Eclipse, Maven and Spring DM for OSGi™ platform

Quick Start

*Author: Oleg Zhurakousky (*oleg.zhurakousky@gmail.com*)*

February 21, 2008

Table of Contents

1. Introduction

This tutorial attempts to introduce the developers on how to setup development environment for working with Spring Dynamic Modules (DM) for OSGi™ platform project using currently available tools based on Eclipse IDE, Eclipse Equinox OSGi™ container and Maven plug-in. And even though it does cover some of the details of OSGi™ as well as Spring DM is not aimed to be the tutorial for either. It is also not aimed to be an Eclipse or Maven tutorial
For more information on OSGi™ and Spring DM please refer to the following websites

Spring DM - http://www.springframework.org/osgi
OSGi™ - http://www.osgi.org/Main/HomePage
Eclipse IDE – http://eclipse.org
Maven - http://maven.apache.org/
Eclipse Equinox - http://www.eclipse.org/equinox/

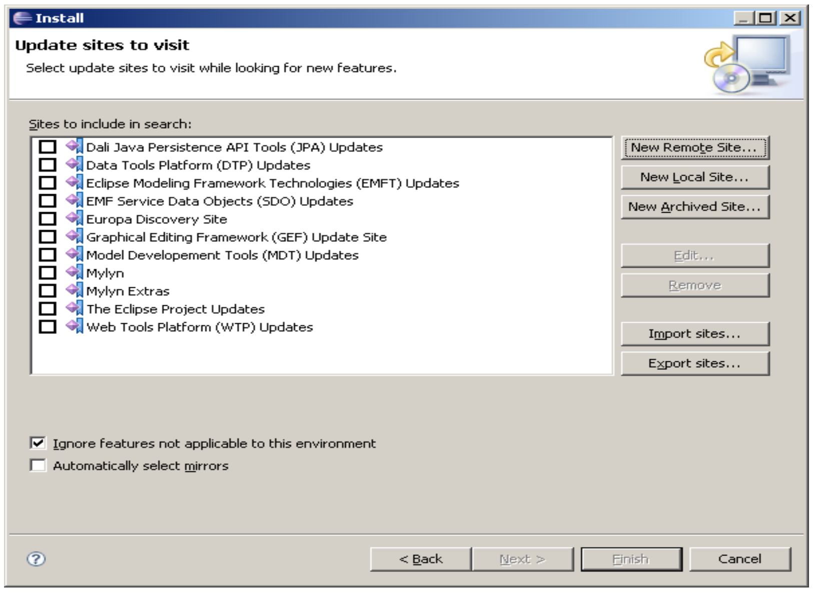
2. Prerequisites

This tutorial is based on Eclipse (Europa) 3.3.1 with Maven Plug-in installed.

You can download Eclipse here (if you don't already have it)

If you already have Maven plug-in installed you can skip to the next chapter otherwise let's install it.

Open up Eclipse. Click on Help (on the menu bar) → **Software Updates → Find and Install → Search for new features to install → Next**



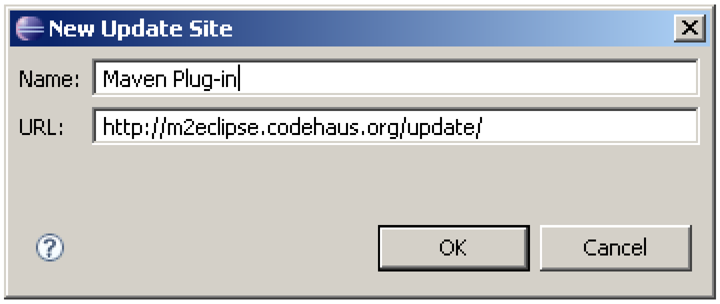
- [ ] Dali Java Persistence API Tools (JPA) Updates
- [ ] Data Tools Platform (DTP) Updates
- [ ] Eclipse Modeling Framework Technologies (EMFT) Updates
- [ ] EMF Service Data Objects (SDO) Updates
- [ ] Europa Discovery Site
- [ ] Graphical Editing Framework (GEF) Update Site
- [ ] Model Developement Tools (MDT) Updates
- [ ] Mylyn
- [ ] Mylyn Extras
- [ ] The Eclipse Project Updates
- [ ] Web Tools Platform (WTP) Updates

- [✓] Ignore features not applicable to this environment
- [ ] Automatically select mirrors

We are going to add a new Update site

Click on **New Remote Site**

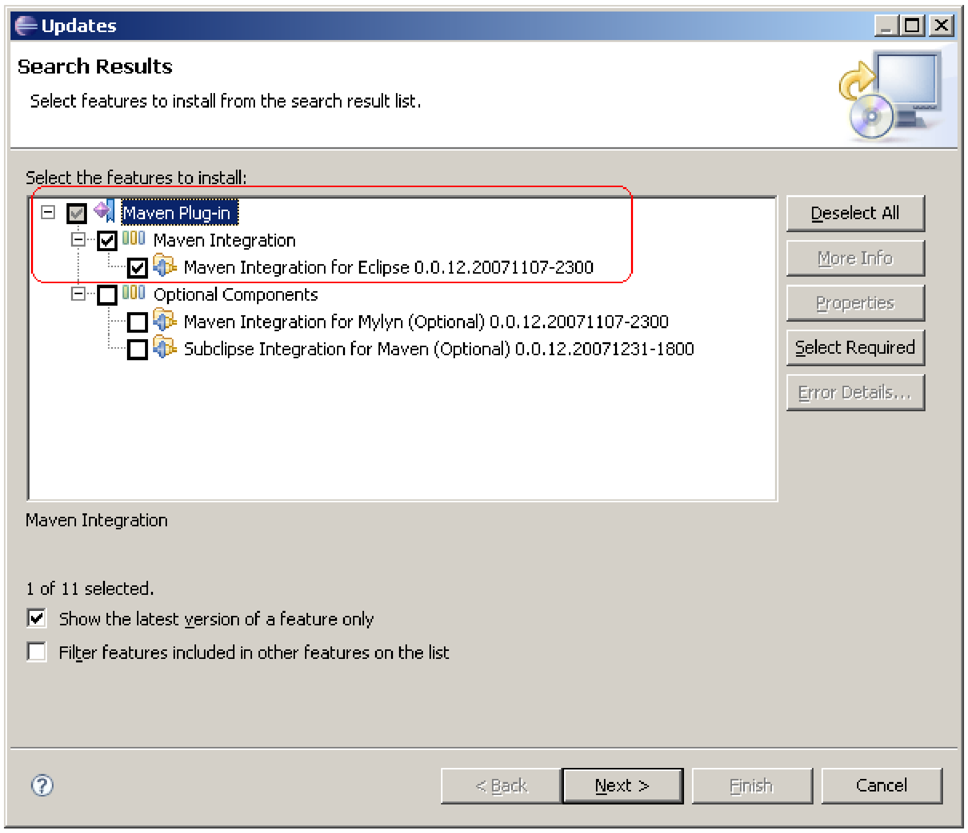
In the **Name** enter: Maven Plug-in
In the **URL** enter: http://m2eclipse.codehaus.org/update/



Name: Maven Plug-in
URL: http://m2eclipse.codehaus.org/update/

Click **OK → Finish**

Expend **Maven Plug-in** and select **Maven integration for Eclipse**



Search Results

Select features to install from the search result list.

Select the features to install:

- [✓] Maven Plug-in
  - [✓] Maven Integration
    - [✓] Maven Integration for Eclipse 0.0.12.20071107-2300
  - [ ] Optional Components
    - [ ] Maven Integration for Mylyn (Optional) 0.0.12.20071107-2300
    - [ ] Subclipse Integration for Maven (Optional) 0.0.12.20071231-1800

Maven Integration

1 of 11 selected.

[✓] Show the latest version of a feature only

[ ] Filter features included in other features on the list

Accept the license agreement and the rest of the defaults.

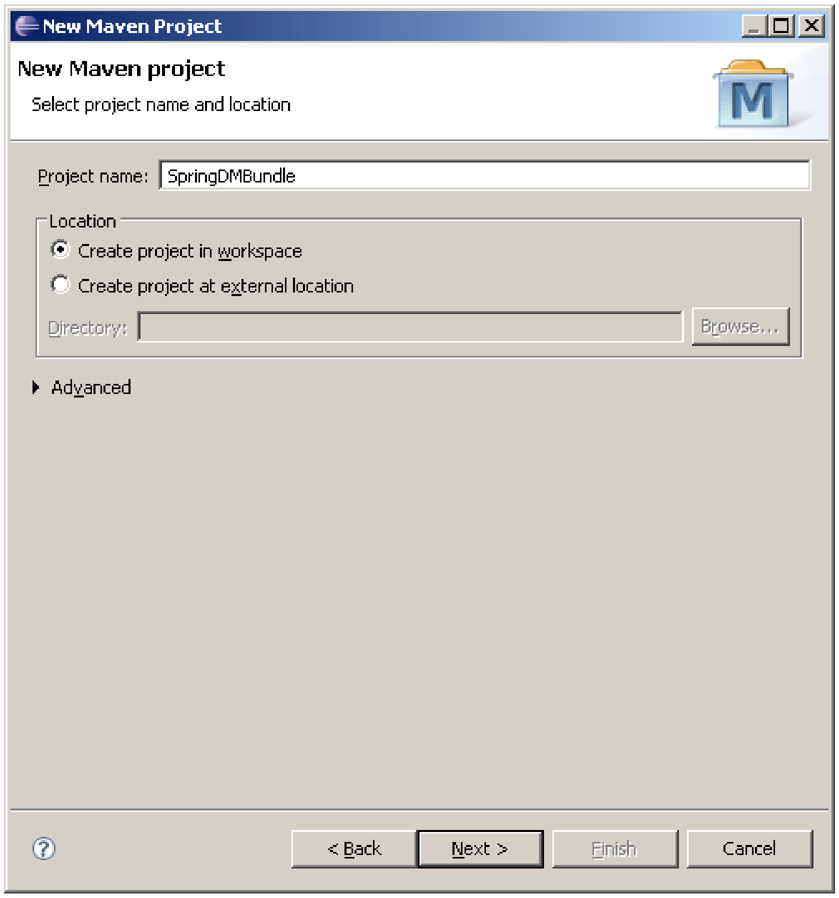
When wizard finishes downloading and installing plug-in it will prompt you to restart Eclipse. Do so.

Now you ready to continue.

3. Create and configure Spring DM project

In this chapter we are going to create a project for our Spring DM bundle.

Right-click on the white space in Project Explorer:
New → Other → Maven → Maven Project


New Maven Project

New Maven project

Select project name and location

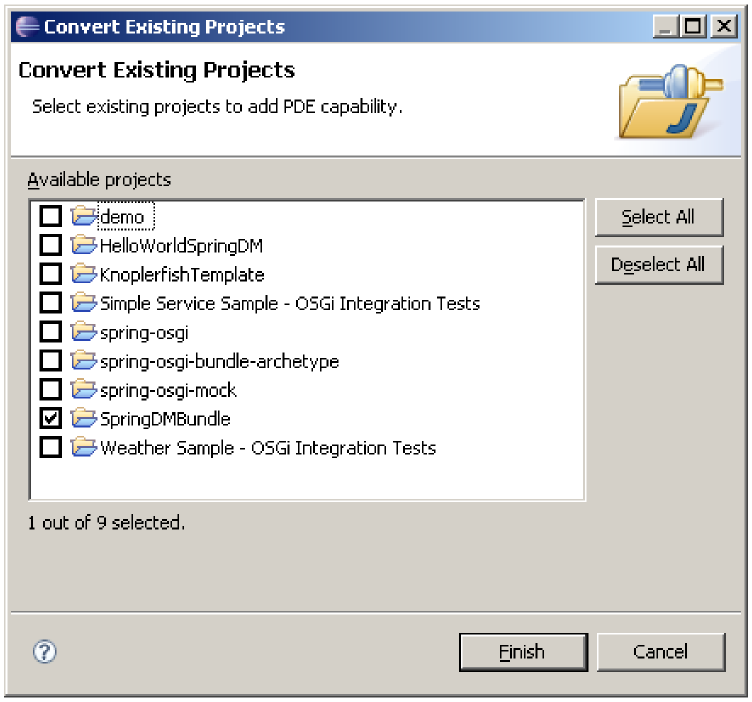
Project name: SpringDMBundle

Location
(●) Create project in workspace
( ) Create project at external location
Directory:

Let's call it ***SpringDMBundle***.

Click **Next → Finish**

Right-click on the newly created project → **PDE Tools → Convert Project to Plug-in Project.**
Make sure it is the only project selected


Convert Existing Projects
Convert Existing Projects
Select existing projects to add PDE capability.
Available projects
Select All
Deselect All
Finish
Cancel

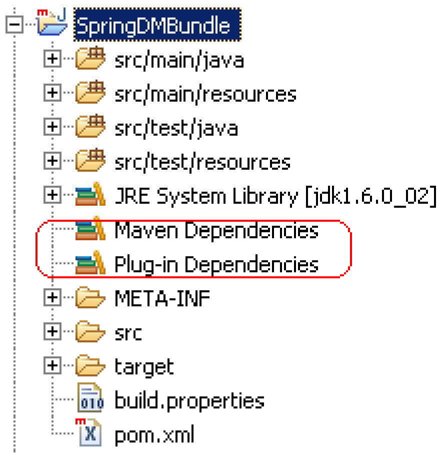
- [ ] demo
- [ ] HelloWorldSpringDM
- [ ] KnoplerfishTemplate
- [ ] Simple Service Sample - OSGi Integration Tests
- [ ] spring-osgi
- [ ] spring-osgi-bundle-archetype
- [ ] spring-osgi-mock
- [✓] SpringDMBundle
- [ ] Weather Sample - OSGi Integration Tests

1 out of 9 selected.

Click on **Finish**

This will configure project as Eclipse Plug-in project which will add **META-INF/MANIFEST.MF** to the root of the project amongst other things.

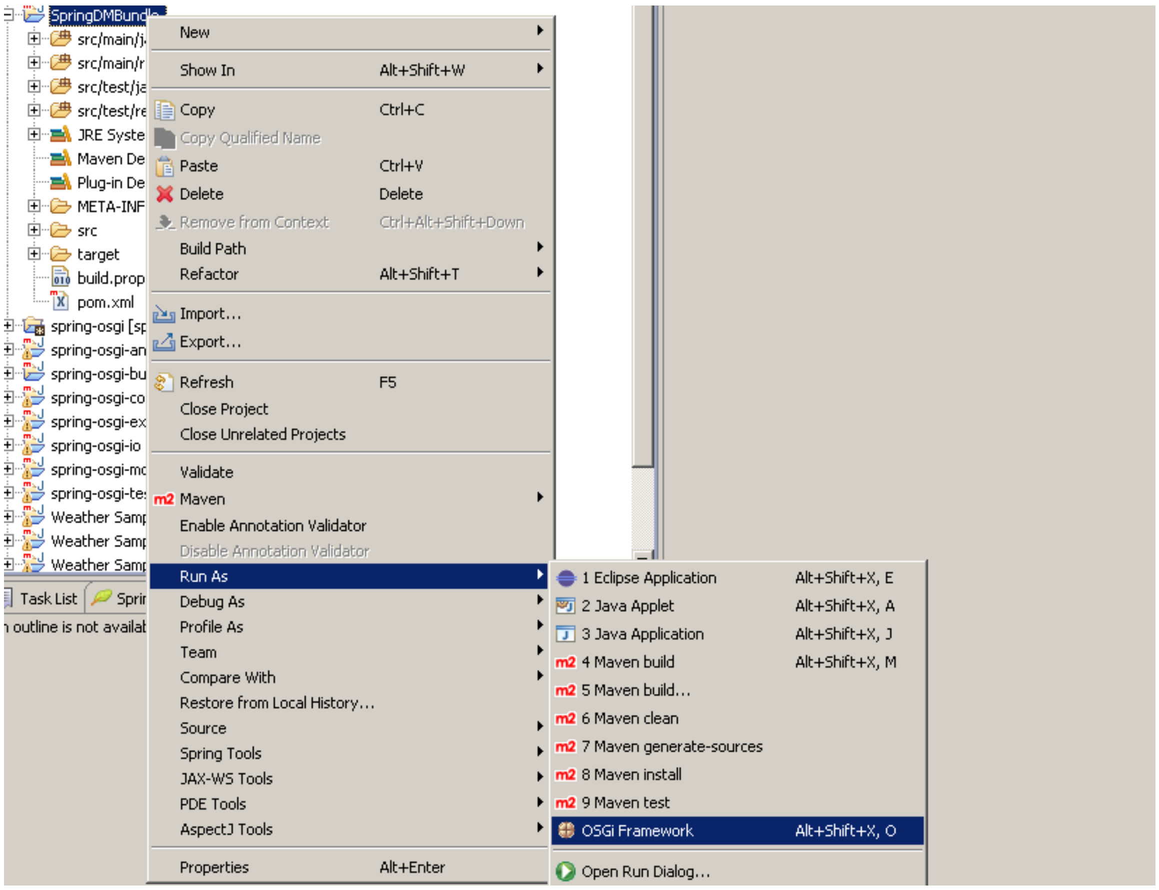
Open up Project Explorer view (**Window → Show View → Other → General → Project Explorer**)



```
SpringDMBundle
  src/main/java
  src/main/resources
  src/test/java
  src/test/resources
  JRE System Library [jdk1.6.0_02]
  Maven Dependencies
  Plug-in Dependencies
  META-INF
  src
  target
  build.properties
  pom.xml
```

You should see libraries for **Maven** and **Plug-in dependencies** there. We are only going to be using Maven for dependency management; however we've added Plug-in nature to the project simply because Eclipse PDE environment is an OSGi™ environment where

Eclipse plug-ins are OSGi™ bundles. This will allow us to use Eclipse provided OSGi ™ Equinox container as well as other Eclipse infrastructure elements be able to deploy and execute Spring DM projects by right-clicking on the project → **Run As** → **OSGi Framework**

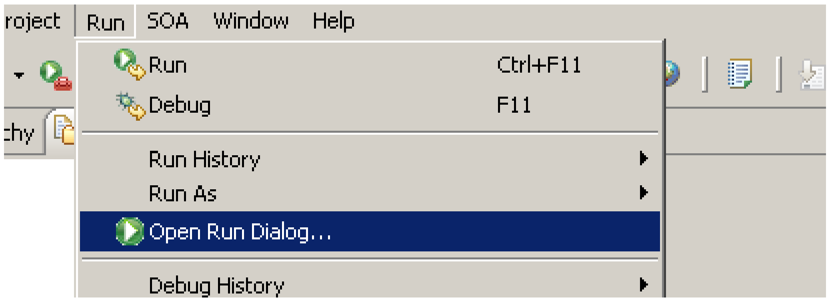


Don’t run anything yet!

