Apache Felix Maven Bundle Plugin (BND)

Contents

This plugin for Maven 2/3 is based on the BND tool from Peter Kriens. The way BND works is by treating your project as a big collection of classes (e.g., project code, dependencies, and the class path). The way you create a bundle with BND is to tell it the content of the bundle's JAR file as a subset of the available classes. This plugin wraps BND to make it work specifically with the Maven 2 project structure and to provide it with reasonable default behavior for Maven 2 projects.

INFO: If you have questions about the maven-bundle-plugin please read the FAQ first. If you still have questions you can ask them on the Felix user list.

*NOTE: test scoped dependencies are **not** included in the classpath seen by BND.*

Since the 1.4.0 release, this plugin also aims to automate OBR (OSGi Bundle Repository) management. It helps manage a local OBR for your local Maven repository, and also supports remote OBRs for bundle distribution. The plug-in automatically computes bundle capabilities and requirements, using a combination of Bindex and Maven metadata.

TIP

Full Maven Site Plugin documentation for the current release of the maven-bundle-plugin

TIP

A complete list of instructions and their format is available from the BND website

Simple Example

Rather than going straight to a detailed list of plugin features, we will first look at a simple example of how to use the plugin to give an immediate flavor. A detailed "how to" will follow.

Assume that we have a simple bundle project that has a pubic API package an several implementation packages, such as:

```
org.foo.myproject.api
org.foo.myproject.impl1
org.foo.myproject.impl2
...
```

If we also assume that we have a bundle activator in one of the implementation packages, then the `<plugins>` section of the POM file for this bundle project would look like this:

```
...
<plugins>
  <plugin>
    <groupId>org.apache.felix</groupId>
    <artifactId>maven-bundle-plugin</artifactId>
    <extensions>true</extensions>
    <configuration>
      <instructions>
        <Export-Package>org.foo.myproject.api</Export-Package>
        <Private-Package>org.foo.myproject.*</Private-Package>
        <Bundle-Activator>org.foo.myproject.impl1.Activator</Bundle-Activator>
```

```
      </instructions>
    </configuration>
  </plugin>
</plugins>
...
```

The `<Export-Package>` and `<Private-Package>` instructions tell the plugin about the contents of the resulting bundle JAR file. The `<Export-Package>` instruction tells the plugin which of the available packages to copy into the bundle *and* export, while the `<Private-Package>` instruction indicates which of the available packages to copy into the bundle *but not* export. If the two sets overlap, as they do in the case, then the export takes precedence. Since we did not specify any values for any other bundle manifest headers, they will assume default values which are described below. One specific behavior to highlight is that the plugin generates the `Import-Package` bundle manifest header based on the contents of the bundle, which means that you generally do not ever need to explicitly specify it yourself. That's it.

Features

The BND library underlying the plugin defines instructions to direct its behavior. For this Maven plugin, these instructions are issued in the plugin configuration section of the POM file, as was illustrated above. BND recognizes three types of instructions:

1. *Manifest headers* - Any instruction that starts with a capital letter will appear in the resulting bundle's manifest file; the value for the header will either be copied, augmented, or generated by BND depending on the instruction.
2. *Variables* - Any instruction starting with a lowercase letter is assumed to be a variable in the form of a name-value pair, such as `version=3.0`, that can be used for property substitution, but is not copied to the manifest.
3. *Directives* - Any instruction starting with a '-' character is considered to be a directive that informs BND to perform some special processing and is not copied to the manifest.

The remainder of this section covers the most important aspects of BND's instructions; for complete details refer to the BND documentation.

Instructions

`<Export-Package>`

The `<Export-Package>` instruction is a list of packages for the bundle to export. These packages are copied into the resulting bundle JAR file from the available classes (i.e., project classes, dependencies, and class path); thus, it is possible to include classes into your bundle that are not associated with

source files in your project. `<Export-Package>` can be specified with package patterns using the '*' wildcard. Also, it is possible to exclude packages using negation by starting the package pattern with '!'. Thus, non-negated patterns indicate which of the available packages to include in the bundle, whereas negated patterns indicate which should not be included in the bundle.

The list of package patterns is ordered and earlier patterns are applied before later patterns. For example, if you specify "`org.foo.,!org.foo.impl`" **the second pattern has no effect since all `org.foo` packages have already been selected by the first pattern. Instead, you should specify** "`!org.foo.impl,org.foo.`", which will export all `org.foo` packages except `org.foo.impl`.

Following standard OSGi R4 syntax, package patterns can include both directives and attributes, which will be copied appropriately into the generated Export-Package manifest header. Besides explicitly listing package version attributes, BND will also determine package versions by examining the source JAR file or from `packageinfo` files in the package directory.

`<Private-Package>`

The `<Private-Package>` instruction is similar in every way to the `<Export-Package>` instruction, except for the fact that these packages will *not* be exported by the bundle. If a package is selected by both the export and private package headers, then the export takes precedence.

