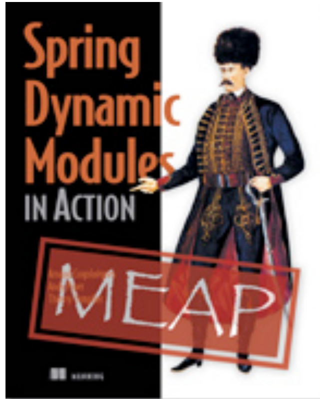
Demystifying OSGi bundles

Excerpted from



Spring Dynamic Modules in Action
EARLY ACCESS EDITION
Arnaud Cogoluegnes, Thierry Templier, and Andy Piper
MEAP Release: June 2009
Softbound print: Summer 2010 (est.) | 450 pages
ISBN: 9781935182306
Manning Publications

This article is taken from the book ***Spring Dynamic Modules in Action.***

OSGi – the dynamic modular platform for Java – uses a slightly different unit of deployment than the plain old JAR, the bundle. Bundles are standard JARs, but with additional metadata in the META-INF/MANIFEST.MF file. This metadata describes amongst others a bundle identity but also the Java packages it consumes or makes available to other bundles. This article covers some theory about this metadata and shows how to create an OSGi bundle from an existing library.

How to use OSGi metadata in the manifest file

To become a OSGi-compliant bundle, a JAR must include some headers in its manifest file. For brevity, we'll qualify the process of transforming a non-OSGi JAR file into a 100% OSGi-compliant bundle with the barbarism "OSGi-ification". What is the most common issue in the OSGi-ification of an existing library? Pretty simple: visibility. Taking the library's point of view, visibility of external dependencies but also giving visibility of its own classes to other bundles if necessary. In OSGi terms, this means we'll have to juggle with the `Import-Package` and `Export-Package` manifest headers.

When a library is built upon other libraries, it uses their classes and obviously imports some of their packages in its own classes. In a standard Java environment, you just need to add these libraries on the classpath, and any class can import their packages and use their classes. The story is different in an OSGi environment: the base libraries must explicitly *export* their packages and modules that want to use them must explicitly *import* them. The whole export/import process is managed by the OSGi platform, thanks to metadata contained in the manifest file of bundles.

IMPORTING PACKAGES

Let's talk first about the process of importing: a library needs to use some classes defined in another library (we assume that this library properly exports these classes, making them visible to other bundles). We can take as an example the ORM module from the Spring Framework: this module includes some support for popular object/relational mapping tools such as Hibernate, iBatis and OpenJPA. If we focus on Hibernate, the `Import-Package` of the ORM module might look like the following:

```
Import-Package: org.hibernate,org.hibernate.cache,org.hibernate.cfg
(...)
```

Hibernate has a lot of packages, and Spring ORM uses most of them, so we did not include the whole list for brevity. The previous snippet is fine regarding package visibility but is not precise enough regarding *version*. In its

2.5.6.A version, the Hibernate support of Spring ORM is only tested against Hibernate 3.2, so this should appear in the manifest. The `Import-Package` header can use the `version` attribute to specify the exact version or version range the bundle needs and defaults to the range [0.0.0, ∞). As we did not use the `version` attribute in our first manifest declaration, the ORM module would use any available version installed in the OSGi container, making a 3.0 Hibernate bundle eligible for use, whereas the ORM module is not compatible with Hibernate 3.0. So when OSGi-ifying a library, a good practice consists of indicating the version of each package in the `Import-Package` header. Spring ORM declares that it works with Hibernate from version 3.2.6.ga, inclusive, to 4.0.0, exclusive:

```
Import-Package: org.hibernate;version="[3.2.6.ga,
4.0.0)",org.hibernate.cache;version="[3.2.6.ga, 4.0.0)",
org.hibernate.cfg;version="[3.2.6.ga, 4.0.0)"
(...)
```

NOTE

The "ga" version qualifier stands for "General Availability" and denotes a stable, production-ready version of software.

