FREE eBook

LEARNING Apache Maven

Free unaffiliated eBook created from
Stack Overflow contributors.

#maven

Table of Contents

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: apache-maven

It is an unofficial and free Apache Maven ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official Apache Maven.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with Apache Maven

Remarks

As described by its official Start Guide:

> Maven is an attempt to apply **patterns** to a project's build infrastructure in order to promote comprehension and productivity by providing a clear path in the use of **best practices**.

Maven is essentially a project management and comprehension tool and as such provides a way to help with managing:

- Builds
- Documentation
- Reporting
- Dependencies
- Version Control
- Releases
- Distribution

Hence, supporting developers across many phases of the whole Software Development Life Cycle (SDLC).

This philosophy is part of Maven in its core: i.e., the word *maven* means *accumulator of knowledge* (in Yiddish).

Maven is about the application of **patterns** in order to achieve an infrastructure which displays the characteristics of visibility, reusability, maintainability, and comprehensibility.

- Maven was born of the very practical desire to make several projects work in the same way, as stated by the official Maven philosophy statement.
- Developers could freely move between projects, knowing clearly how they all worked by understanding how one of them worked
- The same idea extends to testing, generating documentation, generating metrics and reports, and deploying

Versions

Version	Announce	Comment	Release Dates
1.0-beta-2	announce	First (beta) release	2002-03-30
1.0	announce	First official release	2004-07-13

Version	Announce	Comment	Release Dates
2.0	announce	Official **2.0** release	2005-10-20
3.0	announce	Official **3.0** release	2010-10-08

Examples

Installation or Setup

Binary releases of Maven can be downloaded from the Maven website.

The binary comes as a zip archive or as a tar.gz archive. After downloading it, the instructions from the install page can be followed:

- Ensure the `JAVA_HOME` environment variable is set and points to your JDK installation (not JRE). For example, on a Windows machine, this installation folder can correspond `C:\Program Files\Java\jdk1.8.0_51`.
- Extract the distribution archive in the directory of your choice.
- Add the `bin` directory of the created directory (named `apache-maven-3.3.9` for Maven 3.3.9) to the `PATH` environment variable. (Reference to change it on Windows).
- Verify that the set-up is correct by running `mvn -version` on the command line.

There is no need to set the `M2_HOME` or `MAVEN_HOME` environment variable.

Installation on Ubuntu

1. In a terminal run `sudo apt-get install maven`

2. Once the install is over check that it works correctly with `mvn -v` the output should look like:

```
Apache Maven 3.3.9
Maven home: /usr/share/maven
Java version: 1.8.0_121, vendor: Oracle Corporation
Java home: /usr/lib/jvm/java-8-openjdk-amd64/jre
Default locale: en_US, platform encoding: UTF-8
OS name: "linux", version: "4.8.0-parrot-amd64", arch: "amd64", family: "unix"
```

If this does not work, make sure you have a JDK installed in your environment`javac -version`

Configuring Proxy Settings

If your Internet connection is provided via a proxy Maven will not be able to download jars from remote repositories - a common problem faced by companies.

To solve this, Maven needs to be provided the details and credentials of the proxy by going to *{Maven install location}* → *conf* → `settings.xml`. Scroll down to the `<proxies>` tag and enter the details here, using the format mentioned in the comments.

For Eclipse users

Eclipse uses it's own `settings.xml` file for running Maven, whose location can be found by going to the menu *Window → Preferences → Maven → User Settings → User Settings:*. If the file is not available in the location mentioned, simply create it yourself or create a duplicate of the file from the above location *{Maven install location} → conf →* `settings.xml`.

For IntelliJ users

Open the settings and navigate to Maven -> Importing. (This may be nested under Build, Execution, Deployment -> Build Tools ->, depending on the IntelliJ version you're using.)

Set the field named "VM options for importer" like:

```
-DproxySet=true -DproxyHost=<HOST> -DproxyPort=<PORT>
-DproxySet=true -DproxyHost=myproxy.com -DproxyPort=8080
```

Apply and restart IntelliJ.

Installation on Mac OSX with Brew

1. In a terminal run `brew install maven`
2. Once the install is over check that maven works correctly with `mvn -v`. The output should look something like:

```
Apache Maven 3.3.9
Maven home: /usr/local/Cellar/maven/3.3.9/libexec
Java version: 1.8.0_121, vendor: Oracle Corporation
Java home: /Library/Java/JavaVirtualMachines/jdk1.8.0_121.jdk/Contents/Home/jre
Default locale: en_US, platform encoding: UTF-8
OS name: "mac os x", version: "10.12.4", arch: "x86_64", family: "mac"
```

If this does not work, make sure you have a JDK installed in your environment `javac -version`

Read Getting started with Apache Maven online: https://riptutorial.com/maven/topic/898/getting-started-with-apache-maven

Chapter 2: Access Maven informations in code

Introduction

It is sometimes useful to get the maven properties, such as the current version, in code. Here are some ways to to it.

