# Maven and its handling of build dependencies

Aristotelis, Kotsias
kotsias@kth.se

Sandro, Lockwall Rhodin
sandror@kth.se

May 2020

**Abstract**

Apache Maven is one of the most used management tools when it comes to Java and JEE-based projects. It offers a set of features, such as declarative dependency management and standardized directory structure, that simplify the build process and automate the project management task. This essay, attempts to present the features of Apache Maven and its handling of build dependencies. Moreover it aims to provide a conclusive answer as to why the way Maven handles its build dependencies leads to a better environment for developers.

# 1 Introduction

Software developers rely on their tools to develop, build, test and deploy their applications. Management tools, build tools, frameworks, debug tools, containers and developers' integrated development environments (IDEs) play an essential and vital role in the development and maintenance of quality software. Build tools automate the "translation" of the source code of a software program into executable or usable form. Specifically, build tool is a catch-all term that refers to the chain of processes that take place in order to get a piece of software set up and ready for use. A build tool is focused solely on preprocessing, compilation, packaging, testing, and distribution [1]. On the other hand, a project management tool provides a superset of features found in a build tool.

Apache Maven, commonly known as Maven, is an open source, standards-based project management framework that simplifies the building, testing, reporting, and packaging of projects [2] [3]. Maven incorporates some simple concepts but yet so fundamental that make it popular and one of the most used, if not the de facto, open source tool for building, managing, and automating Java and JEE-based projects in enterprises around the world [1]. As a management tool, Maven on top of the build tool features, offers the ability to run reports, facilitate communication between members of a working team and generate a website.

When you use Maven, you describe your project using a well-defined Project Object Model [2] and projects managed with it, have a metadata file known as POM (pom.xml), which includes detailed description of each project. Specifically, the pom.xml file contains information about the project such as configuration managements, version, dependencies, structure, resources and team members [4]. Maven requires some way of dealing with projects that depend on other software to function, these types of software are known as dependencies. They are defined as being essential software on which a project depends to either compile or run. A project could be built without any major, if any, dependencies at all, but for larger projects this is not often the case. In turn, when a larger project is built, dependencies must be checked for, and collected if not already available [5]. Maven does indeed provide functionality in regards to defining dependencies and allowing the setup of from where to collect missing dependencies.

In this essay, the features of Maven are discussed and explored with main focus on the Maven's dependency management.

# 2 Relation to DevOps

DevOps is about the practice of technologies and tools, rather than a technology, in order to deliver high software quality. DevOps revolves around the concepts of continuous integration, automation, continuous delivery, microservices and collaboration [6]. However, none of these are possible without the use

of the right software tools, tools like Apache Maven. Maven fits under different categories in the DevOps loop (since there are many differently structured ones), it does however generally go under the Build category, dealing with the automation of building software [7]. In Figure 1, the Build category has been split into two parts as Verification and/or Packaging and as such that's where we find Maven. Since Maven is a management tool, it provides automated verification and packaging of software by running a simple build command. This is something essential to the core DevOps concept of Automation.

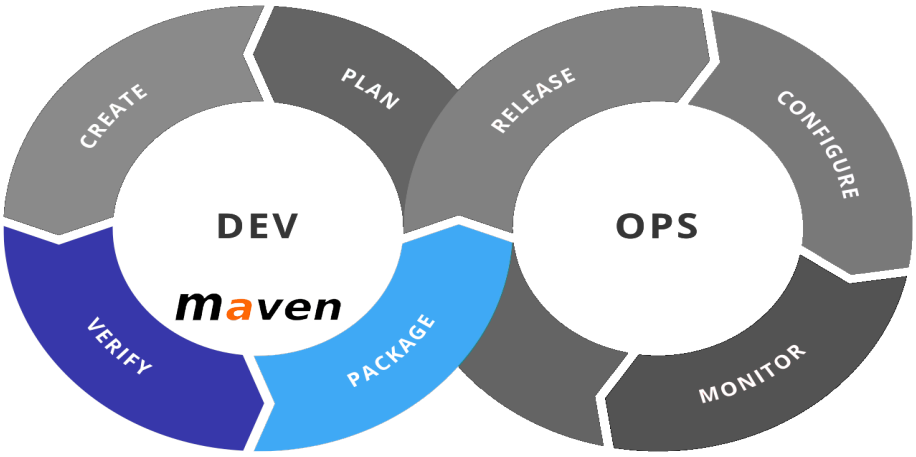


Figure 1: Maven in the DevOps Tool Chain
Source: https://commons.wikimedia.org/wiki/File:Devops-toolchain.svg
Attribution: Kharnagy / CC BY-SA
(https://creativecommons.org/licenses/by-sa/4.0) Changes were made.

