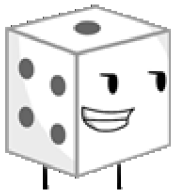


CitizenRandom

Random thoughts in Computer Science and Engineering

Managing OSGi Transitive Dependencies (Part 1)

February 15, 2015 Programming

Contents [hide]

Introduction

This tutorial is a compilation of several techniques used by our development team in order to automate the cumbersome task of collecting all transitive dependencies of a bundle (which can be a very time consuming task). Most of the documentation provided here is the result of weeks of experimentation, research and reverse engineer of the technologies involved. Our goal is to spare you from wasting the same amount of time, effort, and health with the same problems, fruit of the lack of accuracy of the barely existing documentation

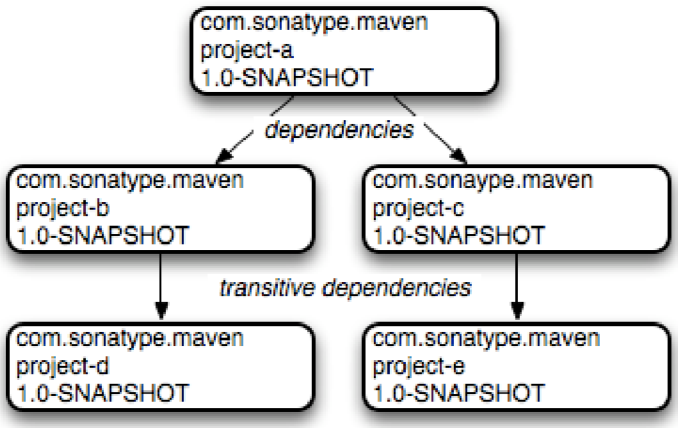
concerning OSGi and its building tools. Please read this article slowly and carefully and it will save you from enormous headaches, we did our best to make it easy to understand. Good luck!

Ps. Keywords such as smartbuildings, lumina, sb-buildmaster, shared-components, etc. Are related to our project only. They show up in the examples of this article just to show you where would you expect to see your project's related items. You should focus on org.osgi.* and alike keywords which are the ones you must ensure if you follow any of these examples.

Advice: Prior to the reading of this tutorial, you should be aware of how a maven bundle plugin project is created.

What is a Transitive Dependency

A transitive dependency consists of a dependency used by another dependency of our project. They are also known as indirect dependencies or *nth* level dependencies. Consider the following example:



Imagine that out bundle is the **project-a**, having as direct dependencies bundle **project-b** and **project-c**. By checking our bundle imports in the MANIFEST.MF closely we'll find out that it also contains transitive dependencies, 2nd level dependencies, which are **project-d** and **project-e**. All these bundles must be present at runtime in the OSGi Framework Session/Instance. However, in real world scenarios these transitive dependencies can be numerous, having more dependencies to satisfy, thus deriving the need to automate the task of collecting these dependencies and deploy them at runtime. This is what this article is about.

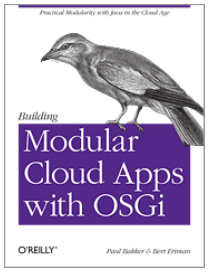
Related Research

Useful Information Found in Books

Unfortunately, according to the literature relating OSGi, transitive dependency solving is a topic barely focused and without a optimal solution to this date. One of the closest reference found until now concerning transitive dependencies is a half-page chapter from the book Building Modular Cloud Apps with OSGi, by Paul Bakker, Bert Ertman.

We quote the mentioned chapter's content here for accessibility purposes only:

Transitive Dependencies (page 79, September 2013: First Edition)



Some libraries and frameworks have a ridiculously long list of transitive dependencies.

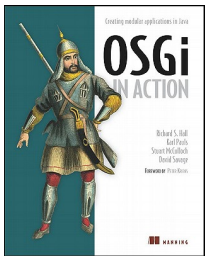
In a non-OSGi project, this can be considered a huge architectural risk. Maven, for example, will pull in transitive dependencies automatically, leaving you in classpath hell, with possibly multiple versions of the same dependencies if you're unlucky. In OSGi, this problem seems to be even more difficult to deal with. We have to add those dependencies to the container, and all the dependencies must be OSGi bundles. This can be a painful and long process to go through.

Luckily there is a better alternative in OSGi. If you need a library that is this messy, it can be better to package the library and dependencies within the bundle that needs them. This way we don't end up with all kind of dodgy exports, and we don't have to deal with the fact that dependencies might not be bundles. In OSGi, a bundle can contain other JAR files, and classes within those JAR files can be used by the bundle as if they were in the bundle itself. This way you isolate the problem of the messy library within a single bundle, making sure that you're overall architecture isn't hurt by this. Of course it's preferable to use libraries that take modularity into account correctly, but at least this is a way to work with less perfect dependencies without losing your own structure.

As you can conclude, this is far from being a solution to the problem since it breaks modularity.

Another interesting reference to point out is the book OSGi in Action: Creating Modular Applications in Java by: Richard S. Hall, Karl Pauls, Stuart McCulloch, David Savage. Where in chapter 6.2.1 we can read:

6.2.1 Making a mega bundle (page 206, ISBN 9781933988917)



A mega bundle comprises a complete application along with its dependencies. Anything the application needs on top of the standard JDKis embedded inside this bundle and made available to the application by extending the Bundle-ClassPath (2.5.3). This is similar to how Java Enterprise applications are constructed. In fact, you can take an existing web application archive (WAR file) and easily turn it into a bundle by adding an identity along with Bundle-ClassPath entries for the various classes and libraries contained within it, as shown in figure 6.6.

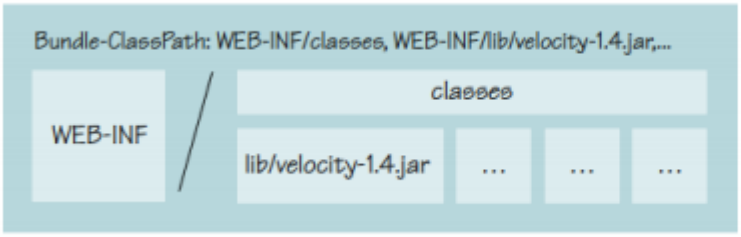


Figure 6.6 Turning a WAR file into a bundle

The key benefit of a mega bundle is that it drastically reduces the number of packages you need to import, sometimes down to no packages at all. The only packages you may need to import are non-java. packages from the JDK (such as javax.* packages) or any packages provided by the container itself. Even then, you can choose to access them via OSGi boot delegation by setting the org.osgi.framework.bootdelegation framework property to the list of packages you want to inherit from the container class path. Boot delegation can also avoid certain legacy problems (see section 8.2 for the gory details).*
The downside is that it reduces modularity, because you can't override boot-delegated packages in OSGi. A mega bundle with boot delegation enabled is close to the classic Java application model; the only difference is that each application has its own class loader instead of sharing the single JDK application class loader.