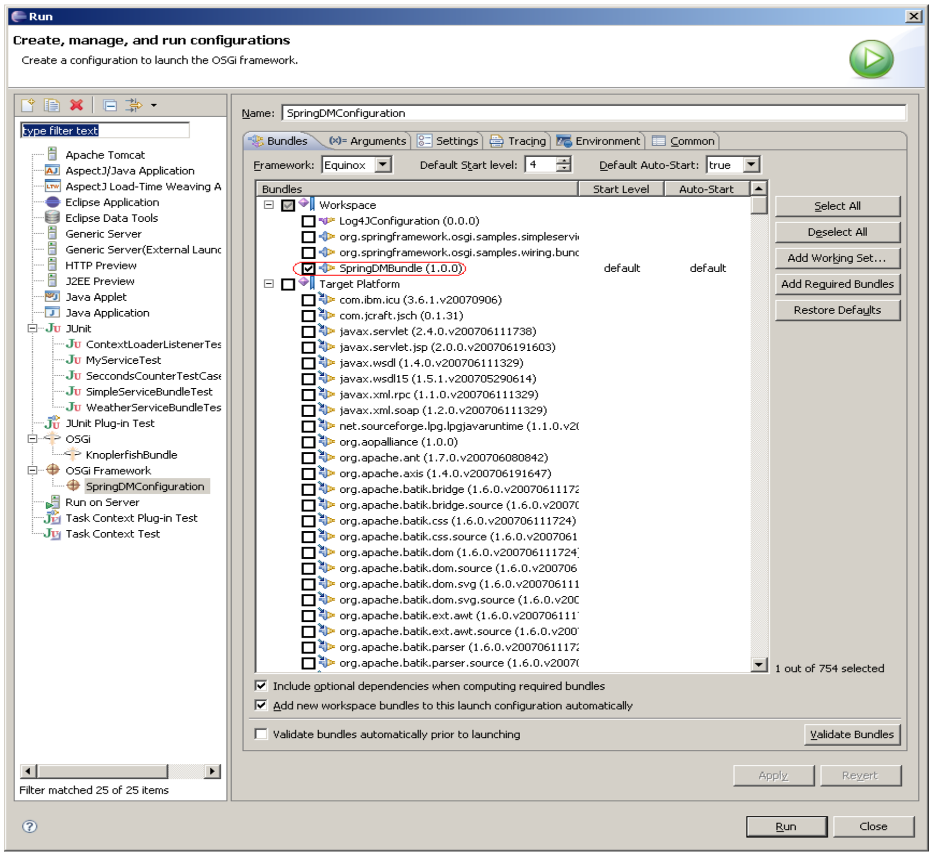
4. Test Configuration

In this chapter we will test our configuration

Let's Test it.
On the Menu bar click on **Run → Open Run Dialog**



Right Click on **OSGi Framework → New**
In the name enter: **SpringDMConfiguration**
Make sure **SpringDMBundle** is checked (you can un-check everything else)



Create, manage, and run configurations
Create a configuration to launch the OSGi framework.

type filter text

- Apache Tomcat
- AspectJ/Java Application
- AspectJ Load-Time Weaving A
- Eclipse Application
- Eclipse Data Tools
- Generic Server
- Generic Server(External Launc
- HTTP Preview
- J2EE Preview
- Java Applet
- Java Application
- JUnit
  - ContextLoaderListenerTes
  - MyServiceTest
  - SeccondsCounterTestCase
  - SimpleServiceBundleTest
  - WeatherServiceBundleTes
- JUnit Plug-in Test
- OSGi
  - KnoplerfishBundle
- OSGi Framework
  - SpringDMConfiguration
- Run on Server
- Task Context Plug-in Test
- Task Context Test

Filter matched 25 of 25 items

Name: SpringDMConfiguration

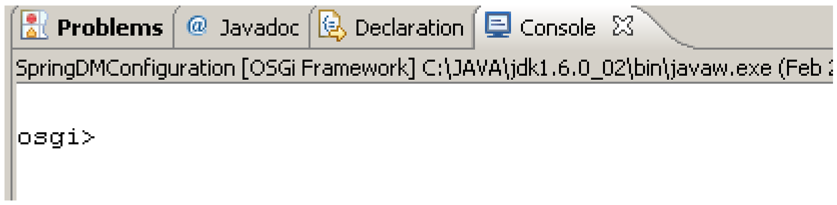
Framework: Equinox   Default Start level: 4   Default Auto-Start: true

Bundles	Start Level	Auto-Start
☑ Workspace		
☐ Log4JConfiguration (0.0.0)		
☐ org.springframework.osgi.samples.simpleservi		
☐ org.springframework.osgi.samples.wiring.bunc		
☑ SpringDMBundle (1.0.0)	default	default
☐ Target Platform		
☐ com.ibm.icu (3.6.1.v20070906)		
☐ com.jcraft.jsch (0.1.31)		
☐ javax.servlet (2.4.0.v200706111738)		
☐ javax.servlet.jsp (2.0.0.v200706191603)		
☐ javax.wsdl (1.4.0.v200706111329)		
☐ javax.wsdl15 (1.5.1.v200705290614)		
☐ javax.xml.rpc (1.1.0.v200706111329)		
☐ javax.xml.soap (1.2.0.v200706111329)		
☐ net.sourceforge.lpg.lpgjavaruntime (1.1.0.v2(		
☐ org.aopalliance (1.0.0)		
☐ org.apache.ant (1.7.0.v200706080842)		
☐ org.apache.axis (1.4.0.v200706191647)		
☐ org.apache.batik.bridge (1.6.0.v2007061117		
☐ org.apache.batik.bridge.source (1.6.0.v2007(		
☐ org.apache.batik.css (1.6.0.v200706111724)		
☐ org.apache.batik.css.source (1.6.0.v2007061		
☐ org.apache.batik.dom (1.6.0.v200706111724		
☐ org.apache.batik.dom.source (1.6.0.v200706		
☐ org.apache.batik.dom.svg (1.6.0.v200706111		
☐ org.apache.batik.dom.svg.source (1.6.0.v200		
☐ org.apache.batik.ext.awt (1.6.0.v200706111		
☐ org.apache.batik.ext.awt.source (1.6.0.v200		
☐ org.apache.batik.parser (1.6.0.v2007061117		
☐ org.apache.batik.parser.source (1.6.0.v2007(		

1 out of 754 selected

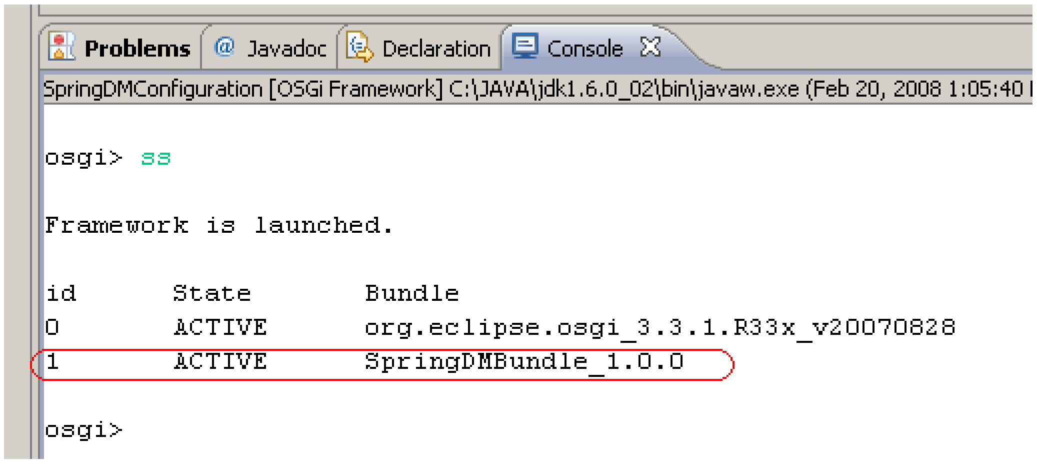
☑ Include optional dependencies when computing required bundles
☑ Add new workspace bundles to this launch configuration automatically
☐ Validate bundles automatically prior to launching

Click on **Apply → Run**
Eclipse Equinox OSGi™ container will start and the Console will show prompt like this:



```
SpringDMConfiguration [OSGi Framework] C:\JAVA\jdk1.6.0_02\bin\javaw.exe (Feb 2

osgi>
```

Enter **“ss”**
And you should see that your project is deployed and in the ACTIVE state.



```
SpringDMConfiguration [OSGi Framework] C:\JAVA\jdk1.6.0_02\bin\javaw.exe (Feb 20, 2008 1:05:40 I

osgi> ss

Framework is launched.

id      State       Bundle
0       ACTIVE      org.eclipse.osgi_3.3.1.R33x_v20070828
1       ACTIVE      SpringDMBundle_1.0.0

osgi>
```

Obviously this bundle is useless since we didn’t write any code, but it is a valid OSGi™ bundle even though it has no relation to Spring or Spring DM.

So let’s add some meat to it.

You can kill the OSGi™ container for now.

5. Set up project class-path dependencies (Maven)

In this chapter we will use Maven to setup class path dependencies for our project.

First let's modify our Maven POM file located at the root of the project.

Open **pom.xml** and paste the contents below into it (make sure to put it after **description** tag and before the closing **project** tag)

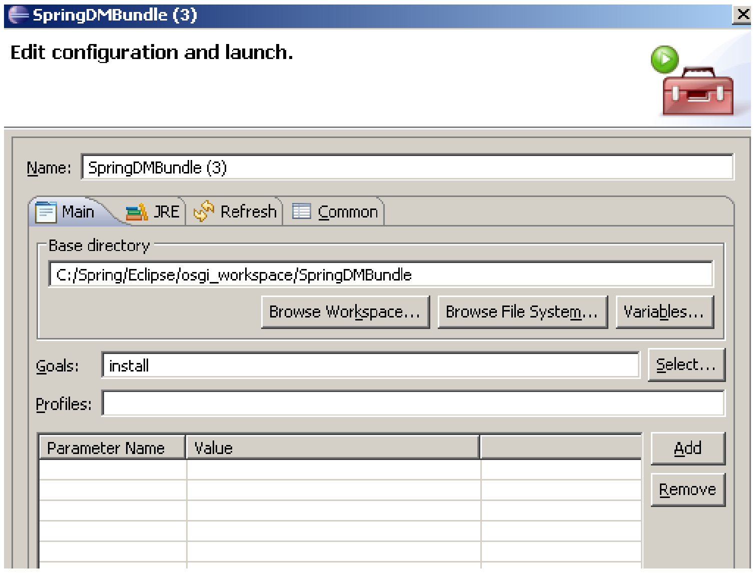
```
<dependencies>
        <dependency>
                <groupId>org.springframework</groupId>
                <artifactId>spring-test</artifactId>
                <version>2.5.1</version>
                <scope>test</scope>
        </dependency>
        <dependency>
                <groupId>org.springframework</groupId>
                <artifactId>spring-core</artifactId>
                <version>2.5.1</version>
                <scope>test</scope>
        </dependency>
        <dependency>
                <groupId>org.springframework</groupId>
                <artifactId>spring-context</artifactId>
                <version>2.5.1</version>
                <scope>test</scope>
        </dependency>
        <dependency>
                <groupId>org.springframework</groupId>
                <artifactId>spring-beans</artifactId>
                <version>2.5.1</version>
                <scope>test</scope>
        </dependency>
</dependencies>

<build>
        <plugins>
                <plugin>
                        <groupId>org.apache.maven.plugins</groupId>
                        <artifactId>maven-jar-plugin</artifactId>
                        <configuration>
                                <archive>
                                        <manifestFile>
                                                META-INF/MANIFEST.MF
                                        </manifestFile>
                                </archive>
                        </configuration>
                </plugin>
        </plugins>
</build>

<pluginRepositories>

        <pluginRepository>
                <id>maven-repo</id>
                <name>maven repo</name>
                <url>http://repo1.maven.org/maven2/</url>
        </pluginRepository>

        <pluginRepository>
                <id>agilejava</id>
                <url>http://agilejava.com/maven/</url>
```

```
		</pluginRepository>
	</pluginRepositories>


	<repositories>
		<repository>
			<id>eclipse-repository</id>
			<name>Eclipse Repository</name>
			<url>http://repo1.maven.org/eclipse/</url>
		</repository>

		<repository>
			<id>safehaus-repository</id>
			<name>Safehaus Repository</name>
			<url>http://m2.safehaus.org</url>
		</repository>

		<repository>
			<id>spring-ext</id>
			<name>Spring External Dependencies Repository</name>
			<url>

	http://springframework.svn.sourceforge.net/svnroot/springframework/repos/repo-ext/
			</url>
		</repository>


		<repository>
			<id>spring-release</id>
			<name>Spring Portfolio Release Repository</name>
			<url>
				http://s3.amazonaws.com/maven.springframework.org/release
			</url>
		</repository>
		<repository>
			<id>spring-external</id>
			<name>Spring Portfolio Release Repository</name>
			<url>
				http://s3.amazonaws.com/maven.springframework.org/external
			</url>
		</repository>
		<repository>
			<id>spring-milestone</id>
			<name>Spring Portfolio Milestone Repository</name>
			<url>
				http://s3.amazonaws.com/maven.springframework.org/milestone
			</url>
		</repository>

		<repository>
			<id>i21-s3-osgi-repo</id>
			<name>i21 osgi artifacts repo</name>
			<snapshots>
				<enabled>true</enabled>
			</snapshots>
			<url>
				http://s3.amazonaws.com/maven.springframework.org/osgi
			</url>
		</repository>

	</repositories>
```

Click **CTRL+Shift+F** to format the contents of the file.

This will define the required dependencies, repositories where to find those dependencies will be downloaded from and few packaging directions.

Save **pom.xml** file and build the project which for now will only install all the dependencies. To build it right-click on the project → **Run-as** → **Maven build**. Enter **install** as your Goal.


SpringDMBundle (3)

Edit configuration and launch.

Name: SpringDMBundle (3)

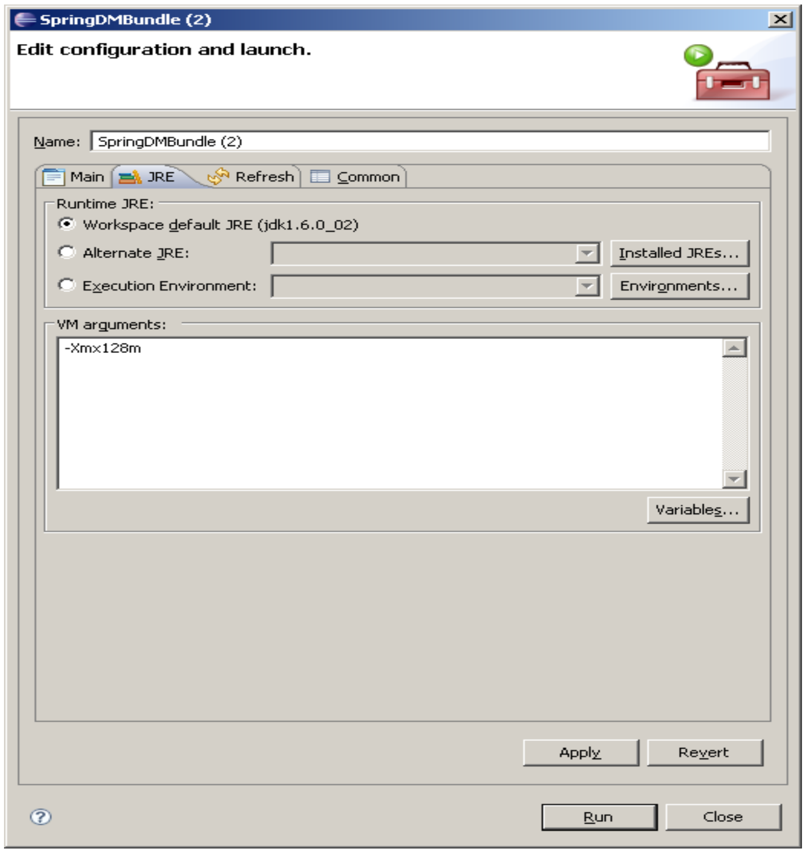
Base directory
C:/Spring/Eclipse/osgi_workspace/SpringDMBundle

Goals: install

Profiles:

Parameter Name	Value	

Click on JRE tab and enter **-Xmx128m** as VM argument. This is optional but on my machine it ran out of heap size so just to be safe.


SpringDMBundle (2)

Edit configuration and launch.

