Baeldung

A Guide to Apache Maven

1: Apache Maven

2: Apache Maven Standard Directory Layout

3: Guide to Maven Profiles

4: Maven Dependency Scopes

5: Plugin Management in Maven

6: Maven Goals and Phases

7: Multi-Module Project with Maven

8: Maven dependencyManagement vs. dependencies Tags

9: Maven Packaging Types

10: The settings.xml File in Maven

1: Apache Maven

1. Overview

Building a software project typically consists of tasks such as downloading dependencies; putting additional jars on a classpath; compiling source code into binary code; running tests; packaging compiled code into deployable artifacts, such as JAR, WAR, and ZIP files; and deploying these artifacts to an application server or repository.

Apache Maven automates these tasks, which minimizes the risk of human error, while building the software manually and separating the work of compiling and packaging our code from that of code construction.

In this chapter, we'll explore this powerful tool for describing, building, and managing Java software projects using a central piece of information, the *Project Object Model (POM)*, which is written in XML.

2. Why Use Maven?

The key features of Maven are:

- **simple project setup that follows best practices:** Maven tries to avoid as much configuration as possible by supplying project templates (named archetypes).
- **dependency management:** it includes automatic updating, downloading, and validating the compatibility, as well as reporting the dependency closures (known also as transitive dependencies).
- **isolation between project dependencies and plugins:** with Maven, project dependencies are retrieved from the dependency repositories, while any plugin's dependencies are retrieved from the plugin repositories, resulting in fewer conflicts when plugins start to download additional dependencies.
- **central repository system:** project dependencies can be loaded from the local file system or public repositories, such as Maven Central.

In order to learn how to install Maven on our system, we can check out this tutorial on Baeldung.

3. Project Object Model

The configuration of a Maven project is done via a *Project Object Model (POM)*, represented by a pom.xml file. The *POM* describes the project, manages dependencies, and configures plugins for building the software.

The *POM* also defines the relationships among modules of multi-module projects. Let's look at the basic structure of a typical *POM* file:

```
<project>
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.baeldung</groupId>
    <artifactId>baeldung</artifactId>
    <packaging>jar</packaging>
    <version>1.0-SNAPSHOT</version>
    <name>com.baeldung</name>
    <url>http://maven.apache.org</url>
    <dependencies>
        <dependency>
            <groupId>org.junit.jupiter</groupId>
            <artifactId>junit-jupiter-api</artifactId>
            <version>5.8.2</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
    <build>
        <plugins>
            <plugin>
            //...
            </plugin>
        </plugins>
    </build>
</project>
```

Now let's take a closer look at these constructs.

3.1. Project Identifiers

Maven uses a set of identifiers, also called coordinates, to uniquely identify a project and specify how the project artifact should be packaged:

- ***groupId*** – a unique base name of the company or group that created the project
- ***artifactId*** – a unique name of the project
- ***version*** – a version of the project
- ***packaging*** – a packaging method (e.g. *WAR / JAR / ZIP*)

The first three of these *(groupId:artifactId:version)* combine to form the unique identifier, and are the mechanism by which we specify which versions of external libraries (e.g. JARs) our project will use.

3.2. Dependencies

These external libraries that a project uses are called dependencies. The dependency management feature in Maven ensures the automatic download of these libraries from a central repository, so we don't have to store them locally.

This is a key feature of Maven, which provides the following benefits:

• uses less storage by significantly reducing the number of downloads off remote repositories

• makes checking out a project quicker

• provides an effective platform for exchanging binary artifacts within our organization and beyond, without the need for building artifacts from source every time

In order to declare a dependency on an external library, we need to provide the *groupId*, *artifactId*, and *version* of the library:

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-core</artifactId>
    <version>5.3.16</version>
</dependency>
```

As Maven processes the dependencies, it'll download the Spring Core library into our local Maven repository.

3.3. Repositories

A repository in Maven is used to hold build artifacts and dependencies of varying types. The default local repository is located in the .m2/repository folder under the home directory of the user.

If an artifact or plugin is available in the local repository, Maven uses it. Otherwise, it's downloaded from a central repository and stored in the local repository. The default central repository is Maven Central.

Some libraries, such as the JBoss server, aren't available at the central repository, but are available at an alternate repository. For those libraries, we need to provide the URL to the alternate repository inside the *pom.xml* file:

```
<repositories>
    <repository>
        <id>JBoss repository</id>
        <url>http://repository.jboss.org/nexus/content/groups/public/</
url>
    </repository>
</repositories>
```

Please note that we can use multiple repositories in our projects.

3.4. Properties

Custom properties can help make our pom.xml file easier to read and maintain. In a classic use case, we would use custom properties to define versions for our project's dependencies.

Maven properties are value-placeholders and are accessible anywhere within a *pom.xml* by using the notation *${name}*, where name is the property:

```
<properties>
    <spring.version>5.3.16</spring.version>
</properties>
```

```
<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-core</artifactId>
        <version>${spring.version}</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>${spring.version}</version>
    </dependency>
</dependencies>
```

Now if we want to upgrade Spring to a newer version, we only have to change the value inside the <*spring.version*> property tag, and all the dependencies using that property in their <*version*> tags will be updated.

Properties are also often used to define build path variables:

```
<properties>
    <project.build.folder>${project.build.directory}/tmp/</project.
build.folder>
</properties>
```

```
<plugin>
    //...
    <outputDirectory>${project.resources.build.folder}</outputDirectory>
    //...
</plugin>
```

3.5. Build

The *build* section is also a very important section of the Maven *POM*. It provides information about the default Maven *goal*, the directory for the compiled project, and the final name of the application. The default *build* section looks like this:

```
<build>
    <defaultGoal>install</defaultGoal>
    <directory>${basedir}/target</directory>
    <finalName>${artifactId}-${version}</finalName>
    <filters>
      <filter>filters/filter1.properties</filter>
    </filters>
    //...
</build>
```

The default output folder for compiled artifacts is named *target*, and the final name of the packaged artifact consists of the *artifactId* and *version*, but we can change it at any time.

3.6. Using *Profiles*

Another important feature of Maven is its support for *profiles*. A *profile* is basically a set of configuration values. By using profiles, we can customize the build for different environments, such as Production/Test/Development:

```
<profiles>
    <profile>
        <id>production</id>
        <build>
            <plugins>
                <plugin>
                //...
                </plugin>
            </plugins>
        </build>
    </profile>
    <profile>
        <id>development</id>
        <activation>
            <activeByDefault>true</activeByDefault>
        </activation>
        <build>
            <plugins>
                <plugin>
                //...
                </plugin>
            </plugins>
        </build>
     </profile>
 </profiles>
```

As we can see in the example above, the default profile is set to development. If we want to run the *production profile*, we can use the following Maven command:

```
mvn clean install -Pproduction
```

Every Maven build follows a specified *lifecycle*. We can execute several build *lifecycle goals*, including the ones to *compile* the project's code, create a *package*, and *install* the archive file in the local Maven dependency repository.

4.1. *Lifecycle Phases*

The following list shows the most important Maven lifecycle phases:

- ***validate*** – checks the correctness of the project
- ***compile*** – compiles the provided source code into binary artifacts
- ***test*** – executes unit tests
- ***package*** – packages compiled code into an archive file
- ***integration-test*** – executes additional tests, which require the packaging
- ***verify*** – checks if the package is valid
- ***install*** – installs the package file into the local Maven repository
- ***deploy*** – deploys the package file to a remote server or repository

4.2. *Plugins and Goals*

A Maven *plugin* is a collection of one or more goals. Goals are executed in phases, which helps to determine the order in which the *goals* are executed.

The rich list of plugins that are officially supported by Maven is available here. **There's also an interesting article on Baeldung about how to build an executable JAR using various plugins.**

To gain a better understanding of which goals are run in which phases by default, take a look at the default Maven *lifecycle* bindings.

To go through any one of the above phases, we just have to call one command:

```
mvn <phase>
```

For example, *mvn clean install* will remove the previously created jar/war/zip files and compiled classes *(clean)*, as well as execute all the phases necessary to install the new archive *(install)*.

Please note that *goals* provided by *plugins* can be associated with different phases of the *lifecycle*.

In this section, we'll use the command line functionality of Maven to create a Java project.

5.1. Generating a Simple Java Project

In order to build a simple Java project, let's run the following command:

```
mvn archetype:generate \
  -DgroupId=com.baeldung \
  -DartifactId=baeldung \
  -DarchetypeArtifactId=maven-archetype-quickstart \
  -DarchetypeVersion=1.4 \
  -DinteractiveMode=false
```

The *groupId* is a parameter indicating the group or individual that created a project, which is often a reversed company domain name. The *artifactId* is the base package name used in the project, and we use the standard *archetype*. Here we're using the latest archetype version to ensure our project is created with the updated and latest structure.
Since we didn't specify the version and packaging type, these will be set to default values; the version will be set to *1.0-SNAPSHOT*, and the packaging will be *jar* by default.
If we don't know which parameters to provide, we can always specify *interactiveMode=true*, so that Maven asks for all the required parameters.

After the command completes, we have a Java project containing an App.java class, which is just a simple "Hello World" program, in the *src/main/java* folder. We also have an example test class in *src/test/java*. The *pom.xml* of this project will look similar to this:

