Test Automation(Selenium) Lecture 2 – *Maven*

PRAGMATIC IT Learning & Outsourcing Center

Lector: Georgi Tsvetanov
Skype: georgi.pragmatic
E-mail: george.tsvetanov@hotmail.com
Facebook: https://www.facebook.com/georgi.tsvetanov.18

www.pragmatic.bg

Summary - overall

- What is Maven
- Lifecycles
- Phases
- Maven Project
- Dependencies
- Repositories
- Plugins and Goals

How to install Maven

- Here are multiple links that are showing the procedure of how to install Maven and then check if it's correctly installed:

- https://maven.apache.org/install.html

- DO NOT forget to set the JAVA_HOME system environment variable to point to the JDK directory on your hard disk

- https://www.youtube.com/watch?v=Jtj-0yhox5s
- https://www.youtube.com/watch?v=NZbQtFr5VG8
- https://www.youtube.com/watch?v=p6LAsta-iDw
- https://www.youtube.com/watch?v=Nn8cmBVdYDs

- Let's go through it now.

Build Lifecycle Basics

- Maven is based around the central concept of a build lifecycle. What this means is that the process for building and distributing a particular artifact (project) is clearly defined.

- There are three built-in build lifecycles:
  - default – which handles project deployment
  - clean – which handles project cleaning
  - site – which handles the creation your project's site documentation

Build Lifecycle is made of Phases (1)

- Each of these build lifecycles is defined by a different list of build phases, wherein a build phase represents a stage in the lifecycle.
- For example, the default lifecycle consists of the following phases (for a complete list of the lifecycle phases, refer to the Lifecycle Reference):

Build Lifecycle is made of Phases (2)

- validate - validate the project is correct and all necessary information is available
- compile - compile the source code of the project
- test - test the compiled source code using a suitable unit testing framework. These tests should not require the code be packaged or deployed
- package - take the compiled code and package it in its distributable format, such as a JAR.
- integration-test - process and deploy the package if necessary into an environment where integration tests can be run.
- verify - run any checks on results of integration tests to ensure quality criteria are met
- install - install the package into the local repository, for use as a dependency in other projects locally
- deploy - done in the build environment, copies the final package to the remote repository for sharing with other developers and projects.