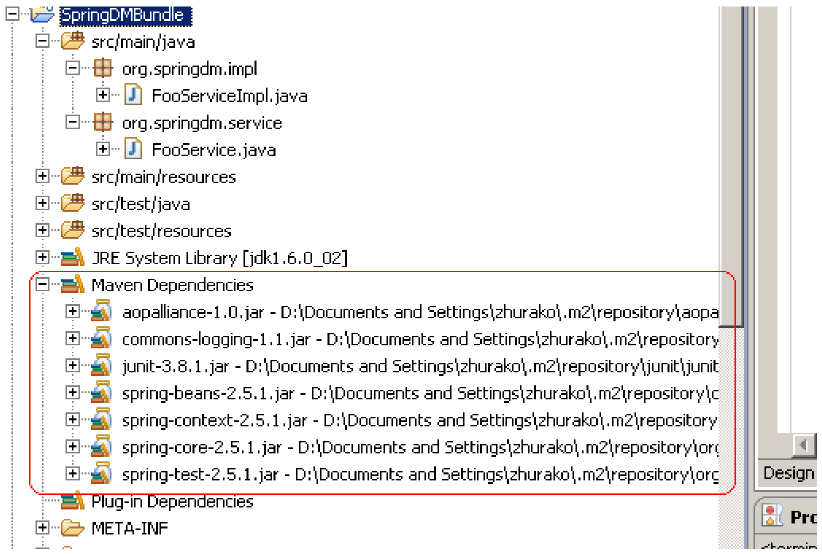
Name: SpringDMBundle (2)

Runtime JRE:
[✓ Workspace default JRE (jdk1.6.0_02)]
Alternate JRE:
Execution Environment:

VM arguments:
-Xmx128m

Click on **Run.**

This will take a bit of time and will depend on the state of your local Maven repository. After successful build your Maven libraries should all be in your class path.



```
SpringDMBundle
  src/main/java
    org.springdm.impl
      FooServiceImpl.java
    org.springdm.service
      FooService.java
  src/main/resources
  src/test/java
  src/test/resources
  JRE System Library [jdk1.6.0_02]
  Maven Dependencies
    aopalliance-1.0.jar - D:\Documents and Settings\zhurako\.m2\repository\aopa
    commons-logging-1.1.jar - D:\Documents and Settings\zhurako\.m2\repository
    junit-3.8.1.jar - D:\Documents and Settings\zhurako\.m2\repository\junit\junit
    spring-beans-2.5.1.jar - D:\Documents and Settings\zhurako\.m2\repository\c
    spring-context-2.5.1.jar - D:\Documents and Settings\zhurako\.m2\repository
    spring-core-2.5.1.jar - D:\Documents and Settings\zhurako\.m2\repository\org
    spring-test-2.5.1.jar - D:\Documents and Settings\zhurako\.m2\repository\org
  Plug-in Dependencies
  META-INF
```

6. Implement Spring DM service

Now we are going to implement a service which we will deploy later on inside of Eclipse Equinox OSGi™ environment.
I am going to assume that the knowledge on how to create classes and interfaces inside of the Eclipse IDE exist therefore I am going to skip the details.

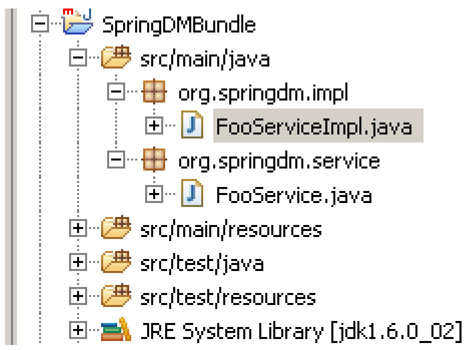
First create **FooService** interface

```
package org.springdm.service;

public interface FooService {
      public void foo();
}
```

Then create **FooServiceImpl** class

```
package org.springdm.impl;

import org.springdm.service.FooService;

public class FooServiceImpl implements FooService {

      public FooServiceImpl(){
            System.out.println(">>> Constructing
FooServiceImpl");
      }

      @Override
      public void foo() {
            System.out.println(">>> Executing foo()");
      }
}
```

You should see something like this



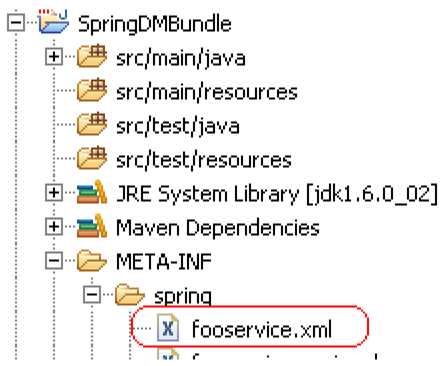
```
SpringDMBundle
  src/main/java
    org.springdm.impl
      FooServiceImpl.java
    org.springdm.service
      FooService.java
  src/main/resources
  src/test/java
  src/test/resources
  JRE System Library [jdk1.6.0_02]
```

Note that in the best traditions of Spring there is nothing Spring specific in these artifacts just a POJO and POJI. And as far as OSGi™ concern that is actually enough with the exception of OSGi™ Activator class that would bootstrap our service. Activator is responsible for lifecycle and registration of the bundle which means Activator usually contains nothing more then a boilerplate code that wires your bundle to OSGi™ container.

Writing Activator is outside of scope of this chapter or tutorial, even though when we get to the Test Client later on we will write a simple Activator for testing purposes. And for those who are interested in more details about Activator, there are plenty examples on how to do it available on the web.

What we want to do is a little more then just deploy an OSGi™ bundle. We want to deploy Spring powered OSGi™ bundle which means we are going to deploy Spring Application Context which need to be recognized somehow by OSGi™ container. We also want to register our POJO service as an OSGi™ service to make it available to other bundles within OSGi™ container.

Spring DM provides convinient infrastructure to do all that.
First we are going to create Application Context definition file **fooservice.xml** and place it in **spring** directory under META-INF

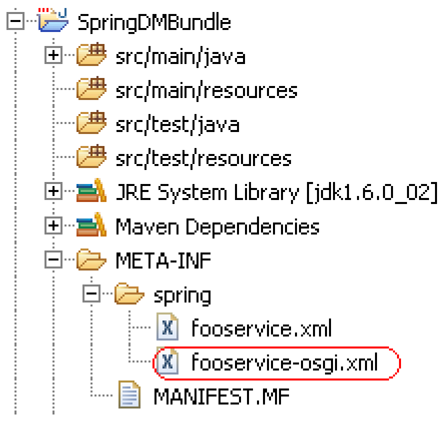


Here are the contents of the file

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">


  <bean name="fooService" class="org.springdm.impl.FooServiceImpl" />

</beans>
```

Nothing special, just a regular Spring bean definition.

We are going to create another Application Context definition file called **fooservice-osgi.xml** and place it in the same directory



And here are the contents of this file:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:osgi="http://www.springframework.org/schema/osgi"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
                      http://www.springframework.org/schema/osgi
http://www.springframework.org/schema/osgi/spring-osgi.xsd">


  <osgi:service id="fooServiceOsgi" ref="fooService"
    interface="org.springdm.service.FooService" />

</beans>
```

Note that we are using a **osgi** name space which will allow Spring DM (OSGIServiceFactoryBean in particular) to use Spring exporter infrastructure to export and register our service as OSGi™ service within OSGi™ container so it could be used by other bundles that might not be Spring enabled.
We could have place both definitions into a single file, but separating them makes testing (we'll get to testing later) of POJO service functionality simpler since it doesn't require OSGi™ specific dependencies and or Spring DM testing infrastructure.

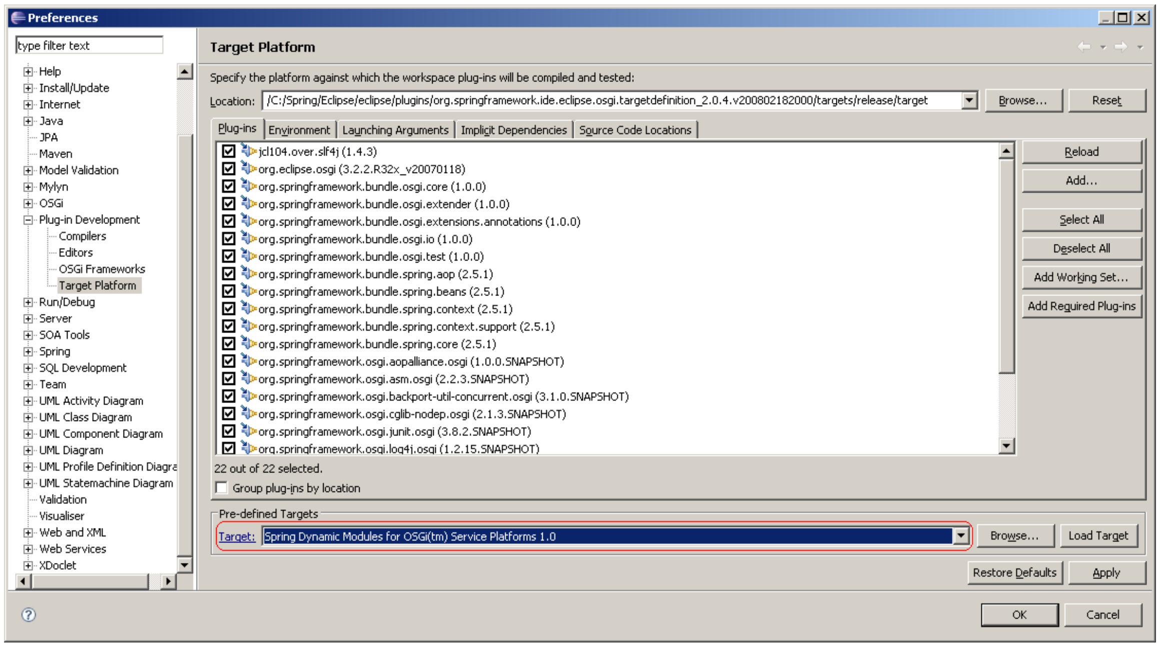
More details on the deployment infrastructure of Spring DM bundles are provided in Spring DM reference documentation (Chapter 4), but in the nutshell here is what is going to happen when we package our bundle:
Once our Spring-powered OSGi™ bundle is deployed into OSGi™ container and brought up to the ACTIVE state, Spring DM Extender Bundle will recognize it and in the similar fashion as ContextLoaderListener loads Web Application Context will load Application Context defined in our bundle.

Then OSGi™ service defined in our **fooservice-osgi.xml** file will be registered within OSGi™ container by OSGIServiceFactoryBean as an OSGi™ service identified by the **FooService.java** interface which is how it is going to be known to the rest of the OSGi™ bundle. The actual implementation becomes internal to the Application Context. You can read more on Exporting Spring Bean as OSGi™ service in Chapter 5 of reference documentation.

Now all we need is integrate Spring DM bundles with Eclipse provided Equinox OSGi™ environment. In Eclipse terms we basically have to setup a Target Platform.

Appendix C of Spring DM reference documentation has a very comprehensive set of instructions on how to do that so I won't be repeating it, so simply **follow the directions there and once done come back**

Click on **Window (menu bar) → Preferences → Plug-in Development → Target Platform**
Select *Spring Dynamic Modules for OSGi™ Service Platform 1.0* as you target and click on **Load Target → OK**



Specify the platform against which the workspace plug-ins will be compiled and tested:

Location: /C:/Spring/Eclipse/eclipse/plugins/org.springframework.ide.eclipse.osgi.targetdefinition_2.0.4.v200802182000/targets/release/target

- ☑ jcl104.over.slf4j (1.4.3)
- ☑ org.eclipse.osgi (3.2.2.R32x_v20070118)
- ☑ org.springframework.bundle.osgi.core (1.0.0)
- ☑ org.springframework.bundle.osgi.extender (1.0.0)
- ☑ org.springframework.bundle.osgi.extensions.annotations (1.0.0)
- ☑ org.springframework.bundle.osgi.io (1.0.0)
- ☑ org.springframework.bundle.osgi.test (1.0.0)
- ☑ org.springframework.bundle.spring.aop (2.5.1)
- ☑ org.springframework.bundle.spring.beans (2.5.1)
- ☑ org.springframework.bundle.spring.context (2.5.1)
- ☑ org.springframework.bundle.spring.context.support (2.5.1)
- ☑ org.springframework.bundle.spring.core (2.5.1)
- ☑ org.springframework.osgi.aopalliance.osgi (1.0.0.SNAPSHOT)
- ☑ org.springframework.osgi.asm.osgi (2.2.3.SNAPSHOT)
- ☑ org.springframework.osgi.backport-util-concurrent.osgi (3.1.0.SNAPSHOT)
- ☑ org.springframework.osgi.cglib-nodep.osgi (2.1.3.SNAPSHOT)
- ☑ org.springframework.osgi.junit.osgi (3.8.2.SNAPSHOT)
- ☑ org.springframework.osgi.log4j.osgi (1.2.15.SNAPSHOT)

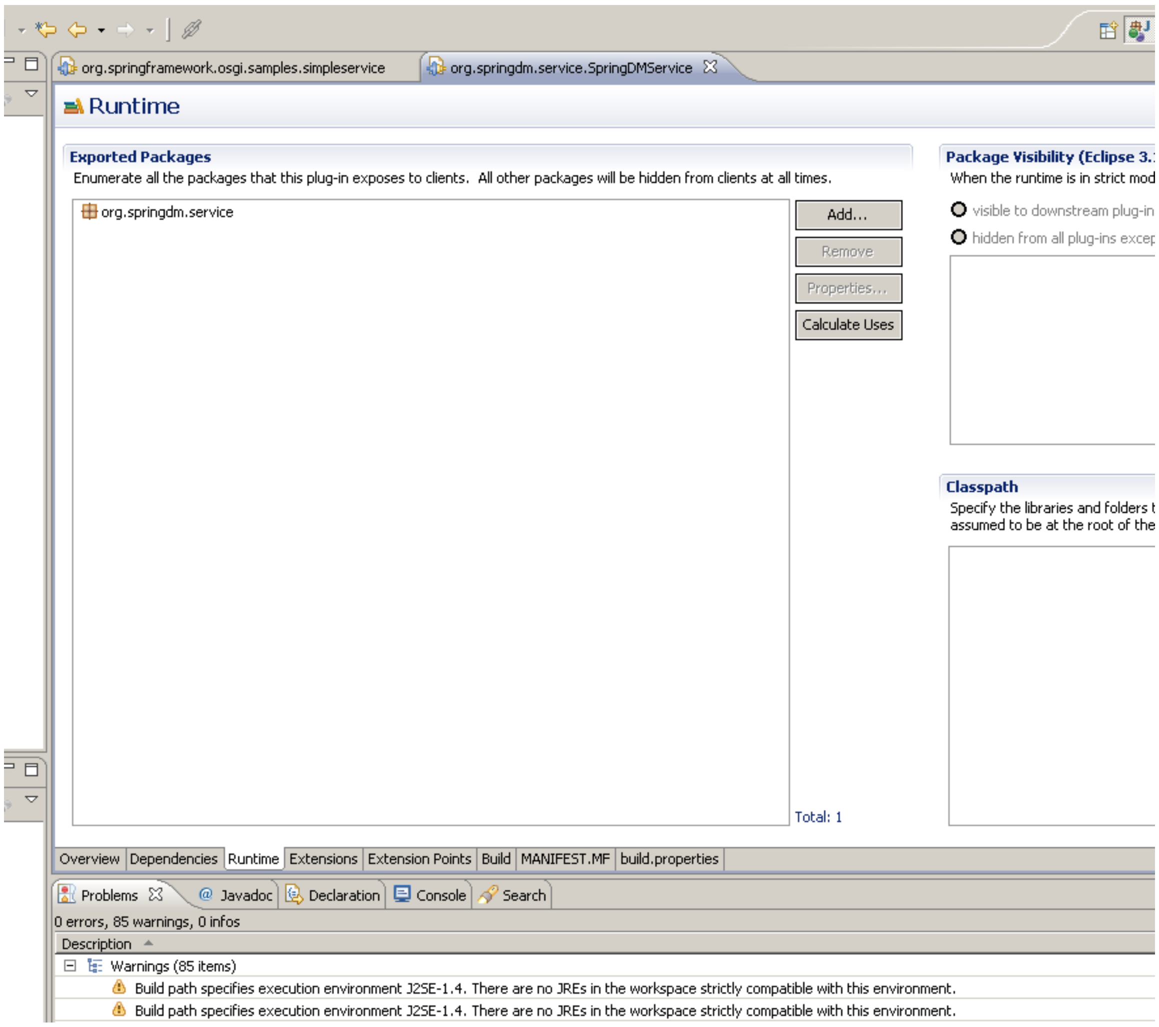
22 out of 22 selected.

Target: Spring Dynamic Modules for OSGi(tm) Service Platforms 1.0

One of the best parts of OSGi™ is the fact that it takes us back to the basics, where Java programs are packaged into JAR files and MANIFEST.MF actually means something. So let's update our MANIFEST.MF file and test our Spring-powered bundle.
When you double-click on the MANIFEST.MF file it will be opened by **Plug-in Manifest Editor** which contains multiple tabs and property windows that will allow you to quickly assemble and/or modify your MANIFEST.MF file.

Below is view of the **Runtime** tab of our Manifest file.

