# INTRODUCTION TO OSGI

## Why OSGI ?

The purpose of this document is to give a basic idea about OSGI, formerly known as Open Service Gateway Initiative. To understand OSGI, first we need to understand why it was developed in the first place. This document assumes the reader have some basic idea about Modular Programming and component based software, because OSGI is basically about how to build and use modularized and component based applications with a given framework. If not, please go through below links

https://www.infoq.com/articles/modular-java-what-is-it

https://www.infoq.com/articles/modular-java-static-modularity

https://www.infoq.com/articles/modular-java-dynamic-modularity

I will discuss about problems before OSGI a developer used to face and what led to the development of technology like OSGI. After reading this document, one can get a basic idea about how to answer following questions :

- Why was OSGI developed ?
- What is the base of OSGI ?
- What are the benefits of using OSGI ?
- Why does it even exists ?

…and so on.

To answer these questions, we will see some basic reasons which will explain the story of developing OSGI.

**Reason 1**: If we have a closer look at Java class path concept, it really hurts. It does bring benefits to use classes in great manner, but it also has some disadvantages. We put all our classes in a single JAR file and we include it in our application's class path to distribute them. At run time Java blindly bags all the classes and classes with the same name overshadow each other and they interact together in unexpected way. This problem is less known in standalone applications, but its common in distributed applications.

OSGI brings a proper system and provides modularity at runtime. It introduces a powerful service model, sometimes referred as Service Oriented Architecture (SOA). To avoid unexpected behavior of classes with same name at runtime, OSGI provides versioning system. So that only the exact class can be used.

**Reason 2:** Suppose we have an application which follows a complex architecture. Complex architecture in the sense, modules in our applications are tightly coupled. Any small modification

in our applications requires rebuilding and deploying the whole application. Which is the common problem any developer/enterprise would like to avoid. Even though somehow, we designed a new architecture which has less tightly coupled modules. But still proper communication between modules, usage of services, and management of external dependencies were hard to manage.

OSGI provides, a module system for Java. It provides a way to export/import Java Packages, efficient management of dependencies and services. It introduces the concept of ***Bundles***, which is nothing but a JAR file with some extra constraints which we will discuss later. Bundles can be dynamically added/removed without stopping the system. Bundles are also referred as "Plug-Ins".

OSGi was originally developed to support high-end embedded systems such as set-top boxes, which motivates the explicit dependencies and versioning, as well as making it fairly light weight when used as a more recent, enterprise-side container.

**Reason 3:** We can see in most of the Non-OSGI or traditional JEE/EAR based applications, they usually have complex modularity. If a team is working on a module which may be dependent on another module which is unknown or yet to be developed. The modules dependencies are exposed at runtime to each other, leaving the module's internal parts also exposed which is insecure.

OSGI has mechanisms to ensure that the modules boundaries are respected by the development team. OSGI introduces a way, in which a development team has a clear idea about a modules dependencies and services. OSGI prevents unwanted coupling, because the dependencies of a modules are explicitly declared. Also, OSGI service platform is secure.

OSGI provides full information of a module at the runtime. Such as we can see each bundle's state, imported/exported packages, versions which makes a bundle isolate.

In OSGI, each plugin or bundle is a versioned artifact, which has its own class loader. Each bundle depends on a specific JAR file which it contains and also on another bundle's JAR. Because of the versioning and isolated class loaders, different versions of the same artifact can be loaded at the same time.

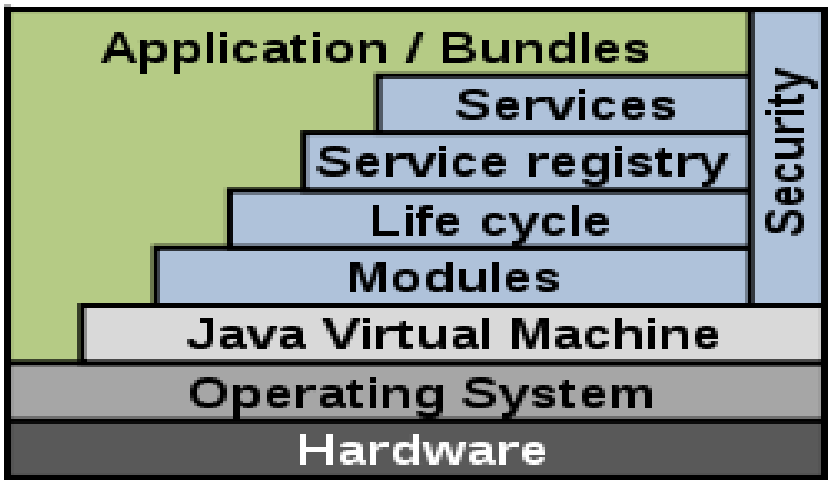
**NOTE** : One of the biggest advantage I see in OSGI is, it has a strong versioning mechanism. The dependencies are verified at the time of deployment itself instead of getting NoClassDefFound Error at run-time.

## What is OSGI ?

The OSGi Alliance, formerly known as the Open Services Gateway initiative, is an open standards organization founded in March 1999 that originally specified and continues to maintain the OSGi standard.

The OSGi specification describes a modular system and a service platform for the Java programming language that implements a complete and dynamic component model, something that does not exist in standalone Java/VM environments. Applications or components, coming in the form of bundles for deployment, can be remotely installed, started, stopped, updated, and uninstalled without requiring a reboot; management of Java packages/classes is specified in great detail. Application life cycle management is implemented via APIs that allow for remote downloading of management policies. The service registry allows bundles to detect the addition of new services, or the removal of services, and adapt accordingly.

**OSGI Architecture :**




OSGi is a Java framework for developing and deploying modular software programs and libraries. Each bundle is a tightly coupled, dynamically loadable collection of classes, jars, and configuration files that explicitly declare their external dependencies (if any).

The above image describes the OSGI architecture. Let's have a closer look at each term:

**Bundles**

Bundles are normal JAR components with extra manifest headers.

**Services**

The services layer connects bundles in a dynamic way by offering a publish-find-bind model for plain old Java interfaces (POJIs) or plain old Java objects (POJOs).

**Services Registry**

The application programming interface for management services (ServiceRegistration, ServiceTracker and ServiceReference).

**Life-Cycle**

The application programming interface for life cycle management (install, start, stop, update, and uninstall) for bundles.

**Modules**

The layer that defines encapsulation and declaration of dependencies (how a bundle can import and export code).

**Security**

The layer that handles the security aspects by limiting bundle functionality to pre-defined capabilities.

Execution Environment

Defines what methods and classes are available in a specific platform. There is no fixed list of execution environments, since it is subject to change as the Java Community Process creates new versions and editions of Java. However, the following set is currently supported by most OSGi implementations:

- CDC-1.0/Foundation-1.0
- CDC-1.1/Foundation-1.1
- OSGi/Minimum-1.0
- OSGi/Minimum-1.1
- JRE-1.1
- From J2SE-1.2 up to J2SE-1.6

**OSGI Bundles Life Cycle**

A Life Cycle layer adds bundles that can be dynamically installed, started, stopped, updated and uninstalled. Bundles rely on the module layer for class loading but add an API to manage the modules in run time. The life cycle layer introduces dynamics that are normally not part of an application. Extensive dependency mechanisms are used to assure the correct operation of the environment. Life cycle operations are fully protected with the security architecture

| Bundle State | Description |
|---|---|
| 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. |
| UNINSTALLED | The bundle has been uninstalled. It cannot move into another state. |

**Getting Started With OSGI :**

To start working with OSGI, we need an OSGI Framework. There are many OSGI Frameworks, which uses OSGI but we are going to use Apache ServiceMix. The serviceMix acts as a Container which will contain the bundles, where they can be dynamically added or removed and many other operations can be performed.

Apache ServiceMix provides runtime environment for OSGI Bundles. In order to get started with ServiceMix, visit official website and get the latest version (currently 7.0.1) of Apache ServiceMix. Apart from serviceMix we are going to use following basic tools :

- ✓ JDK 1.7 or Later
- ✓ Maven 3.x
- ✓ Eclipse IDE

We will discuss following topics for getting started with OSGI :

1. Setting up Maven to use OSGI
2. Simplest Bundle
3. Setting up the apache serviceMix runtime environment
4. Deploying a bundle
5. Simple Bundle
6. ServiceMix basic commands

1. **Setting up Maven to use OSGI :**

   **Step 1 :** Open eclipse IDE. There are other IDEs around but I mostly prefer Eclipse. One may follow this with other IDEs also. Eclipse provides great support for OSGI. It has many built in tools that we'll discuss later.

   **Step 2:** Create a simple Maven Project. And open the Pom.xml. Our goal is to create an OSGI bundle. So, we are going to configure Pom to create a bundle.

   **Step 3:** Add the following plugin under build section of pom.

```
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.felix</groupId>
            <artifactId>maven-bundle-plugin</artifactId>
            <version>2.3.7</version>
            <extensions>true</extensions>
        </plugin>
    </plugins>
</build>
```

   Change the packaging from JAR to "bundle". Maven automatically creates a bundle using this plug-in.

