Maven

Maven is a tool for building, manage project, also is an abstract container for running tasks.

Maven uses:
1 a project object model;
2 a project lifecycle;
3 a dependency management system;
4 and logic for executing plugin goals at defined phases in a lifecycle and a lot more.

Maven uses *convention over configuration* – declarative programming style where all basic requires have already tuned-up or set as a default. Maven's strength comes from the fact that it is "opinionated", it has a defined life-cycle and a set of common plugins that know how to build and assemble software.

Default project structure are:

- /src
    - /main
        - /java
        - /resources
    - /test
        - /java
        - /resources

- /target
- pom.xml

The core of Maven is pretty dumb, it doesn't know how to do much beyond parsing a few XML documents and keeping track of a lifecycle and a few plugins. Maven has been designed to delegate most responsibility to a set of *Maven Plugins* which can affect the *Maven Lifecycle* and offer access to *goals*. Most of the action in Maven happens in plugin goals which take care of things like compiling source, packaging bytecode, publishing sites, and any other task which need to happen in a build.

How it works:
command mvn install --> execute a series of sequential *lifecycle phases* until it reached the install lifecycle phase --> execute a number of default *plugin goals* like compile and create a JAR.

1 POM

POM – project object model, describes all project structure.

POM:

- POM Relationships
    - Coordinates: groupId, artifactId, version
    - Multi-Module
    - Inheritance
    - Dependencies

- General Project Information
    - General
    - Contributors
    - Licenses
- Build Settings
    - Build: directories, extensions, resources, plugins
    - Reporting
- Building Environment
    - Environment Information
    - Maven Information: Profiles

Maven POM sample:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
         http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <!-- The Basics -->
    <groupId>...</groupId>
    <artifactId>...</artifactId>
    <version>...</version>
    <packaging>...</packaging>
    <dependencies>...</dependencies>
    <parent>...</parent>
```

```
    <dependencyManagement>...</dependencyManagement>
    <modules>...</modules>
    <properties>...</properties>

    <!-- Build Settings -->
    <build>...</build>
    <reporting>...</reporting>

    <!-- More Project Information -->
    <name>...</name>
    <description>...</description>
    <url>...</url>
    <inceptionYear>...</inceptionYear>
    <licenses>...</licenses>
    <organization>...</organization>
    <developers>...</developers>
    <contributors>...</contributors>

    <!-- Environment Settings -->
    <issueManagement>...</issueManagement>
    <ciManagement>...</ciManagement>
    <mailingLists>...</mailingLists>
    <scm>...</scm>
    <prerequisites>...</prerequisites>
    <repositories>...</repositories>
    <pluginRepositories>...</pluginRepositories>
    <distributionManagement>...</distributionManagement>
    <profiles>...</profiles>
```

```
</project>
```

Full Maven POM definition can be found at
http://maven.apache.org/ref/3.0.3/maven-model/maven.html

- *General project information* This includes a project's name, the URL for a project, the sponsoring organization, and a list of developers and contributors along with the license for a project.
- *Build settings* In this section, we customize the behavior of the default Maven build. We can change the location of source and tests, we can add new plugins, we can attach plugin goals to the lifecycle, and we can customize the site generation parameters.
- *Build environment*
  The build environment consists of profiles that can be activated for use in different environments. For example, during development you may want to deploy to a development server, whereas in production you want to deploy to a production server. The build environment customizes the build settings for specific environments and is often supplemented by a custom settings.xml in ~/.m2.
- *POM relationships*
  A project rarely stands alone; it depends on other projects, inherits POM settings from parent
  projects, defines its own coordinates, and may include submodules.

The Super POM --initial POM configuration for every Maven project. All Maven project extend super POM, it's like default css in browsers.

`Note that the default Super POM has _disabled snapshot_ artifacts on the Central Repository. If you need to use a snapshot repository, you will need to customize repository settings in your pom.xml or in your settings.xml. _Update policy is set to "never"_, which means that Maven will never automatically update a plugin if a new version is released.`

POM defines a groupId, artifactId, and version: the three required coordinates for every project.

- **groupId** – name for a parent group, will be used in your classes as a part of **Full Qualified Name**, by the naming convention is the reversed domain name: org.spring
- **artifactId** – name for a project, will be name of a jar: spring-boot
- **version** – version: 1.1.1

Executing the effective-pom `$ mvn help:effective-pom` goal should print out an XML document capturing the merge between the Super POM and the POM from your project.

1.1 Versions

Maven uses standard for version configuration, it uses for version based sorting. Sort uses a string based comparison hence you should

use alpha-01 for *qualifier* instead of alpha-1.

<major version>.<minor version>.<incremental version>-<qualifier>

For example, the version “1.3.5” has a *major* version of 1, a *minor* version of 3, and an *incremental* version of 5. The version “5” has a major version of 5 and no minor or incremental version. The qualifier exists to capture milestone builds: alpha and beta releases, and the qualifier is separated from the major, minor, and incremental versions by a hyphen. For example, the version “1.3-beta-01” has a *major* version of 1, a *minor* version of 3, no incremental version and a *qualifier* of “beta-01”.

Snapshot is a project under active development. When you install or deploy 1.0.1-SNAPSHOT project, SNAPSHOT part would be changed to the current date and time 1.0.1-20080207-230803-1. As a default setting, Maven will *not check for SNAPSHOT releases* on remote repositories. To depend on SNAPSHOT releases, users must explicitly enable the ability to download snapshots using a repository or pluginRepository element in the POM. SNAPSHOT versions are for development only.

So for version numbers are used:

- 1.1.1 – plain releases, most stable;
- 1.1.1-alpha-1 – alpha, beta versions;
- 1.1.1-rc release candidate;
- 1.1.1-SNAPSHOT, most unstable.

1.2 Properties and Property References

All kind of properties and property references are described below:

Properties and Property References:

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

  <properties>
    <dependency.version>1.2.3</spring.version>
  </properties>

  <dependencies>
      <dependency>
        <!-- uses POM variable -->
        <groupId>${project.groupId}</groupId>
        <!-- uses environment variable -->
        <artifactId>${env.ARTIFACT_ID}</artifactId>
        <!-- uses POM property -->
        <version>${dependency.version}</version>
        <!-- uses settings.xml in ~/.m2/settings.xml -->
        <scope>${settings.version}</scope>
      </dependency>
  </dependencies>
</project>
```

Use mvn help:effective-pom to find out filled references.