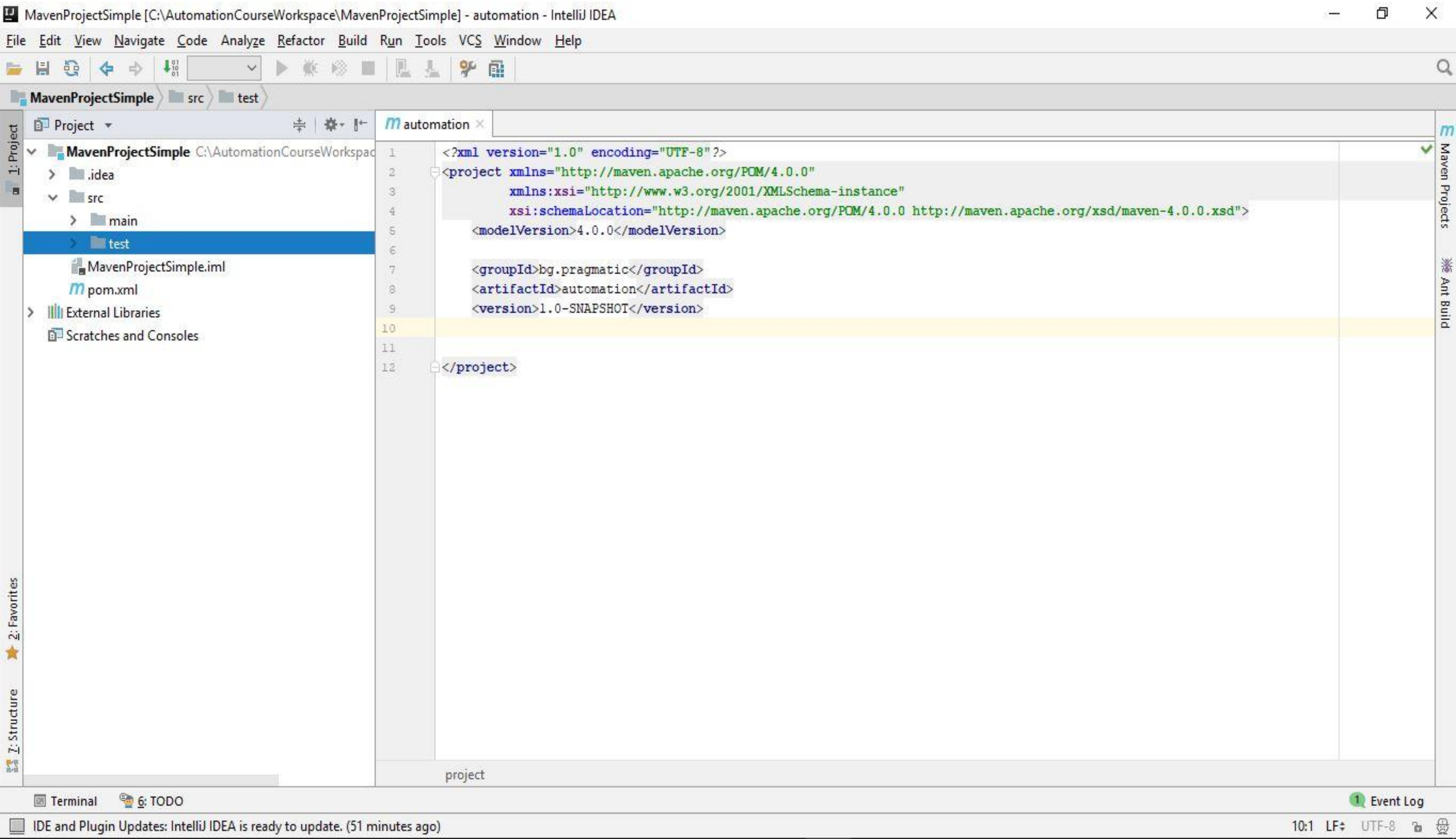
Maven Project

- Every project that is based on Maven, has a file named pom.xml and that's the fundamental configuration unit of work in Maven.
- You can rather create that pom.xml file manually or use a programming IDE like IntelliJ to create a Maven Project by going to -> File -> New -> Project… -> find "Maven" (don't check the checkbox "Create from archetype") -> click on "Next" -> now fill the fields, like the following example:

  Group Id: bg.pragmatic
  Artifact ID: automation

- Then create new folder for the new Project in your workspace -> click "Finish" and you will end up with a directory structure and a project and a pom.xml file in that project like the following one:

Maven Project

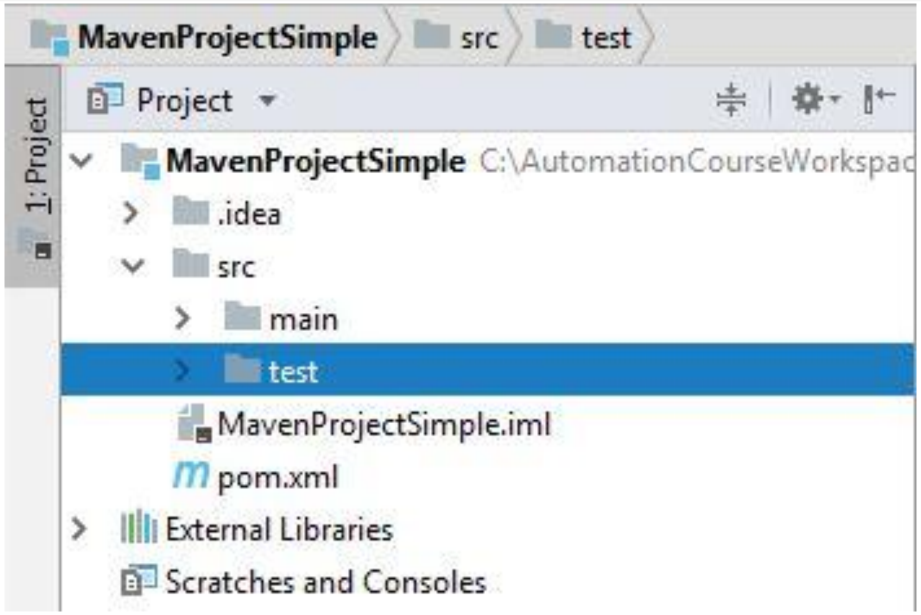

MavenProjectSimple [C:\AutomationCourseWorkspace\MavenProjectSimple] - automation - IntelliJ IDEA
File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help
MavenProjectSimple src test
Project
MavenProjectSimple C:\AutomationCourseWorkspac
.idea
src
main
test
MavenProjectSimple.iml
pom.xml
External Libraries
Scratches and Consoles
automation
project
Terminal
6: TODO
Event Log
IDE and Plugin Updates: IntelliJ IDEA is ready to update. (51 minutes ago)
10:1 LF UTF-8

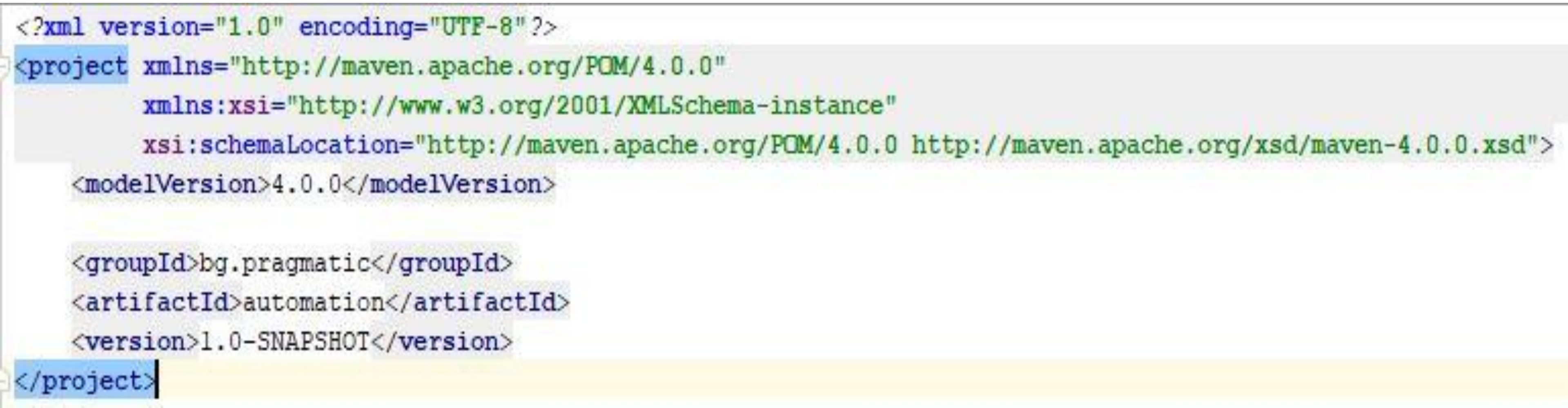
```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>bg.pragmatic</groupId>
    <artifactId>automation</artifactId>
    <version>1.0-SNAPSHOT</version>


</project>
```

Maven Project





```xml
<?xml version="1.0" encoding="UTF-8"?>
<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>bg.pragmatic</groupId>
    <artifactId>automation</artifactId>
    <version>1.0-SNAPSHOT</version>
</project>
```

Import existing Maven Project

- To import some existing Maven project go to File -> New… -> Project from Existing Sources -> navigate to your Project folder for showing where is the root directory of the project you want to import -> click "Ok"

- After you click "Ok" it will read the pom.xml file in that project, it will take into consideration everything in it and use it from now on for your project, rather some plugins or dependencies

pom.xml

- A Project Object Model or POM(pom.xml) is the fundamental unit of work in Maven. It is an XML file that contains information about the project and configuration details used by Maven to build the project. It contains default values for most projects. Examples for this is the build directory, which is <YOUR_PROJECT_DIR>/target/;

the source directory, which is src/main/java;
the test source directory, which is src/test/java; and so on.