Although, we find these *mega-bundle* approaches uncomfortable because they go against the modular design that OSGi tries to preserve throught its nature. By creating a mega-bundle we rise insane difficulty barriers to the task of maintaining each embedded depedency individually. If one of the embedded dependencies gets updated by its third-party development team, we are forced to update the whole mega-bundle and to release a new version of it. Moreover, if we need to maintain an older version of that particular embedded dependency for other bundles to use, we are forced to add extra complexity to the MANIFEST.MF file of the mega-bundle or to separate the dependency from the mega-bundle. So, in our perspective, mega-bundles are just a "quick fix" that should be only used in very well controlled environments at 25ºC in a sunny day with no more than 15% humidity.
Our opinion is somewhat backed-up by the book Modular Java: Creating Flexible Applications with OSGi and Spring, by Craig Walls.

Joe Asks. . . (page 74, ISBN-13: 978-1934356-40-1)

How Do I Decide Whether to Embed a JAR or to Wrap It?

This is a very good question. The answer comes down to a choice of simplicity vs. fine-grained control.

When you wrap a JAR file, you effectively turn it into an OSGi bundle. This means that you can install, start, stop, update, and uninstall it in the OSGi framework just like any other bundle. You need to install it only once for all depending bundles to be able to use it.

Embedded JARs, on the other hand, can be managed only within the scope of the bundle into which they are embedded. This means that you can't upgrade to a newer version of an embedded JAR file without rebuilding the hosting bundle. Also, if more than one bundle depends on a library, then that library's JAR file must be embedded within each bundle that needs it.

With that said, embedding JARs within bundles follows a familiar deployment model that is similar to web application WAR files that have JAR files embedded within them.

As a rule of thumb, if a library is needed by only one bundle and if you will only ever manage that library within the scope of the depending bundle, then you should probably embed it. But if the library is needed by several bundles and/or you want to manage that library independent of other bundles in the OSGi framework, then it may make more sense to wrap the library and deploy it as a full-fledged bundle.

Related Online Threads and Topics

While researching, several threads where opened online, some which without any concrete answer to the opening post's problem. We list these topics bellow:

- Reddit: Guys, I need help with managing OSGi dependencies
- Reddit: OBR START command not found?
- StackOverflow: Managing OSGi Dependency Hell
- StackOverflow: OBR START command not found?
- Creating a New OSGi Project with Maven-Bundle-Plugin

Other Useful Resources

- Bnd Official Website
- BndTools for Eclipse Official Website
- Bundle Plugin for Maven (Maven-Bundle-Plugin)
- Bnd tool for converting jars into bundles OSGi
- Creating OSGi bundles
- Using BND to Create OSGi Bundles
- Using the OSGi Bundle Repository in OSGi and Apache Felix 3.0
- Apache Felix OSGi Bundle Repository (OBR)
- The FAB Deployment Model
- Recommendable Maven repository search engines

Solving Transitive Dependencies

Using BndTools to Wrap Third-Party Jars into OSGi Bundles

BndTools is a plugin for Eclipse that is completely based on bnd. Bnd is used for creating and working with OSGi bundles. Its primary goal is to ease the development of bundles. With OSGi you are forced to provide additional metadata in the JAR's manifest to verify the consistency of your "class path". This metadata must be closely aligned with the class files in the bundle and the policies that a company has about versioning. Maintaining this metdata is an error prone chore because many aspects are redundant.

Wrapping with BND Utility

Wrapping a bundle with BND utility is dead simple. However it cannot solve transitive dependencies, but we include this information here since it is very common to come acoss third-party libraries that are not bundles and must be converted to bundles in order to belong to an OBR repository.

First download the BND utility. You can download the one attached to this article, since it is faster than looking for another place to get it.

So to wrap a given jar to a bundle you just need to run the following command:

```
java -jar bnd-2.1.0.jar wrap other_folder/xxx.jar
```

Where xxx.jar is the bundle you want to wrap.

Note: The Jar you're trying to wrap must not be in the same folder where you're executing the bnd.jar, be-

cause the resulting jar file will be outputed with the same name as the original jar, returning an error if both are in the same folder.

Although it is considered a bad practice, you may want to create a unique jar with all dependencies embedded inside. One way of doing this is in your Maven project execute the command **mvn clean compile assembly:single**, grab the built jar file from the **target** folder and use the BND utility to make a bundle out of it as described above in this chapter.