Spring ORM not only includes support for Hibernate, but also for iBatis, amongst others. So the Spring ORM bundle can apply the same pattern for declaring dependencies on iBatis:

```
 Import-Package: org.hibernate;version="[3.2.6.ga,
4.0.0)",org.hibernate.cache;version="[3.2.6.ga, 4.0.0)",
org.hibernate.cfg;version="[3.2.6.ga, 4.0.0)",
(...)
com.ibatis.common.util;version="[2.3.0.677, 3.0.0)",
com.ibatis.common.xml;version="[2.3.0.677, 3.0.0)",
com.ibatis.sqlmap.client;version="[2.3.0.677, 3.0.0)"
(...)
```

Nice, but let's imagine you're working on an application that uses Hibernate and the support provided by Spring ORM. You provision your OSGi container with the corresponding bundles but you'll soon notice that if you want the Spring ORM bundle to be resolved, you need all of its dependencies in your container, like iBatis or OpenJPA, even if you only use Hibernate. That's a real pain as you'll have to get all these dependencies as OSGi bundles and deal also with their own dependencies. And you'll have to go through all of this for nothing because you don't even use them! Don't panic, there's a solution: these kinds of dependencies can be marked as optional in the manifest, thanks to the `resolution` directive. This directive defaults to `mandatory`, meaning that the bundle won't be able to resolve successfully if the imported package is not present in the container. The `resolution` directive can also take the `optional` value, to indicate that the importing bundle can successfully resolve even if the imported package is not present. Of course, if some code relying on the missing import is called at runtime, it will fail. Spring ORM declares its dependencies on ORM tools as optional, as there is little chance that all these libraries will be used at the same time in an application:

```
Import-Package: org.hibernate;version="[3.2.6.ga, 4.0.0)";resolution:=optional,
org.hibernate.cache;version="[3.2.6.ga,
4.0.0)";resolution:=optional,org.hibernate.cfg;version="[3.2.6.ga,
4.0.0)";resolution:=optional,
(...)
com.ibatis.common.util;version="[2.3.0.677, 3.0.0)";resolution:=optional,
com.ibatis.common.xml;version="[2.3.0.677, 3.0.0)";resolution:=optional,
com.ibatis.sqlmap.client;version="[2.3.0.677, 3.0.0)";resolution:=optional
(...)
```

So when OSGi-ifying libraries or frameworks you should remember the following guidelines with respect to the `Import-Package` header:

- Import the packages the library or framework uses, and pay attention to not import unused packages, which would tie the bundle to unnecessary dependencies.
- Specify the version of the packages, otherwise the library or framework can potentially use classes that it is not meant to use, and code could then break at runtime, or, worse, you could experience unexpected

For Source Code, Free Chapters, the Author Forum and more information about this title go to
http://www.manning.com/cogoluegnes/

behavior.

- Specify the difference between mandatory and optional dependencies by using the `resolution` directive. In some cases, leaving a dependency as mandatory does not make sense and can make your OSGi-ified library much more difficult to use, without any additional benefit.

Enough about what a library can import from other bundles; let's see now how a library can make its classes visible in the OSGi platform.

EXPORTING PACKAGES

The packages that need to be exported by a library really depend on its design. Some libraries clearly make the distinction between their API and their implementation classes, through some kind of special structuring of their packages. For example, interfaces (the API) are located in one package and internal classes (implementation, utilities) are located in an `impl` or `internal` sub package.

NOTE

Splitting API and implementation packages is a good design practice, not only in OSGi.

Nevertheless, usually, the export declarations will end up exporting all the packages of a bundle, as, even if we usually follow the "programming through interface" pattern, in the end, we will always need an implementation that is usually provided by the same library as the API.