Examples

Getting the version number from within a jar

If you package your application in a `jar` using the `maven-jar-plugin` or the `maven-assembly-plugin`, an easy way to get the current pom version is to add an entry in the manifest, which is then available from Java.

The secret is to set the `addDefaultImplementationEntries` flag to true (and the `addDefaultSpecificationEntries` is you also need the artifact id).

jar plugin configuration:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-jar-plugin</artifactId>
      <configuration>
        <archive>
          <manifest>
            <mainClass>...</mainClass>
            <addDefaultImplementationEntries>
              true
            </addDefaultImplementationEntries>
          </manifest>
        </archive>
      </configuration>
    </plugin>
  </plugins>
</build>
```

assembly plugin configuration:

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-assembly-plugin</artifactId>
    <configuration>
        <descriptorRefs>
            <descriptorRef>jar-with-dependencies</descriptorRef>
        </descriptorRefs>
```

```
        <archive>
            <manifest>
              <addDefaultImplementationEntries>true</addDefaultImplementationEntries>
            </manifest>
        </archive>
    </configuration>
    <executions>
        <execution .../>
    </executions>
</plugin>
```

`addDefaultImplementationEntries` instructs Maven to add the following headers to the `MANIFEST.MF` of your jar:

```
Implementation-Title: display-version
Implementation-Version: 1.0-SNAPSHOT
Implementation-Vendor-Id: test
```

Now you can use this line of code anywhere in your jar to access the version number:

```
getClass().getPackage().getImplementationVersion()
```

More information here and here.

Keeping a properties file in sync using maven's property filtering mecanism

As this documentation explains,

> Sometimes a resource file will need to contain a value that can only be supplied at build time. To accomplish this in Maven, put a reference to the property that will contain the value into your resource file using the syntax `${<property name>}`. The property can be one of the values defined in your `pom.xml`, a value defined in the user's `settings.xml`, a property defined in an external properties file, or a system property.

As an example, let's create a simple `info.txt` in `src/main/resources` containing the pom version and the build time.

1. create a `src/main/resources/info.txt` with the following content:

   version=${pom.version} build.date=${timestamp}

2. ask Maven to *expand* the properties by setting `filtering` to true:

   ```
   <build>
       <resources>
           <resource>
              <directory>src/main/resources</directory>
              <filtering>true</filtering>
           </resource>
       </resources>
   </build>
   ```

3. with that, the version will be updated, but unfortunately a bug within Maven prevents the `${maven.build.timestamp}` property from getting passed to the resource filtering mechanism (more info here). So, let's create a `timestamp` property as a workaround ! Add the following to the pom's properties:

```
<properties>
    <timestamp>${maven.build.timestamp}</timestamp>
    <maven.build.timestamp.format>yyyy-MM-dd'T'HH:mm</maven.build.timestamp.format>
</properties>
```

4. run maven, you should find a `info.txt` in `target/classes` with a content like:

```
version=0.3.2
build.date=2017-04-20T13:56
```

Reading a pom.xml at runtime using maven-model plugin

The other examples may be the best and most stable way to get a version number into an application ***statically***. This answer proposes an alternative showing how to do it ***dynamically*** during runtime, using the maven *maven-model* library.

Add the dependency:

```
<dependency>
  <groupId>org.apache.maven</groupId>
  <artifactId>maven-model</artifactId>
  <version>3.3.9</version>
</dependency>
```

In Java, create a `MavenXpp3Reader` to read your pom. For example:

```
package de.scrum_master.app;

import org.apache.maven.model.Model;
import org.apache.maven.model.io.xpp3.MavenXpp3Reader;
import org.codehaus.plexus.util.xml.pull.XmlPullParserException;

import java.io.FileReader;
import java.io.IOException;

public class MavenModelExample {
    public static void main(String[] args) throws IOException, XmlPullParserException {
        MavenXpp3Reader reader = new MavenXpp3Reader();
        Model model = reader.read(new FileReader("pom.xml"));
        System.out.println(model.getId());
        System.out.println(model.getGroupId());
        System.out.println(model.getArtifactId());
        System.out.println(model.getVersion());
    }
}
```

The console log is as follows:

```
de.scrum-master.stackoverflow:my-artifact:jar:1.0-SNAPSHOT
de.scrum-master.stackoverflow
my-artifact
1.0-SNAPSHOT
```

Read Access Maven informations in code online: https://riptutorial.com/maven/topic/9773/access-maven-informations-in-code

Chapter 3: Create a Maven Plugin

Introduction

Maven allows you to implement and use custom plugins. These plugins allow additional behaviour to be bound to any phase of the Maven lifecycle.

