Building Java Projects with Apache Maven

Fritz Solms

January 15, 2015
Relevance of *Apache Maven*

- Primary build tool for *Java Projects*
- Manage from single project descriptor
  - a project’s build,
  - project reporting, and
  - project documentation
- Project descriptor:
  - *Project Object Model (POM)*
Maven = more than just a build tool

- Suggests and enforces for developers, testers, project managers, users
  - proven work patterns, &
  - proven structural patterns.
- Facilitates effective collaboration around
  - development,
  - testing,
  - deployment and distribution,
  - project reporting facilitating observation/monitorability, and
  - project documentation.
Guiding principles of Apache Maven

1. Sensible defaults.
2. Encourage proven patterns including patterns for
   - file/project structure,
   - development process,
   - dependency management and installation.
   making it easier for role players (developers, testers, project managers & system administrators) to move between projects.
3. Increase visibility including
   - code and component visibility for project tracking and reuse,
   - test visibility for bug tracking and fixing,
   - visibility of any other metadata like project vision, participating developers, ...
4. Effectively manage dependencies and resources
5. Integration between development teams
   - local and global
What does Maven provide?

- **A Project Object Model (POM)** within which one specifies
  - project metadata with direct dependency specification,
  - build customizations with enforced unit testing,
  - reporting customizations, distribution information, . . .

- **Uniform project structures** favouring convention over configuration.

- **Dependency management**: with support for resolving dependency chains/webs and retrieval of dependencies from local and remote repositories.

- **Standard processes** for building, reporting, deploying, distribution, . . .

- Remote and local **repositories** with versioned artifacts and metadata facilitating *universal reuse of plugins and projects artifacts*.

- **A plugin framework** which enables plugins to publish goals which can be bound to life cycle phases of either the standard built-in life cycle or to phases of its own life cycle,

- **Support for modularization of projects** with commonality inheritance from parent POM.
Maven versus Ant

- Declarative versus operational approach:
- Project structure / metadata versus explicit build instructions:
- Convention per default versus configure all
- Dependency management:
- Project inheritance:
- Modularization:
- Project reporting and measurement:
- Repositories: Global and local repositories for plugins, libraries and applications.
What is Maven really?

Maven is a *project management tool* which facilitates software project

- building,
- reporting,
- deploying, and
- distribution

in the context of a *globally integrated developer community* with *projects constructed from artifacts developed by different teams*. 
The *Project Object Model* (POM) encapsulates project metadata. It may contain:

- project identifier, name, version, ...
- project contributors and licensing information,
- structural organization (directories) of the project,
- project dependencies,
- relationships with other projects, and
- information about the project’s build environment.
A POM is encoded in XML and its structure, illustrated in figure 10 is defined by an XML schema. At the high level the POM may specify:

- project identifier (coordinate) assembled from artifactId, groupId and version,
- version of the POM model used,
- metadata like project name, description, licensing, contributors and organization information,
- relationships with other projects,
- project dependencies,
- repositories from which dependencies and plugins are sourced,
- build information,
- project artifact distribution information.
- various development, testing and production profiles,
- project measurement and reporting information,
- and a collection of properties which can be used throughout the POM.
Project Identifiers

- Projects artifacts, and dependencies on other projects and plugins are identified using *project coordinates* which take the form of:

\[
<\text{groupId}:\text{artifactId}:\text{packaging}:\text{version}>
\]

- Packaging not part of project identifier.

- For example, a minimal POM

```xml
<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>za.co.solms.mavenCourse</groupId>
  <artifactId>simplestExample</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>simple</name>
</project>
```
Project Identifiers

- Would have as a project identifier

```
1 za.co.solms.mavenCourse: simpleExample: 1.0-SNAPSHOT
```

- and would by default, be packaged in a jar name

```
1 simpleExample-1.0-SNAPSHOT.jar
```
The groupId is to identify the group or organization which takes ownership of the project. The convention is to use the domain name of the group/organization in reverse.
Artifact Identifier

The artifactId is a project identifier which should be unique within the group or organization which owns the project.
Version

This represents an identifier for the version of a project. During active development the version is usually designated as a SNAPSHOT version.
Packaging