2. **Simplest Bundle :**

   **Step 1 :** This may not be a simplest possible bundle. But let's start with a simplest example. Create a package say com.example.hello and a class HelloWorld.

```
package com.example.hello;

public class HelloWorld {

}
```

   Our class doesn't do anything yet. But as I said we are going to create simplest bundle.

   **Step 2:** Build the project with maven clean install.

   **Step 3:** Now go to target folder and open the JAR file. Open the manifest.mf file inside META-INF folder. It would look like below :

```
Manifest-Version: 1.0
Bnd-LastModified: 1527147206718
Build-Jdk: 1.8.0_101
Built-By: aman.verma
Bundle-Description: 01-OSGI-Demo
Bundle-ManifestVersion: 2
Bundle-Name: 01-OSGI-Demo
Bundle-SymbolicName: com.example.osgi.01-OSGI-Demo
Bundle-Version: 1.0.0
Created-By: Apache Maven Bundle Plugin
Export-Package: com.example.hello;version="1.0.0"
Tool: Bnd-1.50.0
```

We can see that the manifest headers are way too different than non-OSGI JAR manifests. The above structure is a typical structure for OSGI Bundles. The two entries **Bundle-ManifestVersion** and **Bundle-SymbolicName** are mendatory. We can see that the **Export-Package** entry contains our package com.example.hello with a version number. We didn't specify any export-package in our Pom. Maven automatically puts all the packages associated with a project in export-package entry, if we don't explicitly specify.

Now our simplest bundle is ready for deployment. To deploy a bundle, we need a runtime environment which we downloaded before.

3. **Setting up the Apache ServiceMix Environment :**

**Step 1:** Extract the download archive, and open the parent folder of *apache-servicemix-7.0.1.* To run the serviceMix Server, make sure the Java_Home is set in environment variables.

**Step2:** Go to bin folder and click on "servicemix.bat" file. This will launch the servicemix console. The console is referred as KARAF console which we will discuss later. Wait until karaf@root appears.

Below image shows how the console looks like :

```
Karaf
Java HotSpot(TM) 64-Bit Server VM warning: ignoring option MaxPermSize=256M; support was removed in 8.0
Please wait while Apache ServiceMix is starting...
100% [========================================================================]

Karaf started in 13s. Bundle stats: 225 active, 225 total

  ____                  _          __  __ _
 / ___|  ___ _ ____   _(_) ___ ___|  \/  (_)_  __
 \___ \ / _ \ '__\ \ / / |/ __/ _ \ |\/| | \ \/ /
  ___) |  __/ |   \ V /| | (_|  __/ |  | | |>  <
 |____/ \___|_|    \_/ |_|\___\___|_|  |_|_/_/\_\

  Apache ServiceMix (7.0.1)

Hit '<tab>' for a list of available commands
and '[cmd] --help' for help on a specific command.
Hit '<ctrl-d>' or 'system:shutdown' to shutdown ServiceMix.

karaf@root>
```

4. **Deploying a Bundle :**

**Step 1:** Copy the bundle which we created before into deploy folder of apache servicemix.
Type command “bundle:list” (without quotes) in karaf console.

```
karaf@root>bundle:list
START LEVEL 100 , List Threshold: 50
 ID | State    | Lvl | Version                                | Name
----------------------------------------------------------------------------------------------------------------
  9 | Active   |  50 | 5.14.5                                 | activemq-karaf
 10 | Active   |  50 | 2.6.3                                  | Jackson-annotations
 11 | Active   |  50 | 2.6.3                                  | Jackson-core
 12 | Active   |  50 | 2.6.3                                  | jackson-databind
 22 | Active   |  50 | 3.1.4                                  | activeio-core
 23 | Resolved |  50 | 5.14.5                                 | activemq-blueprint, Hosts: 25
 24 | Active   |  50 | 5.14.5                                 | activemq-camel
 25 | Active   |  50 | 5.14.5                                 | activemq-osgi, Fragments: 23
 38 | Active   |  50 | 1.1.1                                  | Apache Aries Transaction Manager
 40 | Active   |  50 | 2.16.5                                 | camel-blueprint
 41 | Active   |  50 | 2.16.5                                 | camel-catalog
 42 | Active   |  80 | 2.16.5                                 | camel-commands-core
 43 | Active   |  50 | 2.16.5                                 | camel-core
 44 | Active   |  50 | 2.16.5                                 | camel-cxf
 45 | Active   |  50 | 2.16.5                                 | camel-cxf-transport
 46 | Active   |  50 | 2.16.5                                 | camel-jms
 47 | Active   |  50 | 2.16.5                                 | camel-spring
 48 | Active   |  50 | 2.16.5                                 | camel-xstream
 49 | Active   |  80 | 2.16.5                                 | camel-karaf-commands
 51 | Active   |  50 | 3.2.2                                  | Apache Commons Collections
 52 | Active   |  50 | 2.0.1                                  | Commons JEXL
 54 | Active   |  50 | 3.5.0                                  | Apache Commons Net
 55 | Active   |  50 | 1.6.0                                  | Commons Pool
 56 | Active   |  50 | 2.4.2                                  | Apache Commons Pool
 94 | Active   |  50 | 2.0.0                                  | geronimo-j2ee-connector_1.5_spec
 95 | Active   |  50 | 1.0.1                                  | geronimo-j2ee-management_1.1_spec
 99 | Active   |  50 | 3.4.6                                  | ZooKeeper Bundle
129 | Active   |  80 | 2.0.13                                 | Apache MINA Core
132 | Active   |  50 | 7.0.1                                  | Apache ServiceMix :: ActiveMQ :: Camel
133 | Active   |  50 | 7.0.1                                  | Apache ServiceMix :: ActiveMQ :: Service
136 | Active   |  50 | 1.6.1.5                                | Apache ServiceMix :: Bundles :: dom4j
138 | Active   |  50 | 1.9.2.1                                | Apache ServiceMix :: Bundles :: jasypt
142 | Active   |  50 | 1.1.0.4                                | Apache ServiceMix :: Bundles :: jdom
143 | Active   |  50 | 2.3.0.2                                | Apache ServiceMix :: Bundles :: kxml2
156 | Active   |  50 | 1.7.0.6                                | Apache ServiceMix :: Bundles :: velocity
160 | Active   |  50 | 1.1.4.c                                | Apache ServiceMix :: Bundles :: xpp3
161 | Active   |  50 | 1.4.8.1                                | Apache ServiceMix :: Bundles :: xstream
171 | Active   |  50 | 4.2.0                                  | Apache XBean :: OSGI Blueprint Namespace Handler
202 | Active   |  50 | 0.6.4                                  | JAXB2 Basics - Runtime
215 | Active   |  50 | 2.11.0.v20140415-163722-cac6383e66     | Scala Standard Library
239 | Active   |  80 | 1.0.0                                  | 01-OSGI-Demo
karaf@root>
```

We can see our bundle is deployed without any errors. We can change states of our bundle, view bundle info and perform any operation using various commands. We'll see about the commands later.

5. **Simple Bundle :**

The bundle which we created before doesn't perform any operation. Because we didn't implement our HelloWorld class. If we want our bundle should do something when it starts we must implement **BundleActivator** in a class let's say **Activator.java**. The BundleActivator interface has two methods **start()** and **stop().** We will define our service HelloWorld in start(). So that it automatically get started when a bundle starts. The stop() method invocation stops the bundle. Let's create the Activator.java class and build and create the bundle again. We'll call this as simple bundle.

**Step 1:** If we simply create Activator class which implements BundleActivator interface. We'll get an error "*BundleActivator can't be resolved to a type".* Because the interface BundleActivator is not known by our Application. We must add a dependency for the JAR which contains the interface BundleActivator. See below :

```
<dependencies>
    <dependency>
        <groupId>org.osgi</groupId>
        <artifactId>org.osgi.core</artifactId>
        <version>6.0.0</version>
        <scope>provided</scope>
    </dependency>
</dependencies>
```

**Step 2:** Now create the Activator class and implements start() & stop() methods of BundleActivator.

```
1  package com.example.activator;
2
3  import org.osgi.framework.BundleActivator;
4  import org.osgi.framework.BundleContext;
5
6  import com.example.hello.HelloWorld;
7
8  public class Activator implements BundleActivator{
9
10     private HelloWorld helloWorld;
11
12     @Override
13     public void start(BundleContext context) throws Exception {
14         System.out.println("Hello World Started.");
15         helloWorld = new HelloWorld();
16     }
17
18     @Override
19     public void stop(BundleContext context) throws Exception {
20         helloWorld = null;
21         System.out.println("Hello World Stopped.");
22     }
23 }
```

In this class, we are creating an instance of our HelloWorld class in start() method and again discarding it in stop() method.

