DZone Refcardz

Apache Maven 2

By Matthew McCullough

ABOUT APACHE MAVEN

Maven is a comprehensive project information tool, whose most common application is building Java code. Maven is often considered an alternative to Ant, but as you'll see in this Refcard, it offers unparalleled software lifecycle management, providing a cohesive suite of verification, compilation, testing, packaging, reporting, and deployment plugins.

Maven is receiving renewed recognition in the emerging development space for its convention over configuration approach to builds. This Refcard aims to give JVM platform developers a range of basic to advanced execution commands, tips for debugging Mavenized builds, and a clear introduction to the "Maven vocabulary".

Interoperability and Extensibility

New Maven users are pleasantly surprised to find that Maven offers easy-to-write custom build-supplementing plugins, reuses any desired aspect of Ant, and can compile native C, C++, and .NET code in addition to its strong support for Java and JVM languages and platforms, such as Scala, JRuby, Groovy and Grails.

Hot Tip: All things Maven can be found at http://maven.apache.org

THE MVN COMMAND

Maven supplies a Unix shell script and MSDOS batch file named `mvn` and `mvn.bat` respectively. This command is used to start all Maven builds. Optional parameters are supplied in a space-delimited fashion. An example of cleaning and packaging a project, then running it in a Jetty servlet container, yet skipping the unit tests, reads as follows:

```
mvn clean package jetty:run –Dmaven.test.skip
```

PROJECT OBJECT MODEL

The world of Maven revolves around metadata files named pom.xml. A file of this name exists at the root of every Maven project and defines the plugins, paths and settings that supplement the Maven defaults for your project.

Basic pom.xml Syntax

The smallest valid pom.xml, which inherits the default artifact type of "jar", reads as follows:

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.ambientideas</groupId>
  <artifactId>barestbones</artifactId>
  <version>1.0-SNAPSHOT</version>
</project>
```

Super POM

The Super POM is a virtual pom.xml file that ships inside the core Maven JARs, and provides numerous default settings. All projects automatically inherit from the Super POM, much like the Object super class in Java. Its contents can be viewed in one of two ways:

View Super POM via SVN

Open the following SVN viewing URL in your web browser:

```
http://svn.apache.org/repos/asf/maven/components/branches/maven-2.1.x/pom.xml
```

View Super POM via effective-pom

Run the following command in a directory that contains the most minimal Maven project pom.xml, listed above.

```
mvn help:effective-pom
```

Multi-module Projects

Maven showcases exceptional support for componentization via its concept of multi-module builds. Place sub-projects in sub-folders beneath your top level project and reference each with a module tag. To build all sub projects, just execute your normal `mvn` command and goals from a prompt in the top-most directory.

```
<project>
  <!-- ... -->
  <packaging>pom</packaging>
  <modules>
    <module>servlets</module>
    <module>ejbs</module>
    <module>ear</module>
  </modules>
</project>
```

Artifact Vector

Each Maven project produces an element, such as a JAR, WAR or EAR, uniquely identified by a composite of fields known as groupId, artifactId, packaging, version and scope. This vector of fields uniquely distinguishes a Maven artifact from all others.

Many Maven reports and plugins print the details of a specific artifact in this colon separated fashion:

```
groupid:artifactid:packaging:version:scope
```

An example of this output for the core Spring JAR would be:

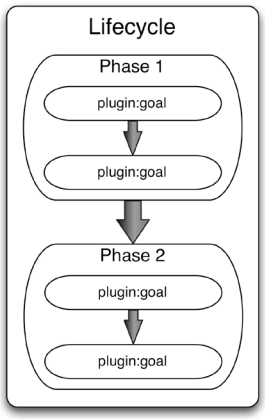
```
org.springframework:spring:jar:2.5.6:compile
```

EXECUTION GROUPS

Maven divides execution into four nested hierarchies. From most-encompassing to most-specific, they are: Lifecycle, Phase, Plugin, and Goal.

Lifecycles, Phases, Plugins and Goals

Maven defines the concept of language-independent project build flows that model the steps that all software goes through during a compilation and deployment process.



`Lifecycles` represent a well-recognized flow of steps (Phases) used in software assembly.

Each step in a `lifecycle` flow is called a `phase`. Zero or more plugin goals are bound to a phase.

A `plugin` is a logical grouping and distribution (often a single JAR) of related goals, such as JARing.

A `goal`, the most granular step in Maven, is a single executable task within a plugin. For example, discrete goals in the jar `plugin` include packaging the jar (jar:jar), signing the jar (jar:sign), and verifying the signature (jar:sign-verify).

Executing a Phase or Goal

At the command prompt, either a phase or a plugin goal can be requested. Multiple phases or goals can be specified and are separated by spaces.

If you ask Maven to run a specific plugin goal, then only that goal is run. This example runs two plugin goals: compilation of code, then JARing the result, skipping over any intermediate steps.