## 3 Structure of the Maven Build dependencies

The Maven dependencies are setup in a POM file, one that is created when first setting up a Maven project. The POM file functions as an extensive guide when the Maven project is built to detail how it should be built, and for example if specified, download all necessary dependencies if they are not already available. The standard, generated POM file is filled with a great amount of information such as where the project is supposed to be built, but for the purpose of this essay only the dependency section will be treated. [8]

The structure of the dependency management in the POM file can be split into a number of parts. First, any and all dependencies are stated within a <dependencies>-tag and a </dependencies>-tag. While any individual dependency are similarly within <dependency>-tags. This is important for the structure of the POM file itself, but not necessarily the most useful information. [9][10]

The more interesting information is found under the <dependency>-tags, since there are a number of other tags contained within it. In particular, the <groupId>, <artifactId>, <version>, <type>, and <scope> tags [9].

The <groupId> tag is a unique project identifier. One that should follow all of the Java package naming conventions. For dependency work, it is used to identify a specific dependency that needs to be gathered from a specific project. [11]

The <artifactId> tag generally identifies what kind of specific jar file should be collected from the specific project identified with the <groupId> tag. In that sense it defines what actually should be collected from the project. [11]

<version> then specifies which version of the jar file should be collected. Since many projects carry many different versions of their builds as their projects develop over time. [2]

However, sometimes what you're collecting isn't a jar file. In that case you need to specify what kind of file you're trying to collect. This is done through the <type>-tag. Generally these are archives, but plugins can be used to define new types to be used. [2]

The <scope> tag indicates when the dependency should be used. Sometimes things only need to be used when compiling the software, other times only when running or testing the software. [9]

A standard dependency can be defined only using the <groupId> <artifactId> and <version> tags, as this is potentially enough information to identify the dependency [8][9]. This can be seen in Figure 2 where a project is defined and dependencies setup in particular with these three tags. It also shows how the <groupId> <artifactId> and <version> tags fit in under the <dependency> tag.

The importance of identifying useful information through the details of the tags should not be underestimated. In particular in regards to the version tag. If semantic versioning is used then the first number in the version tag should specify a major version with changes that may change compatibility with previous versions. The second, a minor version with new features with no changes that cause incompatibility, and the third a patch where bug fixes are made without breaking compatibility [12]. Developers should be careful, since although semantic

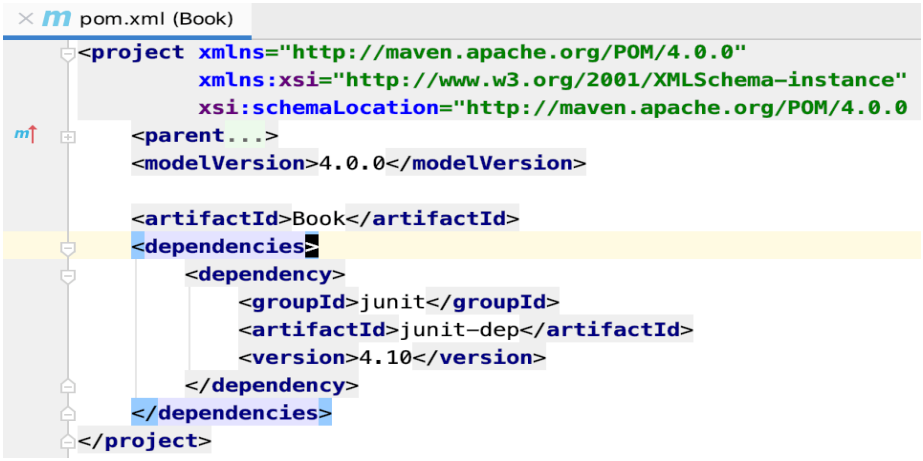


Figure 2: An example setup of a dependency section utilising the three standard dependency tags

Source:
https://www.jetbrains.com/help/idea/work-with-maven-dependencies.html

versioning is commonly used, the study *Semantic Versioning versus Breaking Changes: A Study of the Maven Repository* showed that out of 80589 libraries on the Maven Repository, 35.7% of the minor version updates and 23.8% of the patch ones included changes that would cause compatibility to break [13]. This indicating how common it is for developers to not use a standardised system for versioning and as such making it harder for other developers to know whether or not it is safe to update their projects' dependencies.

# 4 Dependencies and Repositories

Once the dependencies are setup, utilising at least the minimal number of tags, and the Maven project is built, one might ask oneself: How do all of the dependencies get resolved? There are two ways for dependencies for Maven to be resolved, both of which are setup through the POM file. Dependencies that have local files and dependencies that can be found on a remote repository. Local dependencies are rather simple, by default when creating the Maven project, a directory for the dependencies is created and any dependency put into that folder can be grabbed and used by the project. In the beginning however, there are no dependencies stored in the local directory so they have to be obtained somehow. [14]

Obtaining dependencies from non-local sources is primarily done through remote repositories. They can be setup utilising the <repositories> and <repository> tags respectively. Each remote repository needs at least an <id> tag and a <url> tag, providing an identification name and a link to the repository [14].