Each Maven goal is created by implementing a MOJO (Maven Ordinary Java Object): a Java class implemented with annotations that describes how to invoke it.

The goal prefix of a plugin is derived from its artifact name. An artifact `hello-world-plugin` creates a goal prefix `hello-world`. The `hello` goal can then be run with `mvn hello-world:hello`.

Remarks

A Maven plugin is a JAR containing a `maven/plugins.xml` that describes the plugin metadata. This file is generated by the `maven-plugin-plugin`.

Examples

Declaring a Maven artifact as a Maven plugin

An artifact built by Maven can be declared as a Maven plugin by specifying the packaging as `maven-plugin` in the `pom.xml`.

```
<packaging>maven-plugin</packaging>
```

You need to declare a dependency on the plugin API and annotations.

```
<dependency>
    <groupId>org.apache.maven</groupId>
    <artifactId>maven-plugin-api</artifactId>
    <version>3.3.9</version>
    <scope>provided</scope>
</dependency>
<dependency>
    <groupId>org.apache.maven.plugin-tools</groupId>
    <artifactId>maven-plugin-annotations</artifactId>
    <version>3.5</version>
    <scope>provided</scope>
</dependency>
```

You need to add a plugin to generate the metadata.

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-plugin-plugin</artifactId>
    <version>3.5</version>
```

```
</plugin>
```

Creating a goal

Goals are implemented by creating a MOJO. This is a class file annotated with annotations from `maven-plugin-annotations`.

```
@Mojo(name = "hello")
public final class HelloWorldMojo extends AbstractMojo {

    public void execute() throws MojoExecutionException, MojoFailureException {
         getLog().info("Hello world");
    }
}
```

Using plugin configuration

Plugins can be configured by annotating fields with `@Parameter`. The MOJO is then injected with the configuration.

```
@Mojo(name = "greet")
public final class GreetMojo extends AbstractMojo {

    @Parameter(required = true)
    public String name;

    public void execute() throws MojoExecutionException, MojoFailureException {
        getLog().info("Hello " + name);
    }
}
```

The `name` parameter can be configured in the POM:

```
<plugin>
    <groupId>com.mattunderscore</groupId>
    <artifactId>hello-world-plugin</artifactId>
    <version>1.0-SNAPSHOT</version>
    <configuration>
        <name>Matt</name>
    </configuration>
</plugin>
```

If the `greet` goal is run as a standalone goal the `name` parameter can be defined as property on the command line:

```
mvn <plugin name>:greet -Dname=Geri
```

Accessing the project information

The plugin can, among others, access information about the current Maven project being built.

```
@Mojo(name = "project")
public final class ProjectNameMojo extends AbstractMojo {

    @Parameter(defaultValue = "${project}", readonly = true, required = true)
    private MavenProject project;

    public void execute() throws MojoExecutionException, MojoFailureException {
        getLog().info("Hello, this is " + project.getName());
    }
}
```

The above example would print in the console the name of the Maven project it is run on, which is specified in the `<project>/<name>` element of its POM.

The `MavenProject` class used in the plugin requires a dependency of `maven-core` with a (default) `compile` scope in the plugin's POM:

```
<dependency>
    <groupId>org.apache.maven</groupId>
    <artifactId>maven-core</artifactId>
    <version>3.3.9</version>
</dependency>
```

Furthermore, using annotations requires the following dependency in the plugin's POM:

```
<dependency>
  <groupId>org.apache.maven.plugin-tools</groupId>
  <artifactId>maven-plugin-annotations</artifactId>
  <version>3.5</version>
  <scope>provided</scope> <!-- annotations are needed only to build the plugin -->
</dependency>
```

Declare a default phase for a goal

```
@Mojo(name = "hi", defaultPhase = LifecyclePhase.COMPILE)
```

Get the build directory as a file