One guideline you should remember from this article about the `Export-Package` header is the following: specify the version of the exported package. The following snippet shows the first line of the `Export-Package` header from the Spring ORM module manifest (notice the use of the `version` attribute):

```
Export-Package: org.springframework.orm;version="2.5.6.A",
(...)
```

`version` can be different for each exported package, but usually all the exported packages will share the same version, which is (usually) the same as the owning bundle (there are some exceptions, but we are covering most of the cases with that assumption).

`Import-Package` and `Export-Package` are the most important headers to specify when OSGi-ifying libraries, but we'll now also take a look at some other headers, especially those used to identify a bundle.

GIVING AN IDENTITY TO A BUNDLE

In an OSGi environment, a library must be properly identified, as dependency resolution in OSGi builds on bundle identity mechanisms. There are many manifest headers related to identity; we won't describe all of them but we'll focus on three here.

The following snippet (part of Spring Core 2.5.6.A bundle manifest) shows these three manifest headers:

```
Bundle-SymbolicName: org.springframework.core
Bundle-Version: 2.5.6.A
Bundle-Name: Spring Core
```

The `Bundle-SymbolicName` header specifies a unique name for a bundle, usually based on the reverse package (or domain) convention. The header value cannot contain any whitespace (only alphanumeric characters, ".", "_" and "-"). Obviously, the `Bundle-SymbolicName` header is compulsory, does not take a default value and must be set very carefully, as it is the main component of your bundle identity.

The other aspect of a bundle identity is its version, set with the `Bundle-Version` header. Contrary to the `Bundle-SymbolicName`, the version header is not compulsory as it defaults to `0.0.0`, but *it should always be explicitly set*. When setting the bundle version, you should follow the format and semantics of OSGi versioning (major, minor and micro numbers, and qualifier), e.g. `3.2.0.ga`. The symbolic name and version tuple comprises the identity of your bundle: there cannot be two bundles with the same symbolic name and version number installed at the same time in an OSGi container.

The last header, `Bundle-Name`, is not meant to be used directly by the OSGi platform but rather by developers, as it defines a human-readable name for the bundle. Its value can contain spaces and does not have to be unique (even though it should, to avoid confusion), just explanatory enough.

You hold all the cards of the theory about important OSGi metadata. In the second part of this article, we'll cover how to turn an existing library, Apache Commons DBCP, into an OSGi-compliant bundle.

In the first part of this article, we learnt about the main OSGi metadata for a standard JAR to become a fully functional OSGi bundle. In this second part, we'll do some practice by turning the non-OSGi Apache Commons DBCP library. But let's start immediately with some generalities about the available approaches for OSGi-fication.

Conversion by hand

As the deployment unit in OSGi is the JAR file along with some extra metadata, the conversion boils down to a careful editing of the `MANIFEST.MF` file. We know everything about the various manifest headers and how to properly set the corresponding values, but we should not forget some specifics of the JAR packaging:

- The `META-INF/MANIFEST.MF` file must be the first entry in the JAR. So we should still rely on the standard packaging program (the `jar` command) to package our OSGi-powered JAR and not try to package it manually.
- The manifest format has some strict requirements. For instance, lines cannot be longer than 72 characters and the file should end with an empty line.

With these specifics, along with the very sensitive needs of OSGi metadata, the manual editing of an OSGi manifest can end up being a nightmare. Any typo or extra space can break the manifest and be very difficult to track down. Take a look at the manifest of each module of the Spring Framework and imagine the daunting task that it would be to maintain each manually. Imagine doing this for a bunch of Java EE frameworks, like Hibernate or JSF!

Manually editing manifests, without any support from tools, is not a realistic or desirable undertaking, and we'll discuss tools that can help you to reliably package your OSGi bundles.

Conversion using tools

You are a developer and you cannot deny that your life as a developer wouldn't be the same without some of the tools you rely on every day. You will also probably have strong opinions on this subject: developers should not become too dependent on their tools and should know exactly what these tools do for them under the covers.