As I mentioned before one of our goals is to export our service into the OSGi™ container so other bundles can use it and as you can see in **Exported Packages** I did just that by clicking on **Add** button



org.springframework.osgi.samples.simpleservice | org.springdm.service.SpringDMService

Runtime

Exported Packages
Enumerate all the packages that this plug-in exposes to clients. All other packages will be hidden from clients at all times.

org.springdm.service

Add... | Remove | Properties... | Calculate Uses

Total: 1

Package Visibility (Eclipse 3.
When the runtime is in strict mod
visible to downstream plug-in
hidden from all plug-ins excep

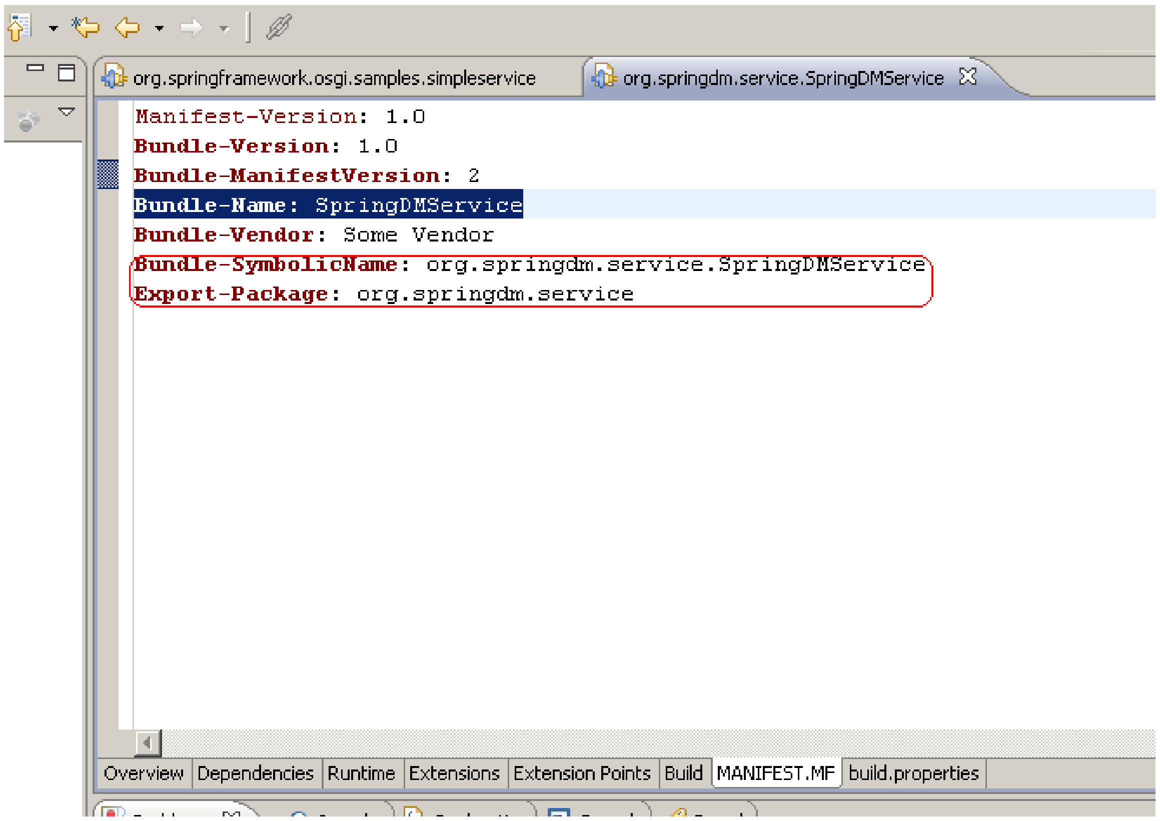
Classpath
Specify the libraries and folders t
assumed to be at the root of the

Overview | Dependencies | Runtime | Extensions | Extension Points | Build | MANIFEST.MF | build.properties

Problems | Javadoc | Declaration | Console | Search

```
0 errors, 85 warnings, 0 infos
Description
Warnings (85 items)
  Build path specifies execution environment J2SE-1.4. There are no JREs in the workspace strictly compatible with this environment.
  Build path specifies execution environment J2SE-1.4. There are no JREs in the workspace strictly compatible with this environment.
```

Also for sanity check click on MANIFES.MF tab to see plain text representation of Manifest. There you can see our exported service.



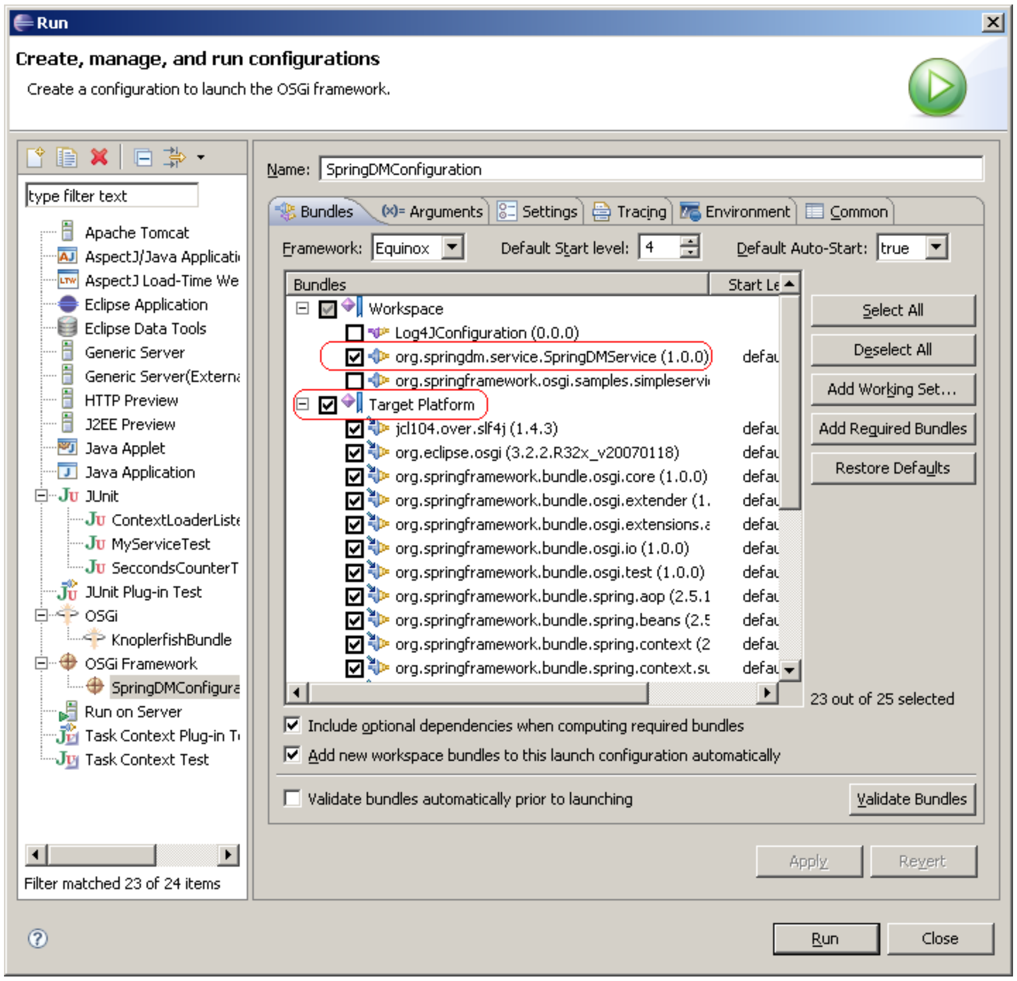
```
Manifest-Version: 1.0
Bundle-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: SpringDMService
Bundle-Vendor: Some Vendor
Bundle-SymbolicName: org.springdm.service.SpringDMService
Export-Package: org.springdm.service
```

For now our Manifest is extremely simple and trivial. Other then **Export-Package** one other important property to point out is **Bundle Symbolic-Name (a.k.a Bundle ID).** It contains the name by which you would reference this bundle from other bundles when need to.

7. Deploy Spring DM service

The fact that your project has Eclipse PDE nature is recognized by PDE runtime environment and that is why you see in under Workspace bundles. This means that while developing there is nothing needs to be done to deploy the service. So let's test it by simply starting OSGi™ Framework runtime environment.

On the Manu bar click on **Run → Open Run Dialog → OSGi(tm) Framework → SpringDMConfiguration**. You should see a familiar window with the exception of Target Platform now being different since we picked Spring DM Target platform earlier.


Run

Create, manage, and run configurations

Create a configuration to launch the OSGi framework.

type filter text

- Apache Tomcat
- AspectJ/Java Applicati
- AspectJ Load-Time We
- Eclipse Application
- Eclipse Data Tools
- Generic Server
- Generic Server(Externa
- HTTP Preview
- J2EE Preview
- Java Applet
- Java Application
- JUnit
  - ContextLoaderListe
  - MyServiceTest
  - SeccondsCounterT
- JUnit Plug-in Test
- OSGi
  - KnoplerfishBundle
- OSGi Framework
  - SpringDMConfigura
- Run on Server
- Task Context Plug-in Te
- Task Context Test

Filter matched 23 of 24 items

Name: SpringDMConfiguration

Bundles | Arguments | Settings | Tracing | Environment | Common

Framework: Equinox   Default Start level: 4   Default Auto-Start: true

Bundles	Start Le
[✓] Workspace	
[ ] Log4JConfiguration (0.0.0)	
[✓] org.springdm.service.SpringDMService (1.0.0)	defau
[ ] org.springframework.osgi.samples.simpleservi	
[✓] Target Platform	
[✓] jcl104.over.slf4j (1.4.3)	defau
[✓] org.eclipse.osgi (3.2.2.R32x_v20070118)	defau
[✓] org.springframework.bundle.osgi.core (1.0.0)	defau
[✓] org.springframework.bundle.osgi.extender (1.	defau
[✓] org.springframework.bundle.osgi.extensions.a	defau
[✓] org.springframework.bundle.osgi.io (1.0.0)	defau
[✓] org.springframework.bundle.osgi.test (1.0.0)	defau
[✓] org.springframework.bundle.spring.aop (2.5.1	defau
[✓] org.springframework.bundle.spring.beans (2.5	defau
[✓] org.springframework.bundle.spring.context (2	defau
[✓] org.springframework.bundle.spring.context.su	defau

Select All | Deselect All | Add Working Set... | Add Required Bundles | Restore Defaults

23 out of 25 selected

[✓] Include optional dependencies when computing required bundles

[✓] Add new workspace bundles to this launch configuration automatically

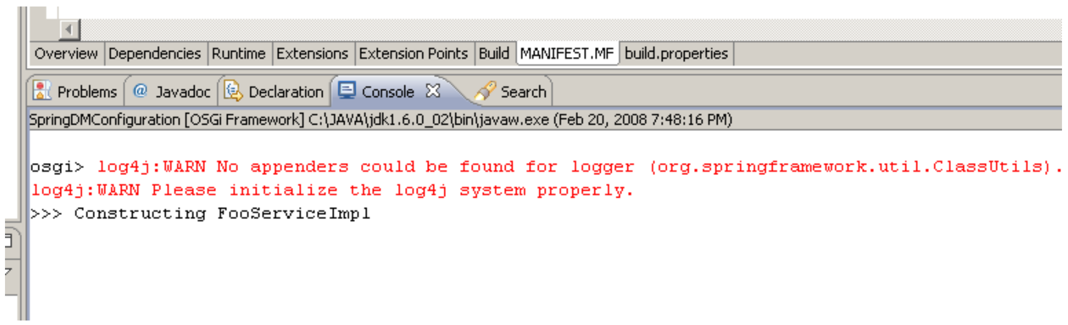
[ ] Validate bundles automatically prior to launching   Validate Bundles

Apply | Revert

Run | Close

Select **SpringDMService** and all of the bundles of **Target Platform**. We obviously don't need all of them but for the sake of saving time we'll just do a Select All.

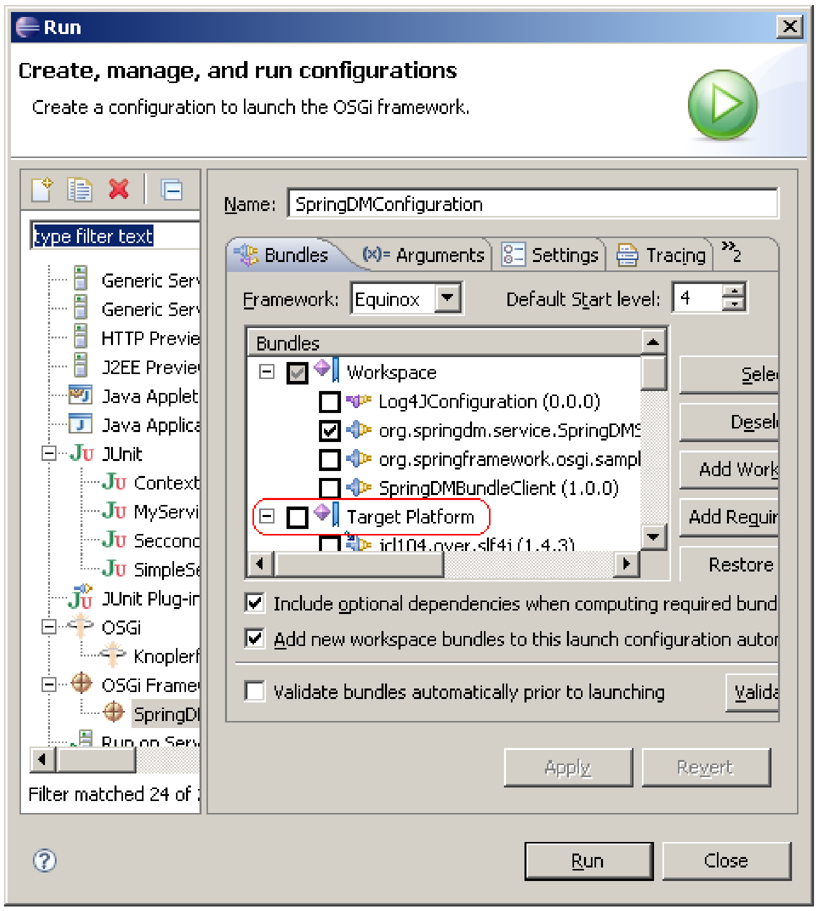
Click **Run**. Your console should look like this



Overview | Dependencies | Runtime | Extensions | Extension Points | Build | MANIFEST.MF | build.properties
Problems | Javadoc | Declaration | Console | Search

```
SpringDMConfiguration [OSGi Framework] C:\JAVA\jdk1.6.0_02\bin\javaw.exe (Feb 20, 2008 7:48:16 PM)

osgi> log4j:WARN No appenders could be found for logger (org.springframework.util.ClassUtils).
log4j:WARN Please initialize the log4j system properly.
>>> Constructing FooServiceImpl
```

This is it! You have deployed your first Spring DM OSGi™ service and the console shows that the service has been constructed. You can type **"ss"** in the console and you'll see your service in the ACTIVE state.
But how do we know that our service was registered and other bundles can see and use it. In other words how do we know that Spring DM did its job?
Well, first of all if we open **Run Dialog** and uncheck **Target Platform** and run the service again



Create, manage, and run configurations

Create a configuration to launch the OSGi framework.

type filter text

Generic Serv
Generic Serv
HTTP Previe
J2EE Previe
Java Applet
Java Applica
JUnit
Context
MyServi
Secconc
SimpleSe
JUnit Plug-ir
OSGi
Knoplerf
OSGi Frame
SpringDI
Run on Serv

Filter matched 24 of

Name: SpringDMConfiguration

Framework: Equinox Default Start level: 4

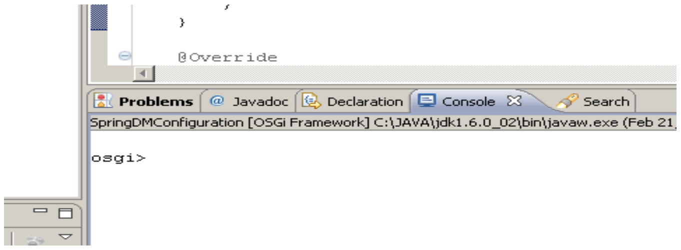
Bundles
- [✓] Workspace
  - [ ] Log4JConfiguration (0.0.0)
  - [✓] org.springdm.service.SpringDMS
  - [ ] org.springframework.osgi.sampl
  - [ ] SpringDMBundleClient (1.0.0)
- [ ] Target Platform
  - [ ] jcl104.over.slf4j (1.4.3)

[✓] Include optional dependencies when computing required bund

[✓] Add new workspace bundles to this launch configuration autor

[ ] Validate bundles automatically prior to launching

Our console output will look like this



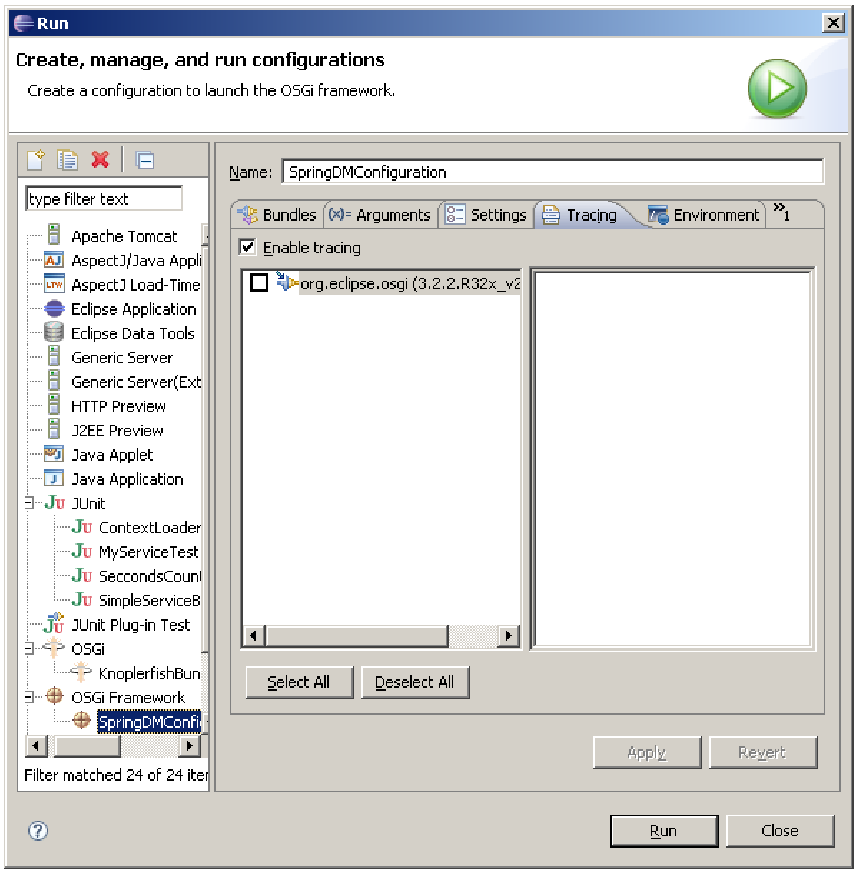
```
}

@Override
```

```
SpringDMConfiguration [OSGi Framework] C:\JAVA\jdk1.6.0_02\bin\javaw.exe (Feb 21,

osgi>
```

And even though our bundle is in ACTIVE (check it by typing **“ss”** in the console) state our service was not registered as an OSGi™ service simply because we did not implement our own OSGi™ Activator inside our bundle nor did we use any framework (i.e., Spring DM) to do it for us.
By re-enabling **Target Platform** which in our case is *Spring DM Target Platform* our service will be constructed and registered.
On the other hand one might say that our *System.out.println* statement inside of the service constructor proves only that the class was constructed. It does not prove anything about service registration. And you are right. To do that we are going to use Eclipse PDE tracing facility which will output certain messages to the console that will show us service registration.
Later on we will also write a quick client bundle that will use our service.

