start method of the CalcServiceActivator, we register the service using the BundleContext object. We need to provide the service type (Class object) along with the service implementation object. In the stop() method we have unregistered the service.
7) As a final step, we need to mention the service package in the export package list of the module. To do this, modify bnd.bnd file and add Export-Package header as below.

```
Bundle-Activator: com.azilen.training.calc.activator.CalcServiceActivator
Bundle-Name: calc-service
Bundle-SymbolicName: com.azilen.training.calc.activator
Bundle-Version: 1.0.0
Export-Package: com.azilen.training.calc.service
```

Our calculator service is now ready to be used.
Note that the service implementation class and its package will be private to the module and won't be available for other modules. In real world, the service api and service implementation classes and interfaces are put in separate modules.

Consuming OSGi Services

We will create a calculator portlet to demonstrate the usage of calculator service.

1) Create an mvc portlet by running the below command in modules/calc directory.
   blade create –t mvc-porlet –p com.azilen.training.calc.web –c CalcPortlet
2) You need to add service module as a compile time dependency in gradle build file to prevent gradle from giving you errors in eclipse.

```
dependencies {
	compileOnly group: "com.liferay.portal", name: "com.liferay.portal.kernel", 
version: "2.0.0"
	compileOnly group: "com.liferay.portal", name: "com.liferay.util.taglib", 
version: "2.0.0"
	compileOnly group: "javax.portlet", name: "portlet-api", version: "2.0"
	compileOnly group: "javax.servlet", name: "javax.servlet-api", version: 
"3.0.1"
	compileOnly group: "jstl", name: "jstl", version: "1.2"
	compileOnly group: "org.osgi", name: "org.osgi.compendium", version: "5.0.0"
	compileOnly project(":modules:calc:calc-service")
}
```

3) Modify the CalcPorlet's code as below

```
package com.azilen.training.calc.web.portlet;

import java.io.IOException;

import javax.portlet.Portlet;
import javax.portlet.PortletException;
import javax.portlet.RenderRequest;
import javax.portlet.RenderResponse;

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

import com.azilen.training.calc.service.CalcService;
import com.liferay.portal.kernel.log.Log;
import com.liferay.portal.kernel.log.LogFactoryUtil;
import com.liferay.portal.kernel.portlet.bridges.mvc.MVCPortlet;

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

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

	private CalcService _calcService;

	@Reference
	public void setCalcService(CalcService calcService) {
		this._calcService = calcService;
	}

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

		_log.info(_calcService.add(10, 15));

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

We have declared a service reference _calcService and provided its setter method.
The @Referene annotation will take care of the service injection. OSGi will inject proper implementation object in the Portlet class.

4) In the default render method i.e. doView(), we have used _calcService to add two integers. When the portlet is put on page and its render method is called, you can see the result of the method in the logs.
5) It is advisable to null-check the service reference as the OSGi services are dynamic and can be brought down at any time by stopping/uninstalling the service module.

Once you are done with understanding of Services and Modules for Liferay 7, you are ready to make start with OOTB customizations.

UNITED STATES	BELGIUM	INDIA
Azilen Technologies LLC 6476, Orchard Lake Road, Ste D, West Bloomfield, MI 48325	Azilen Technologies BVBA Diestseweg 32C, 2440 Geel, Belgium	Azilen Technologies Pvt. Ltd. 404/405,Iscon Mall, Near Shivranjani Cross Road, Satellite, Ahmedabad-380015
Tel: +1-972-325-2243	Tel: +32 3 8082588	Tel: +91-79 4009 3121

Email : info@azilen.com | Website : www.azilen.com