**Step 3:** The OSGI Framework, i.e. ServiceMix doesn't search through the bundle for the Activator class which implements BundleActivator. We must let ServiceMix to know about our Activator class. For this, there must be another entry in manifest.mf file :

*Bundle-Activator: com.example.activator.Activator*

To make this entry appear in our manifest.mf file we must configure Pom.xml and add this Activator class to maven-bundle-plugin.

```
28    <build>
29        <plugins>
30            <plugin>
31                <groupId>org.apache.felix</groupId>
32                <artifactId>maven-bundle-plugin</artifactId>
33                <version>2.3.7</version>
34                <extensions>true</extensions>
35                <configuration>
36                    <instructions>
37                        <Bundle-Activator>com.example.activator.Activator</Bundle-Activator>
38                    </instructions>
39                </configuration>
40            </plugin>
41        </plugins>
42    </build>
```

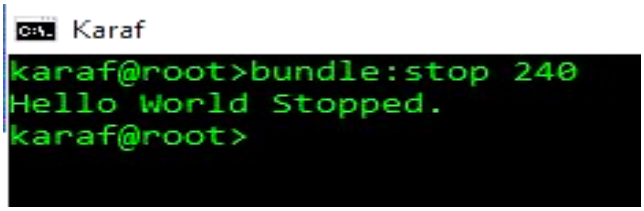
Now build the project, and see the manifest.mf. The Bundle-Activator entry appears.

```
1  Manifest-Version: 1.0
2  Bnd-LastModified: 1527155702315
3  Build-Jdk: 1.8.0_101
4  Built-By: aman.verma
5  Bundle-Activator: com.example.activator.Activator
6  Bundle-Description: 01-OSGI-Demo
7  Bundle-ManifestVersion: 2
8  Bundle-Name: 01-OSGI-Demo
9  Bundle-SymbolicName: com.example.osgi.01-OSGI-Demo
10 Bundle-Version: 1.0.0
11 Created-By: Apache Maven Bundle Plugin
12 Export-Package: com.example.activator;uses:="org.osgi.framework,com.exam
13  ple.hello";version="1.0.0",com.example.hello;version="1.0.0"
14 Import-Package: org.osgi.framework;version="[1.8,2)"
15 Tool: Bnd-1.50.0
```

**Step 4:** Now deploy our new bundle. When bundle starts, the start method gets invoked and a new instance of HelloWorld class is created.

```
Karaf
karaf@root>Hello World Started.
```

If we stop the bundle using bundle:stop command, the stop() gets invoked and the instance of HelloWorld is discarded.




Our HelloWorld class still doesn't have any implementation, its empty.

**Step 5:** If we see above manifest.mf, the Export-Package Entry looks crowded. It contains all the packages in our project including external libraries. We don't want that. We should only export required packages, which we are going to expose, so that other bundles can use the exposed packages or services. In this project we are exposing nothing to another bundle, because we have only one bundle (yet). So, the export-package entry must be empty, or should not exist at all for this example project. This is how security is provided in OSGI Bundles. The dependencies which should be exposed, are only going to be exposed by exporting them.

To export only required dependencies, we can explicitly define them in our pom inside export-package tag.

```
28    <build>
29        <plugins>
30            <plugin>
31                <groupId>org.apache.felix</groupId>
32                <artifactId>maven-bundle-plugin</artifactId>
33                <version>2.3.7</version>
34                <extensions>true</extensions>
35                <configuration>
36                    <instructions>
37                        <Bundle-Activator>com.example.activator.Activator</Bundle-Activator>
38                        <Export-Package>com.example.hello</Export-Package>
39                    </instructions>
40                </configuration>
41            </plugin>
42        </plugins>
43    </build>
```

Now build the project again and see the manifest.mf. We can see that the export-package entry only contains specified package.

```
1  Manifest-Version: 1.0
2  Bnd-LastModified: 1527157785479
3  Build-Jdk: 1.8.0_101
4  Built-By: aman.verma
5  Bundle-Activator: com.example.activator.Activator
6  Bundle-Description: 01-OSGI-Demo
7  Bundle-ManifestVersion: 2
8  Bundle-Name: 01-OSGI-Demo
9  Bundle-SymbolicName: com.example.osgi.01-OSGI-Demo
10 Bundle-Version: 1.0.0
11 Created-By: Apache Maven Bundle Plugin
12 Export-Package: com.example.hello;version="1.0.0"
13 Import-Package: com.example.hello;version="[1.0,2)",org.osgi.framework;v
14  ersion="[1.8,2)"
15 Tool: Bnd-1.50.0
```

6. **ServiceMix Basic Commands :**

There are total 390 commands available in ServiceMix 7.0.1. I am not going to list all the commands here, but some basic commands associated with bundles. We can see the list of all commands at the console by pressing **TAB** key.

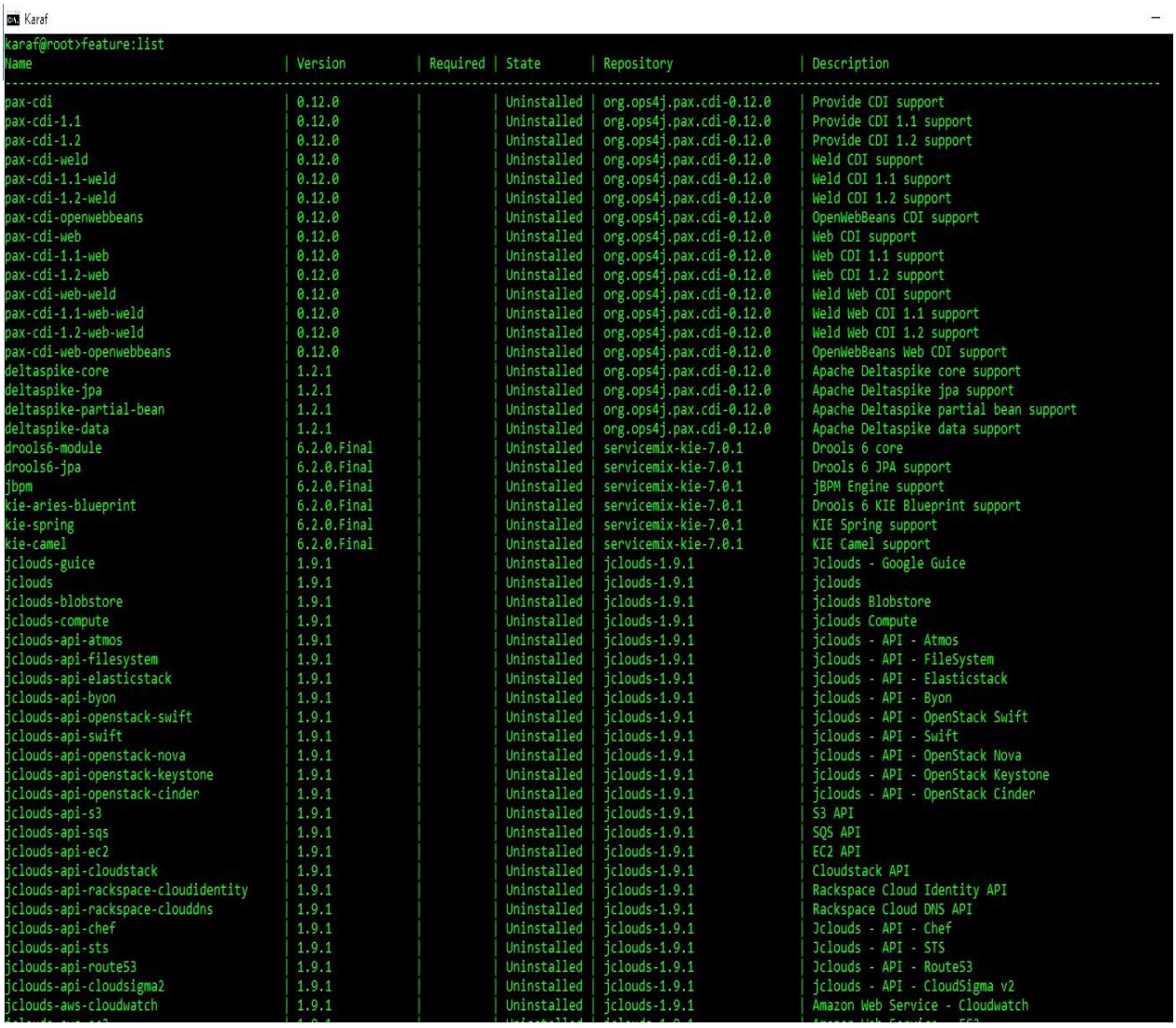
| | |
|---|---|
| *bundle:classes* | Displays list of classes/resources inside a bundle. |
| *bundle:find-class* | Locates specified class in any deployed bundle. |
| *bundle:id* | Gets the bundle id |
| *bundle:headers* | Displays manifest headers of specified bundle |
| *bundle:info* | Displays information about a bundle |
| *bundle:install* | Installs one or more bundles |
| *bundle:list* | Displays list of all the installed bundles |
| *bundle:refresh* | Refreshes the specified bundle |
| *bundle:requirements* | Displays the OSGI requirements of specified bundle |
| *bundle:resolve* | Resolves one or more bundles |
| *bundle:restart* | Restarts one or more bundles |
| *bundle:services* | Lists OSGI services per bundle |
| *bundle:start* | Starts one or more bundles |
| *bundle:status* | Get the status of specified bundle |
| *bundle:stop* | Stops one or more bundles |
| *bundle:tree-show* | Shows the tree of bundles based on wiring information |
| *bundle:uninstall* | Uninstall one or more bundles |
| *bundle:update* | Updates one or more bundles |
| *bundles:watch* | Watches/updates bundles |

(Note) : - Syntax : bundle:start bundle_id Ex: bundle:start 240

| | |
|---|---|
| Ctrl+D | Terminates the console |
| Cmd --help | Get the information/help about any specified command |