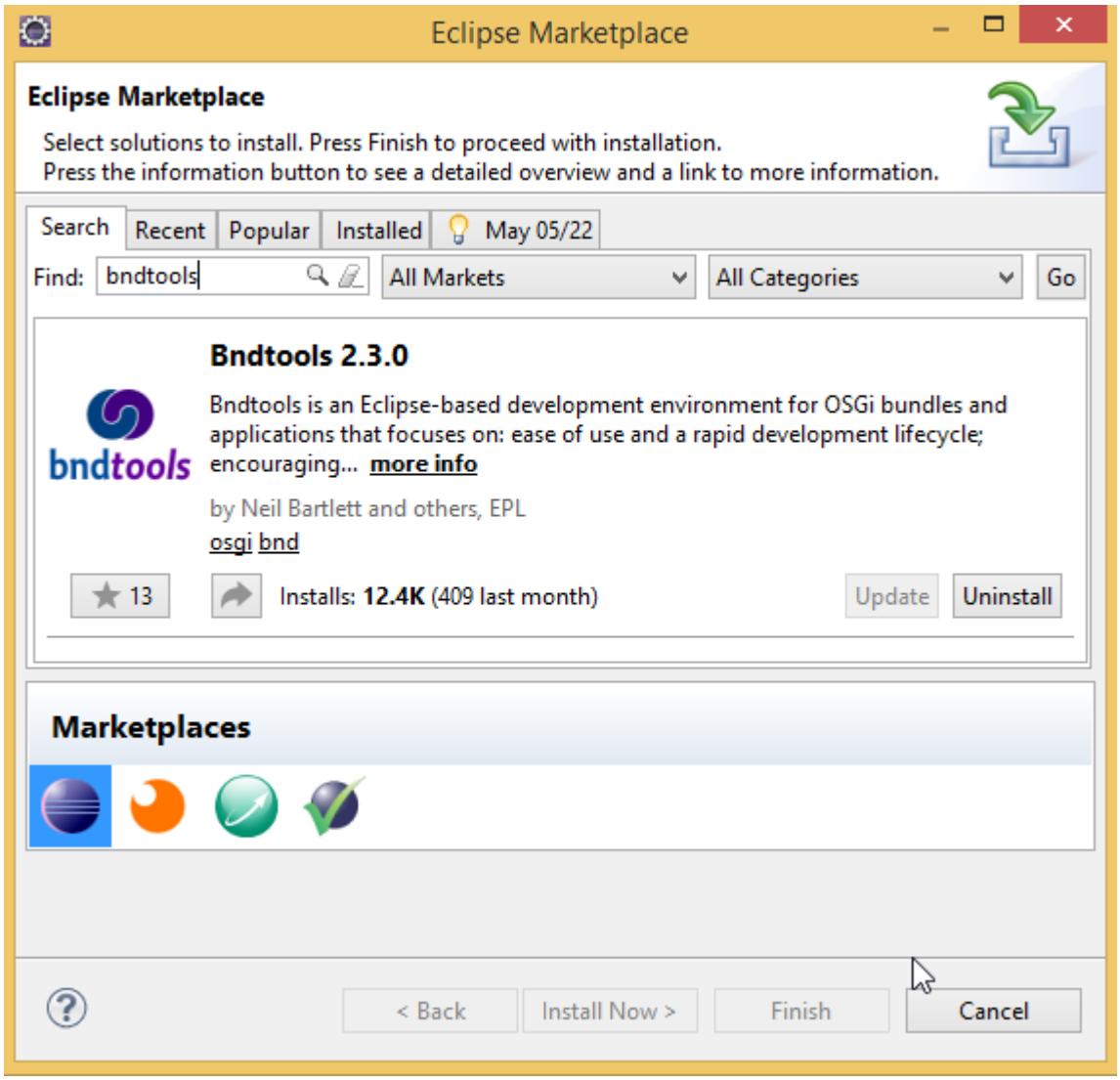
Remember, you need to configure the maven-assembly-plugin in your POM.XML.

```
<plugin>
        <artifactId>maven-assembly-plugin</artifactId>
        <configuration>
          <archive>
            <manifest>
              <mainClass>foo.bar.MyMainClass</mainClass>
            </manifest>
          </archive>
          <descriptorRefs>
            <descriptorRef>jar-with-dependencies</descriptorRef>
          </descriptorRefs>
        </configuration>
</plugin>
```

This same result can also be achieved in a simpler way with Maven-Bundle-Plugin also described in this article.

Wrapping with BndTools in Eclipse

To install BndTools in eclipse just search for it in the Eclipse Market Place as shown in the following figure:



In previous versions of BndTools we used the option **File → New → Other..., Select Bndtools → Wrap JAR as OSGi Bundle Project** to wrap a bundle (as stated in Using BND to Create OSGi Bundles), but now such option got removed in newer versions. Unfortunately, as with many other projects, documentation doesn't get updated at the same rate as the software it documents, so we couldn't find the correct way to do this in the most recent versions of BndTools until this date.

Using Maven to Fetch and Embed Some Transitive Dependencies

Raw Capture of Maven Third-Party Dependencies Into a Folder

In this section we'll see how we can make maven give us all dependencies of one project as jar files. This is useful to let us choose manually which jars we desire to wrap into bundles. Some of these third-party libraries may already be OSGi bundles, depending on their development teams decisions in distributing their libraries as bundles.

To achieve this you must add the following plugin into your (parent) POM.XML file (inside <plugins></plugins> tags, which are inside the <build></build> tags).

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-dependency-plugin</artifactId>
    <executions>
        <execution>
            <id>copy-dependencies</id>
            <phase>package</phase>
            <goals>
                <goal>copy-dependencies</goal>
            </goals>
            <configuration>
                <outputDirectory>${project.build.directory}/thridparty-libs</out
                <overWriteIfNewer>true</overWriteIfNewer>
                <includeScope>runtime</includeScope>
                <excludeGroupIds>${project.groupId}</excludeGroupIds>
                <excludeArtifactIds>...</excludeArtifactIds>
            </configuration>
        </execution>
    </executions>
</plugin>
```

To execute this goal you must run **mvn org.apache.maven.plugins:maven-dependency-plugin:copy-dependencies** or in Eclipse go to **Run As -> Maven Build...** and in the **Goals** text input box type **org.apache.maven.plugins:maven-dependency-plugin:copy-dependencies.**

Note: If you're working with a parent POM, execute this command while selecting that POM.XML in Eclipse, or if you're using a command line, execute this command in the same folder as your parent POM.XML file is located. This way the execution will be applied to every module belonging that parent POM.XML. Otherwise it would only be applied to one particular module.

The dependency jars can then be found inside each module/bundle's **target/thridparty-libs/** folder.

Massive Wrapping of Transitive Dependencies Individually Using a Maven Profile

One powerful way to fetch, wrap and deploy transitive dependencies is using a maven profile based on the one described here: Creating OSGi bundles of your Maven dependencies, which uses the powerful maven-bundle-plugin.

This profile acts as a tool that will iterate over all your project's <dependencies> and download them from the maven repository configured in the POM.XML (or SETTINGS.XML in .m2 folder) and from Maven Central Repository if needed, wrap each one individually using the BND utility and store them in the bundles/ folder for each project (or module of your parent POM if you're using one).

To add this profile to your project you must insert the following declaration into your project's POM.XML or parent POM if you're working with several modules:

```
<profile>
        <id>create-osgi-bundles-from-dependencies</id>
        <build>
            <!-- Wrapped bundles will be inserted in this directory. This may po
                to another directory such as ".../felix-framework/bundles" to de
                into felix automatically. -->
            <directory>${basedir}/bundles</directory>
            <plugins>
                <plugin>
                    <groupId>org.apache.felix</groupId>
                    <artifactId>maven-bundle-plugin</artifactId>
                    <version>2.0.1</version>
                    <extensions>true</extensions>
                    <executions>
                        <execution>
                            <id>wrap-my-dependency</id>
                            <goals>
                                <goal>wrap</goal>
                            </goals>
                            <configuration>
                                <wrapImportPackage>;</wrapImportPackage>
                            </configuration>
                        </execution>
                    </executions>
                </plugin>
            </plugins>
        </build>
    </profile>
</profiles>
```

To execute this profile you must run **mvn -Pcreate-osgi-bundles-from-dependencies bundle:wrap** or in Eclipse go to **Run As -> Maven Build...** and in the **Goals** text input box type **-Pcreate-osgi-bundles-from-dependencies bundle:wrap.**

Note: If you're working with a parent POM, execute this command while selecting that POM.XML in Eclipse, or if you're using a command line, execute this command in the same folder as your parent POM.XML file is located. This way the execution will be applied to every module belonging that parent POM.XML. Otherwise it would only be applied to one particular module.

Embedding Transitive Dependencies Into a Mega-Bundle With Maven-Bundle-Plugin

It is recommended to take a look at the tutorial Creating a New OSGi Project with Maven-Bundle-Plugin before reading this chapter, if this is your first time using the maven-bundle-plugin.

A very easy way to embed all of your transitive dependencies into one mega-bundle is to use the following tag **<Embed-Dependency>*;scope=compile;inline=true</Embed-Dependency>** that will force maven-bundle-plugin to embed all packages that fit into the * wildcard (i.e. all packages) into your generated bundle file. Example:

```
<plugin>
    <groupId>org.apache.felix</groupId>
    <artifactId>maven-bundle-plugin</artifactId>
    <extensions>true</extensions>
    <configuration>
        <instructions>
            <Bundle-SymbolicName>${project.artifactId};singleton:=true</Bundle-S
            <Bundle-Version>${project.version}</Bundle-Version>
            <Export-Package>my.project.exported.packages.*</Export-Package>
            <Embed-Dependency>foo.bar.*;scope=compile;inline=true</Embed-Depende
        </instructions>
    </configuration>
</plugin>
```

Remember that this is a very bad practice! However this will create a standalone bundle that will require no external dependencies (i.e. the MANIFEST.MF Import-Package field should be empty), unless, while packaging the jar, maven couldn't find all the required dependencies. If that's the case, then Import-Package should be present in the MANIFEST.MF with the missing dependencies that should be present at runtime.

Now imagine that you only want to embed just some packages, you can modify the **Embed-Dependency** tag to the following:

<Embed-Dependency>scope=compile;inline=true;artifactId=commons-*</Embed-Dependency> where

commons-* is referring to all artifact ids where the name is commons-<something>

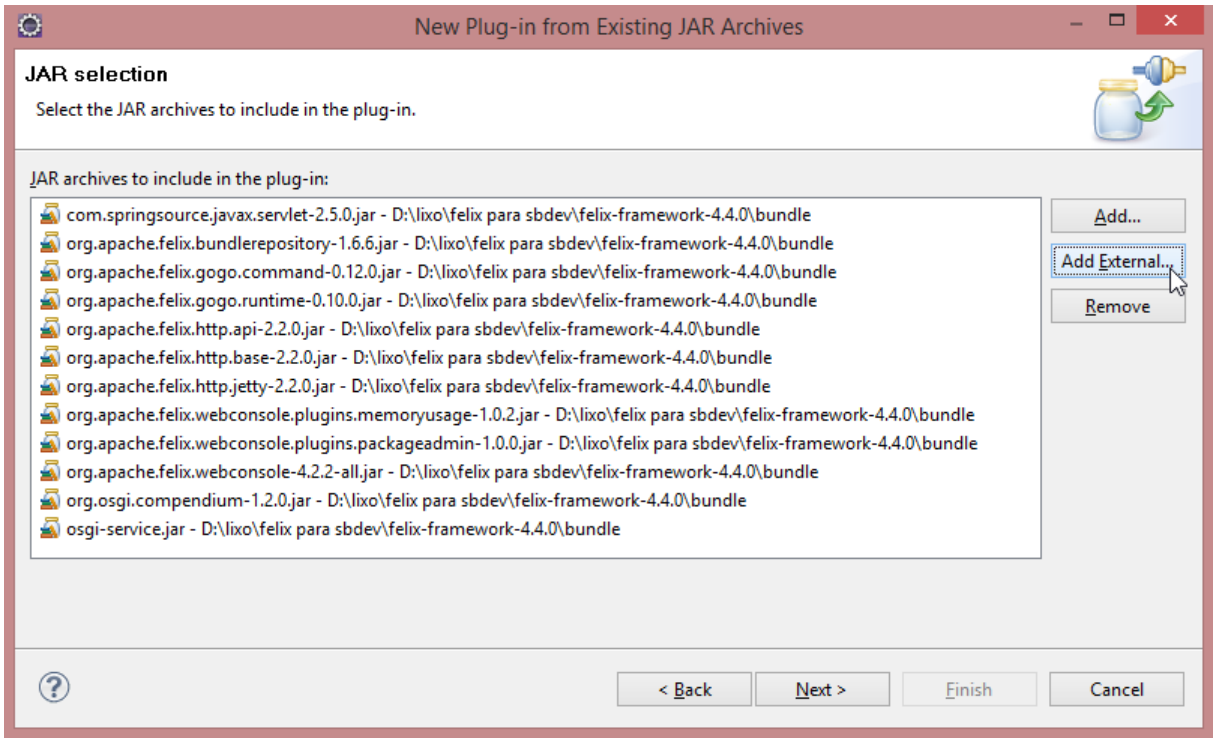
This way only certain packages (the ones fitting the wildcard evaluation) are embedded into the final jar bundle file.

If you don't use the **Embed-Dependency** tag then all dependencies are added to the MANIFEST.MF's Import-Package field and nothing is embedded into the bundle's jar.

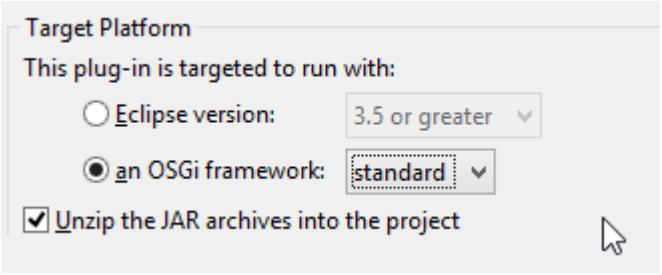
Read more here: http://felix.apache.org/site/apache-felix-maven-bundle-plugin-bnd.html

Embedding Transitive Dependencies Into a Mega-Bundle With Eclipse Plug-in Wizard

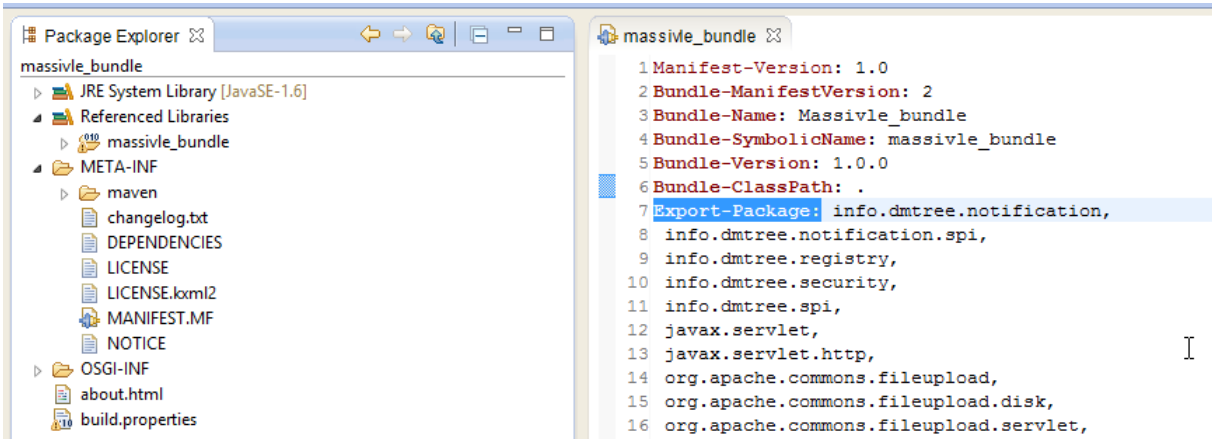
The same result can be achieved with Eclipse simply by selecting **New > Other...**, search for **Plug-in from Existing JAR Archives**. Select the jars you wish to embed by pressing the **Add External...** button.



Press **Next** and fill your **Project name** and select the target platform as in the following image and **Finish**.



You should see a project without any src folder, but containing all imported jars' packages exported in its MANIFEST.MF file.



Now you need to deploy this new (massive) bundle. For that select **Export > Deployable plugins and fragments**, select the project to export in the new window that shows up, select the destination folder and hit **Finish**.

You're done! Now you can deploy your bundle into Felix or other OSGi Framework as any other regular bundle.

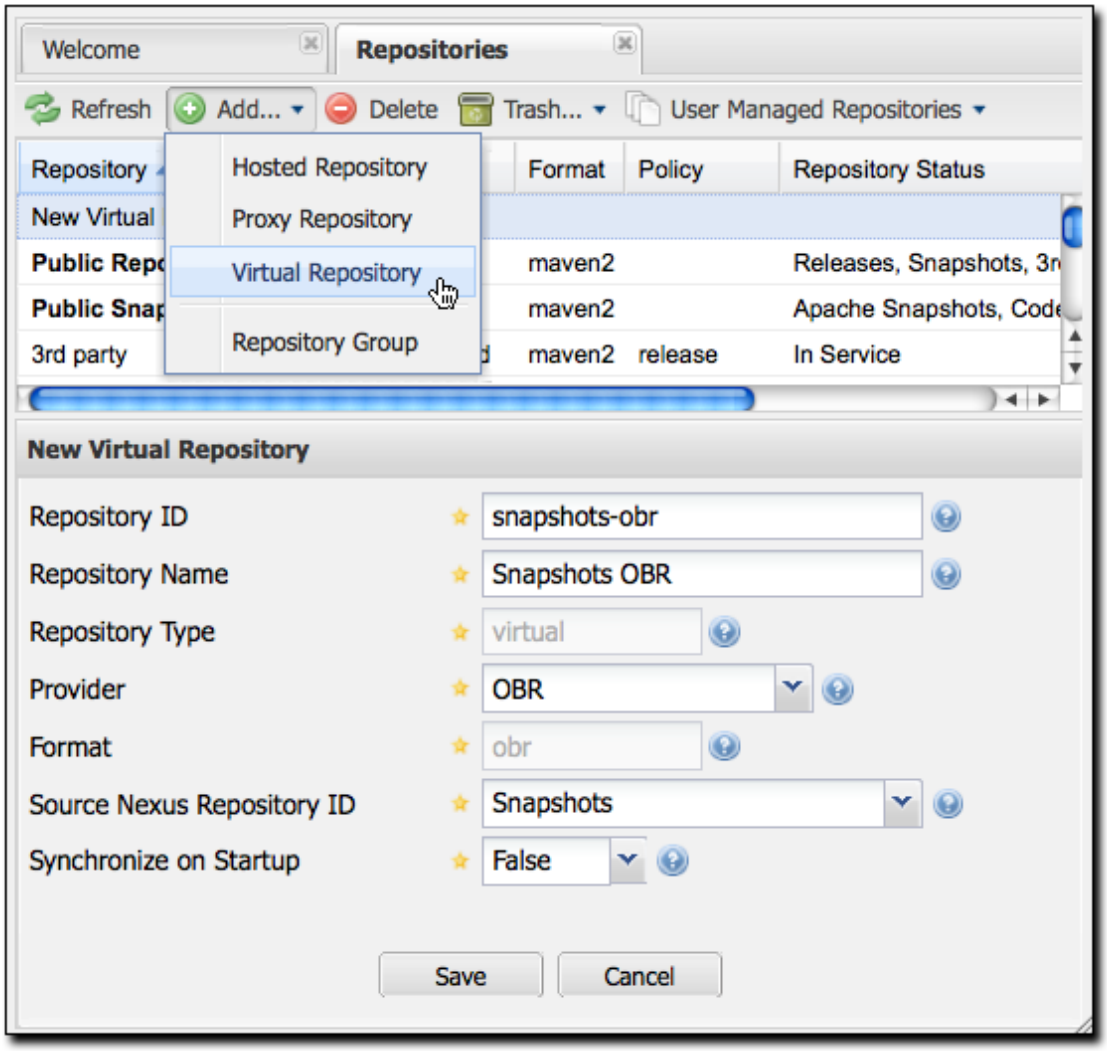
Using OBR Repositories and Apache Felix OSGi Framework

Setting Up an OBR Repository in Nexus

In this section we'll see how to create a virtual OBR repository which will scan a Maven repository for OSGi bundle artifacts and generate the OBR XML metadata in response to a change in the Maven repository.

1. Load the Nexus interface in a web browser by opening the URL http://sb-buildmaster.tagus.ist.utl.pt:8082/nexus/index.html#welcome
2. Login

3. Click on Repositories in the left navigation menu.
4. Click on the Add.. button above the list of Nexus repositories and groups.
5. Select “Virtual Repository” from the resulting dropdown.
6. In the New Virtual Repository window, supply the following values as shown in the figure bellow:



7. Repository ID: snapshots-obr (id example)
8. Repository Name: Snapshots OBR (name example)
9. Provider: OBR
10. Source Nexus Repository we want to attach to this OBR virtual repository, (for example the snapshots maven repository)
11. Synchronize on Startup: False
12. Click the Save button to create the new Virtual repository.

Add an OBR Repository to an Apache Felix OSGi Framework Session

To add an OBR repository to an Apache Felix Framework session you simple need to add the repository XML

descriptor's URL to the Felix's *conf/config.properties* file.

The repository XML descriptor's URL is obtained by appending **/.meta/obr.xml** to the repository URL given by Nexus, for example: http://sb-buildmaster.tagus.ist.utl.pt:8082/nexus/content/shadows/OSGi-Snapshots/.meta/obr.xml

Known Nexus Bug: This URL should be provided automatically by Nexus, but it contains a bug that prevents us from listing the hidden .meta folder in the Browse Storage tab.

You can add several OBR repositories to the configuration file. Here's an example of a configuration file, that also resets the Framework's cache each time the Framework is executed (look for the **obr.repository.url** property):

