Bundle Plugin for Maven

BND是根据你选择的内容生成bundle.该插件借助BND，针对maven2工程结构，来提供maven2工程的默认行为。

注意: 测试范围的依赖并不包含在BND可见的类路径中。

1.4.0发布以后，该插件针对自动化OBR管理。帮助管理一个本地OBR并且支持原创OBR发布。该插件自动计算插件能力和要求，使用Bindex和Maven的metadata。

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 simpleb undle project that has a pubic API package an several implementation packages, such as:

org.foo.myproject.api // 对开API工程

org.foo.myproject.impl1

org.foo.myproject.impl2

...

在实现的包内有一个bundle activator, 那么该工程的pom应该这么写:

...

<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>告诉插件最后生成的bundle文件内容. The <Export-Package> 告诉插件哪些包添加到bundle JAR并且暴露出去, while the <Private-Package>告诉插件哪些包复制添加到bundle jar 但不暴露出去.如果冲突，则优先暴露。因为我们没有指定其他的manifest头, 插件会使用下面列出的默认值.插件自动生成 Import-Package 的内容, 部需要显示地声明

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 有三种类型的指导:

Manifest headers – 以大写开头的字母会出现在生成的bundle的manifest中。该header的值会被复制，参数化，或者被BND生成。

Variables – 以小写开头的instruction被认为是键值对的变量。比如version=3.0 这可以用来做属性替换。但不会被复制到manifest。

Directives – 任何以-开头的instruction会被认作是标签。可以通知BND来进行一下儿特殊的处理，但不复制到manifest。

Instructions

<Export-Package>

The <Export-Package> instruction 是指定bundle要暴露的包。这些包从可用的class中复制到bundle。 (i.e., project classes, dependencies, and class path); 因此，可以在bundle中包含在工程中没有源码的包。可以使用’\*’来泛指，可以使用'!'来去除包。

包格式列表是有序的，前面的会在后面的之前应用。所以要按这样的顺序来写"!org.foo.impl,org.foo.\*", 先写特殊的再写通用的。

依据标准的 OSGi R4 , 包格式可以包含directives 和 attributes, 都会被复制到生成的Export-Package manifest header. 除了显示地列出属性, BND 通过检查 source JAR file 或者从packageinfo来决定包的版本。

<Private-Package>

不export出去，优先级低于export

<Include-Resource>

列除了要被复制到bundle中的资源. 特定的资源被声明为下列格式的语句:

clause ::= assignment | inline | simple

assignment ::= PATH '=' PATH

simple ::= PATH

inline ::= '@' PATH

For the <Include-Resource> instruction, 实际的文件路径相对于 pom.xml, 复制到的目标路径相对于生成的bundle. Assignment或者简单格式，PATH参数可以是个文件或者目录。简单格式将资源放到bundle中只有文件名,比如不包含任何路径。比如包含src/main/resources/a/b.c 会导致资源b.c 被放置到bundle的根目录中 如果PATH指向一个目录，那么目录中的所有内容，会被复制到生成的bundle相对于指定的目录中。.如果资源需要被放置到生成的bundle的子目录内, 那么需要使用 assignment form, 第一个是目标路径，第二个是来源路径. Inline格式要求ZIP or JAR file, 会完全在bundle中展开。

如果资源表达式在 "{ ... }" 里, 那么变量替换会进行, 以这种格式 "${ ... }" syntax.

默认情况下本插件会将工程的Maven resource directories转换成一个单一的<Include-Resource>标签。如果你指定你自己的<Include-Resource> 这回覆盖生成的哪一个。

如果要包含生成的在自己的include-resources中，把{maven-resources}添加到列表中去。

<Import-Package>

The <Import-Package>列出了bundle包含的包中依赖的内容. 默认值是 "\*",导入所有相关的包。该头很少需要被显示地指定。然而，特定情况下如果有哪些不需要的导入，那可以参考Export的设定，比如不导入 "!org.foo.impl,\*"

Default Behavior

To use this plugin, very little information is required by BND. As part of the Maven integration, 本插件为各种 instructions 设置合理的默认值.

比如:

<Bundle-SymbolicName>是通过 Maven2OsgiConverter 计算出来的, 使用下面的算法：

groupId + "." + artifactId, 除去下列情况:

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

计算出来的名字被保存到 $(maven-symbolicname) 属性，当你要向他添加属性或标签.

<Export-Package>默认是工程本地的java源文件,排除默认包内的和包含impl或者internal的包。

(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> 默认是 "\*", 导入所有引用但却不再bundle中的内容.

任何导出的包也被导入来保证一致的类空间.

<Include-Resource>从工程中的Maven resources中生成, 典型的是"src/main/resources/"目录中的内容会被复制到bundle中，来映射标准的maven行为.

<Bundle-Version> 头会是 "${pom.version}" 但是会被转换成maven格式，"MAJOR.MINOR.MICRO.QUALIFIER", 比如"2.1-SNAPSHOT" 会变成 "2.1.0.SNAPSHOT".

<Bundle-Name>是配置的 "${pom.name}".

<Bundle-Description>是配置的"${pom.description}".

<Bundle-License>是配置的"${pom.licenses}".

<Bundle-Vendor>是配置的"${pom.organization.name}".

<Bundle-DocURL>是配置的"${pom.organization.url}".

因为插件为 OSGi R4 创建 bundle,所以写死了 Bundle-ManifestVersion 为 '2'. 另外为所有的导出生成对应的导入，保证包可以被替换，在合作服务时很重要，在pom的插件的configuration配置中可以设置自己需要的值。

Detailed "How To"

Get Maven2

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</Export-Service>

</instructions>

</configuration>

</plugin>

</plugins>

</build>

</project>

注意 <Export-Package> 指定导出Log Service 这个包，即使这个包没有被包含在本工程，但是仍然暴露出来了。通过这样的方式，插件会将Log Servic包复制到生成的bundle中。这样该bundle可以被解析，但不需要再下载compendium Bundle了，因为已经包含了他的包了。 最终的manifest是这样的(注意导入导出如何自动获取版本, 从源代码中的packageinfo 文件中获取):

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;version=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 注意 证书和注意文件等如何从 工程中的src/main/resources/ 目录中复制到bundle):

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

如果你想保留你的打包类型为"jar"，你可以使用 manifest goal 来生成一个a bundle manifest. 这样 maven-jar-plugin 可以使用生成的manifest 添加到最终生成的产物. 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>

如果需要打包为war格式，需要在插件中启用他:

<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>

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|jsch</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>1.2.1</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.version}"</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

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.