This specifies the packaging to be used. Examples include jar, war, ear, pom, and zip.
In order to ensure some level of uniformity across projects you typically will want to define common project parameters in a parent POM which will be inherited across the various child POMs.
A POM declares another POM as its parent by specifying the project identifier and optionally a relative path. For example

```
<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>
  <parent>
    <groupId>za.co.solms.mavenCourse</groupId>
    <artifactId>simpleParent</artifactId>
    <version>1.0</version>
  </parent>
  <groupId>za.co.solms.mavenCourse</groupId>
  <artifactId>simplestExample</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>simple</name>
  <url>http://www.solms.co.za</url>
</project>
```
Child projects inherit all definitions defined in their parent project and only have to either
- define additional definitions which are not inherited,
- override inherited definitions.
Maven enforces a common parent POM called the Super POM from which all projects inherit.

This POM comes packaged with any Maven installation and encapsulates defaults which are shared across all projects including:

- The default remote Maven repository from which dependencies are obtained.
- The default remote Maven plugin repository from which plugins are sourced.
- Default build information including
  - a default directory structure for Maven projects including source and output directories for the code and tests
The super POM

- the default target directory and default build file name which is

```java
1 ${pom.artifactId}-${pom.version}
```

and is appended with the packaging type.

- A default list of plugins which are available for the default Maven build cycle for
  - compiling,
  - creating jars, wars, ears, rars and javadoc,
  - resolving dependencies,
  - deploying and installing, and
  - cleaning the project.

- a default report output directory which is set to

```java
1 target/site
```
The effective POM

- The effective POM is the sum-total of all inherited POM elements.
- Some elements may be overwritten at some level of the parent-child hierarchy.
- The `effective-pom` goal of the `help` plugin can be used to query the effective POM for a specific child POM by executing

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

from within the directory containing the POM.
POM requires direct project dependencies to be explicitly specified. These dependencies may be either

- *external dependencies* on libraries or other resources developed by third parties,
- libraries or other resources developed within other internal projects.
Specifying project dependencies

Project dependencies are specified in a separate dependencies section. The dependency is identified through the standard

```
<groupId>solms.co.za.mavenCourse</groupId>
<artifactId>testLibrary</artifactId>
<version>1.4.2</version>
```
Specifying version ranges

- Instead of locking the dependency into a particular version, you to specify a dependency on a version range.
- Range defines the lowest and highest version numbers which are acceptable.
- This can be done inclusively via square brackets \([1.51,1.9]\) or exclusively via round brackets \((1.5,2)\). For example,

```xml
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>[1.51,4)</version>
</dependency>
```

defines a dependency on any version of JUnit up to, but excluding version 4.0.
Specifying the scope of a dependency

- Maven allows you to define the scope of each project dependency. There are five types of dependencies which include:
  - **compile**: This is the default scope. Dependency is available in all class paths and is included in the package for the project (main) artifact. For example, a library which you are compiling against and whose classes are required at run time.

```xml
<dependency>
  <groupId>solms.co.za.mavenCourse</groupId>
  <artifactId>testLibrary</artifactId>
  <version>1.4.2</version>
  <scope>compile</scope>
</dependency>
```

- **provided**: Used to specify a dependency which would be provided by the deployment/execution environment. For example, the EJB, JTA, JPA, ... APIs would be provided by the application server. Dependencies still included in the class paths for compilation, but are not packaged in resultant artifact.
Specifying the scope of a dependency

```
<dependency>
  <groupId>javax.servlet</groupId>
  <artifactId>javax.servlet-api</artifactId>
  <version>[2.2,2.3)</version>
  <scope>provided</scope>
</dependency>
```

- **runtime**: Dependencies are required during execution and testing, but not for compilation. For example, you might need a particular API/contract jar at compile time, but the actual implementation classes for that API only at run time.

- **test**: Dependencies only required for the compilation and execution of tests. For example, a JUnit dependency would typically be scoped as a *test* dependency:
Specifying the scope of a dependency

```
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>[4,]</version>
  <scope>test</scope>
</dependency>
```

- **system**: Used to specify native system dependencies. If you declare a dependency with this scope, you will have to specify the `systemPath` element. System scope should be used only in very exceptional cases.
Transitive dependencies

- Transitive dependencies are resolved. Maven builds a dependency graph, resolving any conflicts if possible.
- For example, a dependency on the *Jena* framework for processing RDF and OWL may itself depend on *jena.arq* which is its implementation of the SparQL, which in turn depends on *Lucene* and *Xerces*. 
Dependency management

- Across projects and modules one may want to standardize dependencies to particular versions. For this purpose Maven introduces the concept of dependency management.

