# Maven Overview

Chaouki Bayoudhi

## Introduction

**Building a software project typically consists of such tasks 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, minimizing the risk of humans making errors while building the software manually and separating the work of compiling and packaging our code from that of code construction.**

**In this tutorial, we're going to explore this powerful tool for describing, building, and managing Java software projects using a central piece of information — the Project Object Model (POM) — that is written in XML.**

## 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

## Project Object Model (POM)

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:

# POM.XML file Example

```xml
<project>
    <modelVersion>4.0.0</modelVersion>
    <groupId>org.baeldung</groupId>
    <artifactId>org.baeldung</artifactId>
    <packaging>jar</packaging>
    <version>1.0-SNAPSHOT</version>
    <name>org.baeldung</name>
    <url>http://maven.apache.org</url>
    <dependencies>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.12</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
    <build>
        <plugins>
            <plugin>
            //...
            </plugin>
        </plugins>
    </build>
</project>
```

## 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 you specify which versions of external libraries (e.g. JARs) your project will use.**

## Dependencies

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

**This is a key feature of Maven and 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 your organization and beyond without the need for building artifact from source every time**

## Dependencies

In order to declare a dependency on an external library, you need to provide the groupId, artifactId, and the version of the library. Let's take a look at an example:

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-core</artifactId>
    <version>4.3.5.RELEASE</version>
</dependency>
```

As Maven processes the dependencies, it will download Spring Core library into your local Maven repository.

## Repositories

**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 a plug-in is available in the local repository, Maven uses it. Otherwise, it is downloaded from a central repository and stored in the local repository. The default central repository is Maven Central.**

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

## Repositories

```
<repositories>
    <repository>
        <id>JBoss repository</id>

<url>http://repository.jboss.org/nexus/content/gr
oups/public/</url>
    </repository>
</repositories>
```

- you can use multiple repositories in your projects.

## Properties

**Custom properties can help to make your pom.xml file easier to read and maintain. In the classic use case, you would use custom properties to define versions for your 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

**Let's see an example:**

```
<properties>
    <spring.version>4.3.5.RELEASE</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>
```

## Properties

Now if you want to upgrade Spring to a newer version, you 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

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


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

## 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>
```

## 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 you can change it at any time.**

## Using Profiles

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

## Using Profiles

```
<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>
```

## Using Profiles

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

```
mvn clean install -Pproduction
```

## ***Maven Build Lifecycles***

## Introduction

**Every Maven build follows a specified lifecycle. You 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.**

## 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**

## 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 is also an interesting article how to build an executable JAR on Baeldung 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.

## Plugins and Goals

**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) and execute all the phases necessary to install new archive (install).**

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