```
@Parameter(defaultValue = "${project.build.directory}")
private File buildDirectory;
```

Read Create a Maven Plugin online: https://riptutorial.com/maven/topic/8635/create-a-maven-plugin

Chapter 4: Eclipse integration

Examples

Install Maven in Eclipse

You can take advantage of Apache Maven's powerful features in Eclipse by installing the M2Eclipse feature. Follow these steps to install Maven in Eclipse:

1. Open Eclipse and select *Help* → *Install New Software…*
2. In the opened dialog, select the `Add...` button to add a new repository.
3. Fill in the form with the information below and confirm with `OK`:

   Name: `M2Eclipse`

   Location: `http://download.eclipse.org/technology/m2e/releases`
4. After the *Pending...* finishes, `Select All` and select `Next`.
5. ⊙ `accept the terms of the license agreement and select` `Finish`.
6. At the end of the installation you will be asked to restart Eclipse. Select `Yes` to perform the restart.

Check if Eclipse already has M2Eclipse Maven support installed

Go to *Help* → *About Eclipse* → Check if the *m2e feature* is there: .

Configure a custom Maven installation in Eclipse

Eclipse would provide its own embedded Maven enviroment out-of-the-box, which is not recommended whenever a certain Maven version must be used or further configuration must be performed (proxy, mirrors and so on): that is, to have full control over which Maven environment would be used by the IDE.

- Select *Window* → *Preferences* → *Maven* → *Installations*
- Select `Add..` to add a custom/local Maven installation
- Supply the necessary information and select `Finish`:

  Installation home: `... your Maven home ... Directory...`

  Installation name: `Apache Maven x.y.z`
- ☑ select it as default (instead of the default *EMBEDDED* version) and confirm with `OK`.

Read Eclipse integration online: https://riptutorial.com/maven/topic/2315/eclipse-integration

Chapter 5: Generate FIXME/TODO reports using the taglist-maven-plugin

Introduction

This is a small code (xml) snippet to highlight how to use the taglist-maven-plugin to generate customized reports (of TODO, FIXME work ...)

Examples

pom.xml to generate a FIXME report

```
<build>
    <plugins>
        <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>project-info-reports</artifactId>
                <version>2.9</version>
                <executions>
                    <execution>
                        <goals>
                            <goal>index</goal>
                        </goals>
                        <phase>site</phase>
                    </execution>
                </executions>
            </plugin>
    </plugins>
</build>

<reporting>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-project-info-reports-plugin</artifactId>
                <version>2.9</version>
                <reportSets>
                    <reportSet>
                    <reports>
                        <report>index</report>
                        <report>issue-tracking</report>
                    </reports>
                    </reportSet>
                </reportSets>
            </plugin>
            <plugin>
                <groupId>org.codehaus.mojo</groupId>
                <artifactId>taglist-maven-plugin</artifactId>
                <version>2.4</version>
                <configuration>
                    <tagListOptions>
                        <tagClasse>
                            <displayName>FIXME Work</displayName>
```

```
                        <tags>
                            <tag>
                                <matchString>FIXME</matchString>
                                <matchType>ignoreCase</matchType>
                            </tag>
                            <tag>
                                <matchString>@fixme</matchString>
                                <matchType>ignoreCase</matchType>
                            </tag>
                        </tags>
                    </tagClasse>
                    <tagClasse>
                        <displayName>TODO Work</displayName>
                        <tags>
                            <tag>
                                <matchString>TODO</matchString>
                                <matchType>ignoreCase</matchType>
                            </tag>
                            <tag>
                                <matchString>@todo</matchString>
                                <matchType>ignoreCase</matchType>
                            </tag>
                        </tags>
                    </tagClasse>
                </tagListOptions>
            </configuration>
        </plugin>
    </plugins>
</reporting>
```

Then run

```
mvn clean site:site
```

Read Generate FIXME/TODO reports using the taglist-maven-plugin online: https://riptutorial.com/maven/topic/10110/generate-fixme-todo-reports-using-the-taglist-maven-plugin

Chapter 6: Maven Assembly Plugin

Examples

Creating .jar file with all the dependencies of the project

To create a JAR containing all of its dependencies, it is possible to use the built-in descriptor format `jar-with-dependencies`. The following example configures an execution of the Assembly Plugin bound to the `package` phase, using this built-in descriptor and declaring a main class of `com.example`:

```
<plugin>
  <artifactId>maven-assembly-plugin</artifactId>
  <version>2.6</version>
  <executions>
    <execution>
      <id>make-assembly</id>
      <phase>package</phase>
      <goals>
        <goal>single</goal>
      </goals>
      <configuration>
        <archive>
          <manifest>
              <mainClass>com.example</mainClass>
          </manifest>
        </archive>
        <descriptorRefs>
          <descriptorRef>jar-with-dependencies</descriptorRef>
        </descriptorRefs>
      </configuration>
    </execution>
  </executions>
</plugin>
```

Running:

```
mvn clean package
```

on the command-line will result in the jar-with-dependencies to be built and attached to the project.

If more control over this uber-jar is needed, turn to the Maven Shade Plugin.

Read Maven Assembly Plugin online: https://riptutorial.com/maven/topic/2308/maven-assembly-plugin

Chapter 7: Maven Build Cycle

Introduction

Following is a complete list of Maven's default build lifecycle phases. Each of these phases is invoked by adding it to the `mvn` command, e.g. `mvn install`.

Examples

Maven Build Lifecycle Phases

```
validate
```

Validates whether the project is correct and all the required information are available for the build.