- Dependency versions can be defined in the `<dependencyManagement>` section of the parent POM. Dependencies are not automatically inherited by child projects, and will need to be explicitly specified at a lower level, however the version number can be omitted. Consider this top-level POM used by an organization:

```xml
1 <project>
2   <modelVersion>4.0.0</modelVersion>
3   <groupId>za.co.solms</groupId>
4   <artifactId>approvedLibs</artifactId>
5   <version>2.1</version>
6   ...
```
Projects could now inherit the approved versions for standard libraries used within the organization from the parent POM. The child project POMs would not specify version numbers for these libraries (except if a version needs to be overridden):
Dependency management

```xml
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>za.co.solms</groupId>
  <artifactId>courseNotesAssembly</artifactId>
  <version>0.8-SNAPSHOT</version>
  ...
  <dependencies>
    <dependency>
      <groupId>com.hp.hpl.jena</groupId>
      <artifactId>jena</artifactId>
    </dependency>
  </dependencies>
  ...
</project>
```
Exclusions

- One may want to exclude particular library versions. For example, to resolve conflicts caused by a transitive dependency being on a different version of a library than what is required.
- The following POM specifies a dependency on a testLibrary, but excludes a particular transitive dependency.

```xml
<project>
  ...
  <dependencies>
    <dependency>
      <groupId>solms.co.za.mavenCourse</groupId>
      <artifactId>testLibrary</artifactId>
      <version>1.4.2</version>
    </dependency>
    <exclusions>
      <exclusion>
        <groupId>xerces</groupId>
        <artifactId>xercesImpl</artifactId>
      </exclusion>
    </exclusions>
  </dependencies>
  ...
</project>
```
Project modules

- Modules used to specify part-of (aggregation) relationship between projects, and aggregates builds of sub-projects into a single higher-level build.
- POM for the aggregate project is specified which lists a number of sub-projects as modules.
- Sub-projects not identified via standard groupId-artifactId-version identifiers but assumed to be of the same groupId.
- Module name maps onto the artifactId of the sub-project and the sub-project needs to be contained in a sub-directory of the aggregate project.
- Inheritance relationship specified from the child projects perspective, with sub-projects being unaware that they are modules of an aggregate project.
- For example, the modules listed below are sub-projects of an enrollment system:
Project modules

Sub-projects contained in corresponding sub-directories of the enrollment system base directory, each with their own POM.

Common to declare the aggregate project the parent project of the sub-projects. Enables one to specify commonalities across sub-projects and introduce version dependency uniformity.
Build customization

- Build section of the POM used to customize standard Maven build life cycle.
- Involves adding additional build steps to the life cycle for the artifact type being built.
- One typically defines for a custom build step:
  - source and output directories for the sources and tests,
  - directories containing resources required for the build or tests,
  - plugins to be used, goals (services) to be executed, the build phase to which they should be bound, and any dependencies and configurations.
Customizing/configuring plugin behaviour

- Build section of the POM can be used to customize the behaviour of a defined goal. This is done in the configuration sub-element for the plugin element. For example, it is common to customize the clean plugin’s behaviour, by specifying additional directories/files which need to be removed during the clean phase.

```xml
<project>
  ....
  <plugins>
    <plugin>
      <artifactId>maven-clean-plugin</artifactId>
      <configuration>
        <filesets>
          <fileset>
            <directory>temp/generatedClasses</directory>
            <includes>
              <include>*.java</include>
            </includes>
          </fileset>
        </filesets>
      </configuration>
    </plugin>
  </plugins>
</project>
```
Adding a goal to a life cycle phase