**Working with Features :**

ServiceMix contains a list of optional features, which can be installed whenever required such as Spring feature, Web-console Feature, Hibernate, Camel and many more. To see a list of features available, use **feature:list** command.




| Name | Version | Required | State | Repository | Description |
|---|---|---|---|---|---|
| pax-cdi | 0.12.0 | | Uninstalled | org.ops4j.pax.cdi-0.12.0 | Provide CDI support |
| pax-cdi-1.1 | 0.12.0 | | Uninstalled | org.ops4j.pax.cdi-0.12.0 | Provide CDI 1.1 support |
| pax-cdi-1.2 | 0.12.0 | | Uninstalled | org.ops4j.pax.cdi-0.12.0 | Provide CDI 1.2 support |
| pax-cdi-weld | 0.12.0 | | Uninstalled | org.ops4j.pax.cdi-0.12.0 | Weld CDI support |
| pax-cdi-1.1-weld | 0.12.0 | | Uninstalled | org.ops4j.pax.cdi-0.12.0 | Weld CDI 1.1 support |
| pax-cdi-1.2-weld | 0.12.0 | | Uninstalled | org.ops4j.pax.cdi-0.12.0 | Weld CDI 1.2 support |
| pax-cdi-openwebbeans | 0.12.0 | | Uninstalled | org.ops4j.pax.cdi-0.12.0 | OpenWebBeans CDI support |
| pax-cdi-web | 0.12.0 | | Uninstalled | org.ops4j.pax.cdi-0.12.0 | Web CDI support |
| pax-cdi-1.1-web | 0.12.0 | | Uninstalled | org.ops4j.pax.cdi-0.12.0 | Web CDI 1.1 support |
| pax-cdi-1.2-web | 0.12.0 | | Uninstalled | org.ops4j.pax.cdi-0.12.0 | Web CDI 1.2 support |
| pax-cdi-web-weld | 0.12.0 | | Uninstalled | org.ops4j.pax.cdi-0.12.0 | Weld Web CDI support |
| pax-cdi-1.1-web-weld | 0.12.0 | | Uninstalled | org.ops4j.pax.cdi-0.12.0 | Weld Web CDI 1.1 support |
| pax-cdi-1.2-web-weld | 0.12.0 | | Uninstalled | org.ops4j.pax.cdi-0.12.0 | Weld Web CDI 1.2 support |
| pax-cdi-web-openwebbeans | 0.12.0 | | Uninstalled | org.ops4j.pax.cdi-0.12.0 | OpenWebBeans Web CDI support |
| deltaspike-core | 1.2.1 | | Uninstalled | org.ops4j.pax.cdi-0.12.0 | Apache Deltaspike core support |
| deltaspike-jpa | 1.2.1 | | Uninstalled | org.ops4j.pax.cdi-0.12.0 | Apache Deltaspike jpa support |
| deltaspike-partial-bean | 1.2.1 | | Uninstalled | org.ops4j.pax.cdi-0.12.0 | Apache Deltaspike partial bean support |
| deltaspike-data | 1.2.1 | | Uninstalled | org.ops4j.pax.cdi-0.12.0 | Apache Deltaspike data support |
| drools6-module | 6.2.0.Final | | Uninstalled | servicemix-kie-7.0.1 | Drools 6 core |
| drools6-jpa | 6.2.0.Final | | Uninstalled | servicemix-kie-7.0.1 | Drools 6 JPA support |
| jbpm | 6.2.0.Final | | Uninstalled | servicemix-kie-7.0.1 | jBPM Engine support |
| kie-aries-blueprint | 6.2.0.Final | | Uninstalled | servicemix-kie-7.0.1 | Drools 6 KIE Blueprint support |
| kie-spring | 6.2.0.Final | | Uninstalled | servicemix-kie-7.0.1 | KIE Spring support |
| kie-camel | 6.2.0.Final | | Uninstalled | servicemix-kie-7.0.1 | KIE Camel support |
| jclouds-guice | 1.9.1 | | Uninstalled | jclouds-1.9.1 | Jclouds - Google Guice |
| jclouds | 1.9.1 | | Uninstalled | jclouds-1.9.1 | jclouds |
| jclouds-blobstore | 1.9.1 | | Uninstalled | jclouds-1.9.1 | jclouds Blobstore |
| jclouds-compute | 1.9.1 | | Uninstalled | jclouds-1.9.1 | jclouds Compute |
| jclouds-api-atmos | 1.9.1 | | Uninstalled | jclouds-1.9.1 | jclouds - API - Atmos |
| jclouds-api-filesystem | 1.9.1 | | Uninstalled | jclouds-1.9.1 | jclouds - API - FileSystem |
| jclouds-api-elasticstack | 1.9.1 | | Uninstalled | jclouds-1.9.1 | jclouds - API - Elasticstack |
| jclouds-api-byon | 1.9.1 | | Uninstalled | jclouds-1.9.1 | jclouds - API - Byon |
| jclouds-api-openstack-swift | 1.9.1 | | Uninstalled | jclouds-1.9.1 | jclouds - API - OpenStack Swift |
| jclouds-api-swift | 1.9.1 | | Uninstalled | jclouds-1.9.1 | jclouds - API - Swift |
| jclouds-api-openstack-nova | 1.9.1 | | Uninstalled | jclouds-1.9.1 | jclouds - API - OpenStack Nova |
| jclouds-api-openstack-keystone | 1.9.1 | | Uninstalled | jclouds-1.9.1 | jclouds - API - OpenStack Keystone |
| jclouds-api-openstack-cinder | 1.9.1 | | Uninstalled | jclouds-1.9.1 | jclouds - API - OpenStack Cinder |
| jclouds-api-s3 | 1.9.1 | | Uninstalled | jclouds-1.9.1 | S3 API |
| jclouds-api-sqs | 1.9.1 | | Uninstalled | jclouds-1.9.1 | SQS API |
| jclouds-api-ec2 | 1.9.1 | | Uninstalled | jclouds-1.9.1 | EC2 API |
| jclouds-api-cloudstack | 1.9.1 | | Uninstalled | jclouds-1.9.1 | Cloudstack API |
| jclouds-api-rackspace-cloudidentity | 1.9.1 | | Uninstalled | jclouds-1.9.1 | Rackspace Cloud Identity API |
| jclouds-api-rackspace-clouddns | 1.9.1 | | Uninstalled | jclouds-1.9.1 | Rackspace Cloud DNS API |
| jclouds-api-chef | 1.9.1 | | Uninstalled | jclouds-1.9.1 | Jclouds - API - Chef |
| jclouds-api-sts | 1.9.1 | | Uninstalled | jclouds-1.9.1 | Jclouds - API - STS |
| jclouds-api-route53 | 1.9.1 | | Uninstalled | jclouds-1.9.1 | Jclouds - API - Route53 |
| jclouds-api-cloudsigma2 | 1.9.1 | | Uninstalled | jclouds-1.9.1 | jclouds - API - CloudSigma v2 |
| jclouds-aws-cloudwatch | 1.9.1 | | Uninstalled | jclouds-1.9.1 | Amazon Web Service - Cloudwatch |

**The Web-Console Feature :**

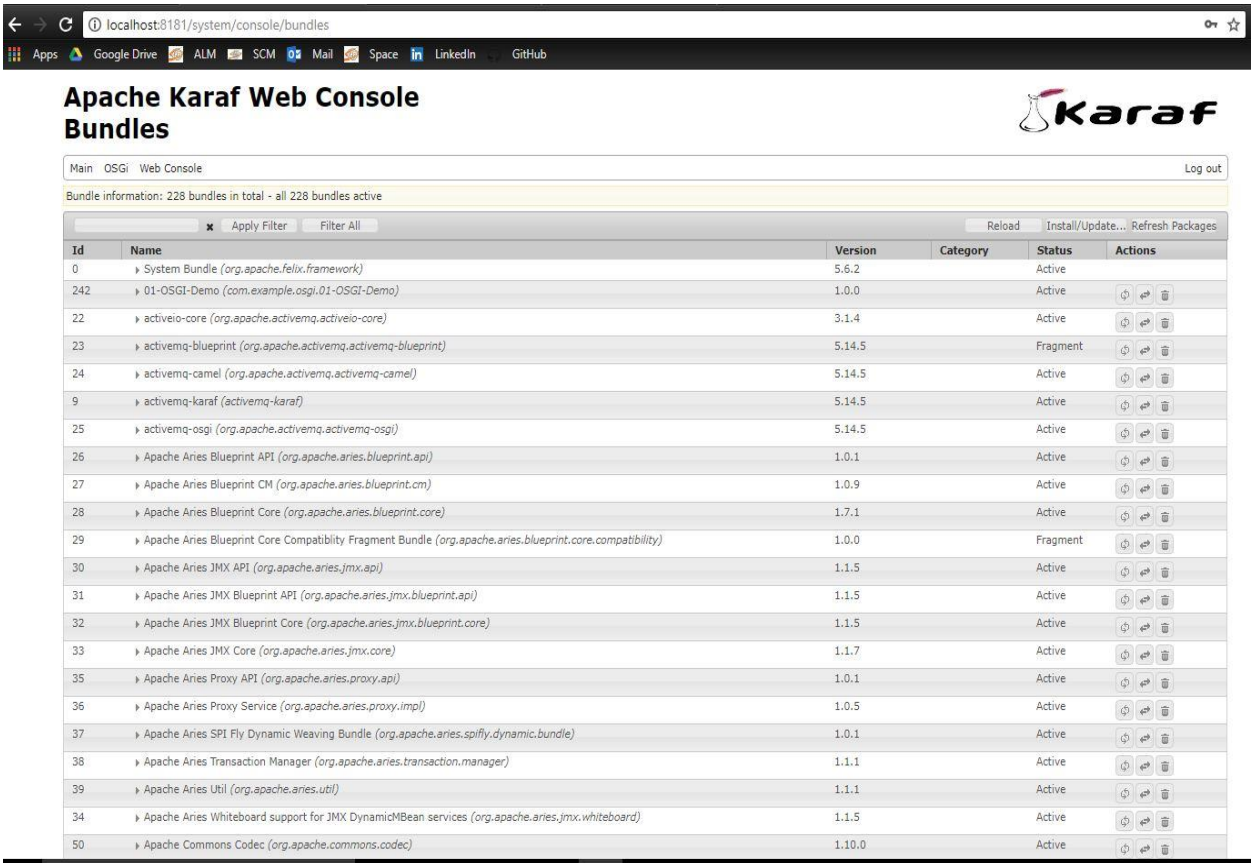
The web-console feature is very important feature in serviceMix. It is nothing but a graphical view of the cmd console with additional features. To access web-console, it must be installed using below command :

