Apache Maven

From Wikipedia, the free encyclopedia

Maven is a build automation tool used primarily for Java projects. The word *maven* means 'accumulator of knowledge' in Yiddish.[3] Maven addresses two aspects of building software: First, it describes how software is built, and second, it describes its dependencies. Contrary to preceding tools like Apache Ant, it uses conventions for the build procedure, and only exceptions need to be written down. An XML file describes the software project being built, its dependencies on other external modules and components, the build order, directories, and required plug-ins. It comes with pre-defined targets for performing certain well-defined tasks such as compilation of code and its packaging. Maven dynamically downloads Java libraries and Maven plug-ins from one or more repositories such as the Maven 2 Central Repository, and stores them in a local cache.[4] This local cache of downloaded artifacts can also be updated with artifacts created by local projects. Public repositories can also be updated.

Apache Maven

maven

Developer(s)	Apache Software Foundation
Initial release	(1 beta) March 2002; 13 years ago
Stable release	3.3.3[1] / April 28, 2015[2]
Development status	Active
Written in	Java
Operating system	Cross-platform
Type	Build tool
License	Apache License 2.0
Website	maven.apache.org

Maven can also be used to build and manage projects written in C#, Ruby, Scala, and other languages. The Maven project is hosted by the Apache Software Foundation, where it was formerly part of the Jakarta Project.

Maven is built using a plugin-based architecture that allows it to make use of any application controllable through standard input. Theoretically, this would allow anyone to write plugins to interface with build tools (compilers, unit test tools, etc.) for any other language. In reality, support and use for languages other than Java has been minimal. Currently a plugin for the .NET framework exists and is maintained,[5] and a C/C++ native plugin is maintained for Maven 2.[6]

Alternative technologies like gradle and sbt as build tools do not rely on XML, but keep the key concepts Maven introduced. With Apache Ivy, a dedicated dependency manager was developed as well that also supports Maven repositories.

Contents [hide]

Example [edit]

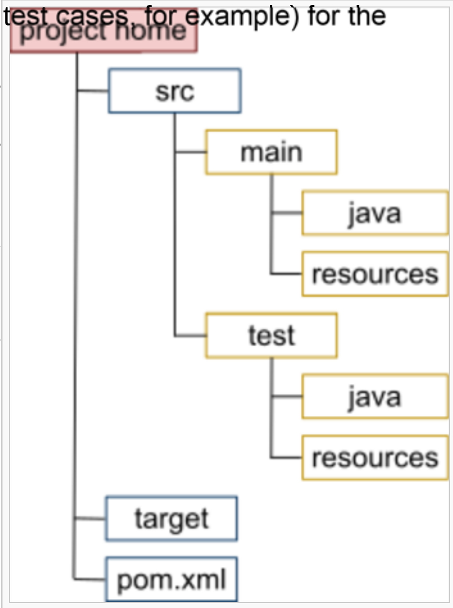
Maven projects are configured using a Project Object Model, which is stored in a `pom.xml`-file. Here's a minimal example:

```xml
<project>
  <!-- model version is always 4.0.0 for Maven 2.x POMs -->
  <modelVersion>4.0.0</modelVersion>

  <!-- project coordinates, i.e. a group of values which
       uniquely identify this project -->

  <groupId>com.mycompany.app</groupId>
  <artifactId>my-app</artifactId>
  <version>1.0</version>

  <!-- library dependencies -->

  <dependencies>
    <dependency>

      <!-- coordinates of the required library -->

      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>

      <!-- this dependency is only used for running and compiling tests -->

      <scope>test</scope>

    </dependency>
  </dependencies>
</project>
```

This POM only defines a unique identifier for the project (*coordinates*) and its dependency on the JUnit framework. However, that is already enough for building the project and running the unit tests associated with the project. Maven accomplishes this by embracing the idea of Convention over Configuration, that is, Maven provides default values for the project's configuration. The directory structure of a normal idiomatic Maven project has the following directory entries:

Directory name	Purpose
project home	Contains the pom.xml and all subdirectories.
src/main/java	Contains the deliverable Java sourcecode for the project.
src/main /resources	Contains the deliverable resources for the project, such as property files.

src/test/java	Contains the testing Java sourcecode (JUnit or TestNG test cases, for example) for the project.
src/test/resources	Contains resources necessary for testing.