- For example, to add a step to create Java classes from XML schemas, a build customization is specified which requests the generate goal of the jaxb2 plugin to create java classes from all XML schemas located in the src/main/resources/schemas directory.
- To use Java 6's built-in JAXB support we specify the source and target Java version for the Maven compile step.
- Finally we include the repositories from which the jaxb2 libraries are sourced. The resultant POM is shown below:

```xml
<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>za.co.solms.example</groupId>
  <artifactId>jaxb-maven-sample-java6</artifactId>
  <packaging>jar</packaging>
  <version>0.1</version>
</project>
```
Adding a goal to a life cycle phase

```xml
<name>JAXB / Maven Sample (Java 6+)</name>

<description>
A Sample Maven 2.x project that illustrates usage of the JAXB Maven plugin to compile XML Schema resources to Java. This is suitable for isolated work on XML documents. For web services, usage of the JAX-WS is recommended.
</description>

<developers>
  <developer>
    <organization>Solms TCD</organization>
    <organizationUrl>http://www.solms.co.za/</organizationUrl>
    <email>info@solms.co.za</email>
  </developer>
</developers>

<build>
  <plugins>
    <!-- Configuration to compile all schemas in the resources/schemas directory. Automatically invoke during the 'generate-sources' phase -->
    <plugin>
      <groupId>org.jvnet.jaxb2.maven2</groupId>
      <artifactId>maven-jaxb2-plugin</artifactId>
    </plugin>
  </plugins>
</build>
```
Adding a goal to a life cycle phase

```xml
<executions>
  <execution>
    <goals>
      <goal>generate</goal>
    </goals>
  </execution>
</executions>

<configuration>
  <schemaDirectory>
    src/main/resources/schemas
  </schemaDirectory>
  <schemaIncludes>
    <include>*.xsd</include>
  </schemaIncludes>
</configuration>

<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-compiler-plugin</artifactId>
  <configuration>
    <source>1.6</source>
    <target>1.6</target>
  </configuration>
</plugin>

<!-- Assume a Java SE 6 environment, which includes JAXB. For Java 5, extra dependencies on the JAXB Implementation should be specified -->
```
Adding a goal to a life cycle phase

```xml
<configuration>
  <plugin>
    <plugins>
      <build>
        <dependencies>
          <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>[4.1,)</version>
            <scope>test</scope>
          </dependency>
        </dependencies>
        <repositories>
          <repository>
            <id>maven2-repository.dev.java.net</id>
            <name>Java.net Maven 2 Repository</name>
            <url>http://download.java.net/maven/2</url>
          </repository>
        </repositories>
      </build>
    </plugins>
  </plugin>
</configuration>
```
Adding a goal to a life cycle phase

```
<url>http://download.java.net/maven/2</url>
</pluginRepository>
</pluginRepositories>
</project>
```
Distribution Information

- Distribution information can be specified in the distributionManagement section of the project’s POM. This is used for deployment, or to publish the resultant artifact for use by other projects.

- One can specify the URL which should be used by other projects when downloading the artifact. This is done in the downloadURL element. A status element is used to specifies the status of the distribution.
Specifying distribution Info

- The distribution management section contains an id, name and url of the repositories for the project’s artifacts, and allows for a main and a snapshot repository.
- For example,

```xml
  ...
  <distributionManagement>
    <repository>
      <uniqueVersion>false</uniqueVersion>
      <id>solmsMain</id>
      <name>Solms Main Repository</name>
      <url>scp://solms.co.za/repository/main</url>
      <layout>default</layout>
    </repository>
  </distributionManagement>
</project>
```
specifies a main and a development repository. The development repository adds time stamp information to the artifact names in order to ensure unique versions.
Specifying login credentials

- Server login credentials should not be distributed with the POM. These should remain on the build server and are specified in the settings.xml file:

```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">
  ...
  <servers>
    <server>
      <id>solms.co.za</id>
      <username>myUserName</username>
      <password>myPassword</password>
      <privateKey>${user.home}/.ssh/id_dsa</privateKey>
      <passphrase>myPassphrase</passphrase>
      <filePermissions>664</filePermissions>
      <directoryPermissions>775</directoryPermissions>
    </server>
    ...
  </servers>
</settings>
```
Maven’s life cycles

- Three standard life cycles to build, clean or document projects are provided. They are not plugin specific:
  - **build life cycle** compile, test, package, install and deploy a project.
  - **clean life cycle** remove generated directories and files so that only the sources remain.
  - **site life cycle** measures the project (e.g. collect test results) and generates documentation containing project metadata and reporting project status.