feature:install webconsole

To access the web-console, use the default URL of serviceMix mentioned below :

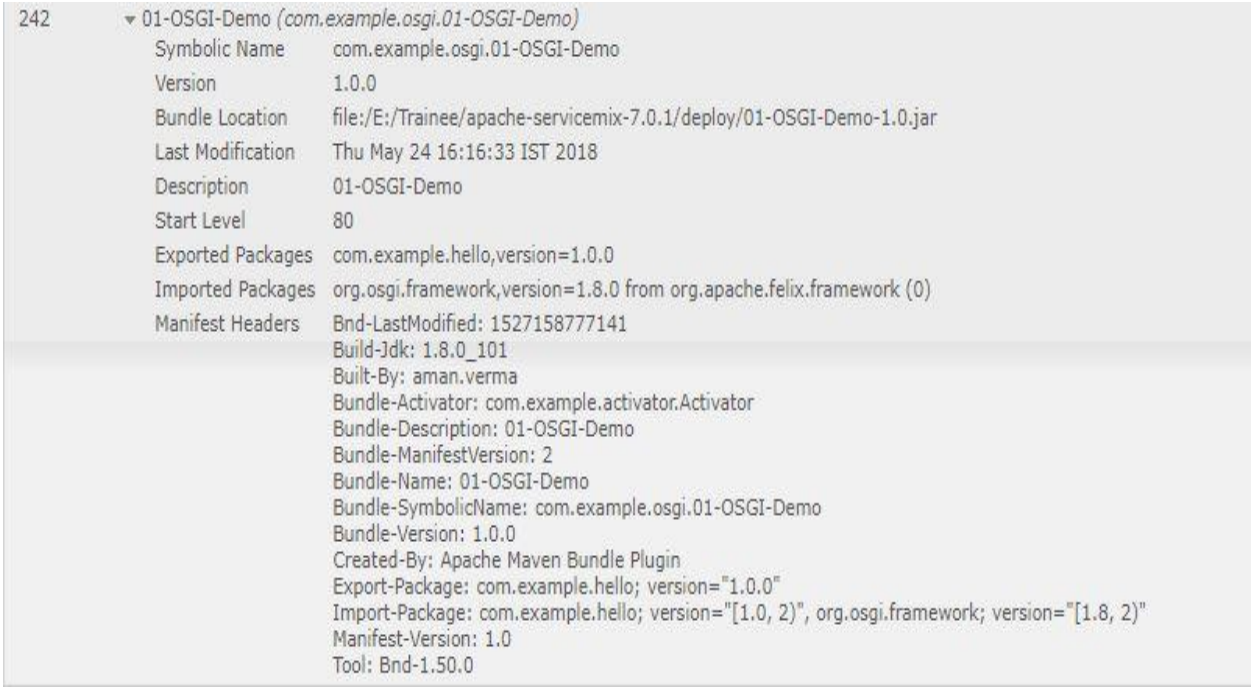
localhost:8181/system/console/bundles

Login using below credentials:
User ID – karaf
Password – karaf
After successful login, the web console appears. See below:




We can see every detail here what we could see in cmd based console. We can access cmd console from web-console using **GoGo** option of menu option Main.
Select Main->Gogo.

We can see manifest headers of a particular bundle by expanding it.




Same way we can install Spring/Hibernate features. For hibernate version 5.x there is no feature available in service mix 7.0.1. To use hibernate version above 5.0 we need to explicitly enforce the hibernate version 5.x. A documentation on similar issue available at space page mentioned below. Please visit and follow the instructions.
https://space.sysbiz.org/display/HOW/Apache+Karaf+Configuration

**Converting any Non-OSGI JAR to OSGI Bundle :**

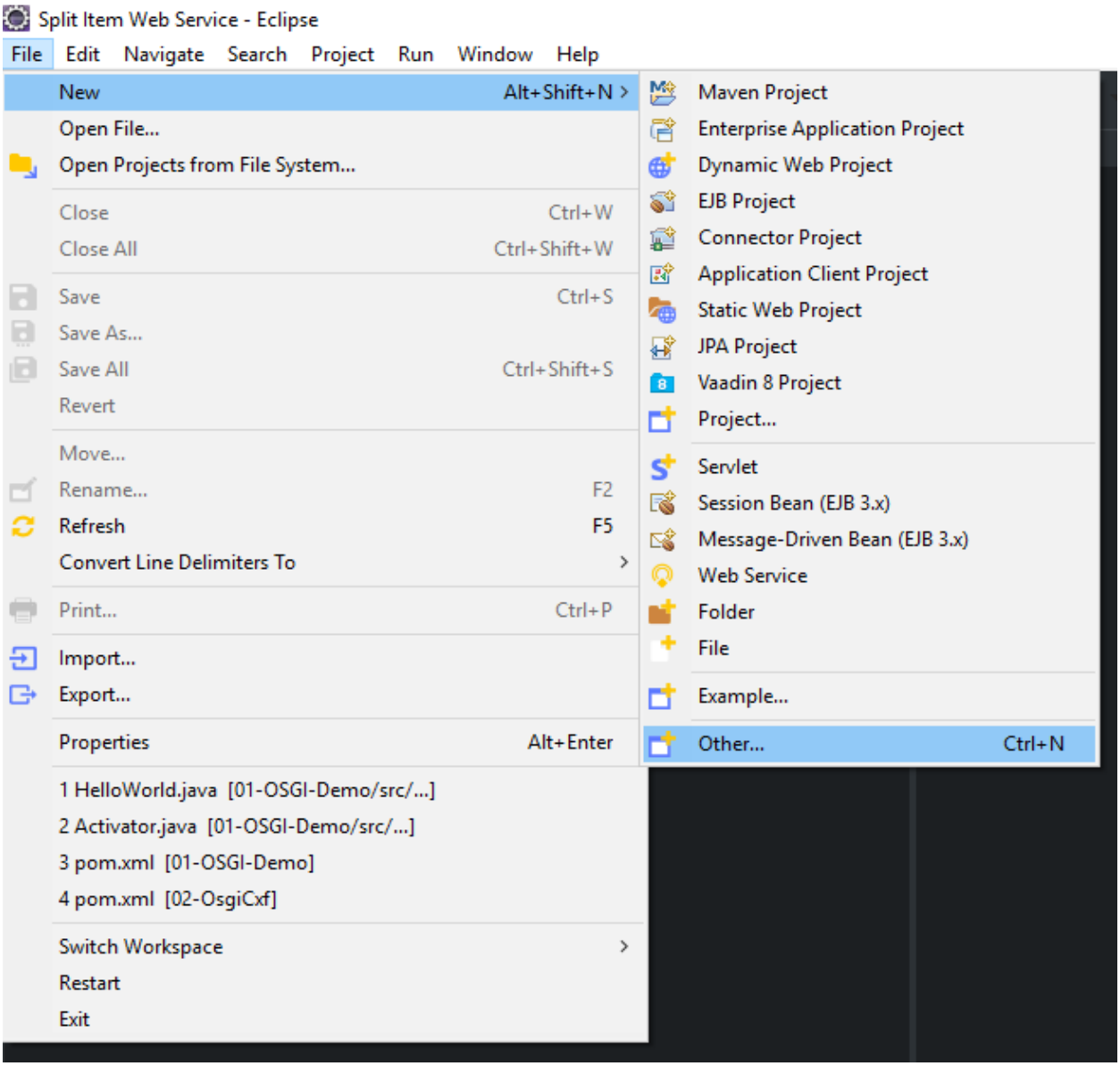
There are 3 ways to convert a non-osgi Jar to osgi bundle.