8. Eclipse PDE Tracing

By now you should be familiar with Eclipse PDE in the scope of this tutorial so some of the details will be skipped.
Open up **Run Dialog → OSGi™ Framework → SpringDMConfiguration**
Click on **Tracing** tab
You should see the following:


Run

Create, manage, and run configurations

Create a configuration to launch the OSGi framework.

type filter text

Apache Tomcat
AspectJ/Java Appli
AspectJ Load-Time
Eclipse Application
Eclipse Data Tools
Generic Server
Generic Server(Ext
HTTP Preview
J2EE Preview
Java Applet
Java Application
JUnit
ContextLoader
MyServiceTest
SeccondsCount
SimpleServiceB
JUnit Plug-in Test
OSGi
KnoplerfishBun
OSGi Framework
SpringDMConfi

Filter matched 24 of 24 iter

Name: SpringDMConfiguration

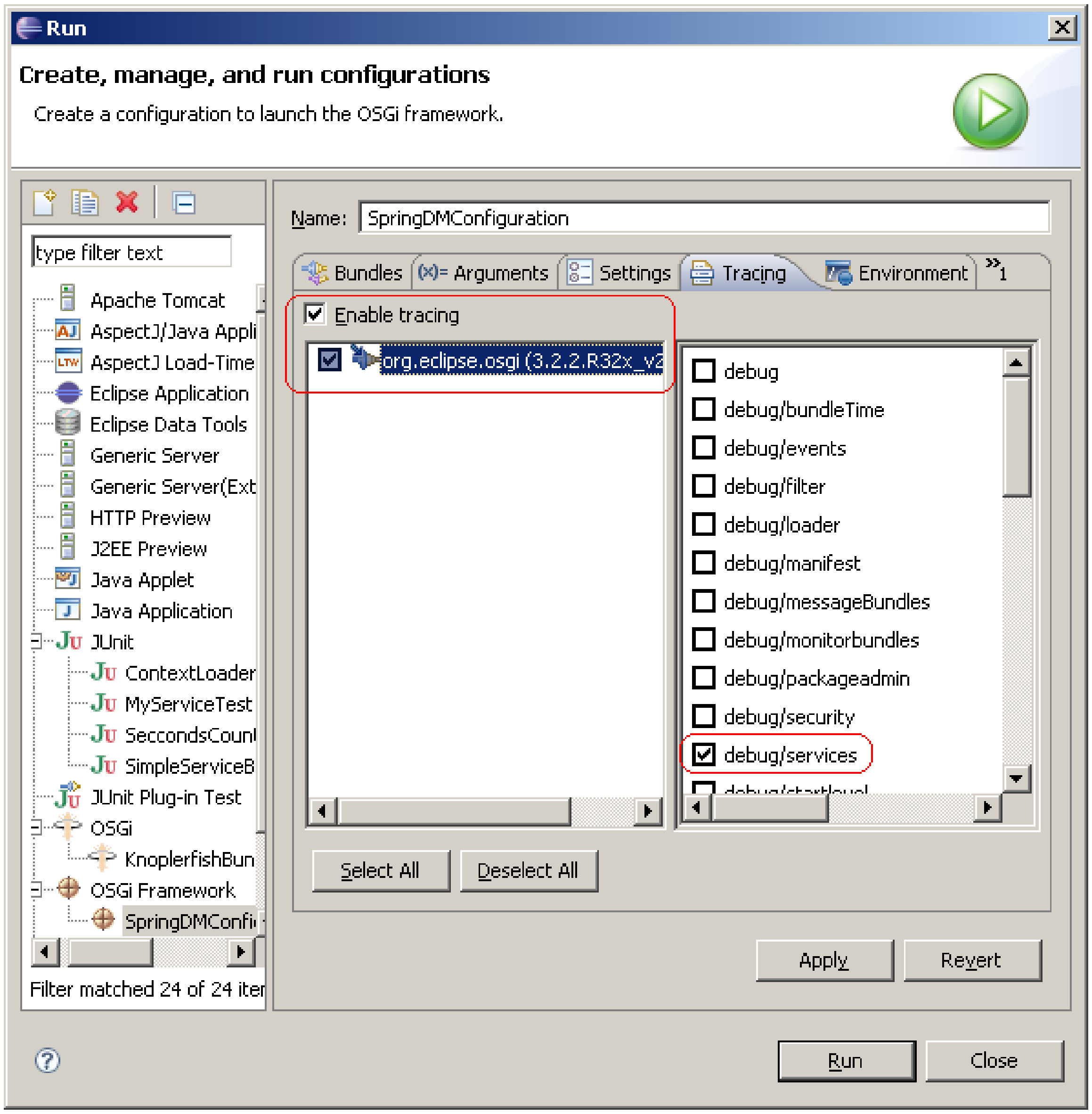
Bundles | Arguments | Settings | Tracing | Environment | »1

[✓] Enable tracing

[ ] org.eclipse.osgi (3.2.2.R32x_v2

Check **Enable Tracing**
Check **org.eclipse.osgi . .**
Check **debug/services**

Your settings should look like this



Run

Create, manage, and run configurations

Create a configuration to launch the OSGi framework.

Name: SpringDMConfiguration

Bundles | Arguments | Settings | Tracing | Environment

[✓] Enable tracing

[✓] org.eclipse.osgi (3.2.2.R32x_v2

- [ ] debug
- [ ] debug/bundleTime
- [ ] debug/events
- [ ] debug/filter
- [ ] debug/loader
- [ ] debug/manifest
- [ ] debug/messageBundles
- [ ] debug/monitorbundles
- [ ] debug/packageadmin
- [ ] debug/security
- [✓] debug/services

Click **Apply → Run**

You will see that your OSGi™ console will start spitting out tons of messages
Look for message similar to this

```
. . . . . .
getServiceReferences(null, "(objectClass=org.xml.sax.EntityResolver)")
getService[initial@reference:file:../../../../../../OSGi(tm)_workspace/S
pringDMBundle/ [23]]({org.xml.sax.EntityResolver}={service.id=22})
>>> Constructing FooServiceImpl
registerService[initial@reference:file:../../../../../../OSGi(tm)_worksp
ace/SpringDMBundle/ [23]]
({org.springdm.service.FooService}={org.springframework.OSGi(tm).bean.na
me=fooService, Bundle-SymbolicName=org.springdm.service.SpringDMService,
Bundle-Version=1.0, service.id=23})
. . . . . .
```

Now you see that your service was successfully registered and available by the name **`org.springdm.service.FooService.`**

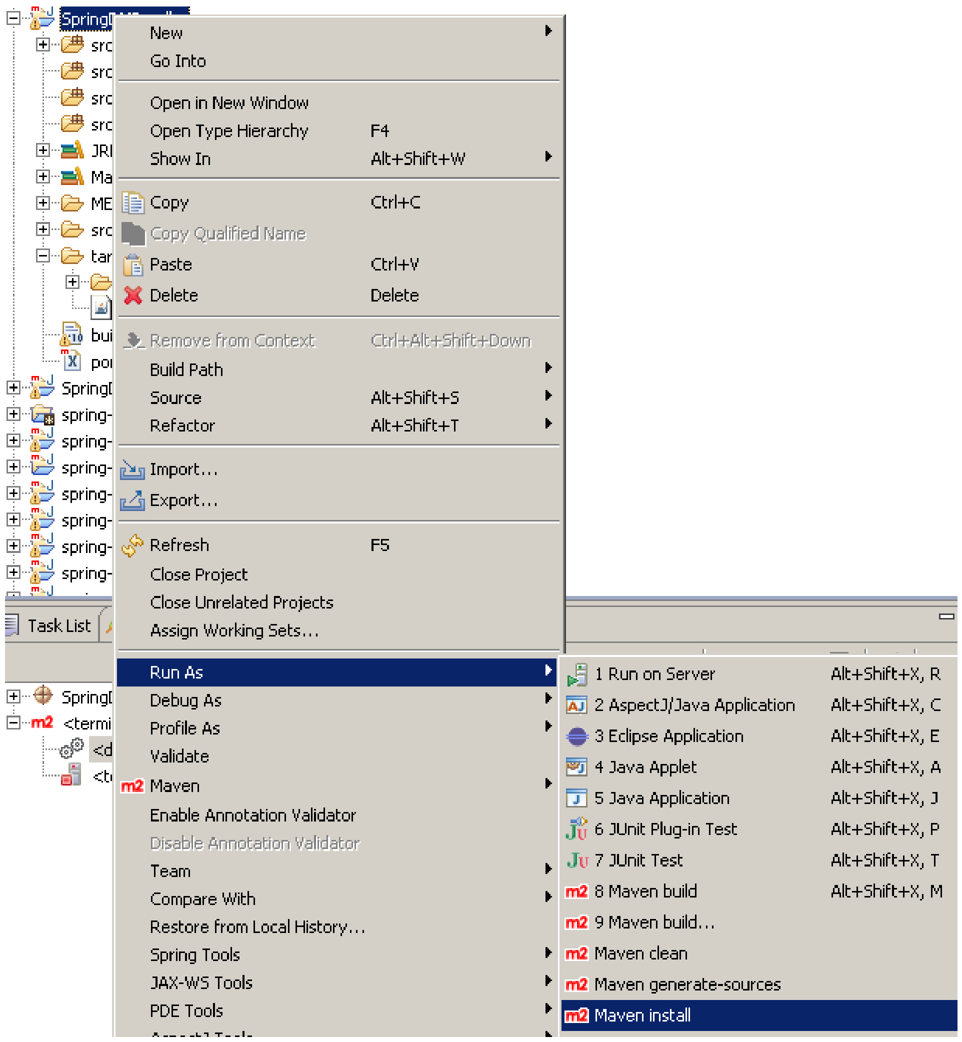
9. Packaging Spring DM OSGi™ bundle

Earlier while creating Maven **pom.xml** file we've added the following direction:

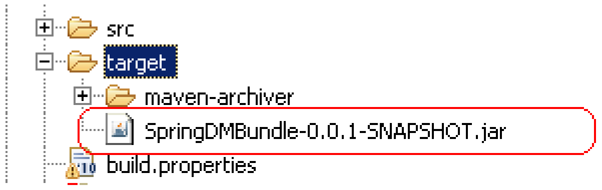
```
<build>
      <plugins>
            <plugin>
                  <groupId>org.apache.maven.plugins</groupId>
                  <artifactId>maven-jar-plugin</artifactId>
                  <configuration>
                        <archive>
                              <manifestFile>
                                    META-INF/MANIFEST.MF
                              </manifestFile>
                        </archive>
                  </configuration>
            </plugin>
      </plugins>
</build>
```

This should copy our META-INF/MANIFEST.MF file into the root of our package when Maven generates the JAR file which will be in **target** directory under root. To generate JAR package do the following:

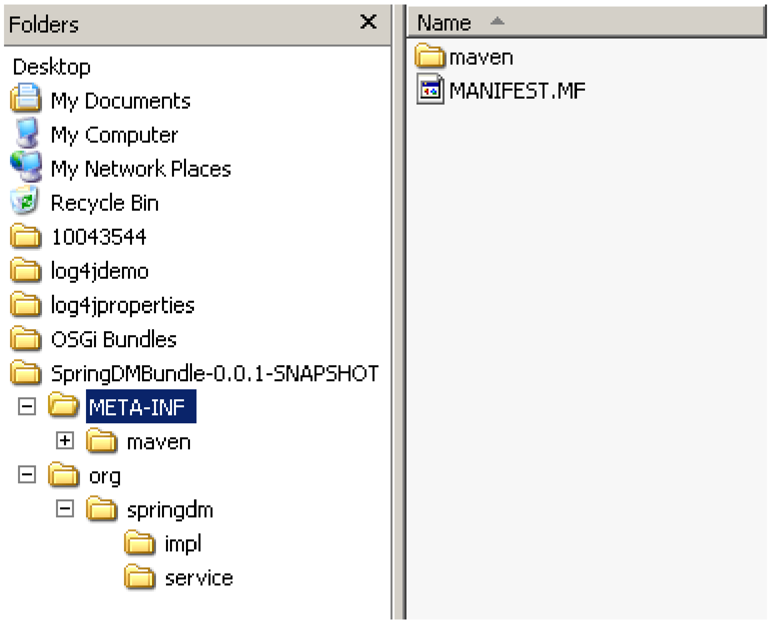
Right click on the project → Run As → Maven install



Refresh your target directory and you will see SpringDMBundle Snapshot packaged as JAR file

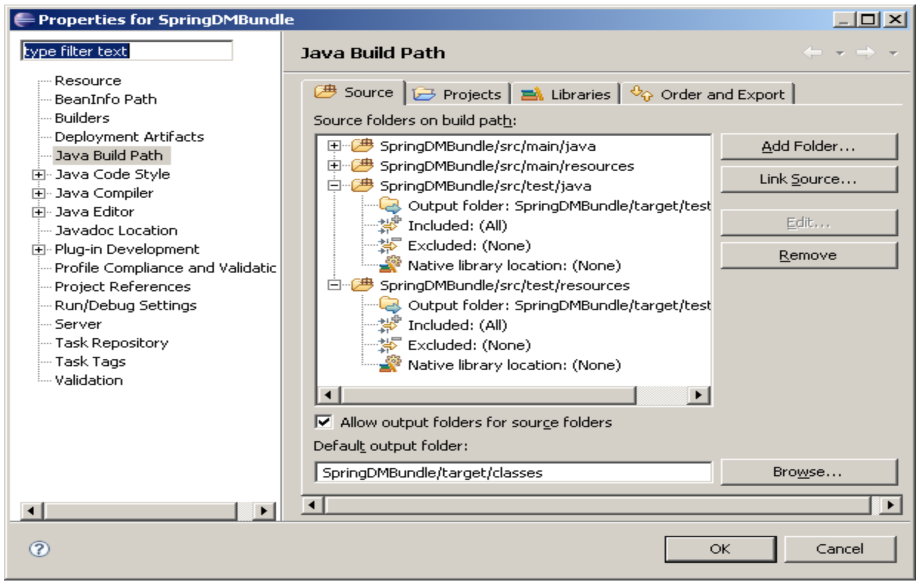


Using your favorite packaging utility un-JAR this file and look what is inside.



There is a problem. Even though META-INF directory does have the MANIFEST we asked Maven to copy, but it is missing **spring** directory which contains our Application Context definition files.
There are several ways to fix it with Maven and/or Eclipse, but since this tutorial is centered on Eclipse development environment we will fix it using Eclipse provided functionality.