Java and Java EE have by now a very large set of tools, both commercial and open source: IDE (for content assistance, debugging…), build tools, continuous integration servers… The good news is that OSGi tooling is getting better and better. We'll focus in this section on tools that can help you package Java libraries into OSGi-compliant JAR files, by adopting a progressive approach, as we'll end up including the OSGi-ification process into a Maven 2 build. As we know you love action, the OSGi-ification of Apache Commons DBCP, the database connection pool library, will be our candidate library.

The Bnd tool

Bnd (http://www.aqute.biz/Code/Bnd) is a tool created by Peter Kriens to help to analyze JAR files and to diagnose and create OSGi R4 bundles. It is used internally by the OSGi alliance for creating OSGi libraries for the various OSGi reference implementations and Technology Compatibility Kits (a.k.a. TCK). Bnd comprises a unique JAR file but can be used from the command line, as an Eclipse plugin or from Ant (yes, a JAR can be all of this!).

Are there any other tools than Bnd?

Bnd is arguably the most popular tool for packaging JARs as OSGi bundles, but OSGi tooling is getting more and more widespread. The latest rival for Bnd is Bundlor, a tool created by the SpringSource team to automate the creation of OSGi bundles. As with Bnd, Bundlor analyzes class files to detect dependencies, but it's also able to

parse different kinds of files to detect *more* dependencies: Spring application context XML files, JPA's `persistence.xml`, Hibernate mapping files and even property files! Bundlor follows a template-based approach, which consists of giving hints for the manifest generation in the guise of a property file (the same approach as used by Bnd). At the time of this writing, Bundlor is still quite new, but can already be used with Ant and Maven 2.

OSGi also gets into your development environment: there has always been the Plugin Development Environment (PDE) in Eclipse, which enables the development of Eclipse plugins, and thus offers some nice support for OSGi (ex.: a dedicated editor for manifest files). More recent is the SpringSource Tool Suite (STS), a dedicated Eclipse distribution, targeting the development of Spring- and SpringSource dm Server-based applications. As SpringSource dm Server applications heavily rely on OSGi, STS offers some support for OSGi. STS used to be a commercial product but is now free since mid 2009.

In this section, we'll use Bnd from the command line to OSGi-ify Commons DBCP 1.2.2. So let's get down to business! Download Bnd from its webpage, DBCP 1.2.2 from http://commons.apache.org/dbcp/ and copy the two JARs into a working directory.

Why Apache Commons DBCP?

Commons DBCP is a very popular database connection pool: Apache Tomcat uses it to provide its datasources and a lot of applications embed a DBCP connection pool (often as a Spring bean). Unfortunately, DBCP is not yet among the OSGi-ified libraries of the Apache Commons family. The OSGi-ification of Commons DBCP happens to be a very good exercise though!

You can't convert a plain JAR file into an OSGi-compliant bundle without knowing a little about it, that's why Bnd comes with the `print` command:

```
java -jar bnd-0.0.313.jar print commons-dbcp-1.2.2.jar
```

Don't be overwhelmed by the output, it is divided into sections; we are going to analyze the most important ones. The first one provides information taken from the manifest:

```
[MANIFEST commons-dbcp-1.2.2.jar]
Ant-Version                          Apache Ant 1.5.3
Build-Jdk                            1.4.2_10
Built-By                             psteitz
Created-By                           Apache Maven
Extension-Name                       commons-dbcp
Implementation-Title                 org.apache.commons.dbcp
Implementation-Vendor                The Apache Software Foundation
Implementation-Vendor-Id             org.apache
Implementation-Version               1.2.2
Manifest-Version                     1.0
Package                              org.apache.commons.dbcp
Specification-Title                  Commons Database Connection Pooling
Specification-Vendor                 The Apache Software Foundation
X-Compile-Source-JDK                 1.3
X-Compile-Target-JDK                 1.3
```