**Method 1: Using Built-In Eclipse Tool Plug-in Development**

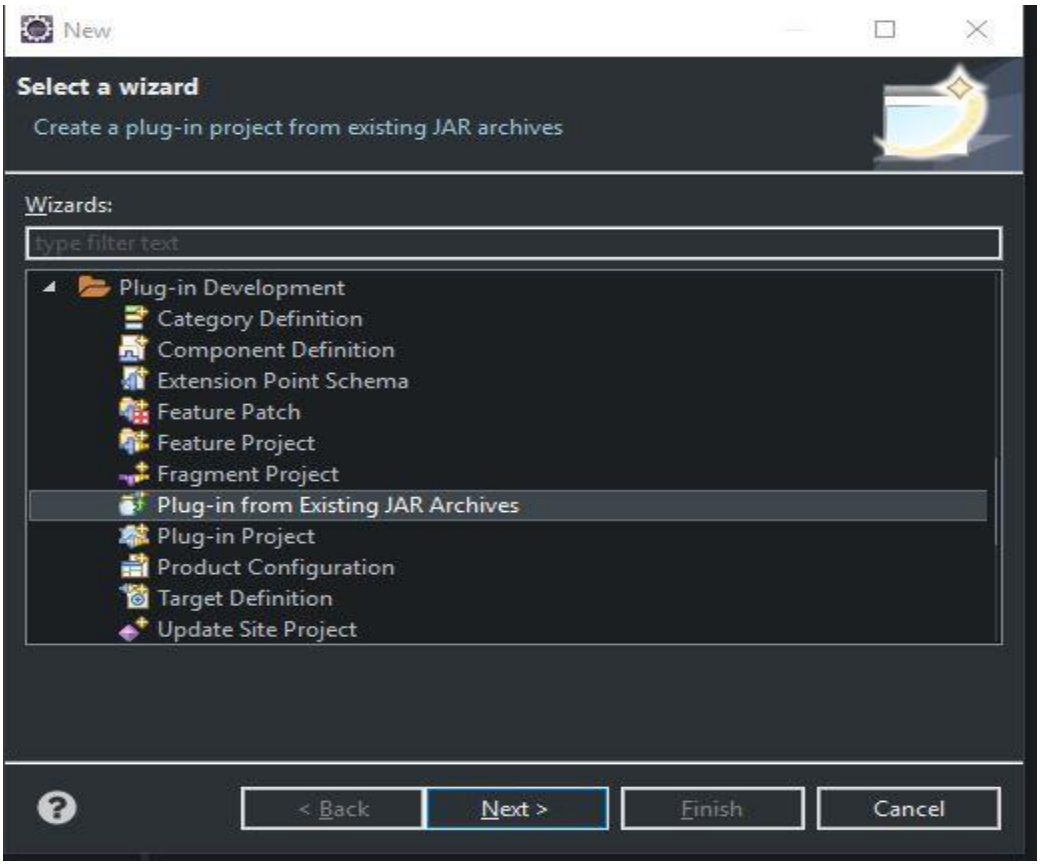
**Step 1 :** Go to File->New->Other.

Or press Ctrl+N.

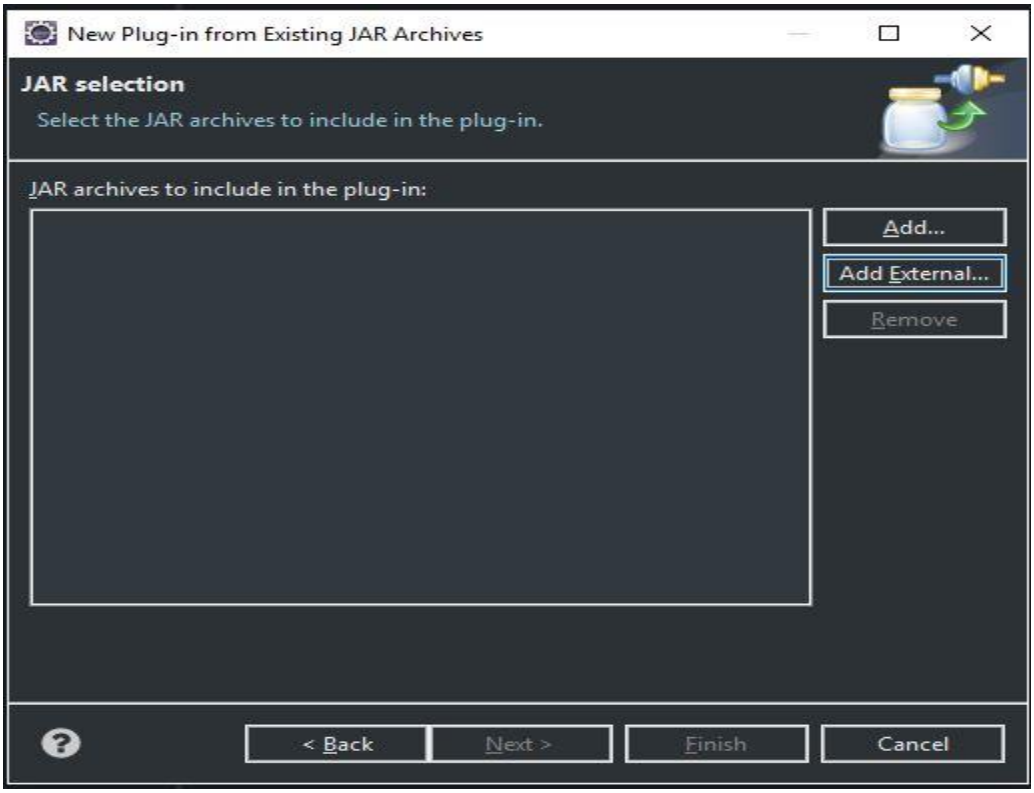
A window will appear.



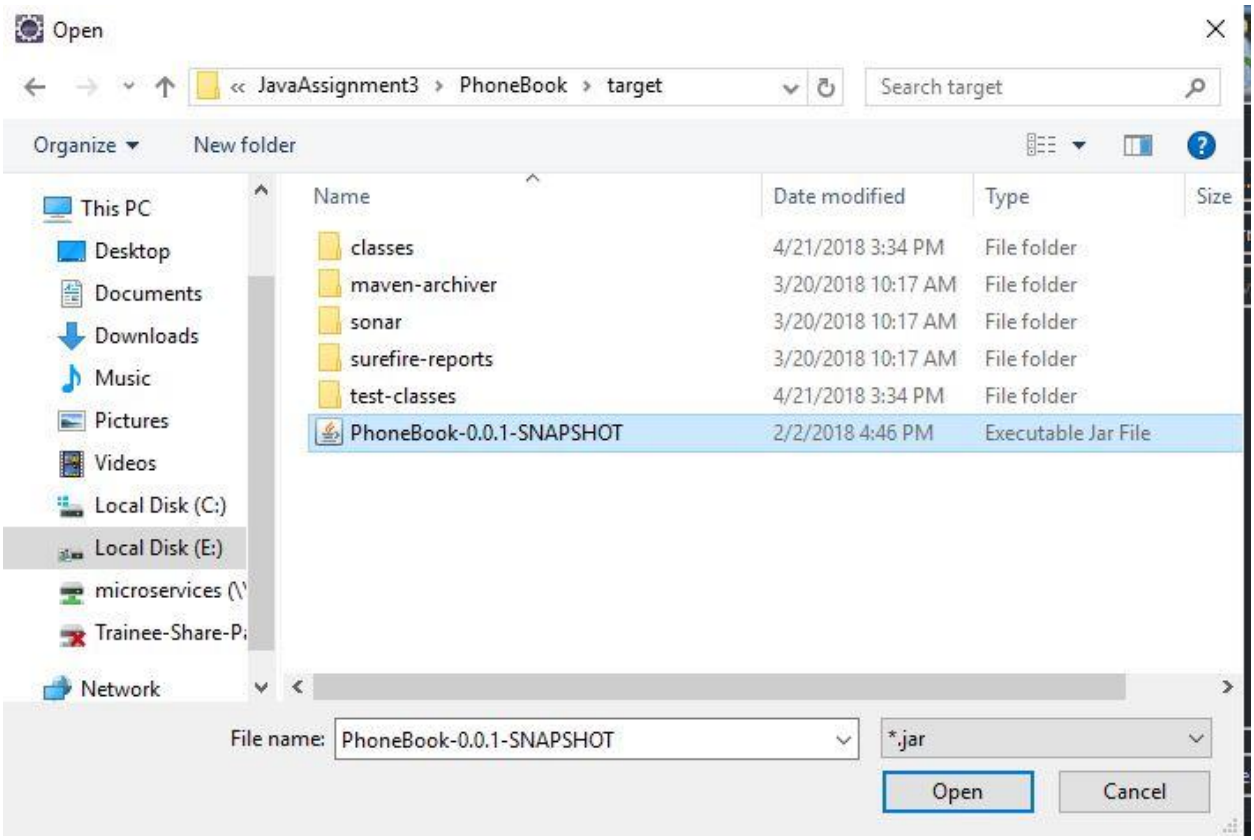

**Step 2:** Select '*Plug-in Development*' and expand it. Then select '*Plug-in from Existing JAR Archives'*. Click Next Button.