1.3 Project Dependencies

Internal dependencies – dependencies for your project, external dependencies – dependencies for your dependencies.

Project Dependencies:

```
  <project>
    ...
    <dependencies>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>3.8.1</version>
        <scope>test</scope>
        <optional>true</optional>
      </dependency>
    </dependencies>
    ...
  </project>
```

Dependency scopes show where in classpath dependency will be available.

Dependency scopes: CPRTS (CoPy RaTS)

Scope	Description	Scope
compile	Default scope. Available everywhere in classpaths, and they are packaged.	compilation, test, package
provided	Like a compile but you expect application or container to provide them. Can be used for testing and compilation. Good example is a Servlet.	compilation, test, (import)
runtime	Runtime dependencies are required to execute and test the system, but they are not required for compilation. JDBC API implementation.	----------, test, package
test	Available only during test compilation and execution phases.	compilation, test, -------
system	As a provided but uses path to JAR in local system.	compilation, test, (import)

Optional – you need both libraries to compile this library project, but you don’t want both libraries to show up as transitive runtime dependencies for the project that uses your library. If you this dependency you would need to add the following dependency element to your POM project.

In difference to <scope>provided</scope> which is used for required dependencies which will be provided by the runtime environment an <optional>true</optional> *dependency is not necessarily meant to be required* (The idea is that some of the dependencies are only used for certain features in the project, and will not be needed if that feature

isn’t used.).

Dependency Version Ranges:

```
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <!-- use any of 3.8, 4.0 or 4.0+ -->
  <version>[3.8,4.0)</version>
  <!-- use any of 3.8.1+ inclusive -->
  <version>[,3.8.1)</version>
  <!-- use any of 3.8.1- exclusive -->
  <version>(3.8.1,)</version>
  <scope>test</scope>
</dependency>
```

1.4 Transitive Dependencies

Maven accomplishes this by building a graph of dependencies and dealing with any conflicts and overlaps that might occur. For example, if Maven sees that two projects depend on the same groupId and artifactId, it will sort out which dependency to use automatically, always favoring the more recent version of a dependency. Although this sounds convenient, there are some edge cases where transitive dependencies can cause some configuration issues. For these scenarios, you can use a dependency exclusion.

Transitive Scope:

Transitive Scope	Direct Scope (used in your POM)
compile	compile provided runtime test
provided	(not allowed to change)
runtime	compile provided runtime test
test	(not allowed to change)

Transitive dependencies which are **provided** and **test** scope usually do not affect a project. Transitive dependencies which are **compile** or **runtime** scoped will have the same scope of the direct dependency .

Often, you will want to replace a transitive dependency with another implementation. For example, if you are depending on a library that depends on the Sun JTA API, you may want to replace the declared transitive dependency.

Conflict Resolution:

```
<dependencies>
  <dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate</artifactId>
    <version>3.2.5.ga</version>
    <exclusions>
      <exclusion>
        <groupId>javax.transaction</groupId>
        <artifactId>jta</artifactId>
```

```
      </exclusion>
    </exclusions>
    </dependency>
  <dependency>
    <groupId>org.apache.geronimo.specs</groupId>
    <artifactId>geronimo-jta_1.1_spec</artifactId>
    <version>1.1</version>
  </dependency>
</dependencies>
```

Alternately named version from a dependency's version - resulting in 2 copies of the same project in the classpath. Normally Maven would capture this conflict and use a single version of the project **preferring a nearest**, but when groupId or artifactId are different, Maven will consider this to be two different libraries. So you need exclude one of dependencies.

Using the dependencyManagement element in a pom.xml allows you to reference a dependency in a child project without having to explicitly list the version. Maven will walk up the parent-child hierarchy until it finds a project with a dependencyManagement element, it will then use the version specified in this dependencyManagement element.

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.sonatype.mavenbook</groupId>
```

```
  <artifactId>a-parent</artifactId>
  <version>1.0.0</version>
  ...
  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>5.1.2</version>
        <scope>runtime</scope>
      </dependency>
    ...
    <dependencies>
  </dependencyManagement>
```

Then, in a child project, you can add a dependency to the MySQL Java Connector using the following dependency XML:

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.sonatype.mavenbook</groupId>
    <artifactId>a-parent</artifactId>
    <version>1.0.0</version>
  </parent>
  <artifactId>project-a</artifactId>
  ...
```

```
  <dependencies>
    <dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <!-- version isn't required because already listed in
 parent -->
    </dependency>
  </dependencies>
</project>
```

1.5 Project Relationships

groupId

A groupId groups a set of related artifacts. Group identifiers generally resemble a Java package name. For example, the groupId org.apache.maven is the base groupId for all artifacts produced by the Apache Maven project. Group identifiers are translated into paths in the Maven Repository; for example, the org.apache.maven groupId can be found in /maven2/org/apache/maven on repo1.maven.org.