The Maven software tool auto-generated this directory structure for a Java project.

Then the command

```
mvn package
```

will compile all the Java files, run any tests, and package the deliverable code and resources into `target/my-app-1.0.jar` (assuming the artifactId is my-app and the version is 1.0.)

Using Maven, the user provides only configuration for the project, while the configurable plug-ins do the actual work of compiling the project, cleaning target directories, running unit tests, generating API documentation and so on. In general, users should not have to write plugins themselves. Contrast this with Ant and make, in which one writes imperative procedures for doing the aforementioned tasks.

Concepts [edit]

Project Object Model [edit]

A Project Object Model (POM) provides all the configuration for a single project. General configuration covers the project's name, its owner and its dependencies on other projects. One can also configure individual phases of the build process, which are implemented as plugins. For example, one can configure the compiler-plugin to use Java version 1.5 for compilation, or specify packaging the project even if some unit tests fail.

Larger projects should be divided into several modules, or sub-projects, each with its own POM. One can then write a root POM through which one can compile all the modules with a single command. POMs can also inherit configuration from other POMs. All POMs inherit from the Super POM[7] by default. The Super POM provides default configuration, such as default source directories, default plugins, and so on.

Plugins [edit]

Most of Maven's functionality is in plugins. A plugin provides a set of goals that can be executed using the following syntax:

```
mvn [plugin-name]:[goal-name]
```

For example, a Java project can be compiled with the compiler-plugin's compile-goal[8] by running `mvn compiler:compile`.

There are Maven plugins for building, testing, source control management, running a web server, generating Eclipse project files, and much more.[9] Plugins are introduced and configured in a <plugins>-section of a `pom.xml` file. Some basic plugins are included in every project by default, and they have sensible default settings.

However, it would be cumbersome if the archetypical build sequence of building, testing and packaging a software project required running each respective goal manually:

```
mvn compiler:compile
mvn surefire:test
mvn jar:jar
```

Maven's lifecycle concept handles this issue.

Plugins are the primary way to extend Maven. Developing a Maven plugin can be done by extending the org.apache.maven.plugin.AbstractMojo class. Example code and explanation for a Maven plugin to create a cloud-based virtual machine running an application server is given in the article *Automate development and management of cloud virtual machines*.[10]

Build lifecycles [edit]

Build lifecycle is a list of named *phases* that can be used to give order to goal execution. One of Maven's standard lifecycles is the *default lifecycle*, which includes the following phases, in this order:[11]

```
 1  validate
 2  generate-sources
 3  process-sources
 4  generate-resources
 5  process-resources
 6  compile
 7  process-test-sources
 8  process-test-resources
 9  test-compile
10  test
11  package
12  install
13  deploy
```

Goals provided by plugins can be associated with different phases of the lifecycle. For example, by default, the goal "compiler:compile" is associated with the "compile" phase, while the goal "surefire:test" is associated with the "test" phase. Consider the following command:

```
mvn test
```

When the preceding command is executed, Maven runs all goals associated with each of the phases up to and including the "test" phase. In such a case, Maven runs the "resources:resources" goal associated with the "process-resources" phase, then "compiler:compile", and so on until it finally runs the "surefire:test" goal.

Maven also has standard phases for cleaning the project and for generating a project site. If cleaning were part of the default lifecycle, the project would be cleaned every time it was built. This is clearly undesirable, so cleaning has been given its own lifecycle.

Standard lifecycles enable users new to a project the ability to accurately build, test and install every Maven project by issuing the single command:

```
mvn install
```

By default, Maven packages the POM file in generated JAR and WAR files. Tools like diet4j[12] can use this information to recursively resolve and run Maven modules at run-time without requiring an "uber"-jar that contains all project code.