- Life cycle is a pipeline of phases which is executed sequentially. The default build life cycle contains, but is not limited to, the following core phases:
  - validate, compile, test, package, deploy and install

- Build process follows the phases in the order of the life cycle, executing any goals/operations which are bound to a phase.
Maven’s life cycles

- Plugin goals are bound to phases in one of the following ways
  - The plugin for the project packaging (project type) binds goals/operations to its life cycle phases.
  - In your build customization you can bind different goals/operations to phases.
- Life cycle used is determined by the plugin for the specified package type (e.g. jar or war). The plugin may hence choose to either
  - bind goals to the standard life cycle phases,
  - inherit a standard life cycle and add phases, or
  - define a new life cycle with its own phases.
Maven’s default build life cycle

Maven’s default build life cycle contains the following phases:

- **validate**: Confirm that project has all information required to build the project.
- **generate-sources**: Generate sources from other artifacts if needed.
- **process-sources**: Pre-process sources before compilation. For example, filtering and obfuscation.
- **generate-resources**: If needed, create resources for inclusion in final assembly.
- **process-resources**: Processing of the generated resources including copying, renaming, and setting of permissions.
- **compile**: Source code compilation.
- **process-classes**: Post-compile processing such as byte-code enhancements or annotation processing.
Maven’s default build life cycle

- **generate-test-sources**: Used if test sources need to be generated from other artifacts.
- **generate-test-resources**: Generate resources for the testing. For example test data generation.
- **process-test-resources**: Filter resources or copy them to locations for packaging.
- **test-compile**: Compile the test sources.
- **test**: Test compiled sources. For example unit testing.
- **prepare-package**: Organise resources for packaging before the actual packaging is done. May include un-packaging certain packaged resources.
- **package**: Create the distributable package, e.g. the jar or war.
- **pre-integration-test**: Set up the environment for the integration test.
Maven’s default build life cycle

- **integration-test**: Perform testing in an environment which mirrors destination environment.
- **post-integration-test**: Clean-up phase which restores the environment within which the integration test was done.
- **verify**: Verify whether the package meets certain quality criteria.
- **install**: Install the package in a local repository so that other dependent projects can use it.
- **deploy**: Copy final package to a remote repository for sharing with other groups and projects.
Maven’s clean life cycle

Maven’s *clean* defines the following phases:

1. pre-clean
2. clean
3. post-clean

By default only the *clean* plugin’s *clean* goal/operation is bound to the *clean* phase.
Maven’s site life cycle

The *site* life cycle is used for generating project documentation. It has the following phases:

1. pre-site
2. site
3. post-site
4. site-deploy
Default goal bindings for the site life cycle

The *site* plugin binds the

- `site:site` goal to the *site phase*, and the
- `site:deploy` goal to the *site-deploy phase*. 

A project's packaging identifies the main plugin that determines the build for the project type. Many plugins simply bind goals to standard life cycle phases. Others define their own life cycle and corresponding goal bindings.

For example, the *jar ejb* and *war* plugins all bind goals to a subset of the default build life cycle. These bindings are in fact the same, except for the bindings for the package phase:

- *process-resources* $\rightarrow$ *resources:resources*
- *compile* $\rightarrow$ *compiler:compile*
- *process-test-resources* $\rightarrow$ *resources:testResources*
Package-based goal bindings

- test-compile $\rightarrow$ compiler:testCompile
- test $\rightarrow$ surefire:test
- package $\rightarrow$ respectively to jar:jar, ejb:ejb and war:war
- install $\rightarrow$ install:install
- deploy $\rightarrow$ deploy:deploy
Project based life cycle goals

- Initial goal binding determined by packaging (project type).
- Additional goal bindings can be specified per project or per parent project (bindings inherited). Specified in POM's `<<build>>` customization element.

```xml
<project>
  ...
  <build>
    <plugins>
      <plugin>
        <groupId>pluginGroupId</groupId>
        <artifactId>pluginArtifactId</artifactId>
        <executions>
          <execution>
            <phase>somePhase</phase>
          </execution>
        </executions>
        <goals>
          <goal>
            someGoal1
          </goal>
        </goals>
      </plugin>
    </plugins>
  </build>
</project>
```
Project based life cycle goals

```xml
<executions>
  <plugin>
    <goals>
      <goal>
        someGoal2
      </goal>
    </goals>
  </plugin>
</executions>
```
Executing Maven

The syntax for executing Maven is

```
1 mvn [option]* [plugin:goal]* [phase]*
```

specifying at least one plugin goal or life cycle phase and zero or more options.
Executing a plugin goal