```
initialize
```

Initializes the build environment, e.g. sets properties or creates directories.

```
generate-sources
```

Generates source code to be processed in the 'compile' phase.

```
process-sources
```

Processes the source code in case some filter need to be applied.

```
generate-resources
```

Generates resources to be included in the artifact.

```
process-resources
```

Processes and copies resources into the output directory (`${basedir}/target/classes`).

```
compile
```

Compiles the project's source code in the source directory (`${basedir}/src/main/[java|groovy|...]`) into the output directory (`${basedir}/target/classes`).

```
process-classes
```

Processes `.class` files generated in the `compile` phase, e.g. to perform bytecode enhancements.

```
generate-test-sources
```

Generates test source code to be processed in the `test-compile` phase.

```
process-test-sources
```

Processes test source code in case some filter need to be applied.

```
generate-test-resources
```

Generates resources for testing.

```
process-test-resources
```

Processes and copies test resources in the resources directory (`${basedir}/src/main/resources`) into the test output directory (`${basedir}/target/test-classes`).

```
test-compile
```

Compiles source code in the test source directory ('${basedir}/src/test/[java|groovy|...]') into the test output directory(`${basedir}/target/test-classes`).

```
process-test-classes
```

Processes test `.class` files generated in the `test-compile` phase, e.g. to perform bytecode enhancements (Maven 2.0.5 and above).

```
test
```

Runs tests using some suitable test framework. Note: These test cases are not considered for packaging and deploying.

```
prepare-package
```

Performs final changes and validations before the package is finally created.

```
package
```

Packages the successfully compiled and tested code into some distributable format like JAR, WAR, EAR into the target directory (`${basedir}/target`).

```
pre-integration-test
```

Performs actions before integration tests are run if they require to apply some changes in the environment for the application.

```
integration-test
```

Processes and possibly deploys the application to an environment where integration tests can be run.

```
post-integration-test
```

Performs actions after the integration tests, like cleaning up the environment which has been created in the `pre-integration-test` phase.

```
verify
```

Checks whether a package is valid and meets required quality criteria.

```
install
```

Installs the artifact into the local repository. Any other local project can use this artifact as one of its dependencies after that (if your IDE doesn't support *workspace dependency resolution* anyway).

```
deploy
```

Copies the package to a remote repository to make it available for other developers.

Read Maven Build Cycle online: https://riptutorial.com/maven/topic/9679/maven-build-cycle

Chapter 8: Maven EAR plugin

Introduction

Here is an example configuration for a basic maven ear plugin for packaging both .war and .jar artifacts

Examples

A basic EAR configuration

```
<dependencies>
    <dependency>
        <groupId>{ejbModuleGroupId}</groupId>
        <artifactId>{ejbModuleArtifactId}</artifactId>
        <version>{ejbModuleVersion}</version>
        <type>ejb</type>
    </dependency>
    <dependency>
        <groupId>{webModuleGroupId}</groupId>
        <artifactId>{webModuleArtifactId}</artifactId>
        <version>{webModuleVersion}</version>
        <type>war</type>
    </dependency>
</depencencies>
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-ear-plugin</artifactId>
                <version>2.9.1</version>
                <configuration>
                    <version>1.4</version><!-- application.xml verion -->
                    <modules>
                        <ejbModule>
                            <groupId>{ejbModuleGroupId}</groupId>
                            <artifactId>{ejbModuleArtifactId}</artifactId>
                        </ejbModule>
                        <webModule>
                            <groupId>{webModuleGroupId}</groupId>
                            <artifactId>{webModuleArtifactId}</artifactId>
                            <contextRoot>/custom-context-root</contextRoot>
                        </webModule>
                    </modules>
                </configuration>
        </plugin>
    </plugins>
</build>
```

Once compiled `mvn clean install`, generates an *.ear* artifact in the *target* directory containing both the ejb (.jar) and the web module, along with the *application.xml* JEE description file.

Read Maven EAR plugin online: https://riptutorial.com/maven/topic/10111/maven-ear-plugin

Chapter 9: Maven install in window

Introduction

how to install maven in window 7

Remarks

how to install maven in window 7 Steps:

1. download the maven form https://maven.apache.org/download.cgi (poffice website) 2.unzip the maven binary folder and save into any floder (good : save in the program files in c drive)
2. Check environment variable value open command prompt and type echo %java_home% its should display path of the jdk like : C:\Program Files\Java\jdk1.8.0_102 if not displayed set the environment variable java_home

Set the environment variables using system m2_home : set the path of the folder where its stored maven_home : same as above set the path Adding to PATH: Add the unpacked distribution's bin directory to your user PATH environment variable path:%m2_home%\bin

To verify open the command prompt and type mvn -version should display this message Apache Maven 3.3.3 (7994120775791599e205a5524ec3e0dfe41d4a06; 2015-04-22T04:57:37-07:00) Maven home: /opt/apache-maven-3.3.3 Java version: 1.8.0_45, vendor: Oracle Corporation Java home: /Library/Java/JavaVirtualMachines/jdk1.8.0_45.jdk/Contents/Home/jre Default locale: en_US, platform encoding: UTF-8 OS name: "mac os x", version: "10.8.5", arch: "x86_64", family: "mac

some time it will not display because mvn folder not running with the admin access make running as administrator

Examples

installing

Check environment variable value e.g. echo %JAVA_HOME% C:\Program Files\Java\jdk1.7.0_51

Read Maven install in window online: https://riptutorial.com/maven/topic/10813/maven-install-in-window

Chapter 10: Maven Surefire Plugin

Syntax

- mvn test
- mvn -Dtest=com.example.package.ExampleTest test

Examples

Testing a Java class with JUnit and the Maven Surefire plugin

The Maven Surefire plugin runs during the test phase of the Maven build process or when `test` is specified as a Maven goal. The following directory structure and minimum `pom.xml` file will configure Maven to run a test.

Directory structure inside the project's root directory:

```
– project_root
  ├– pom.xml
  ├– src
  |  ├– main
  |  |  └– java
  |  └– test
  |     └– java
  └– target
     └– ...
```

`pom.xml` contents:

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.example</groupId>
  <artifactId>company-app</artifactId>
  <version>0.0.1</version>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.11</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

Create a file called `PlusTenTest.java` with the following contents in the project's `src/test/java/com/example/app` directory:

```
package com.example.app;

import static org.junit.Assert.assertEquals;
```

```
import org.junit.Test;

public class PlusTenTest {

    @Test
    public void incrementTest() {
        int result = PlusTen.increment(10);
        assertEquals("PlusTen.increment(10) result", 20, result);
    }
}
```

The annotation `@Test` tells JUnit that it should run `incrementTest()` as a test during the `test` phase of the Maven build process. Now create `PlusTen.java` in `src/main/java/com/example/app`:

```
package com.example.app;

public class PlusTen {
    public static int increment(int value) {
        return value;
    }
}
```

Run the test by opening a command prompt, navigating to the project's root directory and invoking the following command:

```
mvn -Dtest=com.example.app.PlusTenTest test
```

Maven will compile the program and run the test method `incrementTest()` in `PlusTenTest`. The test will fail with the following error:

```
...
Tests run: 1, Failures: 1, Errors: 0, Skipped: 0, Time elapsed: 0.005 sec <<< FAILURE! - in
com.example.app.PlusTenTest
incrementTest(com.example.app.PlusTenTest)  Time elapsed: 0.004 sec  <<< FAILURE!
java.lang.AssertionError: PlusTen.increment(10) result expected:<20> but was:<10>
at org.junit.Assert.fail(Assert.java:88)
at org.junit.Assert.failNotEquals(Assert.java:743)
at org.junit.Assert.assertEquals(Assert.java:118)
at org.junit.Assert.assertEquals(Assert.java:555)
at com.example.app.PlusTenTest.incrementTest(PlusTenTest.java:12)


Results :

Failed tests:
  PlusTenTest.incrementTest:12 PlusTen.increment(10) result expected:<20> but was:<10>

Tests run: 1, Failures: 1, Errors: 0, Skipped: 0

[INFO] ------------------------------------------------------------------------
[INFO] BUILD FAILURE
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 2.749 s
[INFO] Finished at: 2016-09-02T20:50:42-05:00
[INFO] Final Memory: 14M/209M
[INFO] ------------------------------------------------------------------------
```

```
[ERROR] Failed to execute goal org.apache.maven.plugins:maven-surefire-plugin:2.19.1:test
(default-test) on project app: There are test failures.
...
```

The Maven Surefire plugin creates a `/target/surefire-reports/` directory in your project's directory containing the files `com.example.app.PlusTenTest.txt` and `TEST-com.example.app.PlusTenTest.xml` that contain the error details of the beginning of the output above.

Following the test-driven development pattern, modify `PlusTen.java` so that the `increments()` method works correctly:

```
package com.example.app;

public class PlusTen {
    public static int increment(int value) {
        return value + 10;
    }
}
```

Invoke the command again:

```
mvn -Dtest=com.example.app.PlusTenTest test
```

The test passes:

```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running com.example.app.PlusTenTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.028 sec

Results :