artifactId

The artifactId is the project's main identifier. When you generate an artifact, this artifact is going to be named with the artifactId. When you refer to a project, you are going to refer
to it using the artifactId. The artifactId, groupId combination must be unique. In other words, you can't have two separate projects with the same artifactId and groupId; artifactId s are unique within a particular groupId. Note While '.'s are commonly used in groupId s, you should

try to avoid using them in artifactId. This can cause issues when trying to parse a fully qualified name down into the subcomponents.

version

When an artifact is released, it is released with a version number. This version number is a numeric identifier such as “1.0”, “1.1.1”, or “1.1.2-alpha-01”. You can also use what is known as a snapshot version. A snapshot version is a version for a component which is under development, snapshot version numbers always end in SNAPSHOT; for example, “1.0-SNAPSHOT”, “1.1.1-SNAPSHOT”, and “1-SNAPSHOT”. Section 3.3.1.1 introduces versions and version ranges.

classifier

You would use a classifier if you were releasing the same code but needed to produce two separate artifacts for technical reasons. For example, if you wanted to build two separate artifacts of a JAR, one compiled with the Java 1.4 compiler and another compiled with the Java 6 compiler, you might use the classifier to produce two separate JAR artifacts under the same groupId:artifactId:version combination. If your project uses native extensions, you might use the classifier to produce an artifact for each target platform. Classifiers are commonly used to package up an artifact’s sources, JavaDocs or binary assemblies.

1.6 Project Inheritance

```
<project>
  <parent>
```

```
    <groupId>com.training.killerapp</groupId>
    <artifactId>a-parent</artifactId>
    <version>1.0-SNAPSHOT</version>
    <relativePath>../a-parent/pom.xml</relativePath>
  </parent>
  <artifactId>project-a</artifactId>
  ...
</project>
```

Running mvn help:effective-pom in project-a would show a POM that is the result of merging the Super POM with the POM defined by a-parent and the POM defined in project-a. So full dependency three is super-pom <-- a-parent <-- project-a.

When a project specifies a parent project, Maven uses that parent POM as a starting point before it reads the current project's POM. It inherits everything, including the groupId and version number. You'll notice that project-a does not specify either, both groupId and version are inherited from aparent. With a parent element, all a POM really needs to define is an artifactId. If you start using Maven to manage and build large multi-module projects, you will often be creating many projects which share a common groupId and version.

1.7 POM Best Practices

You can create a project with pom packaging that groups dependencies together. For example, let's assume that your application uses Hibernate, a popular Object-Relational mapping framework. Every

project which uses Hibernate might also have a dependency on the Spring Framework and a MySQL JDBC driver. Instead of having to include these dependencies in every project that uses Hibernate, Spring, and MySQL you could create a special POM that does nothing more than declare a set of common dependencies. You could create a project called persistence-deps (short for Persistence Dependencies), and have every project that needs to
do persistence depend on this convenience project:

```
<project>
  <groupId>org.sonatype.mavenbook</groupId>
  <artifactId>persistence-deps</artifactId>
  <version>1.0</version>
  <packaging>pom</packaging>
  <dependencies>
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate</artifactId>
      <version>${hibernateVersion}</version>
    </dependency>
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-annotations</artifactId>
      <version>${hibernateAnnotationsVersion}</version>
    </dependency>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-hibernate3</artifactId>
```

```
      <version>${springVersion}</version>
    </dependency>
    <dependency>
      <groupId>mysql</groupId>
      <artifactId>mysql-connector-java</artifactId>
      <version>${mysqlVersion}</version>
    </dependency>
  </dependencies>
  <properties>
    <mysqlVersion>(5.1,)</mysqlVersion>
    <springVersion>(2.0.6,)</springVersion>
    <hibernateVersion>3.2.5.ga</hibernateVersion>
    <hibernateAnnotationsVersion>3.3.0.ga</hibernateAnnot
ationsVersion>
  </properties>
</project>
```

```
<project>
    <description>This is a project requiring JDBC</descri
ption>
    ...
    <dependencies>
      ...
      <dependency>
        <groupId>org.sonatype.mavenbook</groupId>
        <artifactId>persistence-deps</artifactId>
        <version>1.0</version>
```

```
            <type>pom</type>
        </dependency>
    </dependencies>
</project>
```

If you need to share a large number of dependencies between projects, you could also just establish parent-child relationships between projects and refactor all common dependencies to the parent project, but the disadvantage of the parent-child approach is that a project can have only one parent. Sometimes it makes more sense to group similar dependencies together and reference a pom dependency. This way, your project can reference as many of these consolidated dependency POMs as it needs.

Maven uses the depth of a dependency in the tree when resolving conflicts using a nearest wins approach. Using the dependency grouping technique above pushes those dependencies one level down in the tree. Keep this in mind when choosing between grouping in a pom or using dependencyManagement in a parent POM.

“nearest definition” means that the version used will be the closest one to your project in the tree of dependencies, eg. if dependencies for A, B, and C are defined as A -> B -> C -> D 2.0 and A -> E -> D 1.0, then D 1.0 will be used when building A because the path from A to D through E is shorter. You could explicitly add a dependency to D 2.0 in A to force the use of D 2.0

Maven follows – **Dependency mediation** algorithm to choose between dependencies. Rules are:

- **Nearest first**: A dependency of lower level has a priority over another of the higher depth. (Here, level refers to 1st level, 2nd level transitive dependency etc.) And hence, a direct dependency has priority over transitive dependency.
- **First Found**: At the same level, the first dependency that is found is taken.