When executing a plugin goal only the operation/service representing that goal for the plugin is executed. This is done via

```
  mvn [option]* pluginName:goalName
```

For example,

```
  mvn eclipse:eclipse
```

executes the `eclipse` goal of the `eclipse` plugin which creates an eclipse project for the Maven project.
Executing a life cycle phase

- When executing a life cycle phase, all preceding phases of the life cycle are executed.
- The life cycle is determined by the main plugin which is chosen on the project’s package type. It may be one of the standard life cycles (default build, clean or site) or a custom life cycle.
- In either case, the plugin will bind its goals as well as goals from other plugins to the life cycle phases.
- Maven this
  - determines the package type, and locates its corresponding plugin,
  - obtains the life cycle phases from the plugin, and the goals/operations bound to each of the phases (to the requested life cycle phase), and
  - executes the phases in sequence by executing any plugin goals bound to each phase.
For example,

```
maven test
```

for a jar project executes all phases prior to and including the test phase, i.e. all goals bound to these build cycle phases.

- It is preferable to execute life cycle phases as opposed to plugin goals as the dependencies for the build are resolved.
Maven repositories

- The Maven repository is a central component of Maven, facilitating universal reuse of project artifacts including Maven plugins.
- A repository is simply a storage space for reusable artifacts generated from various projects and the Project Object Model (POM) describing how the artifacts can be built and what the artifact dependencies are.
The Maven repository is simply a file system with a tree structure conforming to

```
<groupId>.<artifactId>.<version>
```

hierarchy. That directory will contain

- the actual artifact,
- the POM for the project with which the artifact was created,
- and a hashing key which is used to verify that the received artifact is not corrupted.
Repository locations

- Maven uses both, remote and local repositories.
- Local and remote repositories are both structured in the same way.
- Since the local and remote repositories have the same structure, they can be simply synchronized.
Remote repositories

Remote repositories one uses typically include
- The central Maven repository containing core maven plugins and globally published maven projects,
- Internal group/organization specific remote repositories where internal projects artifacts are published and shared.

There is no structural difference between internal and external remote repositories. Easy to synchronize.
Accessing remote repositories

- Files can be retrieved from remote repositories either via the HTTP or via a mechanism like SCP using the file://URL protocol.
- Security can be done via HTTPS or simply by restricting the user access to the directory.
- Files can be uploaded to a repository using SCP, FTP or other file copy mechanisms.
Specifying the location of remote repositories

Location of the official central Maven artifact and plugin repositories are specified in the Super-POM:

```xml
<project>
  <modelVersion>4.0.0</modelVersion>
  <name>defaultProject</name>

  <repositories>
    <repository>
      <id>central</id>
      <name>Maven Repository Switchboard</name>
      <layout>default</layout>
      <url>http://repo1.maven.org/maven2</url>
      <snapshots><enabled>false</enabled></snapshots>
    </repository>
  </repositories>
</project>
```
Specifying the location of remote repositories

Further repository and plugin repository specifications can be specified in either the POM for your project or in a common parent POM for your projects.
The local repository

- acts as a cache for remote repositories, and
- hosts the artifacts of local projects.

It has the same structure as the remote repositories and can be synchronized with them.

The location of your local repository is

1. $\{\text{home}\}/.m2/repository
Maven resolves dependencies by sourcing the metadata (POM) for the dependency from either the local cache repository or one of the remote repositories.

From the POM it obtains the transitive dependencies. This is done recursively building up a dependency tree.

Maven attempts to resolve conflicts and source the actual artifacts for the dependencies, by first querying the local cache and then remote caches.

Integrity of sourced dependency is validated using the hash which is also obtained from the repository.
Repository tools

- To create a Maven repository, it is sufficient to simply create the appropriate directory structure containing the artifacts, poms and hash keys and then to serve the file system by some mechanism (typically via HTTP).
- However, Maven repositories provide additional functionality like:
  - searching,
  - providing tools for conveniently navigating the repository, and
  - publishing the metadata in a convenient way.
Archetype = template for project type
  Many templates for many project types

1 mvn archetype:generate -DgroupId=za.co.solms.training.maven
2     -DartifactId=helloWorld