```
1.  <project>
2.      <modelVersion>4.0.0</modelVersion>
3.      <groupId>com.baeldung</groupId>
4.      <artifactId>baeldung</artifactId>
5.      <version>1.0-SNAPSHOT</version>
6.      <name>baeldung</name>
7.      <url>http://www.example.com</url>
8.      <dependencies>
9.          <dependency>
10.             <groupId>junit</groupId>
11.             <artifactId>junit</artifactId>
12.             <version>4.11</version>
13.             <scope>test</scope>
14.         </dependency>
15.     </dependencies>
16. </project>
```

As we can see, the *JUnit* dependency is provided by default.

5.2. Compiling and Packaging a Project

The next step is to compile the project:

```
mvn compile
```

Maven will run through all *lifecycle* phases needed by the *compile* phase to build the project's sources. If we want to run only the test phase, we can use:

```
mvn test
```

Now let's invoke the *package* phase, which will produce the compiled archive *jar* file:

```
mvn package
```

5.3. Executing an Application

Finally, we're going to execute our Java project with the *exec-maven-plugin*. Let's configure the necessary plugins in the *pom.xml*:

```
<build>
    <sourceDirectory>src</sourceDirectory>
    <plugins>
        <plugin>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>3.6.1</version>
            <configuration>
                <source>1.8</source>
                <target>1.8</target>
            </configuration>
        </plugin>
        <plugin>
            <groupId>org.codehaus.mojo</groupId>
            <artifactId>exec-maven-plugin</artifactId>
            <version>3.0.0</version>
            <configuration>
                <mainClass>com.baeldung.java.App</mainClass>
            </configuration>
        </plugin>
    </plugins>
</build>
```

The first plugin, *maven-compiler-plugin*, is responsible for compiling the source code using Java version 1.8. The *exec-maven-plugin* searches for the *mainClass* in our project.

To execute the application, we run the following command:

```
mvn exec:java
```

6. Conclusion

In this chapter, we discussed some of the more popular features of the Apache Maven build tool.

All code examples on Baeldung are built using Maven, so check out our GitHub project website to see various Maven configurations.

2: Apache Maven Standard Directory Layout

1. Overview

Apache Maven is one of the most popular build tools for Java projects. Apart from decentralizing dependencies and repositories, promoting a uniform directory structure across projects is one of its most important aspects.

In this quick chapter, we'll explore the standard directory layout of a typical Maven project.

2. Directory Layout

A typical Maven project has a *pom.xml* file and a directory structure based on defined conventions:

```
└──maven-project
    ├──pom.xml
    ├──README.txt
    ├──NOTICE.txt
    ├──LICENSE.txt
    └──src
        ├──main
        │   ├──java
        │   ├──resources
        │   ├──filters
        │   └──webapp
        ├──test
        │   └──java
```

The default directory layout can be overridden using project descriptors, but this is uncommon and discouraged.

Going ahead in this chapter, we'll uncover more details about each standard file and subdirectory.

3. The Root Directory

This directory serves as the root of every Maven project.

Let's take a closer look at the standard files and subdirectories that are typically found at root:

• ***maven-project/pom.xml*** – defines dependencies and modules needed during the build lifecycle of a Maven project

• ***maven-project/LICENSE.txt*** – licensing information of the project

• ***maven-project/README.txt*** – summary of the project

• ***maven-project/NOTICE.txt*** – information about third-party libraries used in the project

• ***maven-project/src/main*** – contains source code and resources that become part of the artifact

• ***maven-project/src/test*** – holds all the test code and resources

• ***maven-project/src/it*** – usually reserved for integration tests used by the Maven Failsafe Plugin

• ***maven-project/src/site*** – site documentation created using the Maven Site Plugin

• ***maven-project/src/assembly*** – assembly configuration for packaging binaries

4. The *src/main* Directory

As the name indicates, *src/main* is the most important directory of a Maven project. Anything that's supposed to be part of an artifact, be it a *jar* or *war*, should be present here.

Its subdirectories are:

• ***src/main/java*** – Java source code for the artifact

• ***src/main/resources*** – configuration files and others, such as i18n files, per-environment configuration files, and XML configurations

• ***src/main/webapp*** – for web applications, contains resources like JavaScript, CSS, HTML files, view templates, and images

• ***src/main/filters*** – contains files that inject values into configuration properties in the resources folder during the build phase

5. The *src/test* Directory

The directory *src/test* is the place where tests of each component in the application reside.

Note that none of these directories or files will become part of the artifact. Let's see its subdirectories:

• ***src/test/java*** – Java source code for tests

• ***src/main/resources*** – configuration files and others used by tests

• ***src/test/filters*** – contains files that inject values into configuration

• ***src/main/filters*** – contains files that inject values into configuration properties in the resources folder during the test phase

6. Conclusion

In this chapter, we looked at the standard directory layout for an Apache Maven project.

Multiple examples of Maven project structures can be found in the GitHub project.

3: Guide to Maven Profiles

1. Overview

Maven profiles can be used to create customized build configurations, like targeting a level of test granularity, or a specific deployment environment.

In this chapter, we'll learn how to work with Maven profiles.

Normally **when we run *mvn* package, the unit tests are executed** as well. But **what if we want to quickly package the artifact and run it** to see if it works?

First, we'll create a *no-tests* profile that sets the maven.test.skip property to *true*:

```
<profile>
    <id>no-tests</id>
    <properties>
        <maven.test.skip>true</maven.test.skip>
    </properties>
</profile>
```

Next, we'll execute the profile by running the *mvn* package *-Pno-tests* command. **Now the artifact is created and the tests are skipped**. In this case, the *mvn package -Dmaven.test.skip* command would've been easier.

However, this was just an introduction to Maven profiles. Let's take a look at some more complex setups.

In the previous section, we saw how to create one profile. **We can configure as many profiles as we want** by giving them unique ids.

Let's say we wanted to create a profile that only ran our integration tests, and another profile for a set of mutation tests.

We would begin by specifying an id for each one in our *pom.xml* file:

```
<profiles>
    <profile>
        <id>integration-tests</id>
    </profile>
    <profile>
        <id>mutation-tests</id>
    </profile>
</profiles>
```

Within each profile element, **we can configure many elements, such as *dependencies*, *plugins*, *resources*, and *finalName*.**

So, for the above example, we could add plugins and their dependencies separately for *integration-tests* and *mutation-tests*.

Separating tests into profiles can make the default build faster by having it focus, for instance, on just the unit tests.

3.1. Profile Scope

We just placed these profiles in our *pom.xml* file, which declares them only for our project.

But in Maven 3, we can actually add profiles to any of three locations:

1- Project-specific profiles go into the project's *pom.xml* file
2- User-specific profiles go into the user's *settings.xml* file
3- Global profiles go into the global *settings.xml* file

We try to configure profiles in the *pom.xml* whenever possible. This is because we want to use the profiles both on our development machines and build machines. Using the *settings.xml* is more difficult and error-prone, as we have to distribute it across build environments ourselves.

After we create one or more profiles, **we can start using them, or in other words, *activating* them**.

4.1. Seeing Which Profiles Are Active

Let's use the *help:active-profiles* goal to see which profiles are active in our default build:

```
mvn help:active-profiles
```

default build:

```
The following profiles are active:
```

Well, nothing.

We'll activate them in just a moment, but another way to see what's activated is to **include the *maven-help-plugin* in our *pom.xml* and tie the *active-profiles* goal to the *compile* phase:**

```
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-help-plugin</artifactId>
            <version>3.2.0</version>
            <executions>
                <execution>
                    <id>show-profiles</id>
                    <phase>compile</phase>
                    <goals>
                        <goal>active-profiles</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

Now let's look at a few different ways we can use them.

4.2. Using -P

We already saw one way at the beginning, which is that we can **activate profiles with the *-P* argument**. So let's begin by enabling the *integration-tests* profile:

```
mvn package -P integration-tests
```

If we verify the active profiles with the *maven-help-plugin* or the *mvn help:active-profiles -P integration-tests* command, we'll get the following result:

```
The following profiles are active:
 - integration-tests
```

If we want to activate multiple profiles at the same time, we can use a comma-separated list of profiles:

```
mvn package -P integration-tests,mutation-tests
```

4.3. Active by Default

If we always want to execute a profile, we can make one active by default:

```
1.  <profile>
2.      <id>integration-tests</id>
3.      <activation>
4.          <activeByDefault>true</activeByDefault>
5.      </activation>
6.  </profile>
```

Then we can run *mvn* package without specifying the profiles, and we can verify that the *integration-test* profile is active.