The more interesting section is the one starting with `[USES]`, which delivers information about the Java packages of the target JAR:

```
[USES]
org.apache.commons.dbcp                  java.sql
                                         javax.naming
                                         javax.naming.spi
                                         javax.sql
                                         org.apache.commons.jocl
                                         org.apache.commons.pool
                                         org.apache.commons.pool.impl
                                         org.xml.sax
org.apache.commons.dbcp.cpdsadapter      java.sql
                                         javax.naming
```

```
                                          javax.naming.spi
                                          javax.sql
                                          org.apache.commons.dbcp
                                          org.apache.commons.pool
                                          org.apache.commons.pool.impl
(...)
```

We are now aware of the packages that our library depends on. The output ends with an error section:

```
One error
1 : Unresolved references to [javax.naming, javax.naming.spi,
javax.sql, org.apache.commons.pool, org.apache.commons.pool.im
pl, org.xml.sax, org.xml.sax.helpers] by class(es) on the Bund
le-Classpath[Jar:commons-dbcp-1.2.2.jar]: [org/apache/commons/
dbcp/datasources/PerUserPoolDataSource.class,(...)
```

With this huge and monolithic block of text, Bnd tells us that, with respect to the current class path, some packages that our library needs to work are missing. We also notice that Commons DBCP depends on the `org.apache.commons.pool` and `org.apache.commons.pool.impl` packages. Indeed, Commons DBCP relies on the Commons Pool library to handle its pooling algorithm and adds a thin layer on the top of it for database connections.

This dependency implies two things for the OSGi-ification of Commons DBCP, we'll need to:

- Properly import packages from Commons Pool
- Have Commons Pool packaged as an OSGi bundle

We can immediately start the OSGi-ification with the `wrap` command of Bnd:

```
java -jar bnd-0.0.313.jar wrap commons-dbcp-1.2.2.jar
```

This creates a `commons-dbcp-1.2.2.bar` file in the same directory, with an OSGi-compliant manifest and all the defaults for OSGi manifest headers. Unfortunately, Bnd cannot guess the proper values for some very important headers and default values are not always appropriate. That's why Bnd uses a configuration file to supply this information: version, symbolic name, imports and exports can be defined in a way that is close to the manifest format but more editior-friendly and more powerful thanks to the use of variable substitutions and the use of pattern matching.

Where do the .class files come from?

Bnd is not a traditional packaging tool; it does not need a directory with `.class` files to compress them and create a JAR file. It directly locates `.class` files in the classpath and packages them into a JAR file. You can then potentially include into your OSGi bundle all the `.class` files from the classpath you specified when launching Bnd from the command line.

The following snippet shows the Bnd configuration file for converting Commons DBCP into an OSGi bundle:

```
version=1.2.2                                                           #1
Bundle-SymbolicName: org.apache.commons.dbcp                            #2
Bundle-Version: ${version}                                              #3
Bundle-Name: Commons DBCP
Bundle-Description: DBCP connection pool
Export-Package: org.apache.commons.dbcp*;version=${version}             #4
Import-Package:org.apache.commons.pool*;version=1.3.0, [CA]             #5
  org.apache.commons.dbcp*;version=${version},*;resolution:=optional    #5
```

Bnd allows variable substitution so we use this feature for the version at #1, as it is needed at several places in the template. We then specify the bundle symbolic name at #2 and the version at #3, using the `version` variable, with the `${variableName}` pattern. At #4, we specify which packages the bundle will export: notice we use a wildcard (*) to specify that we want to export the `org.apache.commons.dbcp` package and all its sub-packages. We use the `version` variable again to specify the version of the exported packages. At #5, we specify that the bundle imports version 1.3.0 of all the Commons Pool packages it references. Notice that we import the Commons

DBCP packages, with the same version, to ensure a consistent class space. The last wildcard refers to all the remaining packages used by Commons DBCP; and we mark them as optional.

NOTE

With Bnd, always define configuration from the most specific to the most general. If an element is matched twice, the first match always takes precedence. That's the reason the instruction to mark all the dependencies (*) as optional in our `Import-Package` header comes last.