You can run mvn dependency:tree

1.8 Multimodule Project

There is a difference between inheriting from a parent project and being managed by a multimodule project. A parent project is one that passes its values to its children. A multimodule project simply manages a group of other subprojects or modules. The multimodule relationship is defined from the topmost level downwards. When setting up a multimodule project, you are simply telling a project that its build should include the specified modules. Multimodule builds are to be used to group modules together in a single build. The parent-child relationship is defined from the leaf node upwards. The parent-child relationship deals more with the definition of a particular project. When you associate a child with its parent, you are telling Maven that a project's POM is derived from another.

Parent-child relationship:	**Submodule relationship:**
Super POM	Module

<-- Parent POM	–> Submodule 1
<–<-- Module	–> Submodule 2
<–<-- Submodule 1	
<–<-- Submodule 2	

Module is a aggregator for Submodule 1, 2. Submodule know nothing about Module, but lifecycle of Module depends on Submodules. Also Parent POM is a parent for all Module and Submodules.

```
<project>
  ...
  <modules>
    <module>submodule-1</module>
    <module>submodule-2</module>
  </modules>
</project>
```

2 Build Lifecycle

Maven uses objects for describing project. But for describing actions Maven uses *phases*.

A *build lifecycle* is an organized sequence of *phases* that exist to give order to a set of goals. Those *goals* are chosen and bound by the *packaging type* of the project being acted upon. There are three standard lifecycles in Maven: clean, default (sometimes called build)

and site. In this chapter, you are going to learn how Maven ties goals to lifecycle phases and how the lifecycle can be customized. You will also learn about the default lifecycle phases.

Lifecycle strucure:

```
{
  "buildLifecycle": {
    "name": "name",
    "phases": [
      {
        "name": "name",
        "goals": [
          { "name": "name" }
        ]
      }
    ],
    "packagingType": "type"
  }
}
```

2.1 Clean Lifecycle

The interesting phase in the clean lifecycle is the clean phase. The Clean plugin’s clean goal (clean: clean) is bound to the clean phase in the clean lifecycle. The clean:clean goal deletes the output of a build by deleting the build directory. If you haven’t customized the location of the build directory it will be the ${basedir}/target directory as

defined by the Super POM. When you execute the clean:clean goal you do not do so by executing the goal directly with mvn clean:clean, you do so by executing the clean phase of the clean lifecycle. Executing the clean phase gives Maven an opportunity to execute any other goals which may be bound to the pre-clean phase.

Lifecycle strucure:

```
{
  "buildLifecycle": {
    "name": "clean",
    "phases": [
      {
        "name": "pre-clean",
        "goals": [
          ...
        ]
      },
      {
        "name": "clean",
        "goals": [
          { "name": "delete ${basedir}/target directory"
}
        ]
      },
      {
        "name": "post-clean",
        "goals": [
```

```
            ...
          ]
        }
      ],
      "packagingType": "type"
    }
}
```

Simply running the clean:clean goal will not execute the lifecycle at all, but specifying the clean phase will use the clean lifecycle and advance through the three lifecycle phases until it reaches the clean phase.

Triggering a Goal on pre-clean shows an example of build configuration which binds the antrun:run goal to the pre-clean phase to echo an alert that the project artifact is about to be deleted. In this example, the antrun:run goal is being used to execute some arbitrary Ant commands to check for an existing project artifact. If the project's artifact is about to be deleted it will print this to the screen

```
<project>
  ...
  <build>
    <plugins>...<plugin>
    <artifactId>maven-antrun-plugin</artifactId>
    <executions>
      <execution>
        <id>file-exists</id>
```

```
        <!-- adds excecution to this phase of lifecycle -
->
        <phase>pre-clean</phase>
        <goals>
        <!-- adds goal to this phase -->
          <goal>run</goal>
        </goals>
        <configuration>
          <tasks>
            <!-- adds the ant-contrib tasks (if/then/else
 used below) -->
            <taskdef resource="net/sf/antcontrib/antcontr
ib.properties" />
            <available file="${project.build.directory}/$
{project.build.finalName}.${project.packaging}"
            property="file.exists" value="true"/>
            <if>
              <not>
                <isset property="file.exists" />
              </not>
              <then>
                <echo>No ${project.build.finalName}.${pro
ject.packaging} to delete</echo>
              </then>
              <else>
                <echo>Deleting ${project.build.finalName}
.${ project.packaging}</echo>
              </else>
```

```
            </if>
          </tasks>
        </configuration>
      </execution>
    </executions>
    <dependencies>
      <dependency>
        <groupId>ant-contrib</groupId>
        <artifactId>ant-contrib</artifactId>
        <version>1.0b2</version>
      </dependency>
    </dependencies>
    </plugin>
    </plugins>
  </build>
</project>
```

2.2 Default Lifecycle

Most Maven users will be familiar with the default lifecycle. It is a general model of a build process for a software application.

Lifecycle Phase Description:

Lifecycle Phase	Description
validate	Validate the project is correct and all necessary information is available to complete a build

generate-sources	Generate any source code for inclusion in compilation
process-sources	Process the source code, for example to filter any values
generate-resources	Generate resources for inclusion in the package
process-resources	Copy and process the resources into the destination directory, ready for packaging
compile	Compile the source code of the project
process-classes	Post-process the generated files from compilation, for example to do bytecode enhancement on Java classes
generate-test-sources	Generate any test source code for inclusion in compilation
process-test-sources	Process the test source code, for example to filter any values
generate-test-resources	Create resources for testing
process-test-resources	Copy and process the resources into the test destination directory
test-compile	Compile the test source code into the test destination directory
test	Run tests using a suitable unit testing framework. These tests should not require the code be packaged or deployed
prepare-	Prepare build for packaging (without packing)

package	
package	Take the prepare-package code and package it in its distributable format, such as a JAR, WAR, or EAR
pre-integration-test	Perform actions required before integration tests are executed. This may involve things such as setting up the required environment
integration-test	Process and deploy the package if necessary into an environment where integration tests can be run
post-integration-test	Perform actions required after integration tests have been executed. This may include cleaning up the environment
verify	Run any checks to verify the package is valid and meets quality criteria
install	Install the package into the local repository, for use as a dependency in other projects locally
deploy	Copies the final package to the remote repository for sharing with other developers and projects (usually only relevant during a formal release)

Phases: validate, (generate, process sources, resources)compile, (generate, process test sources, resources)test-compile, test, (pre)package, (pre, post)integration-test, verify, install, deploy;
VCT-C ToP In-TV ID

2.3 Site Lifecycle

Maven does more than build software artifacts from project, it can also generate project documentation and reports about the project, or a

collection of projects.