Right-click on the project → **Properties** → **Java Build Path**
Click on **Source** tab
The following window will open up



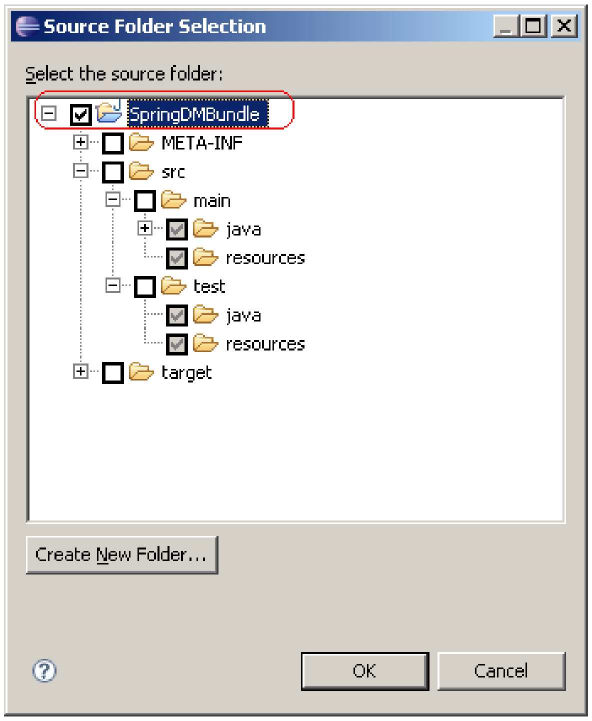
Source folders on build path:

```
SpringDMBundle/src/main/java
SpringDMBundle/src/main/resources
SpringDMBundle/src/test/java
    Output folder: SpringDMBundle/target/test
    Included: (All)
    Excluded: (None)
    Native library location: (None)
SpringDMBundle/src/test/resources
    Output folder: SpringDMBundle/target/test
    Included: (All)
    Excluded: (None)
    Native library location: (None)
```

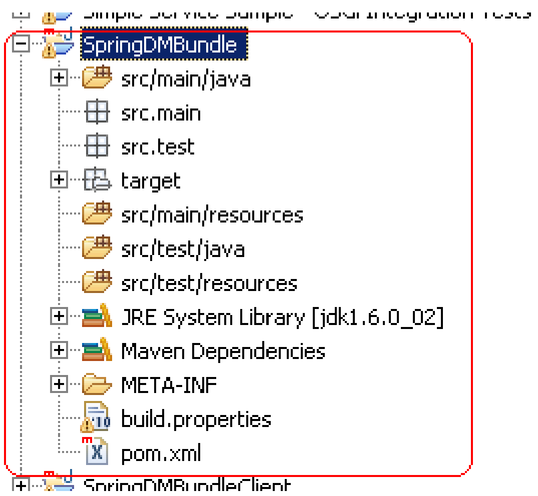
☑ Allow output folders for source folders

Default output folder: SpringDMBundle/target/classes

Click on Add Folder and add root of the project as source folder

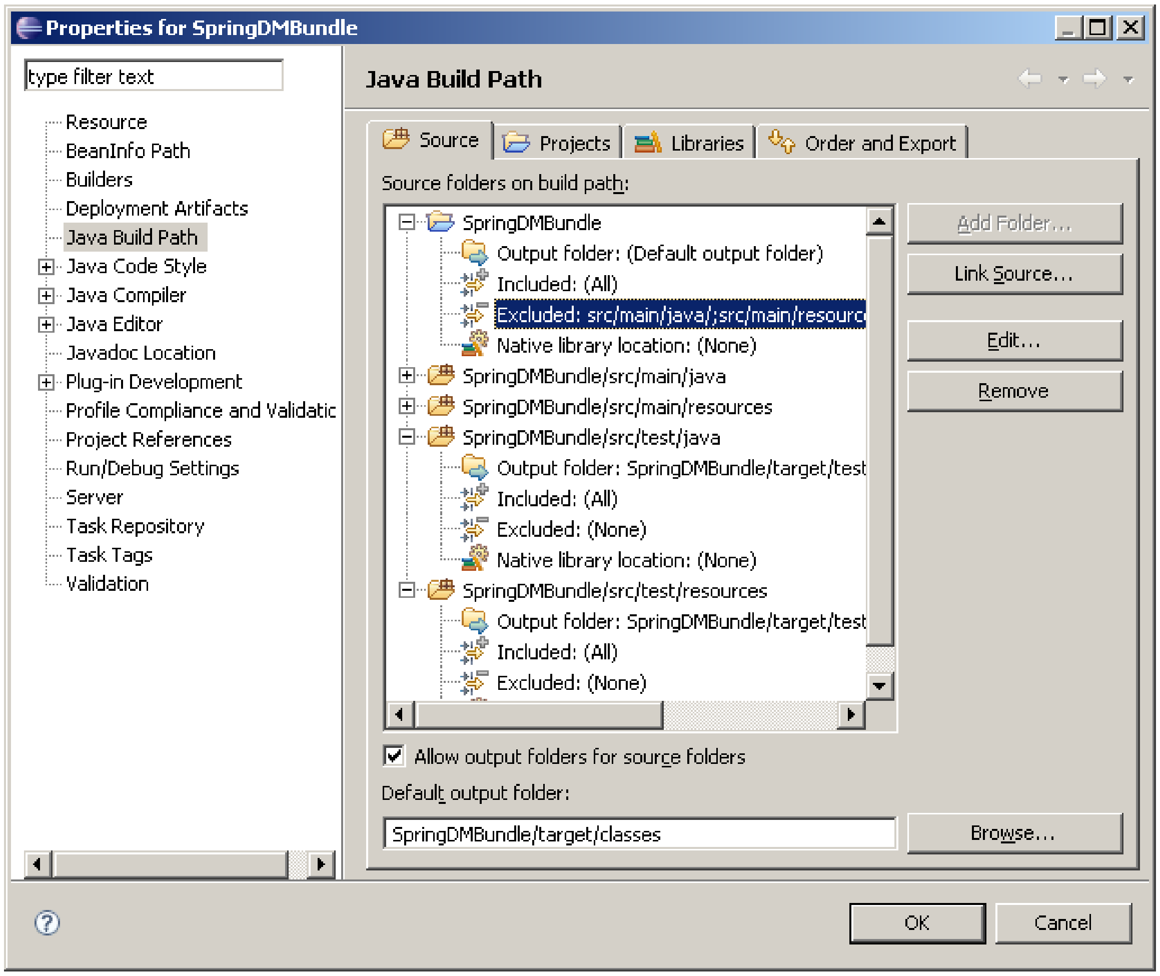


Click **OK** → **OK** (on the next window) → **OK** (on the properties window)
Your project should look like this:



Not out of the woods yet. You can see that **src.main**, **src.test** and **target** folders fell into Source category. We don't want that so we have to exclude them. To do that we would need to add Exclusion filters to our build path.

To do that open up the same project properties window → **Java Build Path** → **Source** Highlight **Excluded** under SpringDMBundle


Properties for SpringDMBundle

type filter text

- Resource
- BeanInfo Path
- Builders
- Deployment Artifacts
- Java Build Path
- Java Code Style
- Java Compiler
- Java Editor
- Javadoc Location
- Plug-in Development
- Profile Compliance and Validatic
- Project References
- Run/Debug Settings
- Server
- Task Repository
- Task Tags
- Validation

Java Build Path

Source | Projects | Libraries | Order and Export

Source folders on build path:

```
SpringDMBundle
    Output folder: (Default output folder)
    Included: (All)
    Excluded: src/main/java/;src/main/resourc
    Native library location: (None)
SpringDMBundle/src/main/java
SpringDMBundle/src/main/resources
SpringDMBundle/src/test/java
    Output folder: SpringDMBundle/target/test
    Included: (All)
    Excluded: (None)
    Native library location: (None)
SpringDMBundle/src/test/resources
    Output folder: SpringDMBundle/target/test
    Included: (All)
    Excluded: (None)
```

Add Folder... | Link Source... | Edit... | Remove

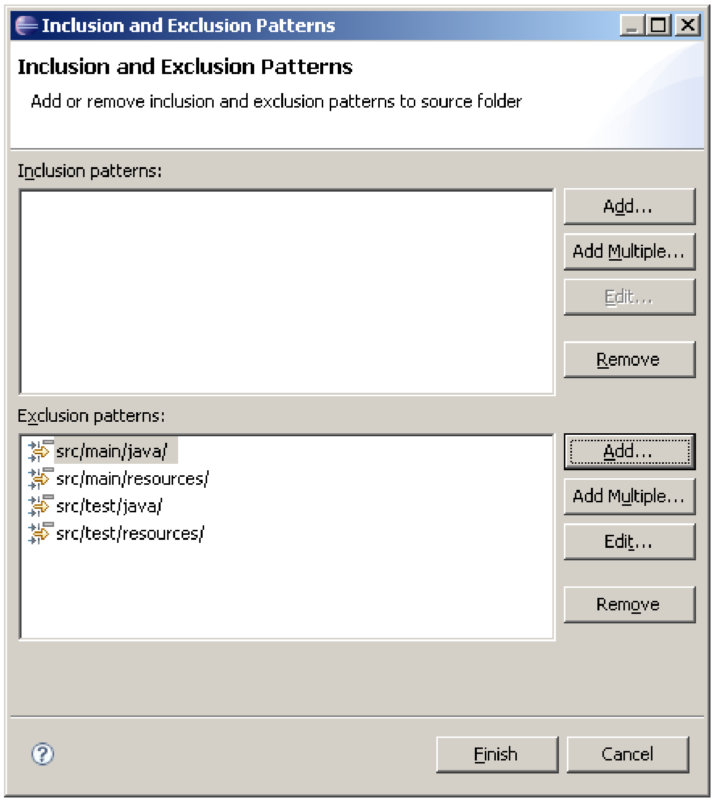
☑ Allow output folders for source folders

Default output folder: SpringDMBundle/target/classes

Browse...

OK | Cancel

Click on **Edit**. You should see the following window.



Inclusion and Exclusion Patterns

Add or remove inclusion and exclusion patterns to source folder

Inclusion patterns:

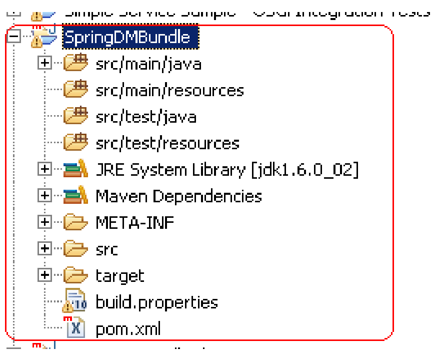
Exclusion patterns:

src/main/java/
src/main/resources/
src/test/java/
src/test/resources/

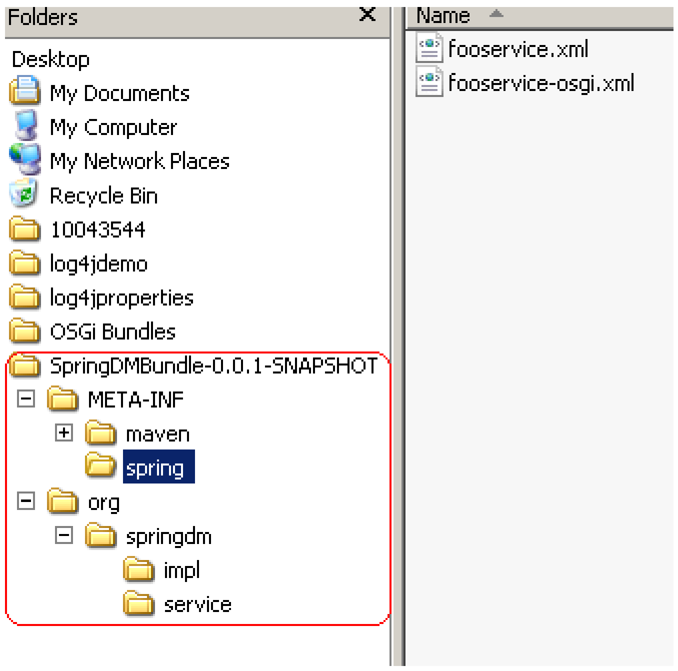
In the **Exclusion patterns** click on **Add → Browse →** and one by one add the following folders and files to the exclusion list:
src
target
.classpath
.project
build.properties.
pom.xml

Click on **Finish → OK**

Now your project looks just like before



Build the project
Right-click on the project → **Run As** → **Maven install**
Refresh the **target** folder and unpack the JAR (just to verify)
You should see the following structure



Now we have a valid Spring DM bundle that you can deploy into any OSGi™ container.

10.Testing

So far we have created SpringDM OSGi™ bundle, deployed it, exported the service and were able to verify its creation and registration.
We missed few things:

1. We did not test service functionality (however trivial it is)
2. We did not attempt to access our service from another bundle.

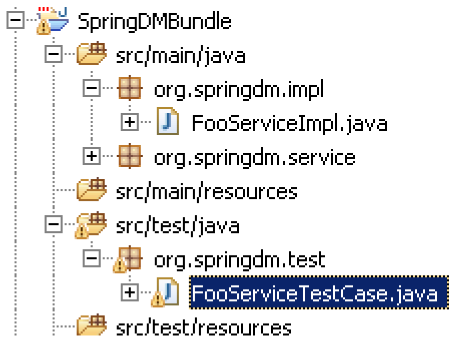
Spring DM provides a testing framework that will allow you to test your services without ever deploying them. The details on how to do that are very well covered in the screen casts available on Spring DM website. You can also look at the **simple-service** available with Spring DM distribution which has a very trivial example demonstrating Spring DM testing platform.
What we'll do here is write a simple unit test to test functionality of our trivial service and then we'll develop SpringDMBundleClient (another Spring DM bundle) that will use our service

10.1.Unit Test

The main purpose of the Unit Test within the scope of this tutorial is to demonstrate the strategy of testing Spring DM services independently from Spring DM and/or OSGi™ infrastructure.
If you remember we created two Application Context files where one was simply maintaining Spring Bean definitions representing our services while another one was using **osgi** namespace to perform post-processing (i.e., register our service as OSGi™ service)

The following Unit Test will test our service and its functionality as if it was a simple Spring Bean (cause it is)

Let's create **FooServiceTestCase**



And here is the code

```
package org.springdm.test;

import junit.framework.TestCase;

import org.springdm.service.FooService;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.FileSystemXmlApplicationContext;

public class FooServiceTestCase extends TestCase {
        private ApplicationContext ac;

        public void setUp(){
                ac = new FileSystemXmlApplicationContext("META-INF/spring/fooservice.xml");
        }

        public void testFooServiceFoo(){
                FooService service = (FooService) ac.getBean("fooService");
                assertNotNull(service);
                service.foo();
        }
}
```

Run it and make sure it passes

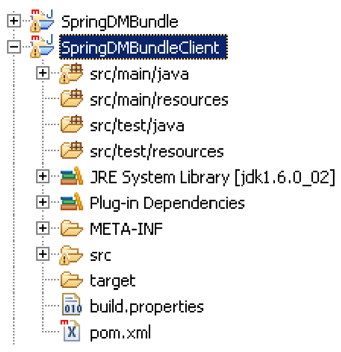
Also run Maven install (I assume by now you know how to do that) and you should see the following output.

```
INFO: Loading XML bean definitions from file
[C:\Spring\Eclipse\osgi_workspace\SpringDMBundle\META-INF\spring\fooservice.xml]
Feb 21, 2008 11:32:27 AM
org.springframework.context.support.AbstractApplicationContext
obtainFreshBeanFactory
INFO: Bean factory for application context
[org.springframework.context.support.FileSystemXmlApplicationContext@1bc887b]:
org.springframework.beans.factory.support.DefaultListableBeanFactory@39e5b5
Feb 21, 2008 11:32:27 AM
org.springframework.beans.factory.support.DefaultListableBeanFactory
preInstantiateSingletons
INFO: Pre-instantiating singletons in
org.springframework.beans.factory.support.DefaultListableBeanFactory@39e5b5:
defining beans [fooService]; root of factory hierarchy
>>> Constructing FooServiceImpl
>>> Executing FooServiceImpl
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.328 sec

Results :

Tests run: 1, Failures: 0, Errors: 0, Skipped: 0

[INFO] jar:jar
```

10.2.Spring DM client bundle

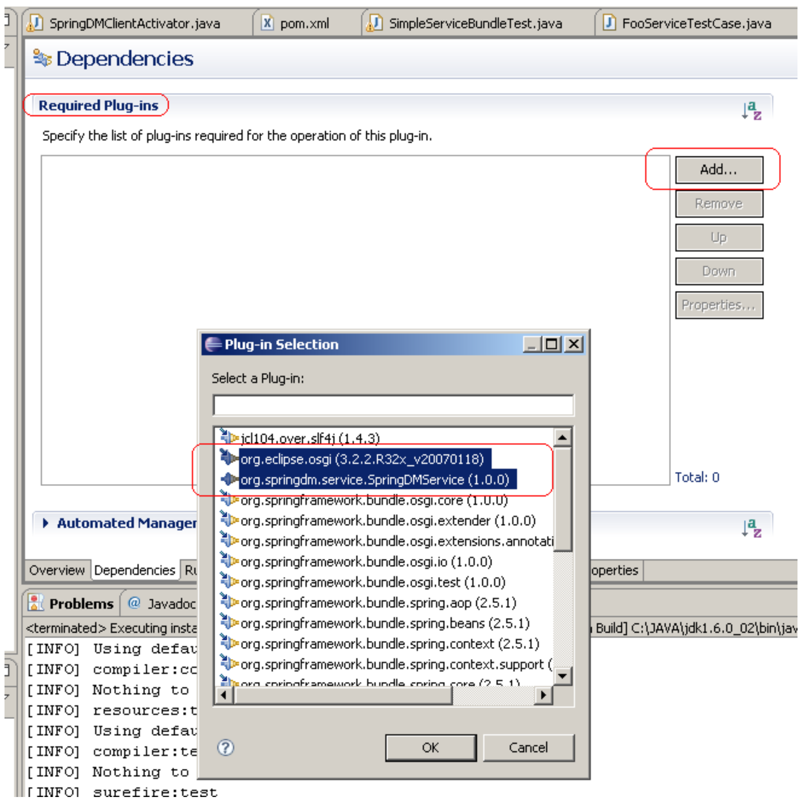
To make our example more exciting and useful let's develop another bundle called SpringDMBundle Client. Follow the steps in Chapter 3.

You should come up with the following project



```
SpringDMBundle
SpringDMBundleClient
    src/main/java
    src/main/resources
    src/test/java
    src/test/resources
    JRE System Library [jdk1.6.0_02]
    Plug-in Dependencies
    META-INF
    src
    target
    build.properties
    pom.xml
```

Open up **META-INF/MANIFEST.MF** file → Click on **Dependencies** tab → **Add** (on the Required Plug-ins) and add **org.eclipse.osgi** and **org.eclipse.service.SpringDMService** (our previous bundle)



SpringDMClientActivator.java | pom.xml | SimpleServiceBundleTest.java | FooServiceTestCase.java

Dependencies

Required Plug-ins

Specify the list of plug-ins required for the operation of this plug-in.

Add... | Remove | Up | Down | Properties...