```
mvn compile:compile jar:jar
```

Conversely, if you ask Maven to execute a `phase`, all phases and bound plugin goals up to that point in the lifecycle are also executed. This example requests the deploy lifecycle phase, which will also execute the verification, compilation, testing and packaging phases.

```
mvn deploy
```

Online and Offline

During a build, Maven attempts to download any uncached referenced artifacts and proceeds to cache them in the `~/.m2/repository` directory on Unix, or in the `%USERPROFILE%/.m2/repository` directory on Windows.

To prepare for compiling offline, you can instruct Maven to download all referenced artifacts from the Internet via the command:

```
mvn dependency:go-offline
```

If all required artifacts and plugins have been cached in your local repository, you can instruct Maven to run in offline mode with a simple flag:

```
mvn <phase or goal> -o
```

Built-in Maven Lifecycles

Maven ships with three lifecycles; clean, default, and site. Many of the phases within these three lifecycles are bound to a sensible plugin goal.

Hot Tip The official lifecycle reference, which extensively lists all the default bindings, can be found at http://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html

The `clean` lifecycle is simplistic in nature. It deletes all generated and compiled artifacts in the output directory.

Clean Lifecycle	
Lifecycle Phase	**Purpose**
`pre-clean`	
`clean`	Remove all generated and compiled artifacts in preperation for a fresh build.
`post-clean`	

The `default` lifecycle defines the most commonly used phases for building an application, ranging from compilation of the code to installation of the completed artifacts, such as a JAR, into a remote Maven repository.

Default Lifecycle	
Lifecycle Phase	**Purpose**
`validate`	Cross check that all elements necessary for the build are correct and present.
`initialize`	Set up and bootstrap the build process.
`generate-sources`	Generate dynamic source code
`process-sources`	Filter, sed and copy source code
`generate-resources`	Generate dynamic resources
`process-resources`	Filter, sed and copy resources files.
`compile`	Compile the primary or mixed language source files.
`process-classes`	Augment compiled classes, such as for code-coverage instrumentation.
`generate-test-sources`	Generate dynamic unit test source code.

process-test-sources	Filter, sed and copy unit test source code.
generate-test-resources	Generate dynamic unit test resources.
process-test-resources	Filter, sed and copy unit test resources.
test-compile	Compile unit test source files
test	Execute unit tests
prepare-package	Manipulate generated artifacts immediately prior to packaging. (Maven 2.1 and above)
package	Bundle the module or application into a distributable package (commonly, JAR, WAR, or EAR).
pre-integration-test	
integration-test	Execute tests that require connectivity to external resources or other components
post-integration-test	
verify	Inspect and cross-check the distribution package (JAR, WAR, EAR) for correctness.
install	Place the package in the user's local Maven repository.
deploy	Upload the package to a remote Maven repository

The `site` lifecycle generates a project information web site, and can deploy the artifacts to a specified web server or local path.

Site Lifecycle	
Lifecycle Phase	**Purpose**
pre-site	Cross check that all elements necessary for the build are correct and present.
site	Generate an HTML web site containing project information and reports.
post-site	
site-deploy	Upload the generated website to a web server

Default Goal

The default goal codifies the author's intended usage of the build script. Only one goal or lifecycle can be set as the default. The most common default goal is `install`.

```
<project>
    [...]
    <build>
        <defaultGoal>install</defaultGoal>
    </build>
    [...]
</project>
```

HELP

Help for a Plugin

Lists all the possible goals for a given plugin and any associated documentation.