Lifecycle strucure:

```
{
  "buildLifecycle": {
    "name": "clean",
    "phases": [
      {
        "name": "pre-site",
        "goals": [
          ...
        ]
      },
      {
        "name": "site",
        "goals": [
          { "name": "generate site" }
        ]
      },
      {
        "name": "post-site",
        "goals": [
          ...
        ]
      },
      {
        "name": "site-deploy",
```

```
        "goals": [
          { "name": "deploy site" }
        ]
      }
    ],
    "packagingType": "type"
  }
}
```

2.4 Package-specific Lifecycles

Maven does more than build software artifacts from project, it can also generate project documentation and reports about the project, or a collection of projects.

The specific goals bound to each phase default to a set of goals specific to a project’s packaging. A project with packaging jar has a different set of default goals from a project with a packaging of war. The packaging element affects the steps required to build a project. For an example of how the packaging affects the build, consider two projects: one with pom packaging and the other with jar packaging. The project with pom packaging will run the site:attach-descriptor goal during the package phase, and the project with jar packaging will run the jar:jar goal instead.

2.5 Common Lifecycle Goals

2.5.1 Process Resources & Process Test

Resources

The process-resources phase “processes” resources and copies them to the output directory. If you haven’t customized the default directory locations defined in the Super POM, this means that Maven will *copy* the files from ${basedir}/src/main/resources to ${basedir}/target/classes or the directory defined in ${project.build.outputDirectory}.

In addition to copying the resources to the output directory, Maven can also apply a *filter* to the resources that allows you to replace tokens within resource file. Just like variables are referenced in a POM using ${…} notation, you can reference variables in your project’s resources using the same syntax. Coupled with build profiles, such a facility can be used to produce build artifacts which target different deployment platforms. This is something that is common in environments which need to produce output for development, testing, staging, and production platforms from the same project.

2.5.2 Compile & test-compile

Most lifecycles bind the Compiler plugin’s compile goal to the compile phase. This phase calls out to compile:compile which is configured to compile all of the source code and copy the bytecode to the build output directory. If you haven’t customized the values defined in the Super POM, compile: compile is going to compile everything from src/main/java to target/classes. The Compiler plugin calls out to javac and uses default source and target settings of 1.3 and 1.1. In other words, the compiler plugin assumes that your Java source conforms to Java 1.3 and that you are targeting a Java 1.1 JVM. If you would like to

change these settings, you'll need to supply the target and source configuration:

```
<project>
  ...
  <build>
    ...
    <plugins>
      <plugin>
        <artifactId>maven-compiler-plugin</artifactId>
        <configuration>
        <source>1.5</source>
        <target>1.5</target>
        </configuration>
      </plugin>
    </plugins>
    ...
  </build>
  ...
</project>
```

2.5.3 Test

Most lifecycles bind the test goal of the Surefire plugin to the test phase. The Surefire plugin is Maven's unit testing plugin, the default behavior of Surefire is to look for all classes ending in *Test in the test source directory and to run them as JUnit tests. The Surefire plugin can also be configured to run TestNG unit tests. After running mvn

test, you should also notice that the Surefire produces a number of reports in target/surefire-reports.

Configuring Surefire to Ignore Test Failures:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-surefire-plugin</artifactId>
      <configuration>
        <testFailureIgnore>true</testFailureIgnore>
      </configuration>
    </plugin>
    ...
  </plugins>
</build>
```

To skip tests altogether: `$ mvn install -Dmaven.test.skip=true` or `$ mvn install -DskipTests=true`

2.5.4 Install

This install:install goal simply installs a project's main artifact to the local repository.

Install for project org.sonatype.mavenbook:simple-test:1.0.2 and artifact target/simpletest-1.0.2.jar clone artifact to ~/.m2/repository/org/sonatype/mavenbook/simple-test/1.0.2/simple-

test-1.0.2.jar.

2.5.5 Deploy

The deploy goal of the Deploy plugin is usually bound to the deploy lifecycle phase. This phase is used to deploy an artifact to a remote Maven repository, this is usually required to update a remote repository when you are performing a release.

3 Build Profiles

Profiles allow for the ability to customize a particular build for a particular environment; profiles enable portability between different build environments.

When you are working in a development environment, your system might be configured to read from a development database instance running on your local machine while in production, your system is configured to read from the production database. Maven allows you to define any number of build environments (build profiles) which can override any of the settings in the pom.xml. You could configure your application to read from your local, development instance of a database in your “development” profile, and you can configure it to read from the production database in the “production” profile. Profiles can also be activated by the environment and platform, you can customize a build to run differently depending the Operating System or the installed JDK version.

A profile in Maven is an alternative set of configuration values which

set or override default values. Using a profile, you can customize a build for different environments. Profiles are configured in the pom.xml and are given an identifier. Then you can run Maven with a command-line flag that tells Maven to execute goals in a specific profile.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.sonatype.mavenbook</groupId>
  <artifactId>simple</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>simple</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
  <profiles>
    <profile>
      <id>production</id>
```

```
      <build>
      <plugins>
        <plugin>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-compiler-plugin</artifactId>
          <configuration>
            <debug>false</debug>
            <optimize>true</optimize>
          </configuration>
        </plugin>
      </plugins>
      </build>
    </profile>
  </profiles>
</project>
```