`<Include-Resource>`

The `<Include-Resource>` instruction is a list of arbitrary resources that should be copied into the bundle JAR file. The specified resources are declared as clauses that can have the following forms:

```
clause ::== assignment | inline | simple
assignment ::== PATH '=' PATH
simple ::== PATH
inline ::== '@' PATH
```

For the `<Include-Resource>` instruction, actual file paths are relative to the `pom.xml`, while file copy destinations are relative to the root of the resulting bundle JAR file. In the case of `assignment` or `simple` forms, the `PATH` parameter can point to a file or directory. The `simple` form will place the resource in the bundle JAR with only the file name, i.e., without any path component. For example, including `src/main/resources/a/b.c` will result in a resource `b.c` in the root of the bundle JAR. If the `PATH` points to a directory, the entire directory hierarchy is copied into the resulting bundle JAR file relative to the specified directory. If a specific resource must be placed into a subdirectory of the bundle jar, then use the `assignment` form, where the first path is the the destination path (including file name if the resource is a file) and the second path is the resource to copy. The `inline` form requires a ZIP or JAR file, which will be completely expanded in the bundle JAR.

If a resource clause is specified inside of "{ …}" brackets, then variable substitution will be performed on the resource, where variables in the resources are denoted with "${ …}" syntax.

By default the bundle plugin converts the project's Maven resource directories into a single `<Include-Resource>` instruction. If you specify your own `<Include-Resource>` instruction, this will replace the generated one. To include the generated list of Maven resources in your own `<Include-Resource>` instruction just add `{maven-resources}` to the list and it will be expanded automatically.

`<Import-Package>`

The `<Import-Package>` instruction is a list of packages that are required by the bundle's contained packages. The default for this header is "**", resulting in importing all referred packages. This header rarely has to be explicitly specified. However, in certain cases when there is an unwanted import, such an import can be removed by using a negation package pattern. The package patterns work in the same way as for `<Export-Package>` , which means they are ordered. For example, if you wanted to import all packages except `org.foo.impl` you would specify " `!org.foo.impl,` "**

Default Behavior

To use this plugin, very little information is required by BND. As part of the Maven integration, the plugin tries to set reasonable defaults for various instructions. For example:

- `<Bundle-SymbolicName>` is computed using the shared Maven2OsgiConverter component, which uses the following algorithm: Get the symbolic name as groupId + "." + artifactId, with the following exceptions:
  - if artifact.getFile is not null and the jar contains a OSGi Manifest with Bundle-SymbolicName property then that value is returned
  - if groupId has only one section (no dots) and artifact.getFile is not null then the first package name with classes is returned. eg. commons-logging:commons-logging -> org.apache.commons.logging
  - if artifactId is equal to last section of groupId then groupId is returned. eg. org.apache.maven:maven -> org.apache.maven
  - if artifactId starts with last section of groupId that portion is removed. eg. org.apache.maven:maven-core -> org.apache.maven.core The computed symbolic name is also stored in the `$(maven-symbolicname)` property in case you want to add attributes or directives to it.
- `<Export-Package>` is now assumed to be the set of packages in your local Java sources, excluding the default package '.' and any packages containing 'impl' or 'internal'. *(before version 2 of the bundleplugin it was based on the symbolic name)*

- Since 2.2.0 you can also use `{local-packages}` inside `<Export-Package>` and it will be expanded to the set of local packages.
- `<Private-Package>` is now assumed to be the set of packages in your local Java sources (note that any packages in both `<Export-Package>` and `<Private-Package>` will be exported). *(before version 2 of the bundleplugin it was assumed to be empty by default)*
- `<Import-Package>` is assumed to be "`*`", which imports everything referred to by the bundle content, but not contained in the bundle. *Any exported packages are also imported by default, to ensure a consistent class space.*
- `<Include-Resource>` is generated from the project's Maven resources, typically "`src/main/resources/`", which will copy the specified project directory hierarchy into the resulting bundle JAR file, mirroring standard Maven behavior.
- `<Bundle-Version>` is assumed to be "`${pom.version}`" but is normalized to the OSGi version format of "`MAJOR.MINOR.MICRO.QUALIFIER`", for example "`2.1-SNAPSHOT`" would become "`2.1.0.SNAPSHOT`".
- `<Bundle-Name>` is assumed to be "`${pom.name}`".
- `<Bundle-Description>` is assumed to be "`${pom.description}`".
- `<Bundle-License>` is assumed to be "`${pom.licenses}`".
- `<Bundle-Vendor>` is assumed to be "`${pom.organization.name}`".
- `<Bundle-DocURL>` is assumed to be "`${pom.organization.url}`".

Since the plugin creates bundles for OSGi R4, it hard-codes `Bundle-ManifestVersion` to be '2'. Additionally, it generates imports for every export to ensure package substitutability, which is very important when working with collaborating services. It is possible to override any of these values (except `Bundle-ManifestVersion`) just by specifying the desired value in the plugin configuration section of the POM file.

Detailed "How To"

Get Maven2

The first step in the process of using the plugin is downloading and installing the latest version of the Maven2 runtime. The latest Maven2 release and instuctions for getting started with Maven2 can be found at the Maven website.