In the default POM file, a repository is already setup to handle the dependencies you have for your Maven project [8]. This is the Maven Central repository and it is commonly used by more than just Maven projects. It is for example one of the recommended repositories for build tools similar to Maven such as Gradle [15]. The repository works as a storage of some of the most commonly used dependencies, with over 4 million indexed jar files at the time of writing [16] [17].

Adding Maven dependencies is often made easier with access to the Central Maven Repository, using the officially recommended search engines (repository.apache.org, search.maven.org and mvnrepository.com) [18]. If you navigate to any of these search engines using a browser you can find and collect necessary jar files and quite likely also find the dependency string needed to be added under the <dependencies> tag to have Maven collect the dependency automatically when building.

This entire setup of how the dependencies are gathered and how Maven is connected to both local and remote repositories is exemplified in Figure 3 which shows exactly that with the remote repository being the standard one, the Maven Central Repository. This is also shown together with how Maven actually builds the project utilising these dependencies.

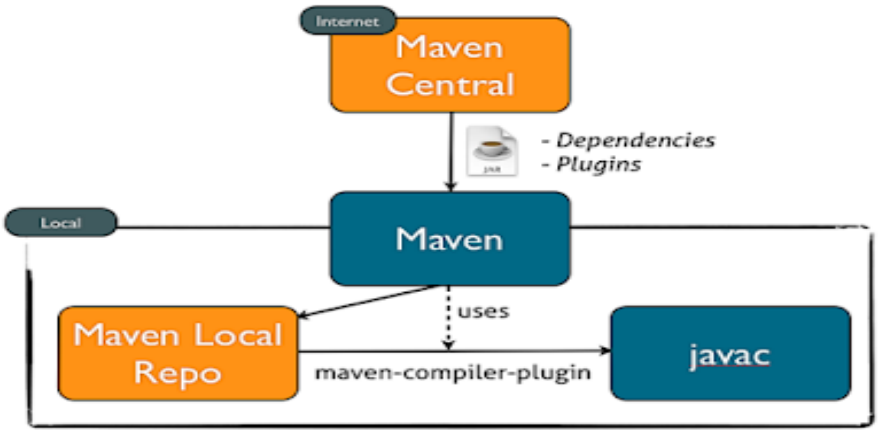


Figure 3: Maven using local and remote repositories
Source: https://javarevisited.blogspot.com/2019/02/10-tools-advanced-java-developers-should-know.html

Something to consider with dependencies that are from a remote source, is that should any dependency be removed from the source, any builds of the software utilising the dependency in a new environment (environment with no local dependencies) will inevitably break. If one is using the Maven Central Repository one might be able to replace the no longer working dependencies with perhaps newer, not compatibility breaking, versions and add them to the project. Although one might expect to be able to find the, old, desired dependency on the Maven Central Repository since it does function like an archive for old dependencies, some are simply removed from it due to for example stability issues. [19]

# 5 Why to use Maven

When starting a new project a considerable amount of time is spent on deciding the layout of the project and folder structure. These decisions can vastly vary among teams and projects something which can make it difficult for new developers to understand and adopt other team's projects or even make it difficult for experienced developers when they jump between projects and find what they are looking for. Maven addresses this problem by adapting the *convention over configuration* concept [1] [3]. Specifically, Maven provides sensible default behaviour for projects without requiring much unnecessary system configuration. For example, source code is assumed to be in *basedir/src/main/java* and resources are assumed to be in *basedir/src/main/resources*. This might seem trivial but yet it is a strong feature considering that in other similar tools, such as Ant, you have to define the locations of these directories in every sub-project whereas Maven requires zero effort. However, when using Maven, the developer is not restricted on this default behavior as Maven allows to customize the defaults in order to adapt to the requirements of the project.

The main benefit of using Maven is the dependency management that offers during the development of Java applications. For small and easy projects, dependencies configuration is not hard and rarely anyone has to do anything with it. However, when a larger and more complicated project is developed, managing the dependencies manually by hand is rather trivial without the use of a tool such as Maven. Specifically, Maven offers tools that deal with dependency management and offers the opportunity to the user to control the version used in transitive dependencies, analyze the dependency tree and control the converge across modules. Moreover, due to Maven's central repository, all developers use the same jar dependencies and also the source distributions' size is reduced due to the fact that the needed jar dependencies can be pulled from the central repository [1].