Total: 0

Automated Manager

Overview | Dependencies | Ru | operties

Problems | Javadoc

<terminated> Executing insta Build] C:\JAVA\jdk1.6.0_02\bin\jav

```
[INFO] Using defau
[INFO] compiler:cc
[INFO] Nothing to
[INFO] resources:t
[INFO] Using defau
[INFO] compiler:te
[INFO] Nothing to
[INFO] surefire:test
```

Plug-in Selection

Select a Plug-in:

```
jcl104.over.slf4j (1.4.3)
org.eclipse.osgi (3.2.2.R32x_v20070118)
org.springdm.service.SpringDMService (1.0.0)
org.springframework.bundle.osgi.core (1.0.0)
org.springframework.bundle.osgi.extender (1.0.0)
org.springframework.bundle.osgi.extensions.annotati
org.springframework.bundle.osgi.io (1.0.0)
org.springframework.bundle.osgi.test (1.0.0)
org.springframework.bundle.spring.aop (2.5.1)
org.springframework.bundle.spring.beans (2.5.1)
org.springframework.bundle.spring.context (2.5.1)
org.springframework.bundle.spring.context.support (
org.springframework.bundle.spring.core (2.5.1)
```

OK | Cancel

Click **OK**. Make sure the entries were added to MANIFEST file by clicking on MANIFEST.MF tab

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: SpringDMBundleClient
Bundle-SymbolicName: SpringDMBundleClient
Bundle-Version: 1.0.0
Require-Bundle: org.eclipse.osgi,
 org.springdm.service.SpringDMService
```

Now let's create Bundle Activator which will be executed by OSGi™ container during initialization. BundleActivator interface defines two methods START and STOP.
We are going to implement START method where we will attempt to access our SpringDMBundle .FooService and execute foo() method.

Why are we doing it? Simply because we can and it is the quickest way to demonstrate how one bundle can access services exported by another bundle. It will also introduce you to OSGi™ API.

The use case for our START method will be simple. Locate *FooService* and execute *foo()* method.

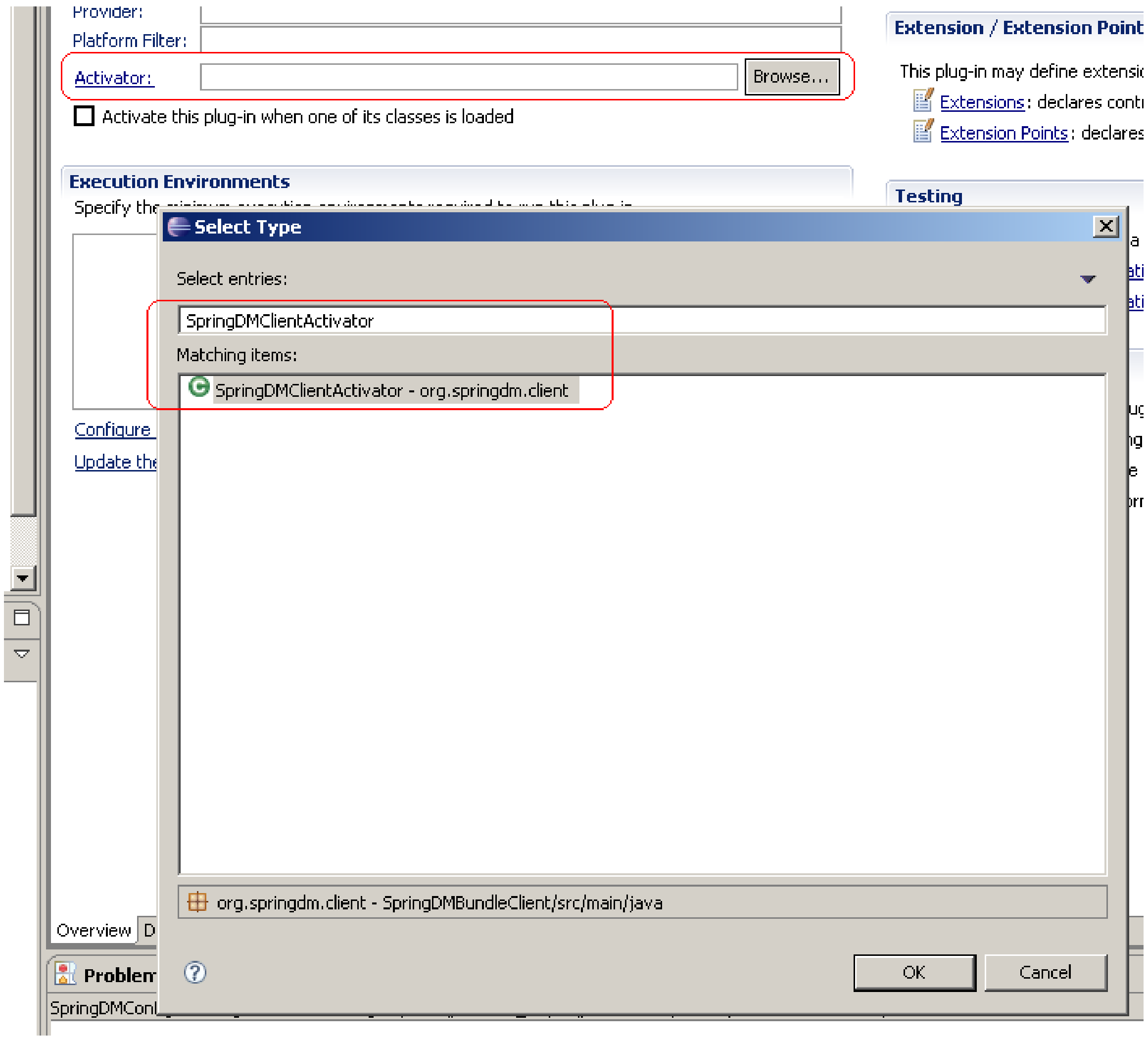
Chapter 3.2 of Spring DM reference documentation states: *"The extender bundle creates applications contexts asynchronously. This behavior ensures that starting an OSGi Service Platform is fast and that bundles with service inter-dependencies do not cause deadlock on startup.. . ."*. This means that even if we were to play with OSGi™ start levels we can't rely on the fact that during initialization and activation of our client bundle our SpringDMBundle.FooService is registered, so we are going to loop 10 times or until service is located (whichever comes first). Once service is located we will retrieve it from the BundleContext and execute it. Very trivial example

Here is the code

```java
package org.springdm.client;

import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;
import org.osgi.framework.ServiceReference;
import org.springdm.service.FooService;

public class SpringDMClientActivator implements BundleActivator {

   public void start(final BundleContext bCtx) throws Exception {

        Thread t = new Thread(new Runnable(){
                boolean registered = false;
                ServiceReference ref = null;
                int safety = 0;
                public void run(){
                   while (!registered && safety < 10){
                        System.out.println("Trying to get the reference to FooService");
                        ref = bCtx.getServiceReference(FooService.class.getName());
                        if (ref != null){
                           break;
                        }
                        try {
                           Thread.sleep(100);
                        } catch (InterruptedException e) {}
                        safety++;
                   }

                   if (ref == null){
                        System.out.println("Service " + FooService.class.getName() +
                                                 " is not registered");
                   } else {
                        System.out.println("Found service");
                        FooService service = (FooService) bCtx.getService(ref);
                        System.out.println("Retrieved service: " +
                                                 service.getClass().getName());
                        service.foo();
                   }
                }

        });
        t.start();
   }

   public void stop(BundleContext bCtx) throws Exception {
        System.out.println("Stopping service");
   }
}
```

Now we have to define Activator in our MANIFEST file.

Open MANIFEST file → Click on Overview tab and let's add **SpringDMClientActivator** we just created by clicking on Browse for Activator text field



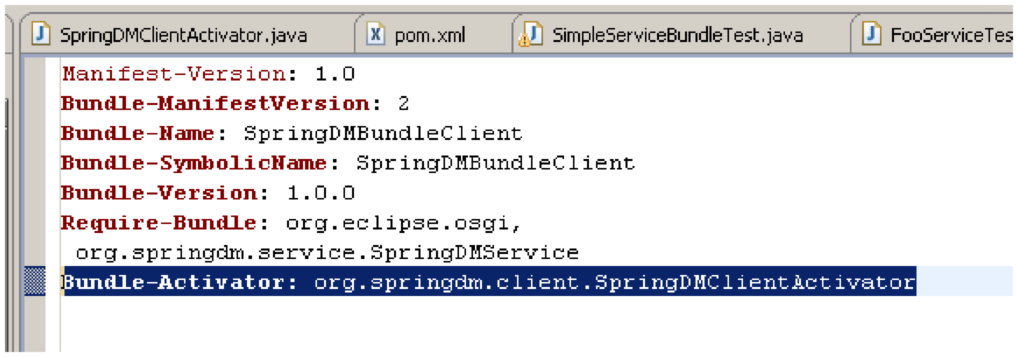
SpringDMClientActivator

Matching items:

SpringDMClientActivator - org.springdm.client

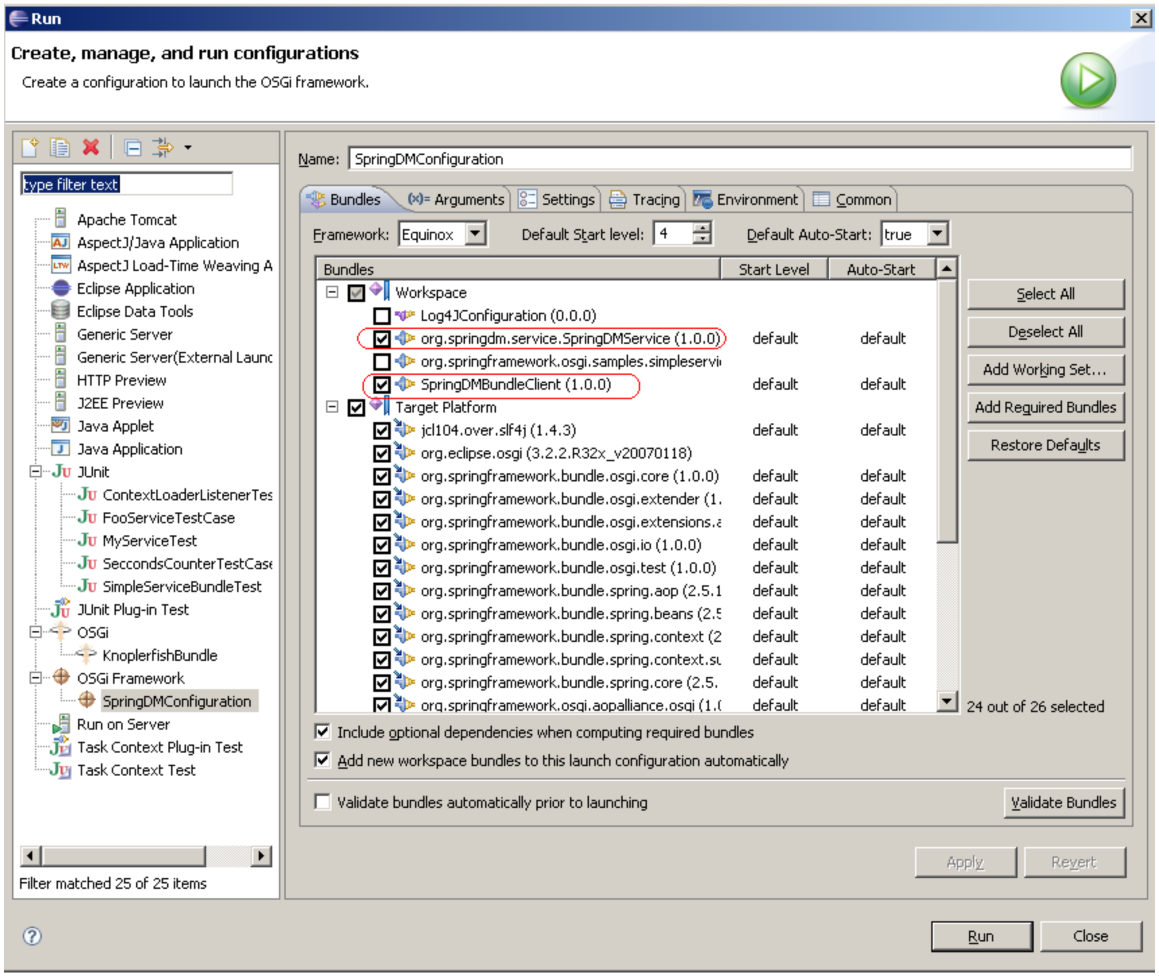
org.springdm.client - SpringDMBundleClient/src/main/java

Click **OK** (save CTRL+S)
Verify that the entry was added by clicking on MANIFEST.MF tab



```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: SpringDMBundleClient
Bundle-SymbolicName: SpringDMBundleClient
Bundle-Version: 1.0.0
Require-Bundle: org.eclipse.osgi,
 org.springdm.service.SpringDMService
Bundle-Activator: org.springdm.client.SpringDMClientActivator
```

This is it. Open **Run Dialog** and make sure SpringDMBundle and SpringDMBundleClient is selected



Run

Create, manage, and run configurations

Create a configuration to launch the OSGi framework.

type filter text

- Apache Tomcat
- AspectJ/Java Application
- AspectJ Load-Time Weaving A
- Eclipse Application
- Eclipse Data Tools
- Generic Server
- Generic Server(External Launc
- HTTP Preview
- J2EE Preview
- Java Applet
- Java Application
- JUnit
  - ContextLoaderListenerTes
  - FooServiceTestCase
  - MyServiceTest
  - SeccondsCounterTestCase
  - SimpleServiceBundleTest
- JUnit Plug-in Test
- OSGi
  - KnoplerfishBundle
- OSGi Framework
  - SpringDMConfiguration
- Run on Server
- Task Context Plug-in Test
- Task Context Test

Filter matched 25 of 25 items

Name: SpringDMConfiguration

Bundles | Arguments | Settings | Tracing | Environment | Common

Framework: Equinox   Default Start level: 4   Default Auto-Start: true

Bundles	Start Level	Auto-Start
☑ Workspace		
☐ Log4JConfiguration (0.0.0)		
☑ org.springdm.service.SpringDMService (1.0.0)	default	default
☐ org.springframework.osgi.samples.simpleservi		
☑ SpringDMBundleClient (1.0.0)	default	default
☑ Target Platform		
☑ jcl104.over.slf4j (1.4.3)	default	default
☑ org.eclipse.osgi (3.2.2.R32x_v20070118)		
☑ org.springframework.bundle.osgi.core (1.0.0)	default	default
☑ org.springframework.bundle.osgi.extender (1.	default	default
☑ org.springframework.bundle.osgi.extensions.a	default	default
☑ org.springframework.bundle.osgi.io (1.0.0)	default	default
☑ org.springframework.bundle.osgi.test (1.0.0)	default	default
☑ org.springframework.bundle.spring.aop (2.5.1	default	default
☑ org.springframework.bundle.spring.beans (2.5	default	default
☑ org.springframework.bundle.spring.context (2	default	default
☑ org.springframework.bundle.spring.context.su	default	default
☑ org.springframework.bundle.spring.core (2.5.	default	default
☑ org.springframework.osgi.aopalliance.osgi (1.(	default	default

Select All | Deselect All | Add Working Set... | Add Required Bundles | Restore Defaults

24 out of 26 selected

☑ Include optional dependencies when computing required bundles

☑ Add new workspace bundles to this launch configuration automatically

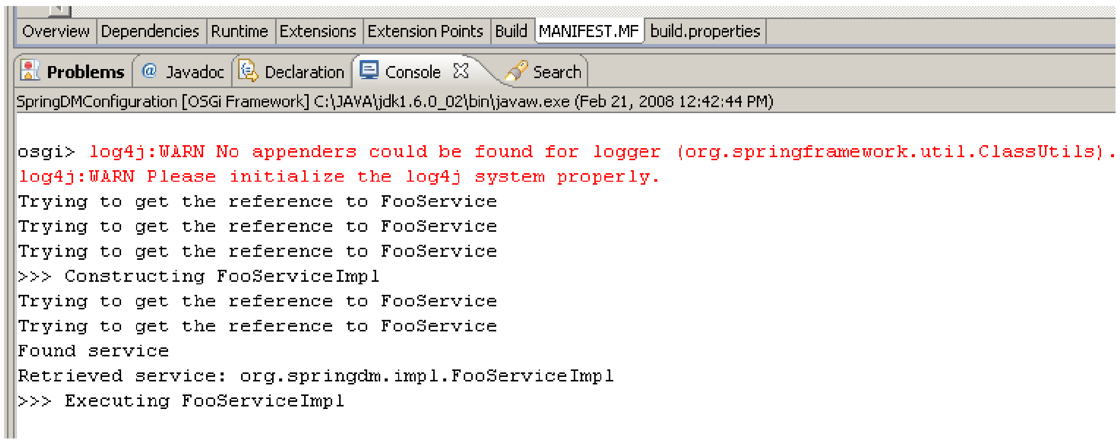
☐ Validate bundles automatically prior to launching   Validate Bundles

Apply Revert

Run Close

Click on **Run**

You should see the following output:



Overview | Dependencies | Runtime | Extensions | Extension Points | Build | MANIFEST.MF | build.properties

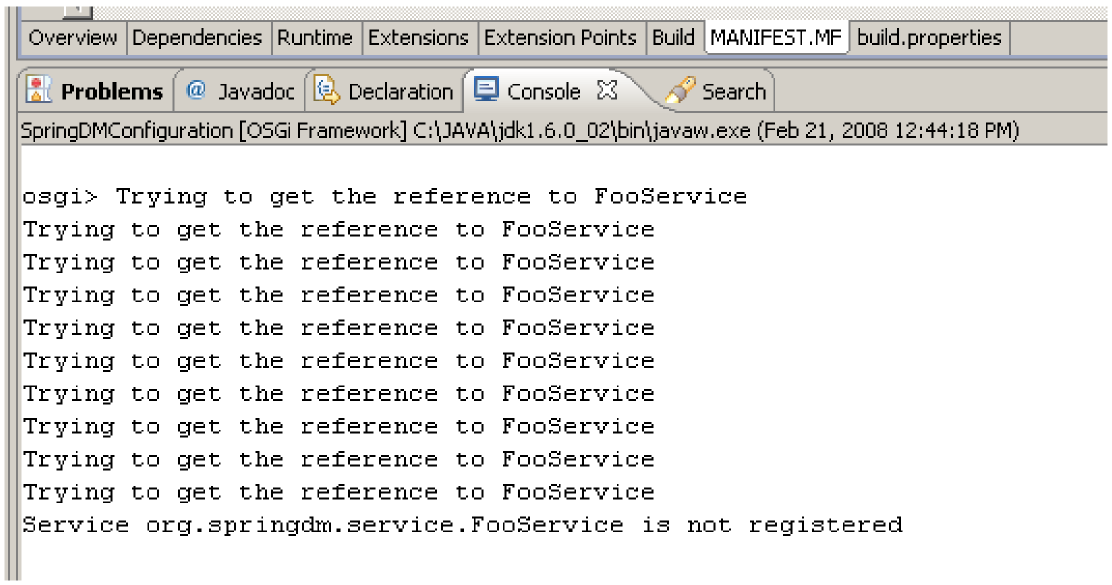
Problems | Javadoc | Declaration | Console | Search

SpringDMConfiguration [OSGi Framework] C:\JAVA\jdk1.6.0_02\bin\javaw.exe (Feb 21, 2008 12:42:44 PM)