**Step 3:** In the JAR Selection Window, click on '*Add External*' Button.



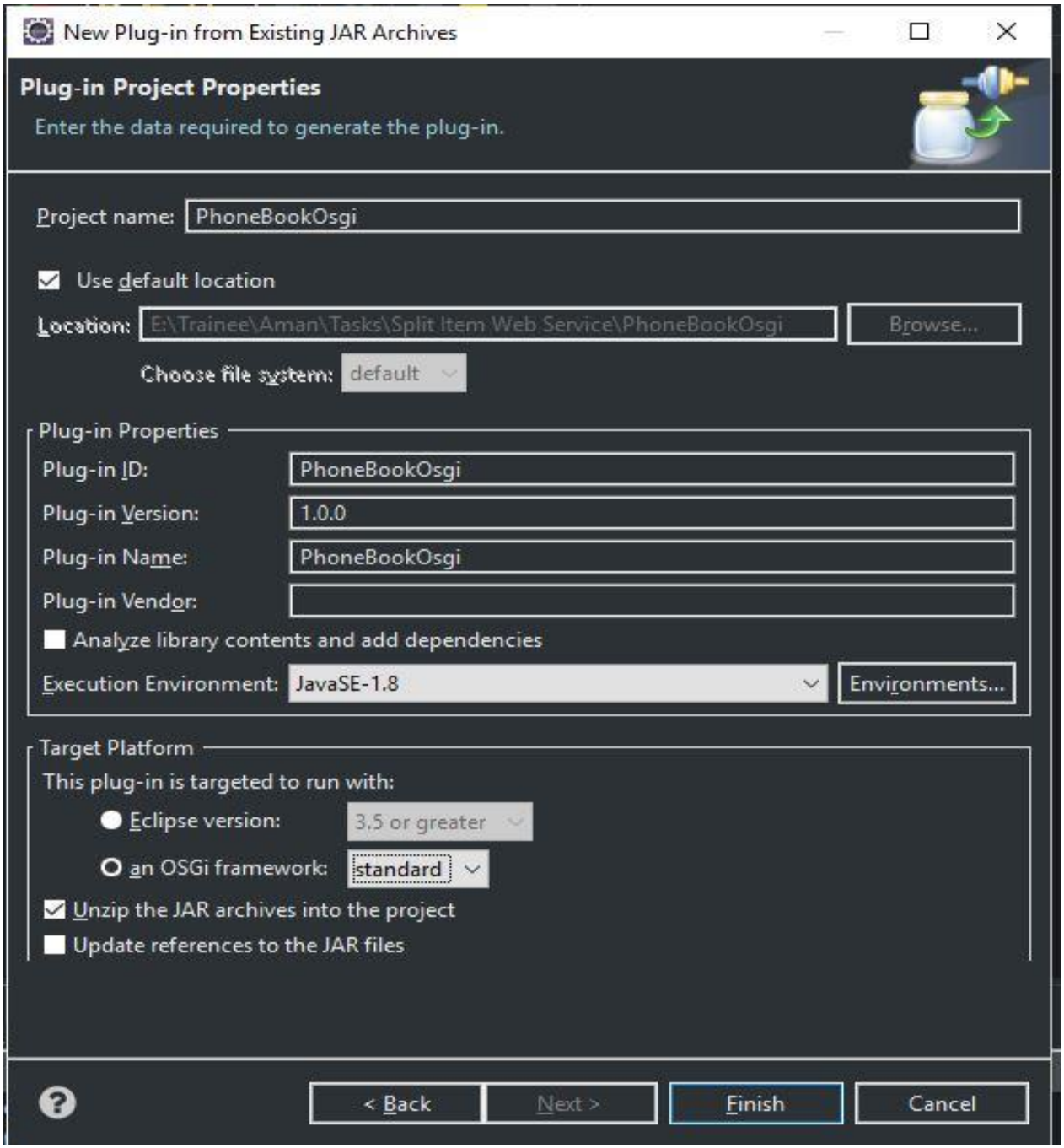
Select any existing non-OSGI JAR from file system, which you want to convert into OSGI JAR, and click on Open Button.




Click Next Button in JAR selection window.

**Step 4:** In the Plug-in Project Properties Window. Enter the required data.

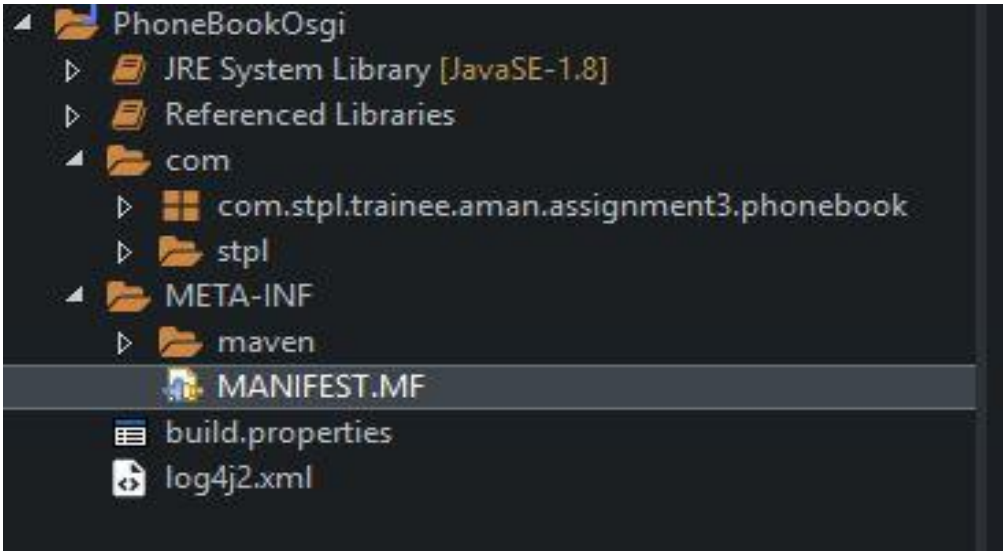
- Give a project name. (Mandatory)
- Select a location where you want this project to be created. Default location will be the workspace location. (Optional)
- Leave Plug-in Id, Plug-in Version, Plug-in Name as it is. If you wish, you can change. But leave the version field untouched.
- Select an execution environment. If we don't select this option, default workspace execution environment will be taken automatically. (Optional)
- Select a target platform from '*an OSGI Framework'* combo box. Select '*standard'.*
- Click Finish.




A new project will be created. The project will include:

- .class files from our existing JAR.
- Manifest.mf file, which is contains the bundle details.
- One build.properties file.
- And other resources of existing JAR file.

The Project Folder Structure would be like:



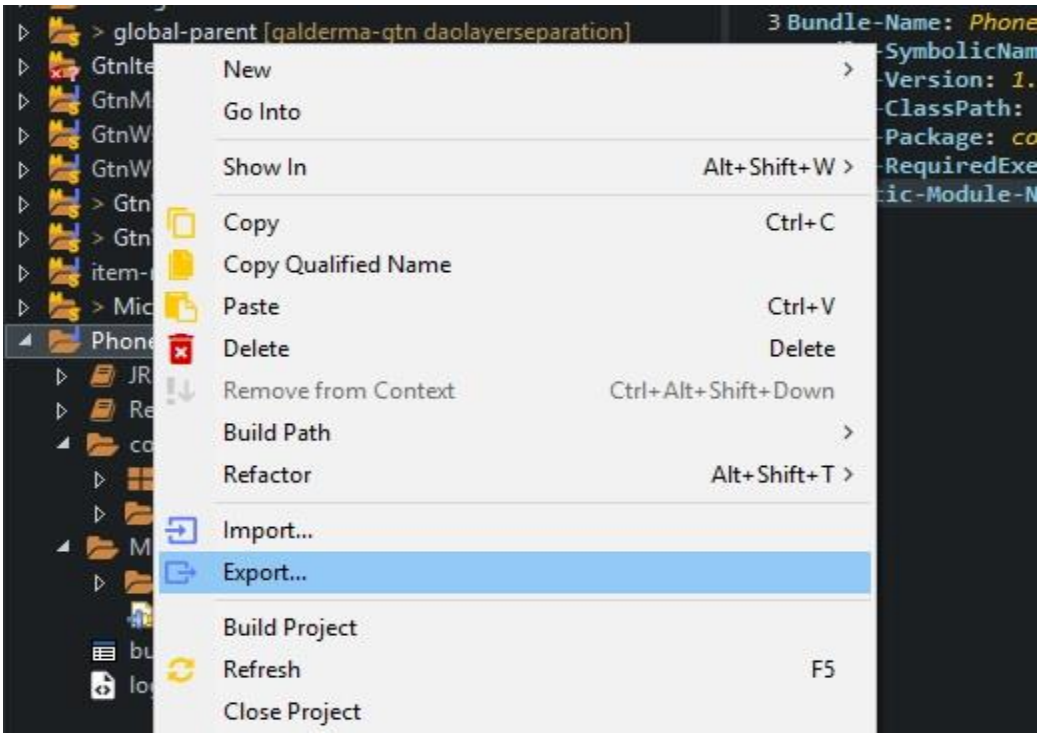

Let's see the manifest.mf file contents:


The file contains details related to OSGI bundle.