1. The profiles element is in the pom.xml, it contains one or more profile elements. Since profiles override the default settings in a pom.xml, the profiles element is usually listed as the last element in a pom.xml.
2. Each profile has to have an id element. This id element contains the name which is used to invoke this profile from the command-line. A profile is invoked by passing the -P<profile_id>
   command-line argument to Maven.
3. A profile element can contain many of the elements which can appear under the project

element of a POM XML Document. In this example, we're overriding the behavior of the Compiler
plugin and we have to override the plugin configuration which is normally enclosed in a build
and a plugins element.

4. We're overriding the configuration of the Maven Compiler plugin. We're making sure that the
bytecode produced by the production profile doesn't contain debug information and that the bytecode has gone through the compiler's optimization routines.

To execute mvn install under the production profile, you need to pass the -Pproduction and verify that the production profile overrides the default Compiler -X as follows:

`$ mvn clean install -Pproduction -X`

Elements Allowed in a Profile:

```
<project>
  <profiles>
    <profile>
      <build>
        <defaultGoal>...</defaultGoal>
        <finalName>...</finalName>
        <resources>...</resources>
        <testResources>...</testResources>
        <plugins>...</plugins>
      </build>
```

```
      <reporting>...</reporting>
      <modules>...</modules>
      <dependencies>...</dependencies>
      <dependencyManagement>...</dependencyManagement>
      <distributionManagement>...</distributionManagement
>
      <repositories>...</repositories>
      <pluginRepositories>...</pluginRepositories>
      <properties>...</properties>
    </profile>
  </profiles>
</project>
```

3.1 Profile Activation

Maven provides a way to “activate” a profile for different environmental parameters, this is called profile activation.

Active profile for JDK version:

```
<profiles>
  <profile>
    <id>jdk16</id>
    <activation>
      <jdk>1.6</jdk>
    </activation>
    <modules> v2
      <module>simple-script</module>
```

```
    </modules>
  </profile>
</profiles>
```

Activations can contain one of more selectors including JDK versions, Operating System parameters, files, and properties. A profile is activated when all activation criteria has been satisfied.

Activation by:

```
<activation>
  <activeByDefault>false</activeByDefault>
  <jdk>1.5</jdk>
  <os>
    <name>Windows XP</name>
    <family>Windows</family>
    <arch>x86</arch>
    <version>5.1.2600</version>
  </os>
  <property>
    <name>customProperty</name>
    <value>BLUE</value>
    <!-- activation by absence of property  -->
    <name>!environment.type</name>
  </property>
  <file>
    <exists>file2.properties</exists>
```

```
    <missing>file1.properties</missing>
  </file>
</activation>
```

Profiles can be in pom.xml, profiles.xml, ~/.m2/settings.xml, or ${M2_HOME}/conf/settings.xml.
For list active profile type `$ mvn help:active-profiles`

There are plugins available that can manipulate the database, run SQL, and plugins like the Maven Hibernate3 plugin which can generate annotated model objects for use in persistence frameworks. A few of these plugins, can be configured in a pom.xml using these properties.

4 Running Maven

Defines a system property `-D, --define <arg>`
Display help information `-h, --help`
Comma-delimited list of profiles to activate `-P, --activate-profiles <arg>`

Forces the use of an alternate POM file `-f, --file <file>`
Alternate path for the user settings file `-s,--settings <arg>`
Alternate path for the global settings file `-gs, --global-settings <file>`

Fail the build afterwards; allow all non-impacted builds to continue `-fae, --fail-at-end`

Stop at first failure `-ff, --fail-fast`
NEVER fail the build, regardless of project result `-fn, --fail-never`
Skip tests and tests compile `-DskipTests=true`

Error handling `-e, --errors`, `-X, --debug`, `-q, --quiet`

Batch mode is essential if you need to run Maven in a non-interactive, continuous integration environment. Maven will never accept answer from the user `-B, --batch-mode`

Fail the build if checksums don't match `-C, --strict-checksums`
Forces a check for updated releases and snapshots on remote repositories `-U, --update-snapshots`
The `-U` option is useful if you want to make sure that Maven is checking for the latest versions of all SNAPSHOT dependencies.

Prevents Maven from building submodules. Only builds the project contained in the current directory. `-N, --non-recursive`

Lists the profiles (project, user, global) which are active for the build `help:active-profiles`

Displays the effective POM for the current build, with the active profiles factored in `help:effective-pom`

Prints out the calculated settings for the project, given any profile enhancement and the inheritance of the global settings into the user-level settings. `help:effective-settings`

Describes the attributes of a plugin. This need not run under an existing project directory. You must at least give the groupId and artifactId of the plugin you wish to describe. `$ mvn help:describe -Dplugin=help -Dfull`
The following command lists all of the information about the Compiler plugin’s compile goal. `$ mvn help:describe -Dplugin=compiler -Dmojo=compile -Dfull`

5 Maven Configuration

Maven plugins are configured using properties that are defined by goals within a plugin. If you look at a goal like the compile goal in the Maven Compiler Plugin you will see a list of configuration parameters like source, target, compilerArgument, fork, optimize, and many others. If you look at the testCompile goal you will see a different list of configuration parameters for the testCompile goal. If you are looking for details on the available plugin goal configuration parameters, you can use the Maven Help Plugin to describe a particular plugin or a particular plugin goal.