Let's issue the `wrap` command with the `properties` option, specifying a Bnd configuration file:

```
java -jar bnd-0.0.313.jar wrap -properties commons-dbcp-1.2.2.bnd [CA]
  commons-dbcp-1.2.2.jar
```

We can now have a look at the manifest file of the generated OSGi bundle; here is an excerpt for the `Export-` and `Import-Package` headers:

```
(...)
Export-Package: org.apache.commons.dbcp.cpdsadapter;uses:="javax.namin
 g,javax.sql,org.apache.commons.pool.impl,org.apache.commons.pool,java
 x.naming.spi,org.apache.commons.dbcp";version="1.2.2",org.apache.comm
 ons.dbcp;uses:="org.apache.commons.pool.impl,org.apache.commons.pool,
 javax.sql,javax.naming,javax.naming.spi,org.xml.sax";version="1.2.2",
 org.apache.commons.dbcp.datasources;uses:="org.apache.commons.dbcp,ja
 vax.sql,org.apache.commons.pool,javax.naming,javax.naming.spi,org.apa
 che.commons.pool.impl";version="1.2.2"
(...)
Import-Package: javax.naming;resolution:=optional,javax.naming.spi;res
 olution:=optional,javax.sql;resolution:=optional,org.apache.commons.d
 bcp;version="1.2.2",org.apache.commons.dbcp.cpdsadapter;version="1.2.
 2",org.apache.commons.dbcp.datasources;version="1.2.2",org.apache.com
 mons.pool;version="1.3.0",org.apache.commons.pool.impl;version="1.3.0
 ",org.xml.sax;resolution:=optional,org.xml.sax.helpers;resolution:=op
 tional
```

Now you can compare the end result with the instructions we specified in the Bnd configuration file; you will now understand that Bnd is a very convenient tool! You now have an OSGi-compliant version of Commons DBCP. As Commons DBCP is not distributed as an OSGi bundle, a good practice is to include `osgi` in the filename: `commons-dbcp-osgi-1.2.2.jar`. If your bundle turns out to be distributed and used by third parties, you can also prefix it with your company name: `com.manning.commons-dbcp-osgi-1.2.2.jar`.

We mentioned that we also need an OSGi-compliant version of Commons Pool, as Commons DBCP is built on this library. Unfortunately, Commons Pool is also not distributed as an OSGi bundle, so we have to again do the conversion ourselves. It turns out to be fairly simple, as we can follow the same process as for Commons DBCP. The following snippet shows the Bnd configuration file for Commons Pool:

```
version=1.3.0
Bundle-SymbolicName=org.apache.commons.pool
Bundle-Version: ${version}
Export-Package: org.apache.commons.pool*;version=${version}
Bundle-Name: Commons Pool
```

Congratulations, you can now create database connection pools with Commons DBCP in an OSGi environment! But perhaps some of you are fond of automation, so we'll see now how to make the OSGi-ification part of a Maven build.

THE FELIX BUNDLE PLUGIN FOR MAVEN 2

We're going to reiterate the OSGi-ification of Commons DBCP, but in a 100% Maven 2 style this time. The Felix Bundle Plugin provides integration between Bnd and Maven 2: it uses Bnd under the covers, providing it with information from the POM file. By using this plugin, you can take advantage of all Maven 2's features (automation, dependency management, standard project structure…) and still package your project as OSGi-compliant bundles. The plugin has reasonable default behavior, making the configuration simple for simple needs.

For the OSGi-fication of Commons DBCP, we start by creating a simple `pom.xml` file:

```
<?xml version="1.0"?>
```

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.manning.sdmia</groupId>                              #A
  <artifactId>commons-dbcp.osgi</artifactId>                        #A
  <version>1.2.2-SNAPSHOT</version>                                 #A
  <packaging>bundle</packaging>                                     #B
  <name>commons-dbcp.osgi</name>                                    #C
  <description>                                                     #C
    OSGified version of Commons DBCP                                #C
  </description>                                                    #C