```
help:describe -Dplugin=<pluginname>
```

Help for POMs

To view the composite pom that's a result of all inherited poms:

```
mvn help:effective-pom
```

Help for Profiles

To view all profiles that are active from either manual or automatic activation:

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

DEPENDENCIES

Declaring a Dependency

To express your project's reliance on a particular artifact, you declare a dependency in the project's pom.xml.

Hot Tip You can use the search engine at repository.sonatype.org to find dependencies by name and get the xml necessary to paste into your pom.xml

```
<project>
  <dependencies>
    <dependency>
      <groupId>com.yourcompany</groupId>
      <artifactId>yourlib</artifactId>
      <version>1.0</version>
      <type>jar</type>
      <scope>compile</scope>
    </dependency>
  </dependencies>
  <!-- ... -->
</project>
```

Standard Scopes

Each dependency can specify a scope, which controls its visibility and inclusion in the final packaged artifact, such as a WAR or EAR. Scoping enables you to minimize the JARs that ship with your product.

Scope	Description
compile	Needed for compilation, included in packages.
test	Needed for unit tests, not included in packages.
provided	Needed for compilation, but provided at runtime by the runtime container.
system	Needed for compilation, given as absolute path on disk, and not included in packages.
import	An inline inclusion of a POM-type artifact facilitating dependency-declaring POM snippets.

PLUGINS

Adding a Plugin

A plugin and its configuration are added via a small declaration, very similar to a dependency, in the `<build>` section of your pom.xml.

```
<build>
  <!-- ... -->
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <configuration>
        <maxmem>512m</maxmem>
      </configuration>
    </plugin>
  </plugins>
</build>
```

Common Plugins

Maven created an acronym for its plugin classes that aggregates "Plain Old Java Object" and "Maven Java Object" into the resultant word, Mojo.

There are dozens of Maven plugins, but a handful constitute some of the most valuable, yet underused features:

surefire	Runs unit tests.
checkstyle	Checks the code's styling
clover	Code coverage evaluation.
enforcer	Verify many types of environmental conditions as prerequisites.
assembly	Creates ZIPs and other distribution packages of apps and their transitive dependency JARs.

Hot Tip The full catalog of plugins can be found at: http://maven.apache.org/plugins/index.html

VISUALIZE DEPENDENCIES

Users often mention that the most challenging task is identifying dependencies: why they are being included, where they are coming from and if there are collisions. Maven has a suite of goals to assist with this.

List a hierarchy of dependencies.

```
mvn dependency:tree
```

List dependencies in alphabetic form.

```
mvn dependency:resolve
```

List plugin dependencies in alphabetic form.

```
mvn dependency:resolve-plugins
```

Analyze dependencies and list any that are unused, or undeclared.

```
mvn dependency:analyze
```

REPOSITORIES

Repositories are the web sites that host collections of Maven plugins and dependencies.

Declaring a Repository

```
<repositories>
    <repository>
      <id>JavaDotNetRepo</id>
      <url>https://maven-repository.dev.java.net</url>
    </repository>
</repositories>
```

The Maven community strongly recommends using a repository manager such as Nexus to define all repositories. This results in cleaner pom.xml files and centrally cached and managed connections to external artifact sources. Nexus can be downloaded from http://nexus.sonatype.org/

Popular Repositories

Central	http://repo1.maven.org/maven2/
Java.net	https://maven-repository.dev.java.net/
Codehaus	http://repository.codehaus.org/
JBoss	http://repository.jboss.org/maven2

Hot Tip A near complete list of repositores can be found at http://www.mvnbrowser.com/repositories.html

PROPERTY VARIABLES

A wide range of predefined or custom of property variables can be used anywhere in your `pom.xml` files to keep string and path repetition to a minimum.

All properties in Maven begin with `${` and end with `}`.

To list all available properties, run the following command.

```
mvn help:expressions
```

Predefined Properties (Partial List)

`${env.PATH}`	Any OS environment variable such as EDITOR, or GROOVY_HOME. Specifically, the PATH environment variable.
`${project.groupId}`	Any project node from the aggregated Maven pom.xml. Specifically, the Group ID of the project
`${project.artifactId}`	Name of the artifact.
`${project.basedir}`	Path of the pom.xml.
`${settings.localRepository}`	The path to the user's local repository.
`${java.home}`	Any Java System Property. Specifically, the Java System Property path to its home.
`${java.vendor}`	The Java System Property declaring the JRE vendor's name.
`${my.somevar}`	A user-defined variable.

Project properties could previously be referenced with a ${pom.basedir} prefix or no prefix at all ${basedir}. Maven now requires that you prefix these variables with the word project ${project.basedir}.

Define a Property

You can define a new custom property in your pom.xml like so:

```
<project>
    [...]
    <properties>
        <my.somevar>My Value</my.somevar>
    </properties>
    [...]
</project>
```

DEBUGGING

Exception Full Stack Traces

If a Maven plugin is reporting an error, to see the full detail of the exception's stack trace run Maven with the -e flag.