To describe a particular plugin, use the help:describe goal from the command line as follows:
`$ mvn help:describe -Dcmd=compiler:compile`

If you need to configure a plugin to use specific versions of dependencies, you can define these dependencies under a dependencies element under plugin. When the plugin executes, it will execute with a classpath that contains these dependencies. Adding

Dependencies to a Plugin is an example of a plugin configuration that overrides default dependency versions and adds new dependencies to facilitate goal execution.

Adding dependencies to Maven Plugin

```
<plugin>
  <groupId>...</groupId>
  <artifactId>...</artifactId>
  <version>...</version>
  <dependencies>...</dependencies>
</plugin>
```

To set a value for a plugin configuration parameter in a particular project, use the XML shown in Configuring a Maven Plugin. Unless this configuration is overridden by a more specific plugin parameter configuration, Maven will use the values defined directly under the plugin element for all goals which are executed in this plugin.

Configuring a Maven Plugin

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-compiler-plugin</artifactId>
  <configuration>
    <source>1.5</source>
    <target>1.5</target>
```

```
  </configuration>
</plugin>
```

You can configure plugin parameters for specific executions of a plugin goal. Setting Configuration Parameters in an Execution shows an example of configuration parameters being passed to the execution of the run goal of the AntRun plugin during the validate phase. This specific execution will inherit the configuration parameters from the plugin's configuration element and merge them with the values defined for this particular execution.

Setting Configuration Parameters in an Execution

```
<plugin>
  <artifactId>maven-antrun-plugin</artifactId>
  <executions>
    <execution>
      <phase>validate</phase>
      <goals>
        <goal>run</goal>
      </goals>
      <configuration>
        <tasks>
          <echo>${PATH}=${env.PATH}</echo>
          <echo>User's Home Directory: ${user.home}</echo
>
          <echo>Project's Base Director: ${basedir}</echo
```

```
>
        </tasks>
      </configuration>
    </execution>
  </executions>
</plugin>
```

6 Maven Assemblies

Because the assembly:assembly goal is an aggregator mojo, it raises some issues in multi-module Maven builds, and it should only be called as a stand-alone mojo from the command-line. Never bind an assembly:assembly execution to a lifecycle phase. assembly:assembly was the original goal in the Assembly plugin, and was never designed to be part of the standard build process for a project. As it became clear that assembly archives were a legitimate requirement for projects to produce, the single mojo was developed. This mojo assumes that it has been bound to the correct part of the build process, so that it will have access to the project files and artifacts it needs to execute within the lifecycle of a large multi-module Maven project. In a multi-module environment, it will execute as many times as it is bound to the different module POMs. Unlike assembly:assembly, single will never force the execution of another lifecycle phase ahead of itself.

Whenever possible, you should definitely stick to using assembly:assembly for assemblies generated from the *command line*, and to assembly:single for assemblies bound to *lifecycle phases*.

6.1 Predefined Assembly Descriptors

Predefined Assembly Descriptors for maven assembly plugin

BSPFat

bin: LICENSE, README, and NOTICE + project’s main artifact.

jar-with-dependencies: JAR + packed runtime dependencies.

project: archives the project directory structure as it exists in your filesystem, the target directory is omitted.

src: archive project source and pom.xml files, along with any LICENSE, README, and NOTICE files that are in the project’s root directory.

When you generate assemblies as part of your normal build process, those assembly archives will be attached to your main project’s artifact. This means they will be installed and deployed alongside the main artifact, and are then resolvable in much the same way. However, these artifacts are attachments, which in Maven means they are derivative works based on some aspect of the main project build.

Assembly artifacts can be used as dependencies using the required coordinates of a project plus the classifier under which the assembly was installed or deployed. If the assembly is not a JAR archive, we also need to declare its type (packaging).

7 Properties and Resource Filtering

POM properties like ${project.groupId}, environment variables and Java System properties

can be referenced, as well as values from your ~/.m2/settings.xml file.

Maven has a feature called Resource Filtering which allows you to replace property
references in any resource files stored under src/main/resources. By default this feature is *disabled*.

You ca use it like this: ${project.*}, ${settings.*}, ${env.*}, ${system-propertiy}

In addition to the implicit properties listed above, a Maven POM, Maven Settings, or a Maven Profile can define a set of arbitrary, user-defined properties.

Projects in a large, multi-module build often share the same `groupId` and `version` identifiers. When you are declaring interdependencies between two modules which share the same groupId and version, it is a good idea to use a property reference for both. A project's `artifactId` is often used as the name of a deliverable.

project.build.*:

- project.build.sourceDirectory
- project.build.scriptSourceDirectory
- project.build.testSourceDirectory
- project.build.outputDirectory
- project.build.testOutputDirectory
- project.build.directory

7.1 Resource Filtering

You can use Maven to perform variable replacement on project resources. When resource filtering is activated, Maven will scan resources for property references surrounded by ${ and }. When it finds these references it will replace them with the appropriate value in much the same way the properties defined in the previous section can be referenced from a POM. This feature is especially helpful when you need to parameterize a build with different configuration values depending on the target deployment platform.

Using Maven resource filtering you can reference Maven properties and then use Maven profiles to define different configuration values for different target deployment environments. To illustrate this feature, assume that you have a project which uses the Spring Framework to configure a BasicDataSource from the Commons DBCP project. Your project may contain a file in src/main/resources named applicationContext. xml which contains the XML listed in Referencing Maven Properties from a Resource.

To illustrate this feature, assume that you have a BasicDataSource from the Commons DBCP project. Your project contain a file in src/main/resources named applicationContext.

Referencing Maven Properties from a Resource:

```
<beans xmlns="http://www.springframework.org/schema/beans
"
```

```
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/  -
        spring-beans-2.5.xsd">
  <bean id="someDao" class="com.example.SomeDao">
    <property name="dataSource" ref="dataSource"/>
  </bean>
  <bean id="dataSource" destroy-method="close"
      class="org.apache.commons.dbcp.BasicDataSource">
    <property name="driverClassName" value="${jdbc.driverClassName}"/>
    <property name="url" value="${jdbc.url}"/>
    <property name="username" value="${jdbc.username}"/>
    <property name="password" value="${jdbc.password}"/>
  </bean>
</beans>
```

Your program would read this file at runtime, and your build is going to replace the references to properties like jdbc.url and jdbc.username with the values you defined in your pom.xml.

To turn on resource filtering, you need to use the resources child element of the build element in a POM.