Using the Plugin

To use the maven-bundle-plugin, you first need to add the plugin and some appropriate plugin configuration to your bundle project's POM. Below is an example of a simple OSGi bundle POM for

Maven2:

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>my-osgi-bundles</groupId>
  <artifactId>examplebundle</artifactId>
  <packaging>bundle</packaging>    <!-- (1) -->
  <version>1.0</version>
  <name>Example Bundle</name>
  <dependencies>
    <dependency>
      <groupId>org.apache.felix</groupId>
      <artifactId>org.osgi.core</artifactId>
      <version>1.0.0</version>
    </dependency>
  </dependencies>
  <build>
    <plugins>
      <plugin>    <!-- (2) START -->
        <groupId>org.apache.felix</groupId>
        <artifactId>maven-bundle-plugin</artifactId>
        <extensions>true</extensions>
        <configuration>
          <instructions>
            <Export-Package>com.my.company.api</Export-Package>
            <Private-Package>com.my.company.*</Private-Package>
            <Bundle-Activator>com.my.company.Activator</Bundle-Activator>
          </instructions>
        </configuration>
      </plugin>    <!-- (2) END -->
    </plugins>
  </build>
</project>
```

There are two main things to note: (1) the `<packaging>` specifier must be "bundle" and (2) the plugin and configuration must be specified (the configuration section is where you will issue instructions to the plugin).

Real-World Example

Consider this more real-world example using Felix' Log Service implementation. The Log Service project is comprised of a single package: `org.apache.felix.log.impl`. It has a dependency on the core OSGi interfaces as well as a dependency on the compendium OSGi interfaces for the specific log service interfaces. The following is its POM file:

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.apache.felix</groupId>
  <artifactId>org.apache.felix.log</artifactId>
  <packaging>bundle</packaging>
  <name>Apache Felix Log Service</name>
  <version>0.8.0-SNAPSHOT</version>
  <description>
    This bundle provides an implementation of the OSGi R4 Log service.
  </description>
  <dependencies>
    <dependency>
      <groupId>${pom.groupId}</groupId>
      <artifactId>org.osgi.core</artifactId>
      <version>0.8.0-incubator</version>
    </dependency>
    <dependency>
      <groupId>${pom.groupId}</groupId>
      <artifactId>org.osgi.compendium</artifactId>
      <version>0.9.0-incubator-SNAPSHOT</version>
    </dependency>
  </dependencies>
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.felix</groupId>
        <artifactId>maven-bundle-plugin</artifactId>
        <extensions>true</extensions>
        <configuration>
          <instructions>
            <Export-Package>org.osgi.service.log</Export-Package>
            <Private-Package>org.apache.felix.log.impl</Private-Package>
            <Bundle-SymbolicName>${pom.artifactId}</Bundle-SymbolicName>
            <Bundle-Activator>${pom.artifactId}.impl.Activator</Bundle-Activator>
            <Export-
Service>org.osgi.service.log.LogService,org.osgi.service.log.LogReaderService</Expo
rt-Service>
          </instructions>
        </configuration>
      </plugin>
    </plugins>
  </build>
</project>
```

Notice that the `<Export-Package>` instruction specifies that the bundle exports the Log Service package, even though this package is not contained in the bundle project. By declaring this, the plugin

will copy the Log Service package into the resulting bundle JAR file. This is useful in this case because now the bundle can resolve without having to download the entire compendium bundle. The resulting manifest for the Log Service bundle looks like this (notice how the imports/exports automatically have version information associated with them, which was obtained from packageinfo files in the source packages):

```
Manifest-Version: 1
Bundle-License: http://www.apache.org/licenses/LICENSE-2.0.txt
Bundle-Activator: org.apache.felix.log.impl.Activator
Import-Package: org.osgi.framework;version=1.3, org.osgi.service.log;v
 ersion=1.3
Include-Resource: src/main/resources
Export-Package: org.osgi.service.log;uses:=org.osgi.framework;version=
 1.3
Bundle-Version: 0.8.0.SNAPSHOT
Bundle-Name: Apache Felix Log Service
Bundle-Description: This bundle provides an implementation of the OSGi
  R4 Log service.
Private-Package: org.apache.felix.log.impl
Bundle-ManifestVersion: 2
Export-Service: org.osgi.service.log.LogService,org.osgi.service.log.L
 ogReaderService
Bundle-SymbolicName: org.apache.felix.log
```

The resulting bundle JAR file has the following content (notice how the LICENSE and NOTICE files were automatically copied from the `src/main/resources/` directory of the project):

```
META-INF/MANIFEST.MF
LICENSE
META-INF/
META-INF/maven/
META-INF/maven/org.apache.felix/
META-INF/maven/org.apache.felix/org.apache.felix.log/
META-INF/maven/org.apache.felix/org.apache.felix.log/pom.properties
META-INF/maven/org.apache.felix/org.apache.felix.log/pom.xml
NOTICE
org/
org/apache/
org/apache/felix/
org/apache/felix/log/
org/apache/felix/log/impl/
org/apache/felix/log/impl/Activator.class
org/apache/felix/log/impl/Log.class
org/apache/felix/log/impl/LogEntryImpl.class
org/apache/felix/log/impl/LogException.class
```

```
org/apache/felix/log/impl/LogListenerThread.class
org/apache/felix/log/impl/LogNode.class
org/apache/felix/log/impl/LogNodeEnumeration.class
org/apache/felix/log/impl/LogReaderServiceFactory.class
org/apache/felix/log/impl/LogReaderServiceImpl.class
org/apache/felix/log/impl/LogServiceFactory.class
org/apache/felix/log/impl/LogServiceImpl.class
org/osgi/
org/osgi/service/
org/osgi/service/log/
org/osgi/service/log/LogEntry.class
org/osgi/service/log/LogListener.class
org/osgi/service/log/LogReaderService.class
org/osgi/service/log/LogService.class
org/osgi/service/log/package.html
org/osgi/service/log/packageinfo
```