```
mvn <yourgoal> -e
```

Output Debugging Info

Whenever reporting a Maven bug, or troubleshooting a problem, turn on all the debugging info by running Maven like so:

```
mvn <yourgoal> -X
```

Debug Maven Core/Plugins

Core Maven operations and plugins can be stepped through with any JPDA-compatible debugger, the most common option being Eclipse. When run in debug mode, Maven will wait for you to connect your debugger to socket port 8000 before continuing with its lifecycle.

```
mvnDebug <yourgoal>

Preparing to Execute Maven in Debug Mode
Listening for transport dt_socket at address: 8000
```

Debug a Unit Test

Your suite or an individual unit test can be debugged in much the same fashion by telling the Surefire test-execution plugin to wait for you to attach a debugger to port 5005.

```
mvn test -Dmaven.surefire.debug

Listening for transport dt_socket at address: 5005
```

SOURCE CODE MANAGEMENT

Configuring SCM

Your project's SCM connection can be quickly configured with just three XML tags, which adds significant capabilities to the scm, release, and reactor plugins.

The connection tag is your read-only view of your repository and developerConnection is the writable link. URL is your web-based view of the source.

```
<scm>
  <connection>scm:svn:http://myvendor.com/ourrepo/trunk</
connection>
  <developerConnection>
    scm:svn:https://myvendor.com/ourrepo/trunk
  </developerConnection>
  <url>http://myvendor.com/viewsource.pl</url>
</scm>
```

Hot Tip Over 12 SCM systems are supported by Maven. The full list can be viewed at http://docs.codehaus.org/display/SCM/SCM+Matrix

Using the SCM Plugin

The core SCM plugin offers two highly useful goals.

The diff command produces a standard Unix patch file with the extension .diff of the pending (uncommitted) changes on disk that can be emailed or attached to a bug report.

```
mvn scm:diff
```

The update-subprojects goal invokes a recursive scm-provider specific update (svn update, git pull) across all the submodules of a multimodule project.

```
mvn scm:update-subprojects
```

PROFILES

Profiles are a means to conditionally turn on portions of Maven configuration, including plugins, pathing and configuration.

The most common uses of profiles are for Windows/Unix platform-specific variations and build-time customization of JAR dependencies based on the use of a specific Weblogic, Websphere or JBoss J2EE vendor.

```
<project>
  [...]
  <profiles>
    <profile>
      <id>YourProfile</id>
      [...settings, build, plugins etc...]
      <dependencies>
        <dependency>
          <groupId>com.yourcompany</groupId>
          <artifactId>yourlib</artifactId>
        </dependency>
      <dependencies>
    </profile>
  </profiles>
  [...]
</project>
```

Profile Definition Locations

Profiles can be defined in pom.xml, profiles.xml (parallel to the pom.xml), ~/.m2/settings.xml, or $M2_HOME/conf/settings.xml.

Hot Tip The full Maven Profile reference, including details about when to use each of the profile definition files, can be found at http://maven.apache.org/guides/introduction/introduction-to-profiles.html

PROFILE ACTIVATION

Profiles can be activated manually from the command line or through an activation rule (OS, file existence, Maven version, etc.). Profiles are primarily additive, so best practices suggest leaving most off by default, and activating based on specific conditions.

Manual Profile Activation

```
mvn <yourgoal> –P YourProfile
```

Automatic Profile Activation

```
<project>
  [...]
  <profiles>
    <profile>
      <id>YourProfile</id>
      [...settings, build, etc...]
      <activation>
        <os>
          <name>Windows XP</name>
          <family>Windows</family>
          <arch>x86</arch>
          <version>5.1.2600</version>
       </os>
        <file>
          <missing>somefolder/somefile.txt</missing>
        </file>
      </activation>
    </profile>
  </profiles>
  [...]
</project>
```

CUTTING A RELEASE

Maven offers excellent automation for cutting a release of your project. In short, this is a plugin-guided ceremony for verifying that all tests pass, tagging your source code repository, and altering the POMs to reflect a product version increment.

The `prepare` goal runs the unit tests, continuing only if all pass, then increments the value in the pom <version> tag to a release version, tags the source repository accordingly, and increments the pom version tag back to a SNAPSHOT version.