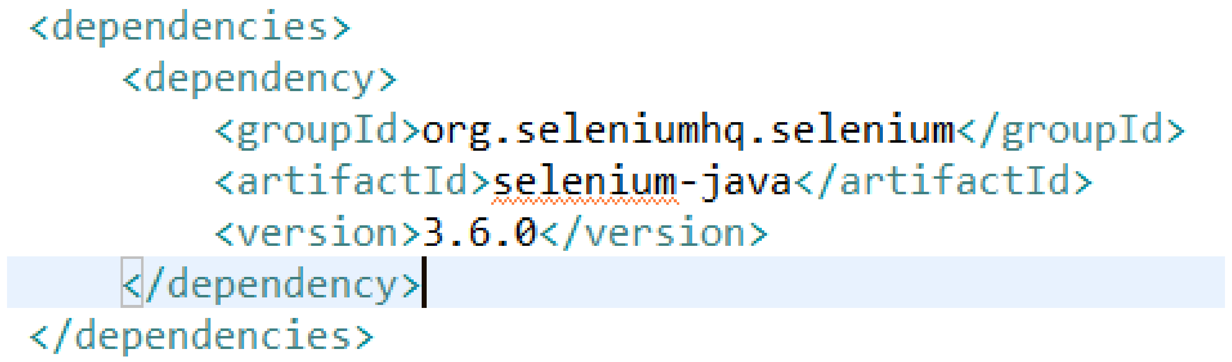
- When executing a task or goal, Maven looks for the POM in the current directory. It reads the POM, gets the needed configuration information, then executes the goal.

Dependencies

- We will take a look on the following:
  - Version
  - Types
  - Transitive Dependencies – this is number 1 reason why people start use maven
- One of the main remote repositories where you can find plenty of the most famous dependencies is:
  https://mvnrepository.com/

More on dependencies

- Dependencies are simply other resources that we want to use inside of our project and maven will pull also the transitive dependencies based on the ones that we list in the pom.xml
- To list a dependency in the dependencies section what you need in the pom.xml is:
  - groupId
  - artifactId
  - version



```
<dependencies>
    <dependency>
        <groupId>org.seleniumhq.selenium</groupId>
        <artifactId>selenium-java</artifactId>
        <version>3.6.0</version>
    </dependency>
</dependencies>
```

Dependencies - Version

- Version is the release number of the artifact that we would like to use.

- If you see SNAPSHOT in the version of any third party library(artifact) like Selenium for example. This allows the Selenium development team to push new code to a repository and our maven project will check for changes every time. In other words every time when the compile phase starts it will check if there is a new code that it needs to pull for that specific library. That way you always have the latest code for that library, but it might be buggy and unstable. It saves the developers from re-releasing versions for development or test purposes.

  - an example how you would see some artifacts use the SNAPSHOT functionality is something like(it has to be with capital letters in order to work):
    - selenium-java-1.0-SNAPSHOT.jar