Dependencies [edit]

A central feature in Maven is dependency management. Maven's dependency-handling mechanism is organized around a coordinate system identifying individual artifacts such as software libraries or modules. The POM example above references the JUnit coordinates as a direct dependency of the project. A project that needs, say, the Hibernate library simply has to declare Hibernate's project coordinates in its POM. Maven will automatically download the dependency and the dependencies that Hibernate itself needs (called transitive dependencies) and store them in the user's local repository. Maven 2 Central Repository[4] is used by default to search for libraries, but one can configure the repositories to be used (e.g., company-private repositories) within the POM.

There are search engines such as The Central Repository Search Engine[13] which can be used to find out coordinates for different open-source libraries and frameworks.

Projects developed on a single machine can depend on each other through the local repository. The local repository is a simple folder structure that acts both as a cache for downloaded dependencies and as a centralized storage place for locally built artifacts. The Maven command `mvn install` builds a project and places its binaries in the local repository. Then other projects can utilize this project by specifying its coordinates in their POMs.

Maven compared with Ant [edit]

The fundamental difference between Maven and Ant is that Maven's design regards all projects as having a certain structure and a set of supported task work-flows (e.g., getting resources from source control, compiling the project, unit testing, etc.). While most software projects in effect support these operations and actually do have a well-defined structure, Maven requires that this structure and the operation implementation details be defined in the POM file. Thus, Maven relies on a convention on how to define projects and on the list of work-flows that are generally supported in all projects.[14]

This design constraint resembles the way that an IDE handles a project, and it provides many benefits, such as a succinct project definition, and the possibility of automatic integration of a Maven project with other development tools such as IDEs, build servers, etc.

But one drawback to this approach is that Maven requires a user to first understand what a project is from the Maven point of view, and how Maven works with projects, because what happens when one executes a phase in Maven is not immediately obvious just from examining the Maven project file. In many cases, this required structure is also a significant hurdle in migrating a mature project to Maven, because it is usually hard to adapt from other approaches.

In Ant, projects do not really exist from the tool's technical perspective. Ant works with XML build scripts defined in one or more files. It processes targets from these files and each target executes tasks. Each task performs a technical operation such as running a compiler or copying files around. Targets are executed primarily in the order given by their defined dependency on other targets. Thus, Ant is a tool that chains together targets and executes them based on inter-dependencies and other Boolean conditions.

The benefits provided by Ant are also numerous. It has an XML language optimized for clearer definition of what each task does and upon what it depends. Also, all the information about what will be executed by an Ant target can be found in the Ant script.

A developer not familiar with Ant would normally be able to determine what a simple Ant script does just by examining the script. This is not usually true for Maven.

However, even an experienced developer who is new to a project using Ant cannot infer what the higher level structure of an Ant script is and what it does without examining the script in detail. Depending on the script's complexity, this can quickly become a daunting challenge. With Maven, a developer who previously worked with other Maven projects can quickly examine the structure of a never-before-seen Maven project and execute the standard Maven work-flows against it while already knowing what to expect as an outcome.

It is possible to use Ant scripts that are defined and behave in a uniform manner for all projects in a working group or an organization. However, when the number and complexity of projects rises, it is also very easy to stray from the initially desired uniformity. With Maven this is less of a problem because the tool always imposes a certain way of doing things.

Note that it is also possible to extend and configure Maven in a way that departs from the Maven way of doing things. This is particularly true for Maven 2 and newer releases, such as Mojos or more formally, plugins and custom [15] project directory structures.

IDE integration [edit]

Add-ons to several popular Integrated Development Environments exist to provide integration of Maven with the IDE's build mechanism and source editing tools, allowing Maven to compile projects from within the IDE, and also to set the classpath for code completion, highlighting compiler errors, etc. Examples of popular IDEs supporting development with Maven include:

- Eclipse
- NetBeans
- IntelliJ IDEA
- JBuilder
- JDeveloper (version 11.1.2)
- MyEclipse

These add-ons also provide the ability to edit the POM or use the POM to determine a project's complete set of dependencies directly within the IDE.