  <dependencies>
    <dependency>                                                    #D
      <groupId>commons-dbcp</groupId>                               #D
      <artifactId>commons-dbcp</artifactId>                         #D
      <version>1.2.2</version>                                      #D
      <scope>provided</scope>                                       #D
    </dependency>                                                   #D
  </dependencies>
</project>
```

#A Defines project identity
#B Sets bundle as packaging
#C Describes bundle
#D Adds Commons DBCP dependency

Notice we clearly state that the project is our own distribution of an OSGi bundle:

- The `groupId` refers to our company
- The `artifactId` is postfixed with `osgi`

Even if Bnd is wrapped in a Maven plugin, it still bases its search of classes on the class path, so we need to add Commons DBCP as a Maven dependency.

We need now to explicitly reference the Felix Bundle Plugin, otherwise the `bundle` packaging does not have any meaning for Maven 2. We do this inside the `build` tag (just before the `dependencies` tag), where we usually configure Maven 2 plugins. Listing 1 shows the configuration of the Felix Bundle Plugin for OSGi-fying Commons DBCP.

Listing 1 Felix Bundle Plugin configuration for the OSGi-fication of Commons DBCP

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.felix</groupId>                           #1
      <artifactId>maven-bundle-plugin</artifactId>                  #1
      <version>2.0.0</version>                                      #1
      <extensions>true</extensions>
      <configuration>                                               #2
        <instructions>                                              #3
          <Bundle-SymbolicName>                                     #4
            org.apache.commons.dbcp                                 #4
          </Bundle-SymbolicName>                                    #4
          <Export-Package>                                          #5
            org.apache.commons.dbcp*;version=${project.version}     #5
          </Export-Package>                                         #5
          <Import-Package>                                          #6
            org.apache.commons.pool*;version="1.3.0",               #6
            *;resolution:=optional                                  #6
          </Import-Package>                                         #6
          <Embed-Dependency>                                        #7
            *;scope=provided;type=!pom;inline=true                  #7
          </Embed-Dependency>                                       #7
        </instructions>
      </configuration>
    </plugin>
  </plugins>
```

```
</build>
```

We start by declaring the plugin at #1 (never omit the version of a plugin with Maven 2 if you don't want your build to break unpredictably). The configuration starts at #2, with a set of *instructions* (#3). We use the `Bundle-SymbolicName` tag to set the corresponding manifest header (#4). At #5, we define the Java packages the bundle will export, thanks to the Export-Package instruction. Notice we can use the exact same syntax as in Bnd files to include sub-packages. This time, we didn't define a variable for the version, as we can refer to the project version directly, with the `${project.version}` variable. We define imported packages the same way as in plain Bnd (#6). At #7, with the `Embed-Dependency` tag, we tell the plugin how to handle dependencies: include all dependencies with `provided` scope (but exclude dependencies of type POM) and copy them inline in the JAR. All set! Any Maven packaging goal (`install` or `package`) will generate a 100% OSGi-compliant bundle!

NOTE

The Commons Pool library can also be easily OSGi-ified with the Felix Bundle Plugin.

OSGi-fying a library and making the process part of a traditional build is fairly simple thanks to the Felix Bundle Plugin, you just need to be careful with the generated OSGi metadata and Bnd will handle the rest. We have now talked a lot about conversion but what about our own modules and applications? We will discuss this topic in the next section.

Packaging your own modules as OSGi bundles

If you understand how to OSGi-ify existing libraries, making your own Java applications and modules OSGi bundles should not be a problem for you. You can apply all the OSGi-ification techniques we have covered so far to your own modules. You can stick with Bnd, choosing the mechanism that suits you best:

- Command line: straight and simple, but difficult to automate.
- Eclipse Plugin: embedded in your development environment, but still difficult to automate.
- Ant task: included in your build, perfect if Ant is your tool of choice for all your builds.
- Maven 2 plugin: included in your build, fits perfectly with any Maven 2 based project.

Packaging existing libraries or your own modules as OSGi bundles should not cause you any trouble by now!