Dependencies – Version - more

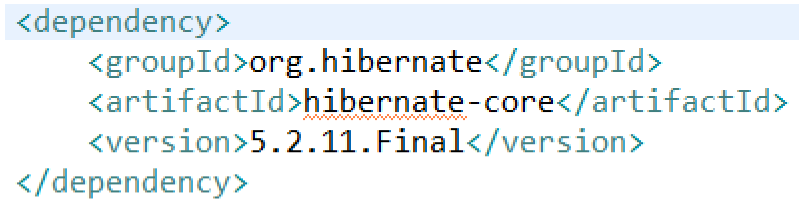
- Other types of example versions you might see(that are not related to Maven like SNAPSHOT, they are naming conventions that some people chosen) are:
  - myapplication-2.0-M1.jar (milestone candidate)
  - myapplication-2.0-RC1.jar (release candidate)
  - myapplication-2.0-RELEASE.jar (an official final release)
  - myapplication-2.0-Final.jar (an official final release)
  - myapplication-2.0-GA.jar (general availability release)
  - myapplication-2.0.jar

Dependencies – Types

- Most of the times you will use some dependency in your project that is one of these:
  - pom, jar, ear, war, rar, par, maven-plugin
  - The default type is jar (if you don’t outline it explicitly)
- The interesting one is of type pom:
  - If we add a dependency which is of type pom, this will automatically download everything that is in that pom into our project. This is used for example when you want to group multiple libraries(for tests for example), that are typical for your company and inform all your colleagues that they don’t need to find all of them one by one, they just need to make that pom as a dependency and they will have it.

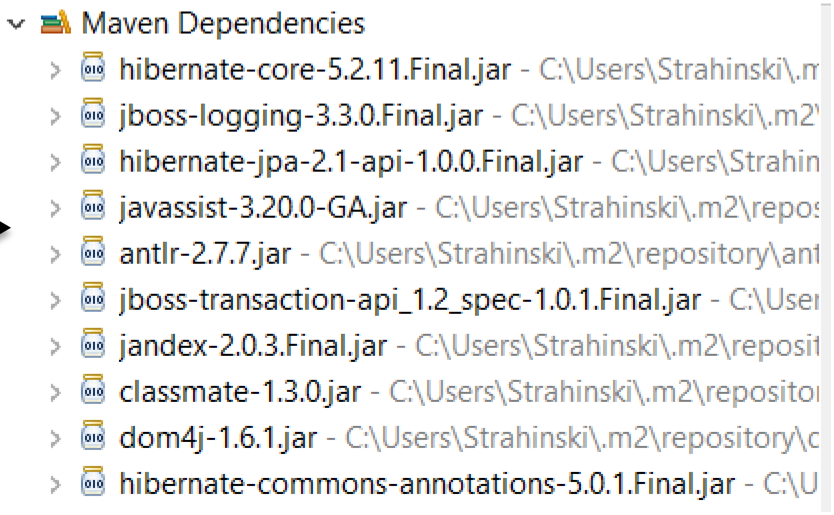
Dependencies - Transitive

- Lets say we would like to use Hibernate ORM Framework in our project, we simply add the dependency in our pom.xml like:



```xml
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>5.2.11.Final</version>
</dependency>
```

→



```
Maven Dependencies
  hibernate-core-5.2.11.Final.jar - C:\Users\Strahinski\.m
  jboss-logging-3.3.0.Final.jar - C:\Users\Strahinski\.m2\
  hibernate-jpa-2.1-api-1.0.0.Final.jar - C:\Users\Strahin
  javassist-3.20.0-GA.jar - C:\Users\Strahinski\.m2\repos
  antlr-2.7.7.jar - C:\Users\Strahinski\.m2\repository\ant
  jboss-transaction-api_1.2_spec-1.0.1.Final.jar - C:\User
  jandex-2.0.3.Final.jar - C:\Users\Strahinski\.m2\reposit
  classmate-1.3.0.jar - C:\Users\Strahinski\.m2\reposito
  dom4j-1.6.1.jar - C:\Users\Strahinski\.m2\repository\c
  hibernate-commons-annotations-5.0.1.Final.jar - C:\U
```

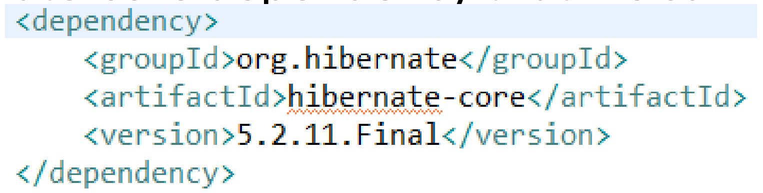
Repositories – Local Repo

- Local Repository
  - If you add some dependency to your pom.xml, maven will automatically download the dependency together with its transitive dependencies and will put it in your local repository for further re-use when needed(not to always download it).

