Apache Felix Bundle Plugin FAQ

Apache Felix Bundle Plugin Frequently Asked Questions

When I embed a dependency why do I see duplicated content?

Having two copies of classes, both unpacked and embedded as jars, is a sign that your Embed-Dependency and Export-Package instructions are overlapping. Export-Package tells BND to pull in classes found in the named packages from the build classpath and add them to the bundle, Embed-Dependency tells BND to embed (or inline if enabled) whole artifacts in the bundle.

so say I have:

```
Export-Package: org.objectweb.asm.*
```

and I have the asm artifact as a compile scope dependency, then I would see the org.objectweb.asm classes unpacked in the bundle, ie. pulled in by BND.

say I now decide to embed asm as a jar, for example with:

```
Embed-Dependency: *;scope=compile|runtime
```

I would now see the asm artifact embedded inside bundle - but I would also see the unpacked classes from before, because I'm still asking BND to pull them in (you will probably also see split package warnings during the build).

ok - so how do I embed asm as a jar, but mark its packages as exported without pulling in the unpacked classes... well, there is another export instruction added for exactly this reason:

```
-exportcontents: org.objectweb.asm.*
```

(this is <_exportcontents> in the POM because a tag can't start with '-')

this instruction is merged with Export-Package, after BND has decided on the content of the bundle - that is, it works just like Export-Package except that it won't alter the content of the bundle.

So by removing org.objectweb.asm.* from Export-Package and using the -exportcontents instruction instead along with Embed-Dependency, I can now embed the asm artifact in my bundle and export its packages:

```
Embed-Dependency: *;scope=compile|runtime
-exportcontents: org.objectweb.asm.*
```

Why do I see more classes in my bundle after upgrading to maven-bundle-plugin 2.0.0?

Before 2.0.0 the maven-bundle-plugin only passed local classes and compile scope dependencies to bnd. This was because the main BundlePlugin mojo used "@requiresDependencyResolution runtime" which asks Maven to only resolve dependencies of compile or runtime scope (the maven-bundle-plugin also explicitly filtered runtime scope dependencies from the classpath passed to bnd). Because bnd only had a fraction of the original classpath used to compile the bundle classes it meant that imported packages weren't augmented with additional information (such as versions and attributes) available from the complete classpath.

In 2.0.0 a conscious decision was made to pass the complete classpath to bnd so it would have the complete set of information used during compilation. To do this the @requiresDependencyResolution setting was changed to "test" so all dependencies will now be resolved. Furthermore only test scope artifacts are now filtered from the final classpath passed to bnd.

For most users passing more of the original compilation classpath to bnd simply means you get more accurate and consistent results. However, if you happened to rely on the old behaviour (perhaps by setting Private-Package: * to pull in all local and compile scope classes) then you will see more classes added to your bundle as the wildcard matches against additional content in the classpath.

There are two solutions when this happens:

- Change your Private-Package / Export-Package instructions to more accurately describe what you want in your bundle
- Add the following setting to remove provided and runtime scope dependencies from the classpath passed to bnd:

```
<plugin>
  <groupId>org.apache.felix</groupId>
  <artifactId>maven-bundle-plugin</artifactId>
  <configuration>
    <excludeDependencies>*;scope=provided|runtime</excludeDependencies>
  </configuration>
</plugin>
```

This second option effectively switches the maven-bundle-plugin back to the old 1.X behaviour.

Please also note that since 2.0.0 the maven-bundle-plugin also sets default Export-Package and Private-Package instructions based on your local source files, so you might well find that simply removing any Private-Package and/or Export-Package instructions from your pom.xml will actually produce the correct result.

Why do I get an exception (Unsupported major.minor version 49.0) when I build with a 1.4 or earlier JDK?

The latest maven-bundle-plugin (2.0.0) uses a new release of bnd which requires Java5. This means you now have to build your bundle projects using a Java5 (or later) JDK. Note that you can still compile classes for earlier releases of the JVM by setting the appropriate source and target levels in your pom.xml:

```
<plugin>
  <artifactId>maven-compiler-plugin</artifactId>
  <configuration>
    <source>1.4</source>
    <target>1.4</target>
  </configuration>
</plugin>
```

When I build a bundle, some classes are built in "target/classes" but they're not included in the final jar.

The only classes that will appear in the bundle are the ones you ask it to include using Export-Package, Private-Package, Include-Resource, and Embed-Dependency - so just because a file exists under target/classes does NOT mean it will end up in the bundle. This is because this is the way the underlying BND tool works (it is primarily pull-based).

Now the bundleplugin will look at your Maven project and add appropriate BND instructions to pull in resource files - and version 2.0.0 will also look at your source directory to set reasonable defaults for Export-Package and Private-Package (unless you set these yourself). So when using bundleplugin 2.0.0 I'd use the default Private-Package and Export-Package to begin with - I would then move towards using an explicit list of packages in Export-Package to add versioning, directives, etc.

The only time I would set Private-Package would be to have more control over what ends up in the bundle - either to remove certain packages or perhaps pull in additional packages not found by the source scanner. Also note that both Export-Package and Private-Package accept wildcards such as "org.example.*" which can reduce the number of entries in the list, but you should be careful not to set either the export or private instruction to "*" as this would pull in the entire classpath... dependencies and all!

How do I remove the generated Maven information from the resulting bundle JAR file?

Use the following archive setting:

```
<configuration>
  <archive>
    <addMavenDescriptor>false</addMavenDescriptor>
  </archive>
</configuration>
```

Put this in either the JAR or bundle plugin configuration.

Why do some generated resources (such as Declarative Services XML files) appear in the final jar, but others don't?

When you use the Service-Component instruction to specify Declarative Services the BND tool scans the project classpath for components and automatically adds its generated XML to the final bundle, therefore Include-Resource is not necessary. But if you generate files under OSGI-INF using another mechanism then they won't end up in the bundle unless you add that directory using Include-Resource (this goes back to the core design decision that BND pulls classes and resources into the bundle, rather than just taking everything under target/classes). We try to provide reasonable defaults on the Maven side in the bundleplugin so local classes and resources will end up in the bundle without additional configuration, but we do this by looking at the effective pom and src/ folder rather than the generated target/classes content.