Adding OSGi metadata to existing projects without changing the packaging type

If you want to keep your project packaging type (for example "jar") but would like to add OSGi metadata you can use the manifest goal to generate a bundle manifest. The maven-jar-plugin can then be used to add this manifest to the final artifact. For example:

```
<plugin>
  <artifactId>maven-jar-plugin</artifactId>
  <configuration>
    <archive>
      <manifestFile>${project.build.outputDirectory}/META-
INF/MANIFEST.MF</manifestFile>
    </archive>
  </configuration>
</plugin>
<plugin>
  <groupId>org.apache.felix</groupId>
  <artifactId>maven-bundle-plugin</artifactId>
  <executions>
    <execution>
      <id>bundle-manifest</id>
      <phase>process-classes</phase>
      <goals>
        <goal>manifest</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

If you want to use packaging types other than "jar" and "bundle" then you also need to enable support for them in the bundleplugin configuration, for example if you want to use the plugin with WAR files:

```
<plugin>
  <groupId>org.apache.felix</groupId>
  <artifactId>maven-bundle-plugin</artifactId>
  <executions>
    <execution>
      <id>bundle-manifest</id>
      <phase>process-classes</phase>
      <goals>
        <goal>manifest</goal>
      </goals>
    </execution>
  </executions>
  <configuration>
    <supportedProjectTypes>
      <supportedProjectType>jar</supportedProjectType>
      <supportedProjectType>bundle</supportedProjectType>
      <supportedProjectType>war</supportedProjectType>
    </supportedProjectTypes>
    <instructions>
      <!-- ...etc... -->
    </instructions>
  </configuration>
</plugin>
```

You'll also need to configure the other plugin to pick up and use the generated manifest, which is written to `${project.build.outputDirectory}/META-INF/MANIFEST.MF` by default (unless you choose a different `manifestLocation` in the maven-bundle-plugin configuration). Continuing with our WAR example:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-war-plugin</artifactId>
  <configuration>
    <archive>
      <manifestFile>${project.build.outputDirectory}/META-
INF/MANIFEST.MF</manifestFile>
    </archive>
  </configuration>
</plugin>
```

Building the Plugin

The plugin is hosted at the Apache Felix project. The following steps describe how to build and install the plugin into your local Maven2 repository.

Using the SVN client of your choice, checkout the maven-bundle-plugin project.

```
$ svn co http://svn.apache.org/repos/asf/felix/trunk/bundleplugin
```

Using Maven2, build and install the maven-bundle-plugin by issuing the following Maven2 command in the project directory that was created as a result of the previous step.

```
$ mvn install
```

Goals

The maven-bundle-plugin also provides additional functionality via some Maven goals. Command-line execution of a goal is performed as follows:

```
mvn org.apache.felix:maven-bundle-plugin:GOAL
```

Where GOAL is one of the following:

- `*bundle*` - build an OSGi bundle jar for the current project configuration options:
  - `*manifestLocation*` defaults to ${project.build.outputDirectory}/META-INF
  - `*unpackBundle*` unpack bundle contents to output directory, defaults to false
  - `*excludeDependencies*` comma-separated list of dependency artifactIds to exclude from the classpath passed to Bnd, use "true" to exclude everything. Version 2 of the bundleplugin now supports the same style of filter clauses in `excludeDependencies` as `Embed-Dependency`.
  - `*classifier*` attach bundle to the project using the given classifier
  - `*supportedProjectTypes*` defaults to "jar","bundle"
- `*bundleall*` - build OSGi bundle jars for all transitive dependencies configuration options:
  - `*wrapImportPackage*` defaults to "*"
  - `*supportedProjectTypes*` defaults to "jar","bundle"
- `*wrap*` - as above, but limited to the first level of dependencies configuration options:
  - `*wrapImportPackage*` defaults to "*"
  - `*supportedProjectTypes*` defaults to "jar","bundle"
- `*manifest*` - create an OSGi manifest for the current project configuration options:

    - `*manifestLocation*` defaults to ${project.build.outputDirectory}/META-INF
    - `*supportedProjectTypes*` defaults to "jar","bundle"
- `*install*` - adds the current bundle project to the local OBR configuration options:
    - `*obrRepository*` path to local OBR, defaults to **<local-maven-repository>** `/repository.xml`
    - `*supportedProjectTypes*` defaults to "jar","bundle"

More GOALs are available in the *1.4.0* release:

- `*ant*` - create an Ant build script to rebuild the bundle
- `*install-file*` - adds a local bundle file to the local OBR configuration options:
    - `*obrRepository*` path to local OBR, defaults to **<local-maven-repository>** `/repository.xml`
    - `*groupId*` Maven groupId for the bundle, taken from *pomFile* if given
    - `*artifactId*` Maven artifactId for the bundle, taken from *pomFile* if given
    - `*version*` Maven version for the bundle, taken from *pomFile* if given
    - `*packaging*` Maven packaging type for the bundle, taken from *pomFile* if given
    - `*classifier*` Maven classifier type, defaults to none
    - `*pomFile*` optional Pom file describing the bundle
    - `*file*` bundle file, defaults to the bundle from the local Maven repository
    - `*obrXml*` optional additional properties for the bundle
- `*deploy*` - adds the current bundle project to a remote OBR configuration options:
    - `*remoteOBR*` name of remote OBR, defaults to NONE (which means no remote OBR deployment)
    - `*obrRepository*` used when the remoteOBR name is blank, defaults to `repository.xml`
    - `*prefixUrl*` optional public URL prefix for the remote repository
    - `*bundleUrl*` optional public URL where the bundle has been deployed
    - `*altDeploymentRepository*` alternative remote repository, *id::layout::url*
    - `*obrDeploymentRepository*` optional OBR specific deployment repository.
    - `*ignoreLock*` ignore remote locking when updating the OBR
    - `*supportedProjectTypes*` defaults to "jar","bundle"
- `*deploy-file*` - adds a local bundle file to a remote OBR configuration options:
    - `*remoteOBR*` name of remote OBR, defaults to an empty string
    - `*obrRepository*` used when the remoteOBR name is blank, defaults to `repository.xml`

    - `*repositoryId*` optional repository id, used to lookup authentication settings
    - `*url*` remote repository transport URL, like `scpexe://host/path/to/obr`
    - `*bundleUrl*` public URL of deployed bundle, like `http://www.foo.org/bundles/foo.jar`
    - `*groupId*` Maven groupId for the bundle, taken from *pomFile* if given
    - `*artifactId*` Maven artifactId for the bundle, taken from *pomFile* if given
    - `*version*` Maven version for the bundle, taken from *pomFile* if given
    - `*packaging*` Maven packaging type for the bundle, taken from *pomFile* if given
    - `*classifier*` Maven classifier type, defaults to none
    - `*pomFile*` optional Pom file describing the bundle
    - `*file*` bundle file, defaults to the bundle from the local Maven repository
    - `*obrXml*` optional additional properties for the bundle
    - `*ignoreLock*` ignore remote locking when updating the OBR
- `*clean*` - cleans the local OBR, removing missing bundles configuration options:
    - `*obrRepository*` path to local OBR, defaults to **<local-maven-repository>**`/repository.xml`
- `*remote-clean*` - cleans a remote OBR, removing missing bundles configuration options:
    - `*remoteOBR*` name of remote OBR, defaults to NONE (which means no remote cleaning)
    - `*obrRepository*` used when the remoteOBR name is blank, defaults to `repository.xml`
    - `*prefixUrl*` optional public URL prefix for the remote repository
    - `*altDeploymentRepository*` alternative remote repository, *id::layout::url*
    - `*obrDeploymentRepository*` optional OBR specific deployment repository.
    - `*ignoreLock*` ignore remote locking when updating the OBR

There are also new instructions available from the underlying BND tool, which continues to be improved independently; for the latest see BND documentation.

The default goal `*bundle*` will be initialized by setting the <packaging>entry to "bundle".

The following features are only available from version 1.2.0 onwards

Embedding dependencies

The Maven Bundle Plugin supports embedding of selected project dependencies inside the bundle by using the `<Embed-Dependency>` instruction:

```
<Embed-Dependency>dependencies</Embed-Dependency>
```

where:

```
dependencies ::== clause ( ',' clause ) *
clause ::== MATCH ( ';' attr '=' MATCH | ';inline=' inline )
attr ::== 'groupId' | 'artifactId' | 'version' | 'scope' | 'type' | 'classifier' |
'optional'
inline ::== 'true' | 'false' | PATH ( '|' PATH ) *
MATCH ::== <globbed regular expression>
PATH ::== <Ant-style path expression>
```

The plugin uses the `<Embed-Dependency>` instruction to transform the project dependencies into `<Include-Resource>` and `<Bundle-ClassPath>` clauses, which are then appended to the current set of instructions and passed onto BND. If you want the embedded dependencies to be at the start or middle of `<Include-Resource>` or `<Bundle-ClassPath>` then you can use `{maven-dependencies}`, which will automatically expand to the relevant clauses.