```
mvn release:prepare
```

After a release has been successfully prepared, run the perform goal. This goal checks out the prepared release and deploys it to the POM's specified remote Maven repository for consumption by other teams and Maven builds.

```
mvn release:perform
```

ARCHETYPES

An archetype is a powerful template that uses your corporate Java package names and project name in the instantiated project and establishes a baseline of dependencies, with a bonus of basic sample code.

You can leverage public archetypes for quickly starting a project that uses a familiar stack, such as Struts+Spring, or Tapestry+Hibernate. You can also create private archetypes within your company to offer new projects a level of consistent dependencies matching your approved corporate technology stack.

Using an Archetype

The default behavior of the generate goal is to bring up

a menu of choices. You are then prompted for various replaceables such as package name and artifactId. Type this command, then answer each question at the command line prompt.

```
mvn archetype:generate
```

Creating Archetypes

An archetype can be created from an existing project, using it as the pattern by which to build the template. Run the command from the root of your existing project.

```
mvn archetype:create-from-project
```

Archetype Catalogs

The Maven Archetype plugin comes bundled with a default catalog of applications it can create, but other projects on the Internet also publish catalogs. To use an alternate catalog:

```
mvn archetype:generate –DarchetypeCatalog=<catalog>
```

A list of the most commonly used catalogs is as follows:

```
local
remote
http://repo.fusesource.com/maven2
http://cocoon.apache.org
http://download.java.net/maven/2
http://myfaces.apache.org
http://tapestry.formos.com/maven-repository
http://scala-tools.org
http://www.terracotta.org/download/reflector/maven2/
```

REPORTS

Maven has a robust offering of reporting plugins, commonly run with the site generation phase, that evaluate and aggregate information about the project, contributors, it's source, tests, code coverage, and more.

Adding a Report Plugin

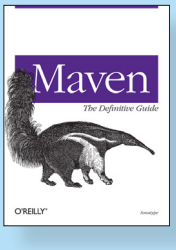
```
<reporting>
  <plugins>

    <plugin>
      <artifactId>maven-javadoc-plugin</artifactId>
    </plugin>
  </plugins>
</reporting>
```

Hot Tip A list of commonly used reporting plugins can be reviewed here http://maven.apache.org/plugins/

ABOUT THE AUTHOR

Matthew McCullough is an Open Source Architect with the Denver, Colorado consulting firm Ambient Ideas, LLC which he co-founded in 1997. He's spent the last 13 years passionately aiming for ever-greater efficiencies in software development, all while exploring how to share these practices with his clients and their team members. Matthew is a nationally touring speaker on all things open source and has provided long term mentoring and architecture services to over 40 companies ranging from startups to Fortune 500 firms. Feedback and questions are always welcomed at matthewm@ambientideas.com

RECOMMENDED BOOK



Several sources for Maven have appeared online for some time, but nothing served as an introduction and comprehensive reference guide to this tool -- until now. Maven: The Definitive Guide is the ideal book to help you manage development projects for software, webapplications, and enterprise applications. And it comes straight from the source.

BUY NOW
books.dzone.com/maven-definitive-guide

DZone

DZone communities deliver over 6 million pages each month to more than 3.3 million software developers, architects and decision makers. DZone offers something for everyone, including news, tutorials, cheatsheets, blogs, feature articles, source code and more. **"DZone is a developer's dream,"** says PC Magazine.

DZone, Inc.
2222 Sedwick Rd Suite 101
Durham, NC 27713
919.678.0300

Refcardz Feedback Welcome
refcardz@dzone.com

Sponsorship Opportunities
sales@dzone.com

ISBN-13: 978-1-934238-57-8
ISBN-10: 1-934238-57-0
50795
9 781934 238578
$7.95