Some built-in features of IDEs are forfeited when the IDE no longer performs compilation. For example, Eclipse's JDT has the ability to recompile a single java source file after it has been edited. Many IDEs work with a flat set of projects instead of the hierarchy of folders preferred by Maven. This complicates the use of SCM systems in IDEs when using Maven.[16][17][18]

History [edit]

Maven, created by Takari's Jason van Zyl, began as a subproject of Apache Turbine in 2002. In 2003, it was voted on and accepted as a top level Apache Software Foundation project. In July 2004, Maven's release was the critical first milestone, v1.0. Maven 2 was declared v2.0 in October 2005 after about six months in beta cycles. Maven 3.0 was released in October 2010 being mostly backwards compatible with Maven 2.

Maven 3 [edit]

Maven 3.0 information began trickling out in 2008. After eight alpha releases, the first beta version of Maven 3.0 was released in April 2010. Maven 3.0 has reworked the core Project Builder infrastructure resulting with the POM's file-based representation being decoupled from its in-memory object representation. This has expanded the possibility for Maven 3.0 add-ons to leverage non-XML based project definition files. Languages suggested include Ruby (already in private prototype by Jason van Zyl), YAML, and Groovy.

Special attention was given to ensuring backward compatibility of Maven 3 to Maven 2. For most projects, upgrading to Maven 3 will not require any adjustments of their project structure. The first beta of Maven 3 saw the introduction of a parallel build feature which leverages a configurable number of cores on a multi-core machine and is especially suited for large multi-module projects.

See also [edit]

Free software portal

- Apache Continuum, a continuous integration server which integrates tightly with Maven
- Apache Jelly, a tool for turning XML into executable code
- Apache Ivy, alternative dependency management tool for Java
- Gradle, a build tool based on convention over configuration
- Hudson
- Jenkins
- List of build automation software

References [edit]

1. ^ Maven 3.3.3
2. ^ Maven Releases History
3. ^ Suereth, Joshua and Farwell, Matthew. *SBT in Action*. Manning Publications. 2015. ISBN 9781617291272
4. ^ *a b* Maven 2 Central Repository
5. ^ .NET Maven Plugin
6. ^ maven-native C/C++ plugin and maven-nar C/C++ plugin
7. ^ Super POM
8. ^ Maven Compiler Plugin
9. ^ Maven - Available Plugins
10. ^ Amies, Alex; Zou P X; Wang Yi S (29 Oct 2011). "Automate development and management of cloud virtual machines". *IBM developerWorks* (IBM).
11. ^ Maven Build Lifecycle Reference
12. ^ diet4j - No more JAR bloat; load your Maven projects as needed at run-time
13. ^ The Central Repository Search Engine,
14. ^ "Maven: The Complete Reference". Sonatype. Retrieved 11 April 2013.
15. ^ "Maven Build Customization". Packt. Retrieved 6 November 2014.
16. ^ Eclipse plugins for Maven
17. ^ IntelliJ IDEA - Ant and Maven support
18. ^ Best Practices for Apache Maven in NetBeans 6.x

Further reading [edit]

- A free online book - O'Brien, Tim et al. "Maven: The Complete Reference". *Sonatype.com*. Sonatype. Retrieved 15 March 2013.
- The anteater book: *Maven: The Definitive Guide*. Sonatype Company. O'Reilly Media, Inc. 2009. p. 470. ISBN 9780596551780. Retrieved 2013-04-17.
- A printed book - Van Zyl, Jason (2008-10-01), *Maven: Definitive Guide* (first ed.), O'Reilly Media, p. 468, ISBN 0-596-51733-5

- "Running JUnit tests from Maven2". *JUnit in Action* (2nd ed.). Manning Publications. 2011. pp. 152–168. ISBN 978-1-935182-02-3.
- *Maven Build Customization*. Packt. 2013. pp. 1–250. ISBN 9781783987221.
- *Mastering Apache Maven 3*. Packt. 2014. p. 298. ISBN 9781783983865.

External links [edit]

- Official website
- The Maven 2 tutorial: A practical guide for Maven 2 users - tutorial at Codehaus.org
- Building Web Applications with Maven 2
- The Maven 2 POM demystified - article at JavaWorld
- Maven for PHP