Tests run: 1, Failures: 0, Errors: 0, Skipped: 0

[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 2.753 s
[INFO] Finished at: 2016-09-02T20:55:42-05:00
[INFO] Final Memory: 17M/322M
[INFO] ------------------------------------------------------------------------
```

Congratulations! You have tested a Java class using JUnit and the Maven Surefire plugin.

Read Maven Surefire Plugin online: https://riptutorial.com/maven/topic/5876/maven-surefire-plugin

Chapter 11: Maven Tomcat Plugin

Examples

Start tomcat using maven plugin.

In the example we will start tomcat 7 using maven plugin, optionally add user/password protection for REST end point. Also adding feature of building war.

Add below section in plugin section of pom for tomcat

```
 <plugin>
    <groupId>org.apache.tomcat.maven</groupId>
    <artifactId>tomcat7-maven-plugin</artifactId>
    <version>2.2</version>
    <configuration>
        <url>http://localhost:8090/manager</url>
        <server>localhost</server>
        <port>8191</port>
        <path>/${project.build.finalName}</path>
        <tomcatUsers>src/main/tomcatconf/tomcat-users.xml</tomcatUsers>
    </configuration>
</plugin>
```

Ensure maven war plugin is added and web.xml is present at location /src/main/webapp/WEB-INF/web.xml. Below is example of war plugin.

```
<plugin>
            <artifactId>maven-war-plugin</artifactId>
            <version>2.3</version>
            </plugin>
            <plugin>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.1</version>
                <configuration>
                    <source>1.7</source>
                    <target>1.7</target>
                    <webResources>
                        <resource>
                            <!-- this is relative to the pom.xml directory -->
                            <directory>/src/main/webapp/WEB-INF/web.xml</directory>
                        </resource>
                    </webResources>
                </configuration>
            </plugin>
```

Optionally, add tomcat-users.xml to location src/main/tomcatconf. It will be copied automatically when tomcat will start.

```
<tomcat-users>
    <user name="user" password="password" roles="admin" />
</tomcat-users>
```

Optionally, add below entry in web.xml to protect REST url.

```
    <!-- tomcat user -->
    <security-constraint>
        <web-resource-collection>
            <web-resource-name>Wildcard means whole app requires authentication</web-resource-
name>
            <url-pattern>/helloworld/*</url-pattern>
            <http-method>GET</http-method>
        </web-resource-collection>
        <auth-constraint>
            <role-name>admin</role-name>
        </auth-constraint>
        <user-data-constraint>
            <transport-guarantee>NONE</transport-guarantee>
        </user-data-constraint>
    </security-constraint>
    <login-config>
        <auth-method>BASIC</auth-method>
    </login-config>
```

Create new maven build from eclipse. Select the war project and in the Goals section add below command.

```
tomcat7:run
```

you will see message.

[INFO] --- tomcat7-maven-plugin:2.2:run (default-cli) @ web-service-ldap2 --- [INFO] Running war on http://localhost:8191/

Read Maven Tomcat Plugin online: https://riptutorial.com/maven/topic/6292/maven-tomcat-plugin

Chapter 12: Perform a Release

Introduction

The standard Maven plugin used by a Release Process is the maven-release-plugin – the configuration for this plugin is minimal:

SCM in the Maven pom:The Release process will interact with the Source Control of the project – this means we need to define the "scm" element in our pom.xml.The "scm" element for a release build should contain enough information to check out the tag that was created for this release.

Remarks

Note: Make sure to use maven release plugin 2.5 or later to avoid maven related issues. The Release Process

```
mvn release:clean
```

The above command will perform the below : delete the release descriptor (release.properties) delete any backup POM files

```
mvn release:prepare
```

Next part of the Release process is Preparing the Release; this will: perform some checks – there should be no uncommitted changes and the project should depend on no SNAPSHOT dependencies change the version of the project in the pom file to a full release number (remove SNAPSHOT suffix) – in our example – 0.0.1 run the project test suites commit and push the changes create the tag out of this non-SNAPSHOT versioned code increase the version of the project in the pom – in our example – 0.0.2-SNAPSHOT commit and push the changes

```
mvn release:perform
```

The latter part of the Release process is Performing the Release; this will: checkout release tag from SCM build and deploy released code This second step of the process relies on the output of the Prepare step – the release.properties.

Examples

POM.xml to perform release to Nexus repository

```
<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>
```

```
<groupId>org.codezarvis.artifactory</groupId>
<artifactId>nexusrelease</artifactId>
<version>0.0.5-SNAPSHOT</version>
<packaging>jar</packaging>

<name>nexusrelease</name>
<url>http://maven.apache.org</url>

<scm>
<connection>scm:git:git@github.com:isudarshan/nexuspractice.git</connection>
<url>scm:git:git@github.com:isudarshan/nexuspractice.git</url>
<developerConnection>scm:git:git@github.com:isudarshan/nexuspractice.git</developerConnection>
<tag>HEAD</tag>
</scm>