```
1  # Licensed to the Apache Software Foundation (ASF) under one
2  # or more contributor license agreements.  See the NOTICE file
3  # distributed with this work for additional information
4  # regarding copyright ownership.  The ASF licenses this file
5  # to you under the Apache License, Version 2.0 (the
6  # "License"); you may not use this file except in compliance
7  # with the License.  You may obtain a copy of the License at
8  #
9  #   http://www.apache.org/licenses/LICENSE-2.0
10 #
11 # Unless required by applicable law or agreed to in writing,
12 # software distributed under the License is distributed on an
13 # "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY
14 # KIND, either express or implied.  See the License for the
15 # specific language governing permissions and limitations
16 # under the License.
17
18 #
19 # Framework config properties.
20 #
21
22 # To override the packages the framework exports by default from the
23 # class path, set this variable.
24 #org.osgi.framework.system.packages=
25
26 # To append packages to the default set of exported system packages,
27 # set this value.
28 #org.osgi.framework.system.packages.extra=
29
30 # The following property makes specified packages from the class path
31 # available to all bundles. You should avoid using this property.
32 #org.osgi.framework.bootdelegation=sun.*,com.sun.*
33
34 # Felix tries to guess when to implicitly boot delegate in certain
35 # situations to ease integration without outside code. This feature
36 # is enabled by default, uncomment the following line to disable it.
37 #felix.bootdelegation.implicit=false
38
39 # The following property explicitly specifies the location of the bundle
40 # cache, which defaults to "felix-cache" in the current working directory.
```

```
41 # If this value is not absolute, then the felix.cache.rootdir controls
42 # how the absolute location is calculated. (See next property)
43 #org.osgi.framework.storage=${felix.cache.rootdir}/felix-cache
44
45 # The following property is used to convert a relative bundle cache
46 # location into an absolute one by specifying the root to prepend to
47 # the relative cache path. The default for this property is the
48 # current working directory.
49 #felix.cache.rootdir=${user.dir}
50
51 # The following property controls whether the bundle cache is flushed
52 # the first time the framework is initialized. Possible values are
53 # "none" and "onFirstInit"; the default is "none".
54 org.osgi.framework.storage.clean=onFirstInit
55
56 # The following property determines which actions are performed when
57 # processing the auto-deploy directory. It is a comma-delimited list of
58 # the following values: 'install', 'start', 'update', and 'uninstall'.
59 # An undefined or blank value is equivalent to disabling auto-deploy
60 # processing.
61 #felix.auto.deploy.action=install,start
62 felix.auto.deploy.action=install,start
63
64 # The following property specifies the directory to use as the bundle
65 # auto-deploy directory; the default is 'bundle' in the working directory.
66 #felix.auto.deploy.dir=bundle
67
68 # The following property is a space-delimited list of bundle URLs
69 # to install when the framework starts. The ending numerical component
70 # is the target start level. Any number of these properties may be
71 # specified for different start levels.
72 #felix.auto.install.1=
73
74 # The following property is a space-delimited list of bundle URLs
75 # to install and start when the framework starts. The ending numerical
76 # component is the target start level. Any number of these properties
77 # may be specified for different start levels.
78 #felix.auto.start.1=
79
80 felix.log.level=1
81
82 # Sets the initial start level of the framework upon startup.
83 #org.osgi.framework.startlevel.beginning=1
84
85 # Sets the start level of newly installed bundles.
86 #felix.startlevel.bundle=1
87
88 # Felix installs a stream and content handler factories by default,
89 # uncomment the following line to not install them.
90 #felix.service.urlhandlers=false
91
92 # The launcher registers a shutdown hook to cleanly stop the framework
93 # by default, uncomment the following line to disable it.
94 #felix.shutdown.hook=false
95
96 #
97 # Bundle config properties.
98 #
```