```
osgi> log4j:WARN No appenders could be found for logger (org.springframework.util.ClassUtils).
log4j:WARN Please initialize the log4j system properly.
Trying to get the reference to FooService
Trying to get the reference to FooService
Trying to get the reference to FooService
>>> Constructing FooServiceImpl
Trying to get the reference to FooService
Trying to get the reference to FooService
Found service
Retrieved service: org.springdm.impl.FooServiceImpl
>>> Executing FooServiceImpl
```

And for the sanity check open **Run Dialog** again and uncheck **Target Platform**. By doing so you will un-deploy all Spring DM bundles.
You Application Context will not be recognized and since no other facilities are provided do export FooService it will not be exported and our client SpringDMClientActivator won't be able to locate it.

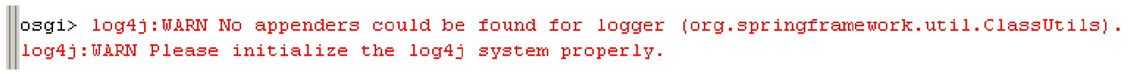
Here is the expected output



```
SpringDMConfiguration [OSGi Framework] C:\JAVA\jdk1.6.0_02\bin\javaw.exe (Feb 21, 2008 12:44:18 PM)

osgi> Trying to get the reference to FooService
Trying to get the reference to FooService
Trying to get the reference to FooService
Trying to get the reference to FooService
Trying to get the reference to FooService
Trying to get the reference to FooService
Trying to get the reference to FooService
Trying to get the reference to FooService
Trying to get the reference to FooService
Trying to get the reference to FooService
Service org.springdm.service.FooService is not registered
```

11.Logging (log4j)

By now you probably annoyed with the following message popping up in your console every now and then

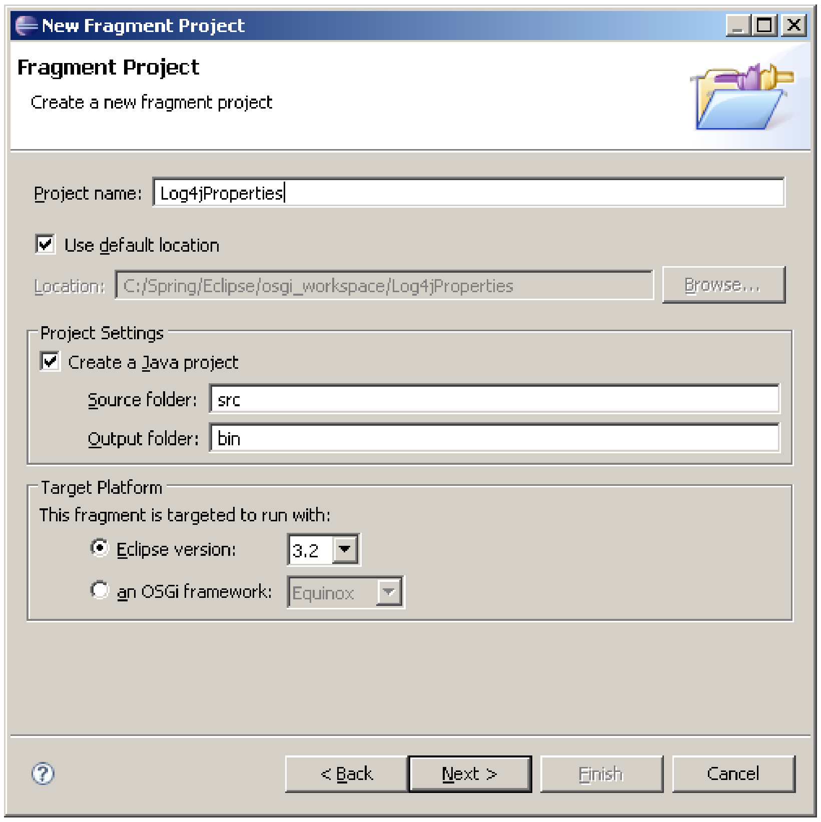


```
osgi> log4j:WARN No appenders could be found for logger (org.springframework.util.ClassUtils).
log4j:WARN Please initialize the log4j system properly.
```

I assume the reader knows what this mean, but how do we fix it within the scope of this tutorial?
Again there are several ways to do that. I am going to demonstrate one which follows the spirit of OSGi™ component architecture and it will also allow us to change logging configuration independently from the components that use logging services. This means we will not be creating and deploying **log4j.properties** file inside of any of the bundles that we have created.
Instead we will create a Fragment bundle which will contain our **log4j.properties** file and will be attached to the bundle providing logging services. The bundle that provides Log4j logging services inside of Spring DM Target Platform is **org.springframework.osgi.log4j.osgi**.

Let's create another project. This time we will create Plug-in Development Environment (PDE) Fragment project that will contain **log4j.properties** file and will be attached to **org.springframework.osgi.log4j.osgi** bundle**.**

New → Other → Plug-in Development → Fragment Project → Next
Let's name it Log4JProperties


New Fragment Project

Fragment Project

Create a new fragment project

Project name: Log4jProperties

☑ Use default location

Location: C:/Spring/Eclipse/osgi_workspace/Log4jProperties

Project Settings

☑ Create a Java project

Source folder: src

Output folder: bin

Target Platform

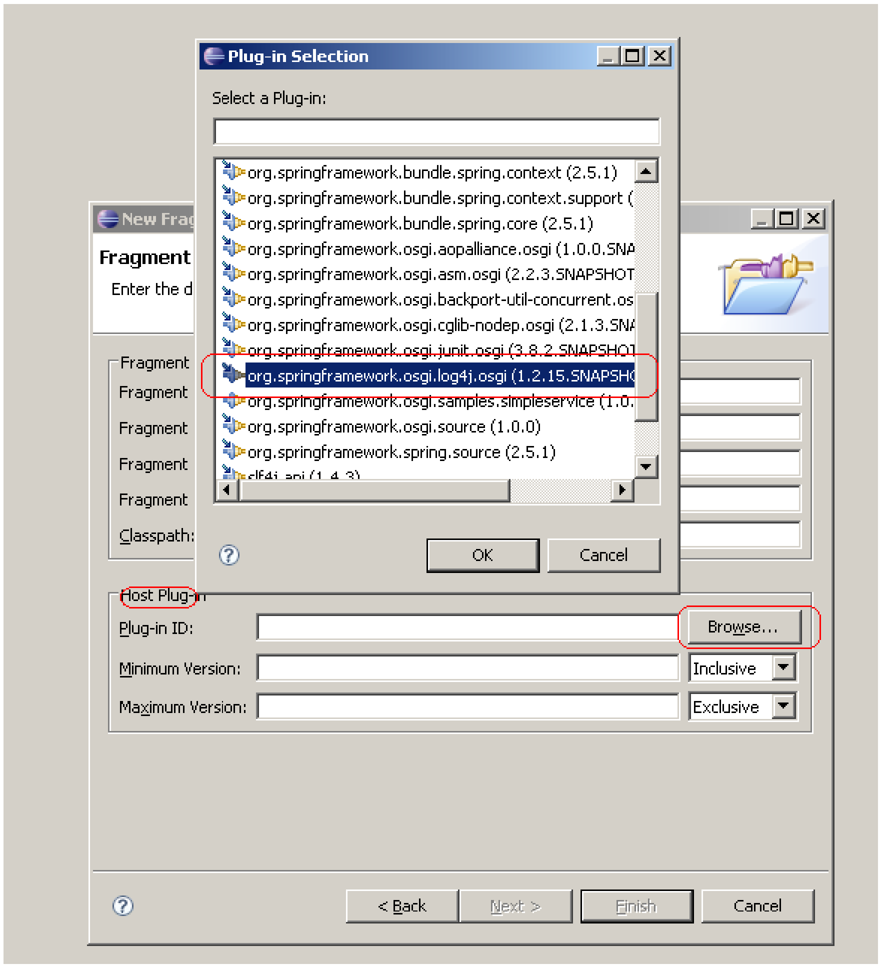
This fragment is targeted to run with:

● Eclipse version: 3.2

○ an OSGi framework: Equinox

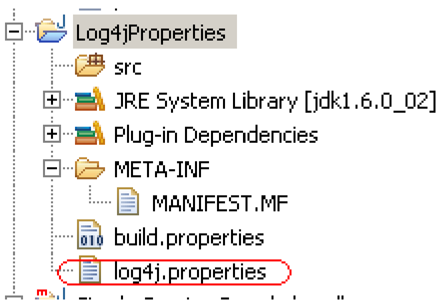
Click **Next.** On the next window in the **Host-plug-in** area click on **Browse** and select **org.springframework.osgi.log4j.osgi** which is the bundle we are going to attach to.

Click **OK → Finish** (see below)



org.springframework.bundle.spring.context (2.5.1)
org.springframework.bundle.spring.context.support (
org.springframework.bundle.spring.core (2.5.1)
org.springframework.osgi.aopalliance.osgi (1.0.0.SNA
org.springframework.osgi.asm.osgi (2.2.3.SNAPSHOT
org.springframework.osgi.backport-util-concurrent.os
org.springframework.osgi.cglib-nodep.osgi (2.1.3.SNA
org.springframework.osgi.junit.osgi (3.8.2.SNAPSHOT
org.springframework.osgi.log4j.osgi (1.2.15.SNAPSHO
org.springframework.osgi.samples.simpleservice (1.0.
org.springframework.osgi.source (1.0.0)
org.springframework.spring.source (2.5.1)
slf4j.api (1.4.3)

Create **log4j.properties** file at the root of the project



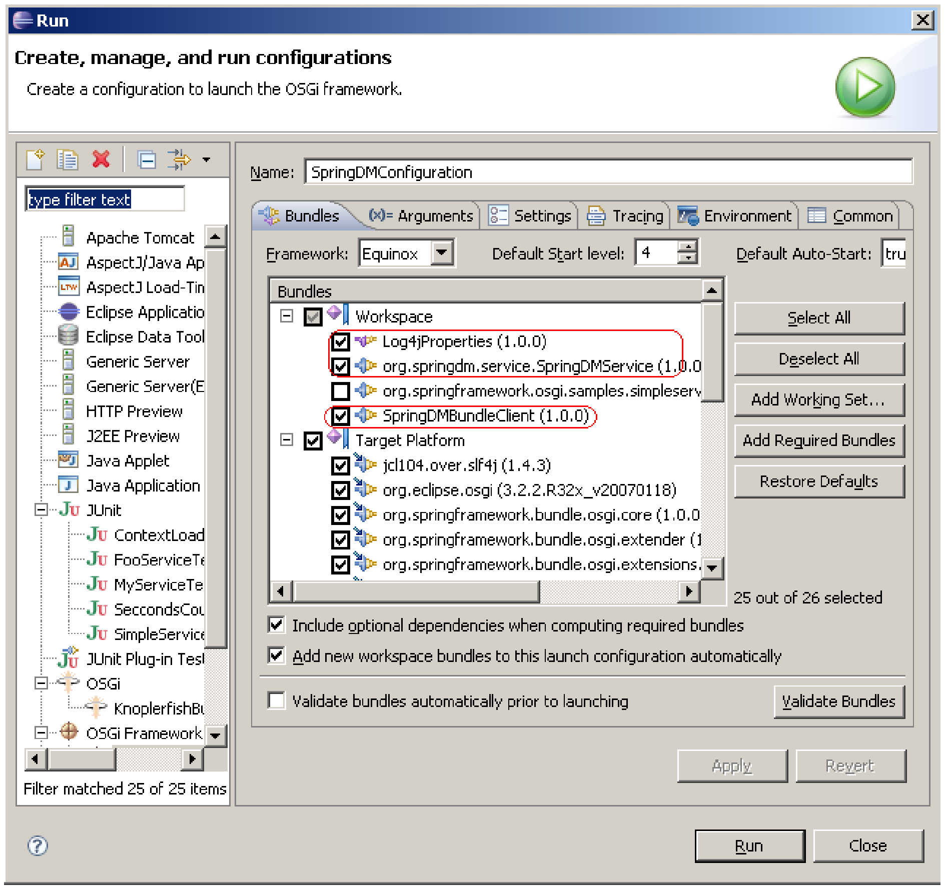
Here is what I have in my **log4j.properties** file

```
log4j.rootLogger=info, default
# default is set to be a ConsoleAppender.
log4j.appender.default=org.apache.log4j.ConsoleAppender
# default uses PatternLayout.
log4j.appender.default.layout=org.apache.log4j.PatternLayout
log4j.appender.default.layout.ConversionPattern=%-4r [%t] %-5p %c %x - %m%n
```

Nothing special

That is all we need.

Open **Run Dialog** and make sure the two bundles plus the **Log4jProperty** bundle is selected



Name: SpringDMConfiguration

Framework: Equinox Default Start level: 4 Default Auto-Start: tru

Bundles
- [✓] Workspace
  - [✓] Log4jProperties (1.0.0)
  - [✓] org.springdm.service.SpringDMService (1.0.0
  - [ ] org.springframework.osgi.samples.simpleserv
  - [✓] SpringDMBundleClient (1.0.0)
- [✓] Target Platform
  - [✓] jcl104.over.slf4j (1.4.3)
  - [✓] org.eclipse.osgi (3.2.2.R32x_v20070118)
  - [✓] org.springframework.bundle.osgi.core (1.0.0
  - [✓] org.springframework.bundle.osgi.extender (1
  - [✓] org.springframework.bundle.osgi.extensions.

25 out of 26 selected

[✓] Include optional dependencies when computing required bundles

[✓] Add new workspace bundles to this launch configuration automatically

[ ] Validate bundles automatically prior to launching

Click **Run** and your output should look similar to this:

```
osgi> 0    [Start Level Event Dispatcher] INFO
org.springframework.osgi.extender.internal.ContextLoaderListener  - Starting
org.springframework.osgi.extender bundle v.[1.0.0]
94   [Start Level Event Dispatcher] INFO
org.springframework.osgi.extender.internal.ContextLoaderListener  - disabled
automatic Spring-DM annotation processing;
[ org.springframework.osgi.extender.annotation.auto.processing=null]
110  [SpringOsgiExtenderThread-1] INFO
org.springframework.osgi.context.support.OsgiBundleXmlApplicationContext  -
Refreshing
org.springframework.osgi.context.support.OsgiBundleXmlApplicationContext@c3014:
display name
[OsgiBundleXmlApplicationContext(bundle=org.springdm.service.SpringDMService,
config=osgibundle:/META-INF/spring/*.xml)]; startup date [Thu Feb 21 13:17:04 EST
2008]; root of context hierarchy
Trying to get the reference to FooService
188  [SpringOsgiExtenderThread-1] INFO
org.springframework.beans.factory.xml.XmlBeanDefinitionReader  - Loading XML bean
definitions from URL [bundleentry://23/META-INF/spring/fooservice-osgi.xml]
Trying to get the reference to FooService
Trying to get the reference to FooService
328  [SpringOsgiExtenderThread-1] INFO
org.springframework.beans.factory.xml.XmlBeanDefinitionReader  - Loading XML bean
definitions from URL [bundleentry://23/META-INF/spring/fooservice.xml]
360  [SpringOsgiExtenderThread-1] INFO
org.springframework.osgi.context.support.OsgiBundleXmlApplicationContext  - Bean
factory for application context
[org.springframework.osgi.context.support.OsgiBundleXmlApplicationContext@c3014]:
org.springframework.beans.factory.support.DefaultListableBeanFactory@1fa681c
375  [SpringOsgiExtenderThread-2] INFO
org.springframework.beans.factory.support.DefaultListableBeanFactory  - Pre-
instantiating singletons in
org.springframework.beans.factory.support.DefaultListableBeanFactory@1fa681c:
defining beans [fooServiceOsgi,fooService]; root of factory hierarchy
Trying to get the reference to FooService
>>> Constructing FooServiceImpl
422  [SpringOsgiExtenderThread-2] INFO
org.springframework.osgi.service.exporter.support.OsgiServiceFactoryBean  -
Publishing service under classes [{org.springdm.service.FooService}]
422  [SpringOsgiExtenderThread-2] INFO
org.springframework.osgi.context.support.OsgiBundleXmlApplicationContext  -
Publishing application context with properties
(org.springframework.context.service.name=org.springdm.service.SpringDMService)
Trying to get the reference to FooService
Found service
Retrieved service: org.springdm.impl.FooServiceImpl
>>> Executing FooServiceImpl
```

Now you can change System.out.println in our bundles to use Log4j.

The benefit of this approach is that you don't have to redeploy your services every time you want to change the logging levels. Simply redeploy the Log4JProperty Fragment bundle and restart **org.springframework.osgi.log4j.osgi** bundle.

12.Conclusion

Spring has been always branded as stateless container. With introduction of Spring DM for OSGi™ environment, not only you can make it state full, but you can introduce the concept of deploying and un-deploying beans (services) similar to the way we deploy WARs and EARs inside of Application Servers, but in a much lighter fashion. Bean(s) are still the same old POJOs that reside inside of Spring Application Context which is wrapped in OSGi™ bundle managed and supported by Spring Dynamic Modules. Such beans become services within OSGi™ container and can interoperate with one another as services in a true SOA fashion but without introducing the complexity of an ESB in the same way Spring addresses J2EE concerns without complexity of J2EE containers.
No, OSGi is not and ESB, it is simply a light container for hosting bundles. However, with Spring Integration and Spring Web Services as OSGi™ bundles . . . could we possibly be looking at the light weight ESB based on Spring DM for OSGi™ Service platform. . .?