However, if we run the Maven command and enable another profile, then the *activeByDefault* profile is skipped. So when we run *mvn package -P mutation-tests*, only the *mutation-tests* profile is active.

When we activate in other ways, the *activeByDefault* profile is also skipped, as we'll see in the next sections.

4.4. Based on a Property

We can activate profiles on the command-line; however, it's sometimes more convenient if they're activated automatically. For instance, **we can base it on a *-D* system property:**

```
<profile>
    <id>active-on-property-environment</id>
    <activation>
        <property>
            <name>environment</name>
        </property>
    </activation>
</profile>
```

We now activate the profile with the *mvn package -Denvironment* command. It's also possible to activate a profile if a property isn't present:

```
<property>
    <name>!environment</name>
</property>
```

Or we can activate the profile **if the property has a specific value:**

```
<property>
    <name>environment</name>
    <value>test</value>
</property>
```

Now we can run the profile with *mvn package -Denvironment=test*.
Finally, we can activate the profile if the property has a value other than the specified value:

```
<property>
    <name>environment</name>
    <value>!test</value>
</property>
```

4.5. Based on the JDK Version

Another option is to enable a profile based on the JDK running on the machine. In this case, we want to enable the profile if the JDK version starts with 11:

```
<profile>
    <id>active-on-jdk-11</id>
    <activation>
        <jdk>11</jdk>
    </activation>
</profile>
```

We can also use ranges for the JDK version, as explained in Maven Version Range Syntax.

4.6. Based on the Operating System

Alternatively, we can activate the profile based on some operating system information. If we aren't sure of the specifics, we can first use the *mvn enforcer:display-info* command, which gives the following output on my machine:

```
Maven Version: 3.5.4
JDK Version: 11.0.2 normalized as: 11.0.2
OS Info: Arch: amd64 Family: windows Name: windows 10 Version: 10.0
```

After that, we can configure a profile that's activated only on Windows 10:

```
<profile>
    <id>active-on-windows-10</id>
    <activation>
        <os>
            <name>windows 10</name>
            <family>Windows</family>
            <arch>amd64</arch>
            <version>10.0</version>
        </os>
    </activation>
</profile>
```

4.7. Based on a File

Another option is to run a profile if a file exists or is missing. So let's create a test profile that only executes if the *testreport.html* isn't yet present:

```
<activation>
    <file>
        <missing>target/testreport.html</missing>
    </file>
</activation>
```

5. Deactivating a Profile

We've seen many ways to activate profiles, but sometimes we need to disable them as well.

To disable a profile, we can use the '!' or '-'.

So to disable the *active-on-jdk-11* profile, we execute:

```
mvn compile -P -active-on-jdk-11
```

6. Conclusion

In this chapter, we learned how to work with Maven profiles in order to create different build configurations.

4: Maven Dependency Scopes

1. Overview

Maven is one of the most popular build tools in the Java ecosystem, and one of its core features is dependency management.

In this chapter, we'll describe and explore the mechanism that helps with managing transitive dependencies in Maven projects - dependency scopes.

2. Transitive Dependency

There are two types of dependencies in Maven: direct and transitive.

Direct dependencies are the ones that we explicitly include in the project.

These can be included using <*dependency*> tags:

```
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
</dependency>
```

On the other hand, transitive dependencies are required by direct dependencies.

Maven automatically includes required transitive dependencies in our project.

We can list all dependencies, including transitive dependencies, in the project using the *mvn dependency:tree* command.

3. Dependency Scopes

Dependency scopes can help to limit the transitivity of the dependencies. They also modify the classpath for different build tasks. **Maven has six default dependency scopes.**

It's important to understand that each scope, except for *import*, has an impact on transitive dependencies.

3.1. Compile

This is the default scope when no other scope is provided.

Dependencies with this scope are available on the classpath of the project in all build tasks. They are also propagated to the dependent projects.

More importantly, these dependencies are also transitive:

```
<dependency>
    <groupId>commons-lang</groupId>
    <artifactId>commons-lang</artifactId>
    <version>2.6</version>
</dependency>
```

3.2. Provided

We use this scope to mark **dependencies that should be provided at runtime by JDK or a container.**

A good use case for this scope would be a web application deployed in a container, where the container already provides some libraries itself.

For example, this could be a web server that already provides the Servlet API at runtime.

In our project, we can define those dependencies with the *provided* scope:

```
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>javax.servlet-api</artifactId>
    <version>4.0.1</version>
    <scope>provided</scope>
</dependency>
```

The *provided* dependencies are available only at compile time and in the test classpath of the project. These dependencies are not transitive.

3.3. Runtime

The dependencies with this scope are required at runtime, but we don't need them for the compilation of the project code.

As a result, dependencies marked with the *runtime* scope will be present in the *runtime* and test classpath, but they'll be missing from the compile classpath.

A JDBC driver is a good example of dependencies that should use the runtime scope:

```
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.28</version>
    <scope>runtime</scope>
</dependency>
```

3.4. Test

We use this scope to indicate that a dependency isn't required at standard runtime of the application, and is only used for test purposes. ***Test* dependencies aren't transitive and are only present for test and execution classpaths.**

The standard use case for this scope is adding a test library, such as JUnit, to our application:

```
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
    <scope>test</scope>
</dependency>
```

3.5. System

The *System* scope is very similar to the *provided* scope. The main difference is that system requires us to point directly to a specific jar on the system.

It's worthwhile to mention that the *system* scope is deprecated. The important thing to remember is that building a project with system scope dependencies may fail on different machines if the dependencies aren't present or are located in a different place than the one *systemPath* points to:

```
<dependency>
    <groupId>com.baeldung</groupId>
    <artifactId>custom-dependency</artifactId>
    <version>1.3.2</version>
    <scope>system</scope>
    <systemPath>${project.basedir}/libs/custom-dependency-1.3.2.jar</
systemPath>
</dependency>
```

3.6. Import

It's only available for the dependency type *pom*.

import indicates that this dependency should be replaced with all effective dependencies declared in its POM.

Here, the *custom-project* dependency will be replaced with all the dependencies declared in the custom-project's *pom.xml* <*dependencyManagement*> section:

```
<dependency>
    <groupId>com.baeldung</groupId>
    <artifactId>custom-project</artifactId>
    <version>1.3.2</version>
    <type>pom</type>
    <scope>import</scope>
</dependency>
```

Each dependency scope affects transitive dependencies in its own way. This means that different transitive dependencies may end up in the project with different scopes. However, dependencies with scopes *provided* and *test* will never be included in the main project.

Let's take a detailed look at what this means:

- For the compile scope, all dependencies with the runtime scope will be pulled in with the runtime scope in the project, and all dependencies with the compile scope will be pulled in with the compile scope in the project.
- For the provided scope, both the runtime and compile scope dependencies will be pulled in with the provided scope in the project.
- For the test scope, both the runtime and compile scope transitive dependencies will be pulled in with the test scope in the project.
- For the runtime scope, both the runtime and compile scope transitive dependencies will be pulled in with the runtime scope in the project.

5. Conclusion

In this quick chapter, we focused on Maven dependency scopes, their purpose, and the details of how they operate.

To dig deeper into Maven, the documentation is a great place to start.

5: Plugin Management in Maven

1. Overview

Apache Maven is a powerful tool that uses plugins to automate and perform all the build and reporting tasks in a Java project.

However, there are likely to be several of these plugins used in the build, along with different versions and configurations, especially in a multi-module project. This can lead to problems of complex POM files with redundant or duplicate plugin artifacts, as well as configurations scattered across various child projects.

In this chapter, we'll see how to use Maven's plugin management mechanism to handle such issues and effectively maintain plugins across the whole project.

Maven has two types of plugins:

- Build – executed during the build process. Examples include Clean, Install, and Surefire plugins. These should be configured in the *build* section of the POM.

- Reporting – executed during site generation to produce various project reports. Examples include Javadoc and Checkstyle plugins. These are configured in the *reporting* section of the project POM.

Maven plugins provide all the useful functionalities required to execute and manage the project build.

For example, we can declare the Jar plugin in the POM:

```
<build>
    ....
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-jar-plugin</artifactId>
            <version>3.2.0</version>
            ....
        </plugin>
    ....
    </plugins>
</build>
```

Here we included the plugin in the build section to add the capability to compile our project into a *jar*.

In addition to the plugins, we can also declare plugins in the *pluginManagement* section of the POM. This contains the *plugin* elements in much the same way as we previously saw. However, by adding the plugin in the *pluginManagement* section, **it becomes available to this POM, and all inheriting child POMs.**

This means that any child POMs will inherit the plugin executions simply by referencing the plugin in their plugin section. All we need to do is add the relevant *groupId* and *artifactId*, without having to duplicate the configuration or manage the version.