```
99
100 org.osgi.service.http.port=8080
101 obr.repository.url=http://felix.apache.org/obr/releases.xml http://sb-buildmast
```

In order to tell Felix Framework to use the linked OBR you need to download the following bundles from here: Apache Felix – Downloads

- Shell – http://mirrors.gigenet.com/apache//felix/org.apache.felix.shell-1.4.3.jar
- Shell Text UI – http://mirrors.gigenet.com/apache//felix/org.apache.felix.shell.tui-1.4.1.jar

And insert them into the **bundles/ directory** inside Felix Framework. Also you must remove the **bundle org.apache.felix.gogo.shell-x.x.x** from the Felix's bundle/ directory, because you're installing a new shell and you don't want the gogo.shell that comes by default with Felix to mess things up, since both read from STDIN.

Now, if you start the Felix Framework (java -jar bin/felix.jar) and type **ps** in the console that shows up you should see the following bundles listed:

```
1  -> ps
2  START LEVEL 1
3     ID   State         Level  Name
4  [   0] [Active     ] [    0] System Bundle (4.4.0)
5  [   1] [Active     ] [    1] Apache Felix Bundle Repository (1.6.6)
6  [   2] [Active     ] [    1] Apache Felix Gogo Command (0.12.0)
7  [   3] [Active     ] [    1] Apache Felix Gogo Runtime (0.10.0)
8  [   4] [Active     ] [    1] Apache Felix Shell Service (1.4.3)
9  [   5] [Active     ] [    1] Apache Felix Shell TUI (1.4.1)
10 [   6] [Active     ] [    1] OSGi R4 Compendium Bundle (4.1.0)
11 [   7] [Active     ] [    1] OSGi R4 Core Bundle (4.1)
12 [   8] [Active     ] [    1] OSGi OBR Service API (1)
13 ->
```

Note: Bundles 6, 7 and 8 are optional and they just provide OSGi APIs to create additional functionality.

We can list all available commands by typing **help** or **obr help** for obr related commands.

Let's tell Felix to show us the available bundles in the registered OBR repositories we inserted in the configuration files by typing **obr list**.

```
1  -> obr list
2  Apache Felix Bundle Repository (1.6.6, ...)
3  Apache Felix Configuration Admin Service (1.2.4, ...)
4  Apache Felix Declarative Services (1.6.0, ...)
5  Apache Felix Dependency Manager (3.0.0)
```

```
 6  Apache Felix EventAdmin (1.0.0)
 7  Apache Felix File Install (3.0.2, ...)
 8  Apache Felix Gogo Command (0.10.0, ...)
 9  Apache Felix Gogo Runtime (0.10.0, ...)
10  Apache Felix Gogo Shell (0.10.0, ...)
11  Apache Felix Gogo Shell Commands (0.2.0)
12  Apache Felix Gogo Shell Console (0.2.0)
13  Apache Felix Gogo Shell Launcher (0.2.0)
14  Apache Felix Http Api (2.0.4)
15  Apache Felix Http Base (2.0.4)
16  Apache Felix Http Bridge (2.0.4)
17  Apache Felix Http Bundle (2.0.4)
18  Apache Felix Http Jetty (2.0.4)
19  Apache Felix Http Proxy (2.0.4)
20  Apache Felix Http Samples - Filter (2.0.4)
21  Apache Felix Http Samples - Whiteboard (2.0.4)
22  Apache Felix HTTP Service Jetty (1.0.1, ...)
23  Apache Felix Http Whiteboard (2.0.4)
24  Apache Felix iPOJO (1.8.0, ...)
25  Apache Felix iPOJO (0.8.0)
26  Apache Felix iPOJO API (1.6.0, ...)
27  Apache Felix iPOJO Arch Command (1.6.0, ...)
28  Apache Felix iPOJO Composite (1.8.0, ...)
29  Apache Felix iPOJO Composite (1.0.0, ...)
30  Apache Felix iPOJO Event Admin Handler (1.8.0, ...)
31  Apache Felix iPOJO Extender Pattern Handler (1.4.0, ...)
32  Apache Felix iPOJO Extender Pattern Handler (1.0.0, ...)
33  Apache Felix iPOJO Gogo Command (1.0.1, ...)
34  Apache Felix iPOJO JMX Handler (1.4.0, ...)
35  Apache Felix iPOJO Temporal Service Dependency Handler (1.6.0, ...)
36  Apache Felix iPOJO URL Handler (1.6.0, ...)
37  Apache Felix iPOJO WebConsole Plugins (1.6.0, ...)
38  Apache Felix iPOJO White Board Pattern Handler (1.2.0, ...)
39  Apache Felix iPOJO White Board Pattern Handler (1.6.0, ...)
40  Apache Felix Log Service (1.0.0)
41  Apache Felix Metatype Service (1.0.2, ...)
42  Apache Felix Preferences Service (1.0.6, ...)
43  Apache Felix Remote Shell (1.0.4, ...)
44  Apache Felix Remote Shell (1.1.2, ...)
45  Apache Felix Shell Service (1.4.2, ...)
46  Apache Felix Shell TUI (1.4.1, ...)
47  Apache Felix UPnP Base Driver (0.8.0)
48  Apache Felix UPnP Extra (0.4.0)
49  Apache Felix UPnP Tester (0.4.0)
50  Apache Felix Web Console Event Plugin (1.0.2)
51  Apache Felix Web Console Memory Usage Plugin (1.0.0)
52  Apache Felix Web Console Memory Usage Plugin (1.0.2)
53  Apache Felix Web Console Service Diagnostics Plugin (0.1.4.SNAPSHOT, ...)
54  Apache Felix Web Console UPnP Plugin (1.0.0)
55  Apache Felix Web Management Console (3.1.2, ...)
56  Apache Felix Web Management Console (3.1.2, ...)
57  Apache Log4j (1.2.17)
58  bayeux-0.0.1 (0.0.0)
59  bayeux-0.0.2 (0.0.0)
60  codebase (1.0.1.SNAPSHOT)
61  eu.smartcampus.api.deviceconnectivity (0.0.1.SNAPSHOT)
62  eu.smartcampus.api.deviceconnectivity.adapters.protocolintegration (0.0.1.SNAPSH
63  eu.smartcampus.api.deviceconnectivity.impls.knxip (0.0.1.SNAPSHOT)
```