Furthermore, Maven deals with transitive dependencies meaning that when an external dependency is included in a project, the developer does not need to download all the other dependencies that the external dependency is dependent

on [9]. This is because Maven will automatically download all the necessary jar files that are needed and also takes care of downloading the right versions. This is a powerful feature of Maven as it is time efficient and build errors that could occur due to incorrect versions of dependent libraries are eliminated.

# 6 Discussion

As it has already been described, Maven is a powerful management tool that automates project's build which is based on POM. The pom.xml file contains information about the project configuration such as source directory and dependencies. The simplicity of how the POM file is structured makes Maven very user friendly and an easy tool to use.

The dependency handling is rather simple, with not too many tags to worry about when constructing the dependency. An added benefit of utilising a Maven project with the Maven Central Repository is that the Maven Central Repository can provide dependency information to be added under the <dependencies> tag. With this we argue that there aren't necessarily any real drawbacks with Maven and its handling of dependencies, other than perhaps the potential for a user to fail to access the correct dependency if they are manually inserting the needed dependency information.

We also argue that Maven is a useful tool for work within the sphere of DevOps since it allows for automation of essential features. Maven also has good support from a community uploading dependencies to things such as the Maven Central Repository and its very simple to understand POM file allows for an easier time using Maven for project management purposes.

# 7 Conclusions

This essay shows that Maven greatly simplifies the build process and automates project management tasks. It offers a strong default behavior with sensible configurations and whereas developers have to spent minimum amount of time and least effort to either set up a project or understand and adopt other teams' project. Moreover, Maven's declarative dependency management through the POM file, provides a convenient way for the developers to download the necessary dependencies and keep track of their versions as they are used in the project. This paired with how simple it is to set up dependencies manually through the tag system and how Maven recommends search engines to help find dependencies allows for an easy to use service. Although, dependencies come with risk of build failures in case of missing dependencies, the strong support of large repositories, such as Maven Central Repository, helps out mitigating this problem. In conclusion, Maven is a project management tool that is both easy to use and well supported.

# References

[1] T. O'Brien, *Maven: The Definitive Guide*. O'Reilly Media, 2008.

[2] Maven. https://maven.apache.org/ref/3.6.1/maven-model/maven.htmldependency. Last Accessed: 2020-04-07.

[3] S. B. Balaji Varanasi, *Introducing Maven*. Apress, Berkeley, CA, 2014.

[4] L. Hernández and H. Costa, "Identifying similarity of software in apache ecosystem – an exploratory study," in *2015 12th International Conference on Information Technology - New Generations*, pp. 397–402.

[5] D. Dilshan. Do you know maven ? a dependency manager or a build tool ? what is pom ? https://medium.com/@dulajdilshan/do-you-know-maven-a-dependency-manager-or-a-build-tool-what-is-pom-bd7dd8b43e80. Last Accessed: 2020-04-25.

[6] (2020, Apr.) What is DevOps? - Amazon Web Services (AWS) @ONLINE. https://aws.amazon.com/devops/what-is-devops/. Library Catalog: aws.amazon.com.

[7] S. Kulshrestha. Devops life cycle - explore about each phase in devops life cycle. https://medium.com/edureka/devops-lifecycle-8412a213a654. Last Accessed: 2020-05-27.

[8] Introduction to the pom. http://maven.apache.org/guides/introduction/introduction-to-the-pom.html. Last Accessed: 2020-04-09.

[9] Introduction to the dependency mechanism. http://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html. Last Accessed: 2020-04-07.

[10] Maven – dependency management. https://howtodoinjava.com/maven/maven-dependency-management/. Last Accessed: 2020-04-09.

[11] Guide to naming conventions on groupid, artifactid, and version. http://maven.apache.org/guides/mini/guide-naming-conventions.html. Last Accessed: 2020-04-07.

[12] T. Preston-Werner, "Semantic versioning 2.0.0," https://semver.org/.

[13] S. Raemaekers, A. van Deursen, and J. Visser, "Semantic versioning and impact of breaking changes in the maven repository," vol. 129, pp. 140–158. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0164121216300243

[14] Introduction to repositories. https://maven.apache.org/guides/introduction/introduction-to-repositories.html. Last Accessed: 2020-04-17.

[15] Declaring repositories. https://docs.gradle.org/current/userguide/declaring_repositories.html. Last Accessed: 2020-04-17.

[16] Central repository. https://mvnrepository.com/repos/central. Last Accessed: 2020-04-21.

[17] Maven - repositories. https://www.tutorialspoint.com/maven/maven_repositories.htm. Last Accessed: 2020-04-21.

[18] Frequently asked technical questions. http://maven.apache.org/general.html. Last Accessed: 2020-04-21.

[19] R. Kikas, G. Gousios, M. Dumas, and D. Pfahl, "Structure and evolution of package dependency networks," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pp. 102–112.