Similar to the dependency management mechanism, this is particularly useful in multi-module projects, as it provides a central location to manage plugin versions and any related configuration.

4. Example

Let's start by creating a simple multi-module project with two submodules. We'll include the Build Helper Plugin in the parent POM, which contains several small goals to assist with the build lifecycle. In our example, we'll use it to copy some additional resources to the project output for a child project.

4.1. Parent POM Configuration

First, we'll add the plugin to the *pluginManagement* section of the parent POM:

```
<pluginManagement>
    <plugins>
        <plugin>
            <groupId>org.codehaus.mojo</groupId>
            <artifactId>build-helper-maven-plugin</artifactId>
            <version>3.3.0</version>
            <executions>
                <execution>
                    <id>add-resource</id>
                    <phase>generate-resources</phase>
                    <goals>
                        <goal>add-resource</goal>
                    </goals>
                    <configuration>
                        <resources>
                            <resource>
                                <directory>src/resources</directory>
                                <targetPath>json</targetPath>
                            </resource>
                        </resources>
                    </configuration>
                </execution>
            </executions>
        </plugin>
    </plugins>
</pluginManagement>
```

This binds the plugin's *add-resource* goal to the *generate-resources* phase in the default POM lifecycle. We also specified the *src/resources* directory containing the additional resources. The plugin execution will copy these resources to the target location in the project output as required.

Next, we'll run the maven command to ensure that the configuration is valid and the build is successful:

```
$ mvn clean test
```

After running this, we can see the target location doesn't yet contain the expected resources.

4.2. Child POM Configuration

Now let's reference this plugin from the child POM:

```
<build>
    <plugins>
        <plugin>
            <groupId>org.codehaus.mojo</groupId>
            <artifactId>build-helper-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>
```

Similar to dependency management, we don't declare the version or any plugin configuration. Instead, child projects inherit these values from the declaration in the parent POM.

Finally, let's run the build again and look at the output:

```
....
[INFO] --- build-helper-maven-plugin:3.3.0:add-resource (add-resource) @
submodule-1 ---

[INFO]

[INFO] --- maven-resources-plugin:2.6:resources (default-resources) @
submodule-1 ---

[INFO] Using 'UTF-8' encoding to copy filtered resources.

[INFO] Copying 1 resource to json
....
```

Here the plugin executes during the build, but only in the child project with the corresponding declaration. As a result, the project output now contains the additional resources from the specified project location, as we expected.

We should note that **only the parent POM contains the plugin declaration and configuration**, whilst the child projects just reference this as needed.

The child projects are also free to modify the inherited configuration if required.

There are some Maven core plugins that are used by default as part of the build lifecycle. For example, the clean and *compiler* plugins don't need to be declared explicitly.

We can, however, explicitly declare and configure these in the *pluginManagement* element in the POM. The main difference is that the **core plugin configuration takes effect automatically without any reference in the child projects**.

Let's try this out by adding the *compiler* plugin to the familiar *pluginManagement* section:

```
<pluginManagement>
    ....
    <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.10.0</version>
        <configuration>
            <source>1.8</source>
            <target>1.8</target>
        </configuration>
    </plugin>
    ....
</pluginManagement>
```

Here we locked down the plugin version, and configured it to use Java 8 to build the project. However, there's no additional *plugin* declaration required in any child projects. The build framework activates this configuration by default. Therefore, this configuration means that the build must use Java 8 to compile this project across all modules.

Overall, it can be **good practice to explicitly declare the configuration and lockdown the versions for any plugins required in a multi-module project**. Consequently, different child projects can inherit only the required plugin configurations from the parent POM and apply them as needed.

This eliminates duplicate declarations, simplifies the POM files, and improves build reproducibility.

6. Conclusion

In this chapter, we learned how to centralize and manage the Maven plugins required for building a project.

First, we looked at plugins and their usage in the project POM. Then we took a more detailed look at the Maven plugin management mechanism, and how it helps to reduce duplication and improve the maintainability of the build configuration.

As always, the example code is available over on GitHub.

6: Maven Goals and Phases

1. Overview

In this chapter, we'll explore different Maven build lifecycles and their phases.

We'll also discuss the core relation between Goals and Phases.

2. Maven Build Lifecycle

The Maven build follows a specific lifecycle to deploy and distribute the target project.

There are three built-in lifecycles:

• default: the main lifecycle, as it's responsible for project deployment

• clean: to clean the project and remove all files generated by the previous build

• site: to create the project's site documentation

Each lifecycle consists of a sequence of phases. The *default* build lifecycle consists of 23 phases, as it's the main build lifecycle.

On the other hand, the *clean* lifecycle consists of 3 phases, while the *site* lifecycle is made up of 4 phases.

3. Maven Phase

A Maven phase represents a stage in the Maven build lifecycle. Each phase is responsible for a specific task.

Here are some of the most important phases in the *default* build lifecycle:

• *validate*: check if all information necessary for the build is available

• *compile*: compile the source code

• *test-compile*: compile the test source code

• *test*: run unit tests

• *package*: package compiled source code into the distributable format (jar, war, ...)

• *integration-test*: process and deploy the package if needed to run integration tests

• *install*: install the package to a local repository

• *deploy*: copy the package to the remote repository

For the full list of each lifecycle's phases, check out the Maven Reference.
Phases are executed in a specific order. This means that if we run a specific phase using the command:

```
mvn <PHASE>
```

It won't only execute the specified phase, but all the preceding phases as well.

For example, if we run the *deploy* phase, which is the last phase in the *default* build lifecycle, it'll execute all the phases before the *deploy* phase as well, which is the entire *default* lifecycle:

```
mvn deploy
```

Each phase is a sequence of goals, and each goal is responsible for a specific task.

When we run a phase, all goals bound to this phase are executed in order.

Here are some of the phases and default goals bound to them:

• *compiler:compile* – the *compile* goal from the *compiler* plugin is bound to the *compile* phase

• *compiler:testCompile* is bound to the *test-compile* phase

• *surefire:test* is bound to the *test* phase

• *install:install* is bound to the *install* phase

• *jar:jar* and *war:war* is bound to the *package* phase

We can list all goals bound to a specific phase and their plugins using the command:

```
mvn help:describe -Dcmd=PHASENAME
```

For example, to list all goals bound to the *compile* phase, we can run:

```
mvn help:describe -Dcmd=compile
```

Then we'd get the sample output:

```
'compile' is a phase corresponding to this plugin:

org.apache.maven.plugins:maven-compiler-plugin:3.1:compile
```

As mentioned above, this means the *compile* goal from the *compiler* plugin is bound to the *compile* phase.

A Maven plugin is a group of goals; however, these goals aren't necessarily all bound to the same phase.

For example, here's a simple configuration of the Maven Failsafe plugin, which is responsible for running integration tests:

```
<build>
    <plugins>
        <plugin>
            <artifactId>maven-failsafe-plugin</artifactId>
            <version>${maven.failsafe.version}</version>
            <executions>
                <execution>
                    <goals>
                        <goal>integration-test</goal>
                        <goal>verify</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

As we can see, the Failsafe plugin has two main goals configured here:

• integration-test: run integration tests
• verify: verify all integration tests passed

We can use the following command to **list all goals in a specific plugin:**

```
mvn <PLUGIN>:help
```

For example, to list all goals in the Failsafe plugin, we can run:

```
mvn failsafe:help
```

And the output will be:

```
This plugin has 3 goals:

failsafe:help
  Display help information on maven-failsafe-plugin.
  Call mvn failsafe:help -Ddetail=true -Dgoal=<goal-name> to display parameter
  details.

failsafe:integration-test
  Run integration tests using Surefire.

failsafe:verify
  Verify integration tests ran using Surefire.
```

To run a specific goal without executing its entire phase (and the preceding phases), we can use the command:

```
mvn <PLUGIN>:<GOAL>
```

For example, to run the integration-test goal from the Failsafe plugin, we need to run:

```
mvn failsafe:integration-test
```

To build a Maven project, we need to execute one of the lifecycles by running one of their phases:

```
mvn deploy
```

This will execute the entire *default* lifecycle. Alternatively, we can stop at the *install* phase:

```
mvn install
```

But usually, we'll clean the project first by running the *clean* lifecycle before the new build:

```
mvn clean install
```

We can also run only a specific goal of the plugin:

```
mvn compiler:compile
```

Note that if we try to build a Maven project without specifying a phase or goal, we'll get an error:

```
[ERROR] No goals have been specified for this build. You must specify a valid
lifecycle phase or a goal
```

7. Conclusion

In this chapter, we discussed Maven build lifecycles, as well as the relation between Maven phases and goals.

7: Multi-Module Project with Maven

1. Overview

In this chapter, we'll learn how to build a multi-module project with Maven.

First, we'll discuss what a multi-module project is, and have a look at the benefits of following this approach. Then we'll set up our sample project. For a good introduction to Maven, check out this tutorial.

2. Maven's Multi-Module Project

A multi-module project is built from an aggregator POM that manages a group of submodules. In most cases, the aggregator is located in the project's root directory and must have packaging of type *pom*.

The submodules are regular Maven projects, and they can be built separately or through the aggregator POM.

By building the project through the aggregator POM, each project that has a packaging type different from *pom* will result in the built archive file.

3. Benefits of Using Multi-Modules

The significant advantage of using this approach is that **we may reduce duplication.**

Let's say we have an application that consists of several modules, a front-end module and a back-end module. Now imagine we work on them and change the functionality, which affects them both. In that case, without a specialized build tool, we'd have to build both components separately, or write a script to compile the code, run tests, and show the results. Then, after we got even more modules in the project, it would become harder to manage and maintain.

In the real world, projects may need certain Maven plugins to perform various operations during the build lifecycle, to share dependencies and profiles, and to include other BOM projects.

Therefore, when leveraging multi-modules, we can **build our application's modules in a single command**, and if the order matters, Maven will figure it out for us. We can also **share a vast amount of configuration with other modules**.

4. Parent POM

Maven supports inheritance in a way that **each *pom.xml* file has the implicit parent POM. It's called Super POM** and can be located in the Maven binaries.

We can create our **own *pom.xml* file**, **which will serve us as the parent project.** Then we can include in it all the configuration with dependencies, and set it as the parent of our child modules, so they'll inherit from it.

Besides inheritance, Maven provides the notion of aggregation. A parent POM that leverages this functionality is called an aggregate POM. Basically, this kind of POM **declares its modules explicitly in its *pom.xml* file**.

5. Submodules

Submodules, or subprojects, are regular Maven projects that inherit from the parent POM. As we already know, inheritance lets us share the configuration and dependencies with submodules. However, if we'd like to build or release our project in one shot, we have to declare our submodules explicitly in the parent POM. Ultimately, our parent POM will be the parent, as well as the aggregate POM.

Now that we understand Maven's submodules and hierarchy, let's build a sample application to demonstrate them. We'll use Maven's command-line interface to generate our projects.

This app will consist of three modules that will represent:

- The *core* part of our domain
- A web *service* providing some REST APIs
- A *webapp* containing user-facing web assets of some sort

Since we'll focus on Maven, the implementation of these services will remain undefined.

6.1. Generating Parent POM

First, let's create a parent project:

```
mvn archetype:generate -DgroupId=com.baeldung -DartifactId=parent-project
```

Once the parent is generated, we have to open the *pom.xml* file located in the parent's directory and add the packaging as *pom*:

```
<packaging>pom</packaging>
```

By setting the packaging to *pom* type, we're declaring that the project will serve as a parent or an aggregator; it won't produce further artifacts.

Now, as our aggregator is done, we can generate our submodules.

However, we need to note, this is where all the configuration to be shared is located, which will eventually be re-used in child modules. Among other things, we can make use of *dependencyManagement* or *pluginManagement* here.

6.2. Creating Submodules

As our parent POM was named *parent-project*, we need to make sure we're in the parent's directory, and then run *generate* commands:

```
cd parent-project
mvn archetype:generate -DgroupId=com.baeldung -DartifactId=core
mvn archetype:generate -DgroupId=com.baeldung -DartifactId=service
mvn archetype:generate -DgroupId=com.baeldung -DartifactId=webapp
```

Notice the command used. It's the same as we used for the parent. The thing here is, these modules are regular Maven projects, yet Maven recognized that they're nested. When we changed the directory to the *parent-project*, it found that the parent has the packaging of type *pom*, and modified the *pom.xml* files accordingly.

In the *parent-project's* pom.xml, it'll add all the submodules inside the *modules* section:

```
<modules>
    <module>core</module>
    <module>service</module>
    <module>webapp</module>
</modules>
```

And in the individual submodules' *pom.xml*, it'll add the parent-project in the *parent* section:

```
<parent>
  <artifactId>parent-project</artifactId>
  <groupId>com.baeldung</groupId>
  <version>1.0-SNAPSHOT</version>
</parent>
```

Finally, Maven will generate the three submodules successfully

It's important to note that submodules can have only one parent. However, we can import many BOMs. More details about the BOM files can be found in this article.

6.3. Building the Project

Now we can build all three modules at once. In the parent's project directory, we'll run:

```
mvn package
```

This will build all the modules. We should see the following output of the command:

```
[INFO] Scanning for projects...
[INFO]
------------------------------------------------------------------------
[INFO] Reactor Build Order:
[INFO] parent-project
[pom]
[INFO] core
[jar]
[INFO] service
[jar]
[INFO] webapp
[war]
```

```
[INFO] Reactor Summary for parent-project 1.0-SNAPSHOT:
[INFO] parent-project ..................................... SUCCESS [  0.272 s]
[INFO] core ............................................... SUCCESS [  2.043 s]
[INFO] service ............................................ SUCCESS [  0.627 s]
[INFO] webapp ............................................. SUCCESS [  0.572 s]
[INFO]
------------------------------------------------------------------------[INFO]
BUILD SUCCESS
[INFO]
------------------------------------------------------------------------
```

The Reactor lists the *parent-project*, but since it's *pom* type, it's excluded, and the build results in three separate *.jar* files for all the other modules. In this case, build occurs in all three of them.

Moreover, Maven Reactor will analyze our project and build it in the proper order. So if our *webapp* module depends on the *service* module, Maven will first build the *service*, then the *webapp*.

6.4. Enable Dependeny Management in Parent Project

Dependency management is a mechanism for centralizing the dependency information for a multi-module parent project and its children.

When we have a set of projects or modules that inherit a common parent, we can put all the required information about the dependencies in the common *pom.xml* file. This will simplify the references to the artifacts in the child *POMs*.

Let's take a look at a sample parent's *pom.xml*:

```
1.  <dependencyManagement>
2.      <dependencies>
3.          <dependency>
4.              <groupId>org.springframework</groupId>
5.              <artifactId>spring-core</artifactId>
6.              <version>5.3.16</version>
7.          </dependency>
8.          //...
9.      </dependencies>
10. </dependencyManagement>
```

By declaring the *spring-core* version in the parent, all submodules that depend on *spring-core* can declare the dependency using only the *groupId* and *artifactId*, and the version will be inherited:

```
<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-core</artifactId>
    </dependency>
    //...
</dependencies>
```

Moreover, we can provide exclusions for dependency management in the parent's *pom.xml*, so that specific libraries won't be inherited by child modules:

```
<exclusions>
    <exclusion>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
    </exclusion>
</exclusions>
```

Finally, if a child module needs to use a different version of a managed dependency, we can override the managed version in the child's *pom.xml* file:

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-core</artifactId>
    <version>4.3.30.RELEASE</version>
</dependency>
```

Please note that while child modules inherit from their parent project, a parent project doesn't necessarily have any modules that it aggregates. On the other hand, a parent project may also aggregate projects that don't inherit from it.

For more information on inheritance and aggregation, please refer to this documentation.

6.5. Updating the Submodules and Building a Project

We can change the *packaging* type of each submodule. For example, let's change the packaging of the *webapp* module to *WAR* by updating the *pom.xml* file:

```
<packaging>war</packaging>
```

We'll also add *maven-war-plugin* in the plugins list:

```
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-war-plugin</artifactId>
            <version>3.3.2</version>
            <configuration>
               <failOnMissingWebXml>false</failOnMissingWebXml>
            </configuration>
        </plugin>
    </plugins>
</build>
```

Now we can test the build of our project by using the *mvn clean install* command. The output of the Maven logs should be similar to this:

```
[INFO] Scanning for projects...
[INFO]
------------------------------------------------------------------------
[INFO] Reactor Build Order:
[INFO]
[INFO] parent-project
[pom]
[INFO] core
[jar]
[INFO] service
[jar]
[INFO] webapp
[war]

//.............

[INFO] Reactor Summary for parent-project 1.0-SNAPSHOT:
[INFO]
[INFO] parent-project ..................................... SUCCESS [  0.272 s]
[INFO] core ............................................... SUCCESS [  2.043 s]
[INFO] service ............................................ SUCCESS [  0.627 s]
[INFO] webapp ............................................. SUCCESS [  1.047 s]
```

7. Conclusion

In this chapter, we discussed the benefits of using Maven multi-modules. We also distinguished between regular Maven's parent POM and an aggregate POM. Finally, we explored how to set up a simple multi-module to start playing with.

Maven is a great tool, but it's complex on its own. If we want to learn more details about Maven, we can look at the Sonatype Maven reference or Apache Maven guides. If we seek advanced usages of Maven's multi-modules set-up, we can look at how the Spring Boot project leverages its usage.

All the code examples used in this chapter are available over Github.