```
64 eu.smartcampus.api.deviceconnectivity.impls.lifx (0.0.1.SNAPSHOT)
65 eu.smartcampus.api.deviceconnectivity.impls.meterip (0.0.1.SNAPSHOT)
66 eu.smartcampus.api.deviceconnectivity.impls.modbus (0.0.1.SNAPSHOT)
67 eu.smartcampus.api.deviceconnectivity.wrappers.pubsub (0.0.1.SNAPSHOT)
68 eu.smartcampus.api.deviceconnectivity.wrappers.rest (0.0.1.SNAPSHOT)
69 eu.smartcampus.api.historydatastorage (0.0.1.SNAPSHOT)
70 eu.smartcampus.api.osgi (0.0.1.SNAPSHOT)
71 FormattedText Plugin (1.0.0.HEAD)
72 jlifx (0.0.0)
73 JUnit (4.8.1)
74 lumina.api (0.0.1.SNAPSHOT)
75 lumina.api.abstractions (0.0.1.SNAPSHOT)
76 lumina.app.keygen (0.0.1.SNAPSHOT)
77 lumina.db (0.0.1.SNAPSHOT)
78 lumina.extensions.drivers.knx (0.0.1.SNAPSHOT)
79 lumina.extensions.drivers.rs232jSSC (0.0.1.SNAPSHOT)
80 lumina.kernel (0.0.1.SNAPSHOT)
81 lumina.license (0.0.1.SNAPSHOT)
82 modbus4J (0.0.0)
83 OSGi OBR Service API (1.0.0)
84 OSGi R4 Compendium Bundle (4.0.0)
85 Rxtx (2.1.7)
86 seroUtils (0.0.0)
87 Servlet 2.1 API (1.0.0)
88 shared.extensions.base.properties (1.0.1.SNAPSHOT)
89 shared.extensions.base.properties (2.1.0.SNAPSHOT, ...)
90 shared.extensions.base.usermanager (1.0.1.SNAPSHOT)
91 shared.extensions.base.usermanager (1.0.1.SNAPSHOT)
92 shared.properties.api (2.1.0.SNAPSHOT, ...)
93 shared.thirdparty.rxtx (1.0.1.SNAPSHOT)
94 shared.usermanager.api (1.0.1.SNAPSHOT)
95 shared.usermanager.api (1.0.1.SNAPSHOT)
96 shared.usermanager.osgi (1.0.1.SNAPSHOT)
97 shared.usermanager.wrappers.rest (1.0.1.SNAPSHOT)
98 shared.usermanager.wrappers.rest (1.0.1.SNAPSHOT)
99 ->
```

We can now ask Felix to start a bundle directly from the OBR repository, **also downloading its dependencies automatically**! For example if we type **obr start "lumina.extensions.drivers.rs232jSSC"** Felix will download that bundle and its transitive dependencies, resolve them all and start. If some of the dependencies is missing from the repositories it will complain about that. You can use **bnd.jar wrap** command to wrap a bundle and upload it to the attached maven repository in nexus, so that the missing bundles can be listed and used by Felix.

After starting lumina.extensions.drivers.rs232jSSC bundle and if all went correctly you should see the following active bundles:

```
1 -> obr start "lumina.extensions.drivers.rs232jSSC"
2 Target resource(s):
3 -------------------
4    lumina.extensions.drivers.rs232jSSC (0.0.1.SNAPSHOT)
5 Required resource(s):
```

```
6  ---------------------
7     shared.extensions.base.properties (2.0.0.SNAPSHOT)
8     JUnit (4.8.1)
9     lumina.api (0.0.1.SNAPSHOT)
10    lumina.kernel (0.0.1.SNAPSHOT)
11    codebase (1.0.1.SNAPSHOT)
12    lumina.api.abstractions (0.0.1.SNAPSHOT)
13 Deploying...done.
14 -> ps
15 START LEVEL 1
16    ID   State         Level  Name
17 [   0] [Active     ] [    0] System Bundle (4.4.0)
18 [   1] [Active     ] [    1] Apache Felix Bundle Repository (1.6.6)
19 [   2] [Active     ] [    1] Apache Felix Gogo Command (0.12.0)
20 [   3] [Active     ] [    1] Apache Felix Gogo Runtime (0.10.0)
21 [   4] [Active     ] [    1] Apache Felix Shell Service (1.4.3)
22 [   5] [Active     ] [    1] Apache Felix Shell TUI (1.4.1)
23 [   6] [Active     ] [    1] OSGi R4 Compendium Bundle (4.1.0)
24 [   7] [Active     ] [    1] OSGi R4 Core Bundle (4.1)
25 [   8] [Active     ] [    1] OSGi OBR Service API (1)
26 [   9] [Active     ] [    1] JUnit (4.8.1)
27 [  10] [Active     ] [    1] shared.extensions.base.properties (2.0.0.SNAPSHOT)
28 [  11] [Active     ] [    1] lumina.api (0.0.1.SNAPSHOT)
29 [  12] [Active     ] [    1] codebase (1.0.1.SNAPSHOT)
30 [  13] [Active     ] [    1] lumina.kernel (0.0.1.SNAPSHOT)
31 [  14] [Active     ] [    1] lumina.api.abstractions (0.0.1.SNAPSHOT)
32 [  15] [Active     ] [    1] lumina.extensions.drivers.rs232jSSC (0.0.1.SNAPSHOT
33 ->
```

Note: You can also specify a bundle's version while starting, for example **obr start "lumina.extensions.drivers.rs232jSSC";0.0.1.SNAPSHOT**.

Here you can find a Felix Framework already containing the required bundles to support the OBR functionality: felix-framework-4.4.0 obr configured.zip

Conclusion

Now you're aware of the several ways we have at our hands to manage transitive dependencies and which tools can be used to battle against the dependency hell. However each technique has its own advantages and disadvantages, so use them wisely.

Currently there's no favorite way of dealing with transitive dependencies in our development team, but the most used techniques are **Wrapping with BND Utility** to wrap jars into bundles and upload them into one of Nexus Maven/OBR repositories, **Add an OBR Repository to an Apache Felix OSGi Framework Session** to solve transitive dependencies at runtime and **Massive Wrapping of Transitive Dependencies Individually Using a Maven Profile** as an alternative when we cannot use OBR repositories.

EDIT: You should check the Part 2 of this tutorial for novelty methods of solving the OSGi transitive dependency hell (link).

- Pedro D.

No comments yet

Add a comment as Pedro D.