<distributionManagement>
<!-- Publish the versioned snapshot here -->
<repository>
<id>codezarvis</id>
<name>codezarvis-nexus</name>
<url>http://localhost:8080/nexus/content/repositories/releases</url>
</repository>

<!-- Publish the versioned releases here -->
<snapshotRepository>
<id>codezarvis</id>
<name>codezarvis-nexus</name>
<url>http://localhost:8080/nexus/content/repositories/snapshots</url>
</snapshotRepository>
</distributionManagement>

<properties>
<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
</properties>

<dependencies>
<dependency>
<groupId>junit</groupId>
<artifactId>junit</artifactId>
<version>3.8.1</version>
<scope>test</scope>
</dependency>
</dependencies>

<build>
<pluginManagement>
<plugins>
<plugin>
<artifactId>maven-release-plugin</artifactId>
<version>2.5.2</version>
<executions>
<execution>
<id>default</id>
<goals>
<goal>perform</goal>
</goals>
<configuration>
<pomFileName>${project.name}/pom.xml</pomFileName>
</configuration>
</execution>
```

```
</executions>
</plugin>
</plugins>
</pluginManagement>
</build>
</project>
```

Read Perform a Release online: https://riptutorial.com/maven/topic/9680/perform-a-release

Chapter 13: POM - Project Object Model

Examples

POM structure

Project Object Model is the basic unit of Maven and defines the project structure, dependencies, etc.

The following are very minimal to create a POM:

- `project` root
- `modelVersion` – should be set to `4.0.0`
- `groupId` – the ID of the project's group
- `artifactId` – the ID of the artifact (project)
- `version` – the version of the artifact under the specified group

`groupId`, `artifactId` and `version` are called *Maven coordinates* and sometimes abbreviated with *GAV*. They uniquely identify the resulting artifact of a project in a Maven repository (and *should* do so in the entire universe).

A minimal sample POM looks like:

```
<project>
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.sample</groupId>
  <artifactId>sample-app</artifactId>
  <version>0.0.1-SNAPSHOT</version>
</project>
```

POM Inheritance

Inheritance is the biggest asset of the POM, where the following can be managed from super POM to child POM.

- dependencies
- developers and contributors
- plugin lists (including reports)
- plugin executions with matching ids
- plugin configuration

The following enables the inheritance

```
  <parent>
    <groupId>com.sample</groupId>
    <artifactId>sample-app-parent</artifactId>
    <version>1.0.0</version>
```

```
  </parent>
```

POM structure looks like

```
<project>
  <parent>
    <groupId>com.sample</groupId>
    <artifactId>sample-app-parent</artifactId>
    <version>1.0.0</version>
  </parent>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.sample</groupId>
  <artifactId>sample-app</artifactId>
  <version>0.0.1-SNAPSHOT</version>
</project>
```

POM Aggregation

The modules of a multi-module project are aggregated from a hierarchical structure.

The root `pom` packing should look like:

```
<packaging>pom</packaging>
```

The following will be the directory structure of the project:

```
 |-- sample-app
  \   `-- pom.xml
   |-- sample-module-1
   |   `-- pom.xml
   |-- sample-module-2
   |   `-- pom.xml
```

Root POM:

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.sample</groupId>
  <artifactId>sample-app</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>pom</packaging>

  <modules>
    <module>sample-module-1</module>
    <module>sample-module-2</module>
  </modules>
  <dependencyManagement>
    ...
  </dependencyManagement>
</project>
```

Read POM - Project Object Model online: https://riptutorial.com/maven/topic/2310/pom---project-object-model

Credits

S. No	Chapters	Contributors
1	Getting started with Apache Maven	A_Di-Matteo, Adonis, Community, darkend, Nate Vaughan, ngreen, Ray, Stephen Leppik, tobybot11, Tunaki
2	Access Maven informations in code	Derlin
3	Create a Maven Plugin	Gerold Broser, Matt Champion, Tunaki, Vince
4	Eclipse integration	A_Di-Matteo, Gerold Broser, karel, kartik, Radouane ROUFID
5	Generate FIXME/TODO reports using the taglist-maven-plugin	Mahieddine M. Ichir
6	Maven Assembly Plugin	Jean-Rémy Revy, Tunaki, wallenborn, zygimantus
7	Maven Build Cycle	Gerold Broser, isudarsan
8	Maven EAR plugin	Mahieddine M. Ichir
9	Maven install in window	shashigura
10	Maven Surefire Plugin	Gerold Broser, Nate Vaughan
11	Maven Tomcat Plugin	kartik
12	Perform a Release	isudarsan
13	POM - Project Object Model	Gerold Broser, JF Meier, Radouane ROUFID, VinayVeluri, Vince