The MATCH section accepts alternatives, separated by *	*, and can be negated by using *!* at the *beginning* of the MATCH. Use * to represent zero or more unknown characters and ? to represent zero or one character. You can also use standard Java regexp constructs. There is no need to escape the . character inside MATCH. The first MATCH in a clause will filter against the artifactId.

some examples:

```
<!-- embed all compile and runtime scope dependencies -->
<Embed-Dependency>*;scope=compile|runtime</Embed-Dependency>

<!-- embed any dependencies with artifactId junit and scope runtime -->
<Embed-Dependency>junit;scope=runtime</Embed-Dependency>

<!-- inline all non-pom dependencies, except those with scope runtime -->
<Embed-Dependency>*;scope=!runtime;type=!pom;inline=true</Embed-Dependency>

<!-- embed all compile and runtime scope dependencies, except those with
artifactIds in the given list -->
<Embed-
Dependency>*;scope=compile|runtime;inline=false;artifactId=!cli|lang|runtime|tidy|j
sch</Embed-Dependency>
```

```
<!-- inline contents of selected folders from all dependencies -->
<Embed-Dependency>*;inline=images/**|icons/**</Embed-Dependency>
```

examples of using `{maven-dependencies}`:

```
<Include-Resource>
  {maven-resources}, {maven-dependencies},
  org/foo/Example.class=target/classes/org/foo/Example.class
</Include-Resource>

<Bundle-ClassPath>.,{maven-dependencies},some.jar</Bundle-ClassPath>
```

By default matched dependencies are embedded in the bundle as `artifactId-version.jar`. This behaviour can be modified using the following instructions:

- `<Embed-StripVersion>true</Embed-StripVersion>` - removes the version from the file (ie. *artifactId.jar*)
- `<Embed-StripGroup>false</Embed-StripGroup>` - adds the groupId as a subdirectory (ie. *groupId/artifactId-version.jar*)
- `<Embed-Directory>directory</Embed-Directory>` - adds a subdirectory (ie. *directory/artifactId-version.jar*)

Normally the plugin only checks direct dependencies, but this can be changed to include the complete set of transitive dependencies with the following option:

```
<Embed-Transitive>true</Embed-Transitive>
```

If you want a dependency inlined instead of embedded add the `inline=true`. For example to inline all *compile* and *runtime* scoped dependencies use:

```
<Embed-Dependency>*;scope=compile|runtime;inline=true</Embed-Dependency>
```

Embed-Dependency and Export-Package

If you embed a dependency with `<Embed-Dependency>`, and your `<Export-Package>` or `<Private-Package>` instructions match packages inside the embedded jar, you will see some duplication inside the bundle. This is because the `<Export-Package>` and `<Private-Package>` instructions will result in classes being inlined in the bundle, even though they also exist inside the embedded jar. If you want to export packages from an embedded dependency without such duplication then you can either inline the dependency, or use a new BND instruction called `<_exportcontents>`.

`<_exportcontents>` behaves just like Export-Package, except it doesn't change the content of the bundle, just what content should be exported.

OBR integration

The latest Maven Bundle Plugin automatically updates the local OBR repository.xml file during the install phase, using a default location of:

```
<LOCAL-MAVEN-REPOSITORY>/repository.xml
```

You can configure the location of the OBR repository by using the command line:

```
mvn clean install -DobrRepository=<PATH_TO_OBR>
```

or in the configuration section for the maven-bundle-plugin in your Maven POM:

```
<groupId>org.apache.felix</groupId>
<artifactId>maven-bundle-plugin</artifactId>
<extensions>true</extensions>
<configuration>
  <obrRepository>PATH_TO_OBR</obrRepository>
  <instructions>
    <!-- bnd instructions -->
  </instructions>
</configuration>
```

to disable OBR installation set the obrRepository to NONE, for example:

```
<groupId>org.apache.felix</groupId>
<artifactId>maven-bundle-plugin</artifactId>
<extensions>true</extensions>
<configuration>
  <obrRepository>NONE</obrRepository>
  <instructions>
    <!-- bnd instructions -->
  </instructions>
</configuration>
```

Eclipse/PDE integration

It is possible to configure the Maven Bundle Plugin to put the bundle manifest where Eclipse/PDE expects it, and use the Maven Dependency Plugin to arrange for any embedded dependencies to appear in a local directory that matches the Bundle-ClassPath entries. Here is an example POM that does this:

```
<project>

  <properties>
    <bundle.symbolicName>org.example</bundle.symbolicName>
    <bundle.namespace>org.example</bundle.namespace>
  </properties>

  <modelVersion>4.0.0</modelVersion>
  <groupId>examples</groupId>
  <artifactId>org.example</artifactId>
  <version>1.0-SNAPSHOT</version>

  <name>${bundle.symbolicName} [${bundle.namespace}]</name>

  <packaging>bundle</packaging>

  <build>
    <resources>
      <resource>
        <directory>src/main/resources</directory>
      </resource>
      <resource>
        <directory>.</directory>
        <includes>
          <include>plugin.xml</include>
        </includes>
      </resource>
    </resources>
    <plugins>
      <plugin>
        <groupId>org.apache.felix</groupId>
        <artifactId>maven-bundle-plugin</artifactId>
        <version>2.5.0</version>
        <extensions>true</extensions>
        <!--
          the following instructions build a simple set of public/private classes
into an OSGi bundle
        -->
        <configuration>
          <manifestLocation>META-INF</manifestLocation>
          <instructions>
            <Bundle-SymbolicName>${bundle.symbolicName}</Bundle-SymbolicName>
            <Bundle-Version>${pom.version}</Bundle-Version>
            <!--
              assume public classes are in the top package, and private classes are
under ".internal"
            -->
```

```
            <Export-
Package>!${bundle.namespace}.internal.*,${bundle.namespace}.*;version="${pom.versio
n}"</Export-Package>
            <Private-Package>${bundle.namespace}.internal.*</Private-Package>
            <Bundle-
Activator>${bundle.namespace}.internal.ExampleActivator</Bundle-Activator>
            <!--
              embed compile/runtime dependencies using path that matches the copied
dependency folder
            -->
            <Embed-Dependency>*;scope=compile|runtime;inline=false</Embed-
Dependency>
            <Embed-Directory>target/dependency</Embed-Directory>
            <Embed-StripGroup>true</Embed-StripGroup>
          </instructions>
        </configuration>
      </plugin>
      <plugin>
        <artifactId>maven-dependency-plugin</artifactId>
        <executions>
          <execution>
            <id>copy-dependencies</id>
            <phase>package</phase>
            <goals>
              <goal>copy-dependencies</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>

  <dependencies>
    <dependency>
      <groupId>org.osgi</groupId>
      <artifactId>osgi_R4_core</artifactId>
      <version>1.0</version>
      <scope>provided</scope>
      <optional>true</optional>
    </dependency>
    <dependency>
      <groupId>org.osgi</groupId>
      <artifactId>osgi_R4_compendium</artifactId>
      <version>1.0</version>
      <scope>provided</scope>
      <optional>true</optional>
    </dependency>
    <dependency>
```

```
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>compile</scope>
      <optional>true</optional>
    </dependency>
  </dependencies>

</project>
```