See generated POM & Project Tree
**Compile**

1. `mvn compile`

- executes all life cycle phases up to and including the compile phase.
- compiled classes stored in `target/classes`
- Prior phases executed including `process-resources`
  - Copies resources from `src/main/resources` to phase `target/classes`
  - so that in class path
  - will be included in packaging
Test

- To execute up to the test phase:

  1. mvn test

- includes compiling and executing test programs.
- Test results saved in

  1. target/surefire-reports
To execute life cycle phases up to & including creating jar

mvn package

Resultant jar stored in

target
Install

- To execute all phases up to & including package phase

  `mvn install`

- Will do compilation, test compilation & execution, packaging
- Will copy resultant artifact into your local maven repository.
Deploy phase

- Last phase of default build life cycle
- Requires distribution information in POM
- Specify authentication details in a separate settings.xml
  - NOT distributed when the project is deployed.
Specifying Distribution Indo

```xml
    <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>za.co.solms.training.maven</groupId>
      <artifactId>helloWorld</artifactId>
      <packaging>jar</packaging>
      <version>1.0-SNAPSHOT</version>

      ...  

      <distributionManagement>
        <snapshotRepository>
          <uniqueVersion>true</uniqueVersion>
          <id>solmsRepository</id>
          <name>Solms Internal Repository</name>
          <url>scp://solms.co.za/var/maven/repository</url>
        </snapshotRepository>
      </distributionManagement>

    </project>
```
SSH key for login without password

1. `ssh-keygen -t dsa`
2. Generating public/private dsa key pair.
3. Enter file in which to save the key (`/home/clientUser/.ssh/id_dsa`):
4. Enter passphrase (empty for no passphrase):
5. Enter same passphrase again:
6. Your identification has been saved in `/home/clientUser/.ssh/id_dsa`.
7. Your public key has been saved in `/home/clientUser/.ssh/id_dsa.pub`.
8. The key fingerprint is:
10. The key's randomart image is:
11. `+---[ DSA 1024]-----+
12. ...+-----------------------+

Use ssh to create `.ssh` in user’s home directory on server:

1. `ssh serverUser@serverMachine mkdir -p .ssh`
2. Password:

    1. `cat .ssh/id_dsa.pub | ssh serverUser@serverMachine 'cat >> .ssh/authorized_keys'`
    2. Password:

You can test that password no longer required via...
Specifying Auth Settings

- **settings.xml:**

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<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">
  <servers>
    <server>
      <id>repository</id>
      <username>fritz</username>
      <privateKey>~/ssh/id_dsa</privateKey>
    </server>
  </servers>
</settings>
```
Executing Specific Plugin Goals

- May want to
  - execute a goal which is not bound to any of default life cycle phases, or
  - execute goal without any of goals bound to previous life cycle phases.
- E.g. Execute compiler’s compile goal

```
mvn compiler:compile
```
Executing a Program from Maven

- **Execute Java class:**

  ```
  mvn exec:java -Dexec.mainClass=za.co.solms.training.maven.App
  ```

- **Execute binary:**

  ```
  mvn exec:exec -Dexec.executable="binaryExecutable"
  -Dexec.workingdir="~/temp" -Dexec.args="-arg1 -arg2"
  ```

- **e.g. Directory listing:**

  ```
  > mvn exec:exec -Dexec.executable="ls" -Dexec.args="-l -a"
  [INFO] Scanning for projects...
  [INFO] Searching repository for plugin with prefix: 'exec'.
  [INFO]
  [INFO] Building helloWorld
  [INFO] task-segment: [exec:exec]
  [INFO]
  [INFO] [exec:exec]
  [INFO] total 8
  [INFO] drwxr-xr-x 3 fritz fritz 80 Dec 13 06:28 ..
Generating Documentation

- Use *site* plugin to generate documentation for project:
  
  ```
  mvn site
  ```

- `index.html` stored in `target/site`
- Deploy documentation onto project documentation server via
  
  ```
  mvn site−deploy
  ```

- Requires that site distribution information provided in a `site` sub-element of `distributionManagement`:
  
  ```
  <project>
  ...  
  <distributionManagement>
      <site>
          <id></id>
          <name></name>
          <url></url>
      </site>
  </distributionManagement>
  </project>
  ```
Cleaning the Project

- use clean plugin
  - removes all generated files and directories
  - leaves all sources incl resources and POM

`mvn clean`
JAXWS Example

- Look at source and execute