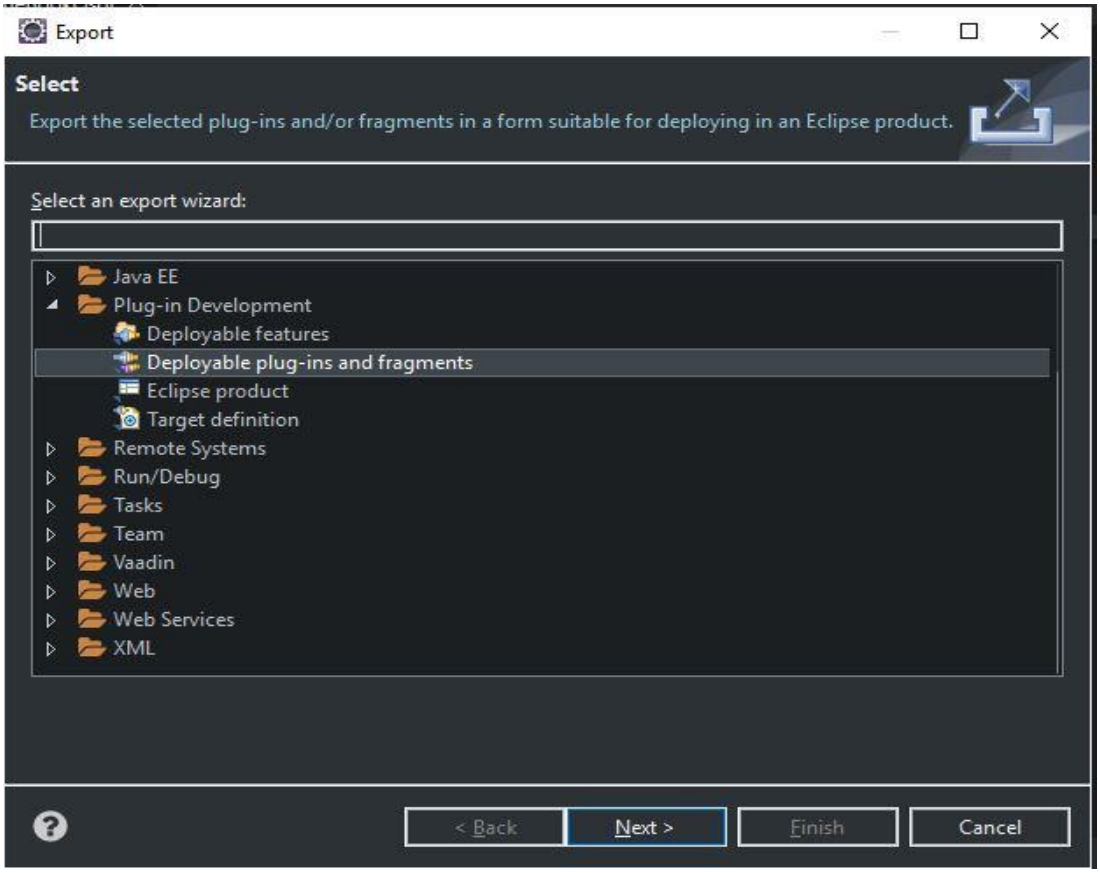
Now we are going to bundle the class files and this manifest.mf file into a single JAR,

Which will be an OSGI bundle.

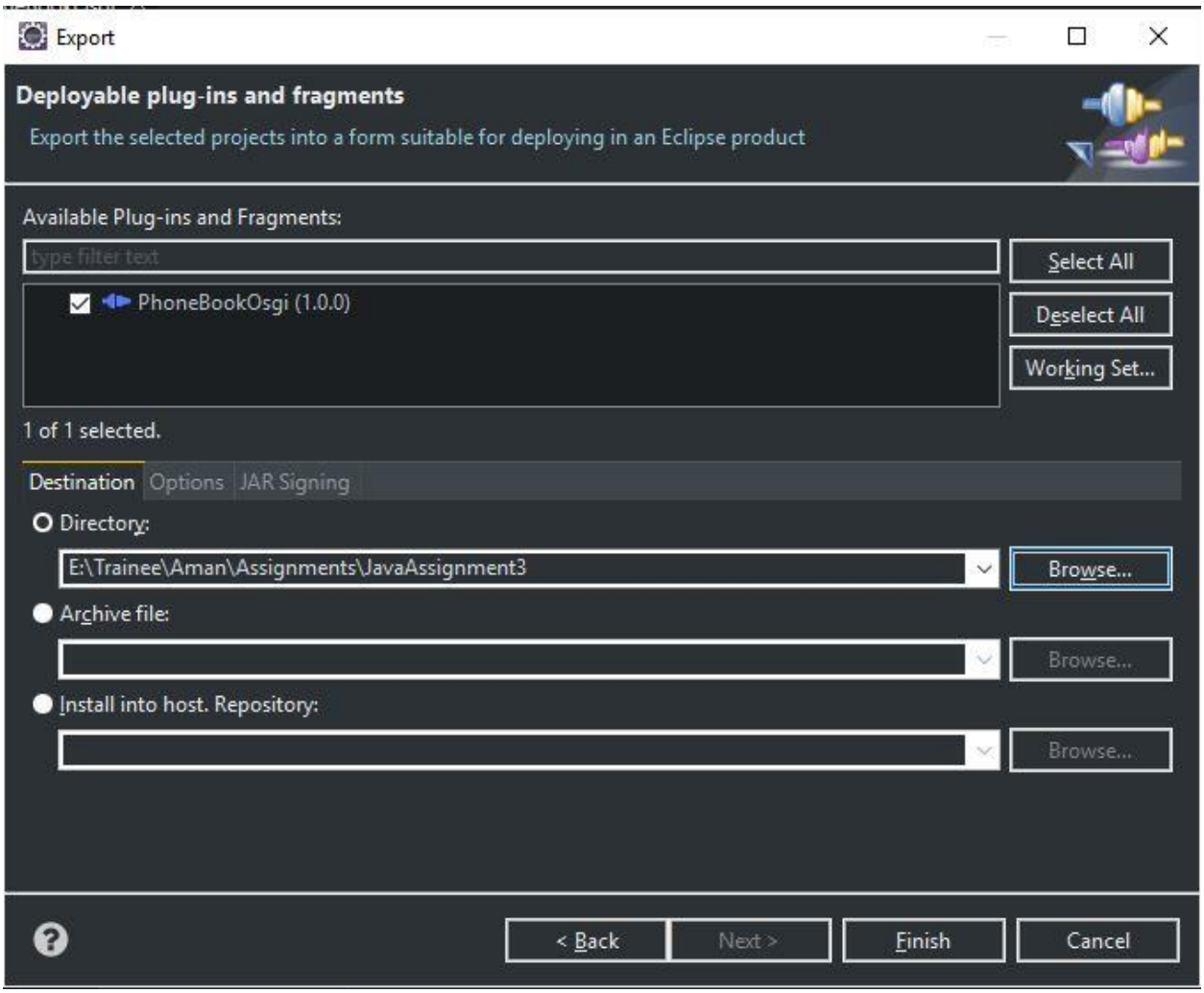
**Step 5:** Right click on project and select Export option.




**Step 6:** In export project window, again select '*Plug-in Development'* and expand. Then select '*Deployable Plug-ins and fragments'* and click on Next button.



**Step 7:** In deployable plug-ins and fragments window, select directory where you want the New JAR file (bundle) to be created. Eclipse will create a plugin directory under selected directory and the JAR file under plugin directory. Click on Finish Button.




Our bundle is created. Now we can deploy our bundle using steps that I described earlier.

**Method 2: Using Maven Project**

**Step 1:** Create a simple maven project.

**Step 2:** Open pom.xml. Add the dependency to the JAR which we are going to convert

Into OSGI bundle.

**Step 3:** Add maven-bundle-plugin in pom.

**Step 4:** Change the packaging from JAR/POM to bundle.

02-NonOsgiToOsgiBundle/pom.xml

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

    <modelVersion>4.0.0</modelVersion>
    <groupId>com.example.osgi</groupId>
    <artifactId>02-NonOsgiToOsgiBundle</artifactId>
    <version>1.0</version>
    <packaging>bundle</packaging>
    <name>02-NonOsgiToOsgiBundle</name>
    <description>02-NonOsgiToOsgiBundle</description>

    <properties>
        <maven.compiler.source>1.8</maven.compiler.source>
        <maven.compiler.target>1.8</maven.compiler.target>
    </properties>

    <dependencies>

        <dependency>
            <groupId>com.aman.assignment1</groupId>
            <artifactId>Log4jDemo</artifactId>
            <version>1.0-SNAPSHOT</version>
        </dependency>

    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.felix</groupId>
                <artifactId>maven-bundle-plugin</artifactId>
                <version>2.3.7</version>
                <extensions>true</extensions>
                <configuration>
                    <instructions>
                        <Export-Package>Specify Packages to be exported</Export-Package>
                    </instructions>
                </configuration>
            </plugin>
        </plugins>
    </build>

</project>
```

**Step 5:** Build the project using maven clean install goal. We'll get a JAR which is an OSGI compatible bundle.

**Method 3: Manual Method**

We can convert a non-OSGI JAR to OSGI bundle by manually creating manifest.mf and replacing the existing manifest.mf into the JAR. This approach is error prone and not recommended. But saves time.