8: Maven dependencyManagement vs. dependencies Tags

1. Overview

In this chapter, we'll review two important Maven tags, *dependencyManagement* and *dependencies*.

These features are especially useful for multi-module projects.

We'll review the similarities and differences between the two tags, and then we'll look at some common mistakes developers make when using them that can cause confusion.

In general, we use the *dependencyManagement* tag to avoid repeating the *version* and *scope* tags when we define our dependencies in the *dependencies* tag. In this way, the required dependency is declared in a central POM file.

2.1. dependencyManagement

This tag consists of a *dependencies* tag, which in itself might contain multiple *dependency* tags. Each *dependency* is supposed to have at least three main tags: *groupId*, *artifactId*, and *version*. Let's see an example:

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.apache.commons</groupId>
            <artifactId>commons-lang3</artifactId>
            <version>3.12.0</version>
        </dependency>
    </dependencies>
</dependencyManagement>
```

The above code just declares the new artifact, *commons-lang3*, but it doesn't really add it to the project dependency resource list.

2.2. dependencies

This tag contains a list of dependency tags. Each dependency is supposed to have at least two main tags: groupId and artifactId.

Let's see a quick example:

```
<dependencies>
    <dependency>
        <groupId>org.apache.commons</groupId>
        <artifactId>commons-lang3</artifactId>
        <version>3.12.0</version>
    </dependency>
</dependencies>
```

The *version* and *scope* tags can be inherited implicitly if we've used the *dependencyManagement* tag before in the POM file:

```
<dependencies>
    <dependency>
        <groupId>org.apache.commons</groupId>
        <artifactId>commons-lang3</artifactId>
    </dependency>
</dependencies>
```

Both of these tags aim to declare some third-party or sub-module dependency. They complement each other.

In fact, we usually define the *dependencyManagement* tag once, preceding the *dependencies* tag. This is used in order to declare the dependencies in the POM file. **This declaration is just an announcement, and it doesn't really add the dependency to the project.**

Let's see an example of adding the JUnit library dependency:

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.13.2</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

As we can see in the above code, there's a *dependencyManagement* tag that in itself contains another *dependencies* tag.

Now let's see the other side of the code, which adds the actual dependency to the project:

```
<dependencies>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
    </dependency>
</dependencies>
```

The current tag is very similar to the previous one. Both of them will define a list of dependencies. Of course, there are small differences, which we'll cover soon.

The same *groupId* and *artifactId* tags are repeated in both code snippets, and there's a meaningful correlation between them; both of them refer to the same artifact.

As we can see, there isn't any version tag present in our later *dependency* tag. Surprisingly, it's valid syntax, and it can be parsed and compiled without any problem. The reason can be guessed easily; it'll use the version declared by *dependencyManagement*.

4.1. Structural Difference

As we covered earlier, the main structural difference between these two tags is the logic of inheritance. We define the version in the *dependencyManagement* tag, and then we can use the mentioned version without specifying it in the next *dependencies* tag.

4.2. Behavioral Difference

***dependencyManagement* is just a declaration, and it doesn't really add a dependency**. The declared *dependencies* in this section must be used later by the *dependencies* tag. It's only the *dependencies* tag that causes real dependency to happen. In the above example, the *dependencyManagement* tag won't add the *junit* library into any scope; it's just a declaration for the future *dependencies* tag.

Nearly all Maven-based open-source projects use this mechanism.

Now let's see an example from the Maven project itself. We can see the hamcrest-core dependency, which exists in the Maven project. It's declared first in the *dependencyManagement* tag, and then it's imported by the main *dependencies* tag:

```
dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.hamcrest</groupId>
            <artifactId>hamcrest-core</artifactId>
            <version>2.2</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

```
<dependencies>
    <dependency>
        <groupId>org.hamcrest</groupId>
        <artifactId>hamcrest-core</artifactId>
        <scope>test</scope>
    </dependency>
</dependencies>
```

A very common use case for this feature is a multi-module project.

Imagine we have a big project that consists of different modules. Each module has its own dependencies, and each developer might use a different version for the used dependencies. That could lead to a mesh of different artifact versions, which can also create difficult and hard-to-resolve conflicts.

The easy solution to this problem is using the *dependencyManagement* tag in the root POM file (usually called the "parent"), and then using the *dependencies* in the child's POM files (sub-modules) and even the parent module itself (if applicable).

So if we have a single module, does it make sense to use this feature or not? Although it's very useful in multi-module environments, it can also be a rule of thumb to obey it as a best practice even in a single-module project. This helps the project readability, and also makes it ready to extend to a multi-module project.

One common mistake is defining a dependency only in the *dependencyManagement* section, and not including it in the *dependencies* tag. As a result, we'd encounter compile or runtime errors, depending on the mentioned scope. Let's see an example:

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.apache.commons</groupId>
            <artifactId>commons-lang3</artifactId>
            <version>3.12.0</version>
        </dependency>
        ...
    </dependencies>
</dependencyManagement>
```

Imagine the above POM code snippet, and then suppose we're going to use this library in a sub-module source file:

```
import org.apache.commons.lang3.StringUtils;

public class Main {

    public static void main(String[] args) {
        StringUtils.isBlank(" ");
    }
}
```

This code won't compile because of the missing library. The compiler complains about an error:

```
[ERROR] Failed to execute goal compile (default-compile) on project sample-
module: Compilation failure
[ERROR] ~/sample-module/src/main/java/com/baeldung/Main.java:[3,32] package
org.apache.commons.lang3 does not exist
```

To avoid this error, it's enough to add the below *dependencies* tag to the sub-module POM file:

```
<dependencies>
    <dependency>
        <groupId>org.apache.commons</groupId>
        <artifactId>commons-lang3</artifactId>
    </dependency>
</dependencies>
```

8. Conclusion

In this chapter, we compared Maven's *dependencyManagement* and *dependencies* tags. Then we reviewed their similarities and differences, and learned how they work together.

As usual, the code for these examples is available over on GitHub.

9: Maven Packaging Types

1. Overview

The packaging type is an important aspect of any Maven project. It specifies the type of artifact the project produces. Generally, a build produces a *jar, war, pom*, or other executable.

Maven offers many default packaging types and also provides the flexibility to define a custom one.

In this chapter, we'll take a deep dive into Maven packaging types. First, we'll look at the build lifecycles in Maven. Then we'll discuss each packaging type, what they represent, and their effect on the project's lifecycle. Finally, we'll learn how to define a custom packaging type.

2. Default Packaging Types

Maven offers many default packaging types that include a *jar, war, ear, pom, rar, ejb*, and maven-plugin. **Each packaging type follows a build lifecycle that consists of phases**.

Usually, every phase is a sequence of goals and performs a specific task.

Different packaging types may have different goals in a particular phase. For example, in the package phase of a *jar* packaging type, *maven-jar-plugin's jar* goal is executed. Conversely, for a war project,

maven-war-plugin's war goal is executed in the same phase.

2.1. *jar*

Java archive, or jar, is one of the most popular packaging types. Projects with this packaging type produce a compressed zip file with the *jar* extension. It may include pure Java classes, interfaces, resources, and metadata files.

To begin with, let's look at some of the default goal-to-build-phase bindings for the *jar*:

- resources: resources
- compiler: compile
- resources: testResources
- compiler: testCompile
- surefire: test
- **jar: jar**
- install: install
- deploy: deploy
- install: install
- deploy: deploy

Without delay, let's define the packaging type of a jar project:

```
1.  <packaging>jar</packaging>
```

If nothing has been specified, Maven assumes the packaging type is a *jar*.

2.2. *war*

Simply put, a web application archive, or war, contains all files related to a web application. It may include Java servlets, JSPs, HTML pages, a deployment descriptor, and related resources. Overall, war has the same goal bindings as a *jar*, but with one exception; the package phase of the *war* has a different goal, which is war.

Without a doubt, *jar* and war are the most popular packaging types in the Java community. A detailed difference between these two might be an interesting read.

Let's define the packaging type of a web application:

```
1.  <packaging>war</packaging>
```

The other packaging types, *ejb*, *par*, and *rar*, also have similar lifecycles, but each has a different package goal:

```
ejb:ejb or par:par or rar:rar
```

2.3. *ear*

Enterprise application archive, or *ear*, is a compressed file that contains a J2EE application. It consists of one or more modules that can be web modules (packaged as a war file), EJB modules (packaged as a *jar* file), or both of them.

To put it differently, the ear is a superset of jars and *wars*, and requires an application server to run the application, whereas war requires only a web container or web server to deploy it. The aspects that distinguish a web server from an application server, and what those popular servers are in Java, are important concepts for a Java developer.

Let's define the default goal bindings for the *ear*:

- ear: generate-application-xml
- resources: *resources*
- **ear: *ear***
- install: *install*
- deploy: *deploy*

Here's how we can define the packaging type of such projects:

```
1.  <packaging>ear</packaging>
```

2.4. *pom*

Among all packaging types, *pom* is the simplest one. It helps to create aggregators and parent projects.

An aggregator, or multi-module project, assembles submodules coming from different sources. These submodules are regular Maven projects, and follow their own build lifecycles. The aggregator POM has all the references of submodules under the *modules* element.

A parent project allows us to define the inheritance relationship between POMs. The parent POM shares certain configurations, plugins, and dependencies, along with their versions. Most elements from the parent are inherited by its children, but exceptions include artifactId, name, and *prerequisites*.

This is because there are no resources to process and no code to compile or test. Therefore, the artifacts of pom projects generate themselves instead of any executable.

Let's define the packaging type of a multi-module project:

```
1.  <packaging>pom</packaging>
```

Such projects have the simplest lifecycle that consists of only two steps: *install* and *deploy*.

2.5. *maven-plugin*

Maven offers a variety of useful plugins; however, there might be cases when default plugins aren't sufficient enough. In those cases, the tool provides the flexibility to create a maven-plugin, according to the project's needs.

To create a plugin, set the packaging type of the project:

```
1.  <packaging>maven-plugin</packaging>
```

The *maven-plugin* has a lifecycle similar to jar's lifecycle, but with two exceptions:

- plugin: *descriptor* is bound to the generate-resources phase
- plugin: *addPluginArtifactMetadata* is added to the package phase

For this type of project, a *maven-plugin-api* dependency is required.

2.6. *ejb*

Enterprise Java Beans, or *ejb*, help to create scalable, distributed server-side applications. EJBs often provide the business logic of an application.

A typical EJB architecture consists of three components: Enterprise Java Beans (EJBs), the EJB container, and an application server.

Now let's define the packaging type of the EJB project:

```
1. <packaging>ejb</packaging>
```

The *ejb* packaging type also has a similar lifecycle as *jar* packaging, but with a different package goal. The package goal for this type of project is ejb:*ejb*.

The project, with *ejb* packaging type, requires a *maven-ejb-plugin* to execute lifecycle goals. Maven provides support for EJB 2 and 3. If no version is specified, then the default version 2 is used.

2.7. *rar*

Resource adapter, or *rar*, is an archive file that serves as a valid format for the deployment of resource adapters to an application server. Basically, it's a system-level driver that connects a Java application to an enterprise information system (EIS).

Here's the declaration of packaging type for a resource adapter:

```
1. <packaging>rar</packaging>
```

Every resource adapter archive consists of two parts: a *jar* file that contains source code, and a *ra.xml* that serves as a deployment descriptor.

Again, the lifecycle phases are the same as a *jar* or war packaging with one exception; **the *package* phase executes the *rar* goal that consists of a *maven-rar-plugin* to package the archives**.

So far, we've looked at the various packaging types that Maven offers by default. Now let's imagine we want our project to produce an artifact with a *.zip* extension. In this case, the default packaging types can't help us.

Maven also provides some more packaging types through plugins. With the help of these plugins, we can define a custom packaging type and its build lifecycle. Some of these types are:

- msi
- rpm
- tar
- tar.bz2
- tar.gz
- tbz
- zip

To define a custom type, we have to define its *packaging type* and phases in its lifecycle. For this, we'll create a *components.xml* file under the src/main/*resources/META-INF/plexus* directory:

```
<component>
   <role>org.apache.maven.lifecycle.mapping.LifecycleMapping</role>
   <role-hint>zip</role-hint>
   <implementation>org.apache.maven.lifecycle.mapping.
DefaultLifecycleMapping</implementation>
   <configuration>
      <phases>
         <process-resources>org.apache.maven.plugins:maven-resources-
plugin:resources</process-resources>
         <package>com.baeldung.maven.plugins:maven-zip-plugin:zip</
package>
         <install>org.apache.maven.plugins:maven-install-
plugin:install</install>
         <deploy>org.apache.maven.plugins:maven-deploy-plugin:deploy</
deploy>
      </phases>
   </configuration>
</component>
```

Up to this point, Maven doesn't know anything about our new packaging type and its lifecycle. To make it visible, we'll add the plugin in the *pom* file of the project and set *extensions* to *true*:

```
<plugins>
    <plugin>
        <groupId>com.baeldung.maven.plugins</groupId>
        <artifactId>maven-zip-plugin</artifactId>
        <extensions>true</extensions>
    </plugin>
</plugins>
```

Now the project will be available for a scan, and the system will look into the *plugins* and *components.xml* files, too.

In addition to all of these types, Maven offers a lot of other packaging types through external projects and plugins. For example, *nar* (native archive), *swf*, and *swc* are packaging types for the projects that produce Adobe Flash and Flex content. For such projects, we need a plugin that defines custom packaging and a repository that contains the plugin.

4. Conclusion

In this chapter, we looked at the various packaging types available in Maven. We also became familiar with what these types represent, and how they differ in their lifecycles. Finally, we learned how to define a custom packaging type and customize the default build lifecycle.

All code examples on Baeldung are built using Maven. Be sure to check out our various Maven configurations over on GitHub.

10: The settings.xml File in Maven

1. Overview

While using Maven, we keep most of the project-specific configuration in the *pom*.xml.

Maven provides a settings file, *settings.xml*, which allows us to specify which local and remote repositories it'll use. We can also use it to store settings that we don't want in our source code, such as credentials.

In this chapter, we'll learn how to use the *settings.xml*. We'll look at proxies, mirroring, and profiles. We'll also discuss how to determine the current settings that apply to our project.

The *settings.xml* file configures a Maven installation. It's similar to a *pom.xml* file, but is defined globally or per user.

Let's explore the elements we can configure in the *settings.xml* file. The main *settings* element of the *settings.xml* file can contain nine possible predefined child elements:

```
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0
https://maven.apache.org/xsd/settings-1.0.0.xsd">
    <localRepository/>
    <interactiveMode/>
    <offline/>
    <pluginGroups/>
    <servers/>
    <mirrors/>
    <proxies/>
    <profiles/>
    <activeProfiles/>
</settings>
```

2.1. Simple Values

Some of the top-level configuration elements contain simple values:

```
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0 https://
maven.apache.org/xsd/settings-1.0.0.xsd">
    <localRepository>${user.home}/.m2/repository</localRepository>
    <interactiveMode>true</interactiveMode>
    <offline>false</offline>
</settings>
```

The *localRepository* element points to the path of the system's local repository. **The local repository is where all the dependencies from our projects get cached**. The default is to use the user's home directory; however, we could change it to allow all logged-in users to build from a common local repository.

The *interactiveMode* flag defines if we allow Maven to interact with the user asking for input. This flag defaults to *true*.

The *offline* flag determines if the build system may operate in offline mode. This defaults to *false*; however, we can switch it to true in cases where the build servers cannot connect to a remote repository.

2.2. Plugin Groups

The *pluginGroups* element contains a list of child elements that specify a *groupId*. A *groupId* is the unique identifier of the organization that created a specific Maven artifact:

```
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0 https://
maven.apache.org/xsd/settings-1.0.0.xsd">
    <pluginGroups>
        <pluginGroup>org.apache.tomcat.maven</pluginGroup>
    </pluginGroups>
</settings>
```

Maven searches the list of plugin groups when a plugin is used without a groupId provided at the command line. The list contains the groups *org.apache.maven.plugins* and *org.codehaus.mojo* by default.

The *settings.xml* file defined above allows us to execute truncated Tomcat plugin commands:

```
mvn tomcat7:help
mvn tomcat7:deploy
mvn tomcat7:run
```

2.3. Proxies

We can configure a proxy for some or all of Maven's HTTP requests. The *proxies* element allows a list of child proxy elements, but **only one proxy can be active at a time:**

```
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0 https://
maven.apache.org/xsd/settings-1.0.0.xsd">
    <proxies>
        <proxy>
            <id>baeldung-proxy</id>
            <active>true</active>
            <protocol>http</protocol>
            <host>baeldung.proxy.com</host>
            <port>8080</port>
            <username>demo-user</username>
            <password>dummy-password</password>
            <nonProxyHosts>*.baeldung.com|*.apache.org</nonProxyHosts>
        </proxy>
    </proxies>
</settings>
```

We define the currently active proxy via the active flag. Then with the *nonProxyHosts* element, we specify which hosts aren't proxied. The delimiter used depends on the specific proxy server. The most common delimiters are pipe and comma.

2.4. Mirrors

Repositories can be declared inside a project *pom.xml*. This means that the developers sharing the project code get the right repository settings out of the box.

We can use mirrors in cases where we want to **define an alternative mirror for a particular repository**. This overrides what's in the *pom.xml*.

For example, we can force Maven to use a single repository by mirroring all repository requests:

```
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0 https://
maven.apache.org/xsd/settings-1.0.0.xsd">
    <mirrors>
        <mirror>
            <id>internal-baeldung-repository</id>
            <name>Baeldung Internal Repo</name>
            <url>https://baeldung.com/repo/maven2/</url>
            <mirrorOf>*</mirrorOf>
        </mirror>
    </mirrors>
</settings>
```

We may define only one mirror for a given repository, and Maven will pick the first match. Normally, **we should use the official repository** distributed worldwide via CDN.

2.5. Servers

Defining repositories in the project *pom.xml* is a good practice. However, we shouldn't put security settings, such as credentials, into our source code repository with the *pom.xml*. Instead, we **define this secure information in the *settings.xml* file**:

```
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0 https://
maven.apache.org/xsd/settings-1.0.0.xsd">
    <servers>
        <server>
            <id>internal-baeldung-repository</id>
            <username>demo-user</username>
            <password>dummy-password</password>
            <privateKey>${user.home}/.ssh/bael_key</privateKey>
            <passphrase>dummy-passphrase</passphrase>
            <filePermissions>664</filePermissions>
            <directoryPermissions>775</directoryPermissions>
            <configuration></configuration>
        </server>
    </servers>
</settings>
```

We should note that the *ID* of the server in the *settings.xml* needs to match the *ID* element of the repository mentioned in the *pom.xml*. The XML also allows us to use placeholders to pick up credentials from environment variables.

3. Profiles

The *profiles* element enables us to create multiple profile child elements differentiated by their *ID* child element. The *profile* element in the *settings.xml* is a truncated version of the same element available in the *pom.xml*.

It can contain only four child elements: *activation*, *repositories*, *pluginRepositories*, and *properties*. These elements configure the build system as a whole, instead of any particular project.

It's important to note that values from an active profile in *settings.xml* will **override any equivalent profile values in a *pom.xml*** or ***profiles.xml*** file. Profiles are matched by *ID*.

3.1. Activation

We can use profiles to modify certain values only under given circumstances. We can specify those circumstances using the *activation* element. Consequently, profile **activation occurs when all specified criteria are met**:

```
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0 https://
maven.apache.org/xsd/settings-1.0.0.xsd">
    <profiles>
        <profile>
            <id>baeldung-test</id>
            <activation>
                <activeByDefault>false</activeByDefault>
                <jdk>1.8</jdk>
                <os>
                    <name>Windows 10</name>
                    <family>Windows</family>
                    <arch>amd64</arch>
                    <version>10.0</version>
                </os>
                <property>
                    <name>mavenVersion</name>
                    <value>3.0.7</value>
                </property>
                <file>
                    <exists>${basedir}/activation-file.properties</
exists>
                    <missing>${basedir}/deactivation-file.properties</
missing>
                </file>
            </activation>
        </profile>
    </profiles>
</settings>
```

There are four possible activators, and not all of them need to be specified:

- *jdk*: activates based on the JDK version specified (ranges are supported)
- *os*: activates based on operating system properties
- *property*: activates the profile if Maven detects a specific property value
- *file*: activates the profile if a given filename exists or is missing

In order to check which profile will activate a certain build, we can use the Maven help plugin:

```
mvn help:active-profiles
```

The output will display the currently active profiles for a given project:

```
[INFO] --- maven-help-plugin:3.2.0:active-profiles (default-cli) @ core-java-
streams-3 ---

[INFO]
Active Profiles for Project 'com.baeldung.core-java-modules:core-java-streams-
3:jar:0.1.0-SNAPSHOT':

The following profiles are active:
 - baeldung-test (source: com.baeldung.core-java-modules:core-java-streams-
3:0.1.0-SNAPSHOT)
```

3.2. Properties

Maven properties can be thought of as named placeholders for a certain value. The values are **accessible within a *pom.xml* file using the *${property_name}* notation:**

```
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0 https://
maven.apache.org/xsd/settings-1.0.0.xsd">
    <profiles>
        <profile>
            <id>baeldung-test</id>
            <properties>
                <user.project.folder>${user.home}/baeldung-tutorials</
user.project.folder>
            </properties>
        </profile>
    </profiles>
</settings>
```

Four different types of properties are available in *pom.xml* files:

- Properties using the *env* prefix return an environment variable value, such as *${env.PATH}*.
- Properties using the *project* prefix return a property value set in the *project* element of the *pom.xml*, such as *${project.version}*.
- Properties using the *settings* prefix return the corresponding element's value from the *settings.xml*, such as *${settings.localRepository}*.
- We may reference all properties available via the *System.getProperties* method in Java directly, such as *${java.home}*.
- We may use properties set within a *properties* element without a prefix, such as *${junit.version}*.

3.3. Repositories

Remote repositories contain collections of artifacts that Maven uses to populate our local repository. Different remote repositories may be needed for particular artifacts. Maven **searches the repositories enabled under the active profile**:

```
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0 https://
maven.apache.org/xsd/settings-1.0.0.xsd">
    <profiles>
        <profile>
            <id>adobe-public</id>
            <repositories>
              <repository>
                  <id>adobe-public-releases</id>
                  <name>Adobe Public Repository</name>
                  <url>https://repo.adobe.com/nexus/content/groups/
public</url>
                  <releases>
                      <enabled>true</enabled>
                      <updatePolicy>never</updatePolicy>
                  </releases>
                  <snapshots>
                      <enabled>false</enabled>
                  </snapshots>
              </repository>
          </repositories>
        </profile>
    </profiles>
</settings>
```

We can use the *repository* element to enable only release or snapshots versions of artifacts from a specific repository.

3.4. Plugin Repositories

There are two standard types of Maven artifacts, dependencies and plugins. As Maven plugins are a special type of artifact, we may **separate plugin repositories from the others**:

```
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0 https://
maven.apache.org/xsd/settings-1.0.0.xsd">
    <profiles>
        <profile>
            <id>adobe-public</id>
            <pluginRepositories>
               <pluginRepository>
                  <id>adobe-public-releases</id>
                  <name>Adobe Public Repository</name>
                  <url>https://repo.adobe.com/nexus/content/groups/
public</url>
                  <releases>
                      <enabled>true</enabled>
                      <updatePolicy>never</updatePolicy>
                  </releases>
                  <snapshots>
                      <enabled>false</enabled>
                  </snapshots>
              </pluginRepository>
          </pluginRepositories>
        </profile>
    </profiles>
</settings>
```

Notably, the structure of the *pluginRepositories* element is very similar to the *repositories* element.

3.5. Active Profiles

The *activeProfiles* element contains child elements that refer to a specific profile *ID*. **Maven automatically activates any profile referenced here**:

```
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0 https://
maven.apache.org/xsd/settings-1.0.0.xsd">
    <activeProfiles>
        <activeProfile>baeldung-test</activeProfile>
        <activeProfile>adobe-public</activeProfile>
    </activeProfiles>
</settings>
```

In this example, every invocation of *mvn* is run as though we've added *-P baeldung-test,adobe-public* to the command line.

A *settings.xml* file is usually found in a couple of places:

- Global settings in Mavens home directory: *${maven.home}/conf/settings.xml*
- User settings in the user's home: *${user.home}/.m2/settings.xml*

If both files exist, their contents are merged. **Configurations from the user settings take precedence.**

4.1. Determine File Location

In order to determine the location of global and user settings, we can run Maven using the debug flag, and search for "*settings*" in the output:

```
$ mvn -X clean | grep "settings"

[DEBUG] Reading global settings from C:\Program Files (x86)\Apache\apache-
maven-3.6.3\bin\..\conf\settings.xml
[DEBUG] Reading user settings from C:\Users\Baeldung\.m2\settings.xml
```

4.2. Determine Effective Settings

We can use the Maven help plugin to **find out the contents of the combined global and user settings:**

```
mvn help:effective-settings
```

This describes the settings in XML format:

```
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0 https://
maven.apache.org/xsd/settings-1.0.0.xsd">
    <localRepository>C:\Users\Baeldung\.m2\repository</localRepository>
    <pluginGroups>
        <pluginGroup>org.apache.tomcat.maven</pluginGroup>
        <pluginGroup>org.apache.maven.plugins</pluginGroup>
        <pluginGroup>org.codehaus.mojo</pluginGroup>
    </pluginGroups>
</settings>
```

4.3. Override the Default Location

Maven allows us to override the location of the global and user settings via the command line:

```
$ mvn clean --settings c:\user\user-settings.xml --global-settings c:\user\
global-settings.xml
```

We can also use the shorter –s version of the same command:

```
$ mvn clean --s c:\user\user-settings.xml --gs c:\user\global-settings.xml
```

5. Conclusion

In this chapter, we **explored the configurations available in Maven's *settings.xml* file.**

We learned how to configure proxies, repositories, and profiles. Next, we looked at the difference between global and user settings files, and how to determine which are in use.

Finally, we looked at determining the effective settings used, and overriding default file locations.