To generate the Eclipse metadata use:

```
mvn clean package eclipse:eclipse -Declipse.pde install
```

and you should now be able to import this as an existing Eclipse project.

FYI: the above POM was generated using the `pax-create-bundle` command from Pax-Construct and then tweaked to demonstrate using the Maven Dependency Plugin to handle embedded jars in Eclipse.

With the original Pax-Construct generated POM you would simply use:

```
mvn clean package pax:eclipse
```

to create the appropriate Eclipse files and manifest, and also handle any embedded entries. The pax:eclipse goal extends eclipse:eclipse, and supports the same parameters.

Unpacking bundle contents to 'target/classes'

Once in a while you may create a bundle which contains additional classes to the ones compiled from `src/main/java`, for example when you embed the classes from another jar. This can sometimes cause unforeseen problems in Maven, as it will use the output directory (`target/classes`) rather than the final bundle, when compiling against projects in the same reactor (ie. the same build).

The easiest way to get around this Maven 'feature' is to unpack the contents of the bundle to the output directory after the packaging step, so the additional classes will be found where Maven expects them. Thankfully there is now an easy option to do this in the bundle-plugin:

```
<groupId>org.apache.felix</groupId>
<artifactId>maven-bundle-plugin</artifactId>
<extensions>true</extensions>
<configuration>
  <unpackBundle>true</unpackBundle>
  <instructions>
```

```
    <!-- bnd instructions -->
  </instructions>
</configuration>
```

Using an existing MANIFEST.MF file

If you have an existing manifest, you can add this to the Bnd instructions, like so:

```
<_include>src/main/resources/META-INF/MANIFEST.MF</_include>
<Export-Package>org.example.*</Export-Package>
```

Bnd will use it when calculating the bundle contents, and will also copy across all manifest attributes starting with a capital letter. As shown in the above example, you could use this to include a non-OSGi manifest which you then customize with extra OSGi attributes.

The following features are only available from version 1.4.0 onwards

bundle:ant

The *ant* goal creates a customized `build.xml` Ant script along with a collection of BND instructions and properties, taken from the current project and stored in `maven-build.bnd`. You also need to run `*ant:ant*` to create the standard Ant support tasks to download Maven dependencies and perform compilation, etc.

The customized Ant script uses the BND tool to rebuild the bundle, so any source changes should be reflected in the (re)generated manifest.

Example:

```
mvn ant:ant bundle:ant

ant clean package
```

bundle:install-file

The *install-file* goal updates the local OBR with the details of a bundle from the local filesystem.

Configuration:

- *obrRepository* path to local OBR, defaults to **<local-maven-repository>**`/repository.xml`
- *groupId* Maven groupId for the bundle, taken from *pomFile* if given
- *artifactId* Maven artifactId for the bundle, taken from *pomFile* if given

- *version* Maven version for the bundle, taken from *pomFile* if given
- *packaging* Maven packaging type for the bundle, taken from *pomFile* if given
- *classifier* Maven classifier type, defaults to none
- *pomFile* optional Pom file describing the bundle
- *file* bundle file, defaults to the bundle from the local Maven repository
- *obrXml* optional additional properties for the bundle

Example:

```
mvn org.apache.felix:maven-bundle-plugin:1.4.0:install-file \
  -DpomFile=myPom.xml -Dfile=foo-1.0.jar
```

bundle:deploy

The *deploy goal* updates the remote OBR with the details of the deployed bundle from the local Maven repository. The remote OBR is found by querying the `<distributionManagement>` section of the project, unless `-DaltDeploymentRepository` is set. See http://maven.apache.org/plugins/maven-deploy-plugin/deploy-mojo.html for more details about these particular settings.

(If the project has an `obr.xml` file somewhere in its resources, then it will be automatically detected and applied.)