```
<project>
```

```
  ...
  <properties>
    <jdbc.driverClassName>com.mysql.jdbc.Driver</jdbc.dri
verClassName>
    <jdbc.url>jdbc:mysql://localhost:3306/development_db<
/jdbc.url>
    <jdbc.username>dev_user</jdbc.username>
    <jdbc.password>s3cr3tw0rd</jdbc.password>
  </properties>
  ...
  <build>
    <resources>
      <resource>
        <!-- enable resource filtering -->
        <directory>src/main/resources</directory>
        <filtering>true</filtering>
      </resource>
    </resources>
  </build>
  ...
  <profiles>
    <profile>
      <id>production</id>
      <!-- add properties -->
      <properties>
        <jdbc.driverClassName>oracle.jdbc.driver.OracleDr
iver</jdbc.driverClassName>
        <jdbc.url>jdbc:oracle:thin:@proddb01:1521:PROD</j
```

```
dbc.url>
        <jdbc.username>prod_user</jdbc.username>
        <jdbc.password>s00p3rs3cr3t</jdbc.password>
      </properties>
    </profile>
  </profiles>
</project>
```

8 Site Generation

Maven provides you with the ability to write simple web pages and render those pages against a consistent project template. Maven can publish site content in multiple formats including XHTML and PDF.

To build the site and preview the result in a browser, you can run mvn site:run, this will build the site and start an embedded instance of Jetty.

9 Lifecycle

A custom lifecycle must be packaged in the plugin under the META-INF/maven/lifecycle.xml file. You can include a lifecycle under src/main/resources in META-INF/maven/lifecycle.xml. The following lifecycle. xml declares a lifecycle named zipcycle that contains only the zip goal in a package phase.

Define a Custom Lifecycle in lifecycle.xml:

```
<lifecycles>
  <lifecycle>
    <id>zipcycle</id>
    <phases>
      <phase>
        <id>package</id>
        <executions>
          <execution>
            <goals>
            <goal>zip</goal>
            </goals>
          </execution>
        </executions>
      </phase>
    </phases>
  </lifecycle>
</lifecycles>
```

10 Maven Archetype

Archetypes use for starting development of project with custom parent POM which adds completely usable environment for development such kind of project. To see all archetypes run `$ mvn archetype:generate`

11 Settings

The settings element in the settings.xml file contains elements used to

define values which configure Maven execution. There are two locations where a settings.xml file may live:

- Maven Installation Directory: $M2_HOME/conf/settings.xml
- User-specific Settings File: ~/.m2/settings.xml

System.xml:

```
<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
    http://maven.apache.org/xsd/settings-1.0.0.xsd">
  <localRepository/>
  <interactiveMode/>
  <usePluginRegistry/>
  <offline/>
  <pluginGroups/>
  <servers/>
  <mirrors/>
  <proxies/>
  <profiles/>
  <activeProfiles/>
</settings>
```

localRepository: ${user.dir}/.m2/repository.

interactiveMode: true if Maven should attempt to interact with the user for input

usePluginRegistry: true if Maven should use the ${user.dir}/.m2/plugin-registry.xml file to manage plugin versions, defaults to false.

offline_pluginGroups: used for searching groupId of plugin if it not provided in the command line. This list contains org.apache.maven.plugins by default.

servers: used for storing credentials and for deployment.

mirrors: it's copy of maven server

profiles: It consists of the activation, repositories, pluginRepositories and properties elements. The profile elements only include these four elements because they concern themselves with the build system as a whole (which is the role of the settings.xml file), not about individual project object model settings.

activation: Activations are the key of a profile. Like the POM's profiles, the power of a profile comes from its ability to modify some values only under certain circumstances: jdk, os, property, file. Profile also can be activated as `-Ptest` Maven parameter. See all profiles `mvn help:active-profiles`.

properties: by using the notation ${X}, where X is the property. They come in five different styles, all accessible from the settings.xml file: env.X (environment variable), project.x (all POM vars), settings.x (settings), x (java system properties).

repositories: repositories are remote collections of projects. Different remote repositories may contain different projects, and under the active profile they may be searched for a matching release or snapshot artifact.

- releases, snapshots: these are the policies for each type of

artifact, Release or snapshot. For example, one may decide to enable only snapshot downloads, possibly for development purposes.

- enabled: true or false for whether this repository is enabled for releases or snapshots.
- updatePolicy: This element specifies how often updates should attempt to occur. Maven will compare the local POMs timestamp to the remote. The choices are: always, daily (default), interval:X (where X is an integer in minutes) or never.
- checksumPolicy: check checksums and make an action: ignore, fail, or warn on missing or incorrect checksums.

  plugin Repositories: remote location where Maven can find plugins.

  activeProfiles: this contains a set of activeProfile elements, which each have a value of a profile id. Any profile id defined as an act activeProfile will be active, regardless of any environment settings.