Configuration:

- *remoteOBR* name of remote OBR, defaults to NONE (which means no remote OBR deployment)
- *obrRepository* used when the remoteOBR name is blank, defaults to `repository.xml`
- *altDeploymentRepository* alternative remote repository, *id::layout::url*
- *ignoreLock* ignore remote locking when updating the OBR

This goal is part of the "bundle" packaging lifecycle, but is disabled by default - to enable just set the `remoteOBR` parameter.

bundle:deploy-file

The *deploy-file* goal updates the remote OBR with the details of a deployed bundle from the local filesystem. The remote OBR is found using the `-DrepositoryId` and `-Durl` parameters. See http://maven.apache.org/plugins/maven-deploy-plugin/deploy-file-mojo.html for more details about these particular settings.

You can use the `-DbundleUrl` parameter to give the public location of the deployed bundle, which may differ from the remote OBR location.

Configuration:

- *remoteOBR* name of remote OBR, defaults to an empty string
- *obrRepository* used when the remoteOBR name is blank, defaults to `repository.xml`
- *repositoryId* optional repository id, used to lookup authentication settings
- *url* remote repository transport URL, like `scpexe://host/path/to/obr`
- *bundleUrl* public URL of deployed bundle, like `http://www.foo.org/bundles/foo.jar`
- *groupId* Maven groupId for the bundle, taken from *pomFile* if given
- *artifactId* Maven artifactId for the bundle, taken from *pomFile* if given
- *version* Maven version for the bundle, taken from *pomFile* if given
- *packaging* Maven packaging type for the bundle, taken from *pomFile* if given
- *classifier* Maven classifier type, defaults to none
- *pomFile* optional Pom file describing the bundle
- *file* bundle file, defaults to the bundle from the local Maven repository
- *obrXml* optional additional properties for the bundle
- *ignoreLock* ignore remote locking when updating the OBR

Example:

```
mvn org.apache.felix:maven-bundle-plugin:1.4.0:deploy-file \
  -DpomFile=myPom.xml -Dfile=foo-1.0.jar -Durl=file:/tmp/example/OBR \
  -DbundleUrl=http://www.foo.org/bundles/foo.jar
```

bundle:clean

Sometimes you would like to clean your local OBR because it contains bundles that are no longer in your local Maven repository. This case often occurs when artifacts were deleted manually. The maven-bundle-plugin provides a simple goal to check for missing bundles, and remove them from the local OBR.

Configuration:

- *obrRepository* path to local OBR, defaults to **<local-maven-repository>** `/repository.xml`

Example:

```
mvn bundle:clean
```

bundle:index

The `index` goal allows the creation of an OBR repository based on a set of jars in a maven repository.

Configuration:

- *obrRepository* path to local OBR, defaults to **<local-maven-repository>** `/repository.xml`
- *urlTemplate* template for generating urls for OBR resources
- *mavenRepository* path to the maven repository, defaults to **<local-maven-repository>**

Possible values for the `urlTemplate` are:

- *maven* this will create a maven based url such as `mvn:groupid/artifactid/version`
- pattern with the following placeholders:
  - `%v` bundle version
  - `%s` bundle symbolic name
  - `%f` file name
  - `%p` file path

Concurrent updates

With a remote OBR, several uploads may occur at the same time. However, the remote OBR is centralized in one file, so concurrent modification must be avoided. To achieve this, the plug-in implements a locking system. Each time the plug-in tries to modify the file it sets a file based lock. If it can't take the lock, it will wait and retry. After 3 attempts the upload process fails. To bypass this lock add `-DignoreLock` to the command-line (or add `<ignoreLock>true<ignoreLock>` to the configuration section of your Pom).

FTP protocol

Not all protocols are supported by Maven out of the box. For example the ftp protocol requires the *wagon-ftp* component. To enable the ftp protocol add this to your Pom:

```
<build>
  <extensions>
    <extension>
      <groupId>org.apache.maven.wagon</groupId>
      <artifactId>wagon-ftp</artifactId>
      <version>1.0-alpha-6</version>
    </extension>
  </extensions>
</build>
```

How the plug-in computes the description of the bundle

The description of the bundle comes from three different sources:

- Bindex : Bindex is a tool that analyzes a bundle manifest to generate OBR description
- pom.xml : by analyzing the pom file, various information is collected (symbolic name …)
- obr.xml : this file contains customized description and capabilities for the bundle

These sources are merged together using the following precedence:

```
Bindex
| (overrides)
pom.xml
| (overrides)
obr.xml
```

A warning message is displayed when existing information is overridden.

Known issues & limitations

1. obr.xml (file given by the user to add properties not found by Bindex) must be correct, because the plug-in does not check its syntax.

Feedback

Subscribe to the Felix users mailing list by sending a message to users-subscribe@felix.apache.org; after subscribing, email questions or feedback to users@felix.apache.org.

Contents

- Embedding dependencies
- OBR integration
- Eclipse/PDE integration
- Unpacking bundle contents to 'target/classes'
- Using an existing MANIFEST.MF file

The following features are only available from version 1.4.0 onwards

- bundle:ant
- bundle:install-file
- bundle:deploy
- bundle:deploy-file
- bundle:clean
- bundle:index
- Concurrent updates
- FTP protocol
- How the plug-in computes the description of the bundle
- Known issues & limitations

Feedback