  - Your local repository lives in your home directory of your user in a directory named .m2, for example on my PC it's:

    C:\Users\<YOUR_USER>\.m2\repository

  - So the hibernate-core dependency that we saw earlier will be downloaded under:



```xml
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>5.2.11.Final</version>
</dependency>
```

C:\Users\<YOUR_USER>\.m2\repository\org\hibernate\hibernate-core\5.2.11.Final

Repositories – Remote Repo

- There is the so call super pom.xml, which exists in your Maven installation where the central Apache Maven Repositories are defined and you're NOT suppose to edit that super pom.xml, because there are ways to override things that are defined there in your own pom.xml. Normally that super pom.xml since version 3 of Maven is under:

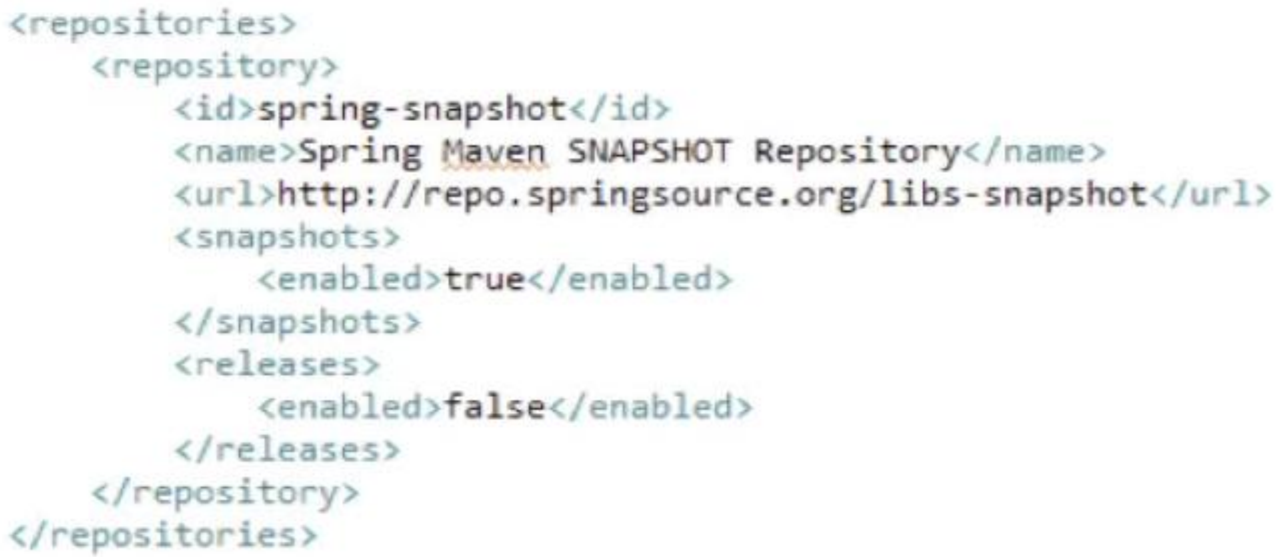
  lib/maven-model-builder-3.0.3.jar:org/apache/maven/model/pom-4.0.0.xml

- You can also check it how it looks on: http://maven.apache.org/ref/3.5.0/maven-model-builder/super-pom.html

Repositories – Remote Repo – For Dependencies

- The default location where the super pom.xml points to download from a remote repository is:
  https://repo.maven.apache.org/maven2

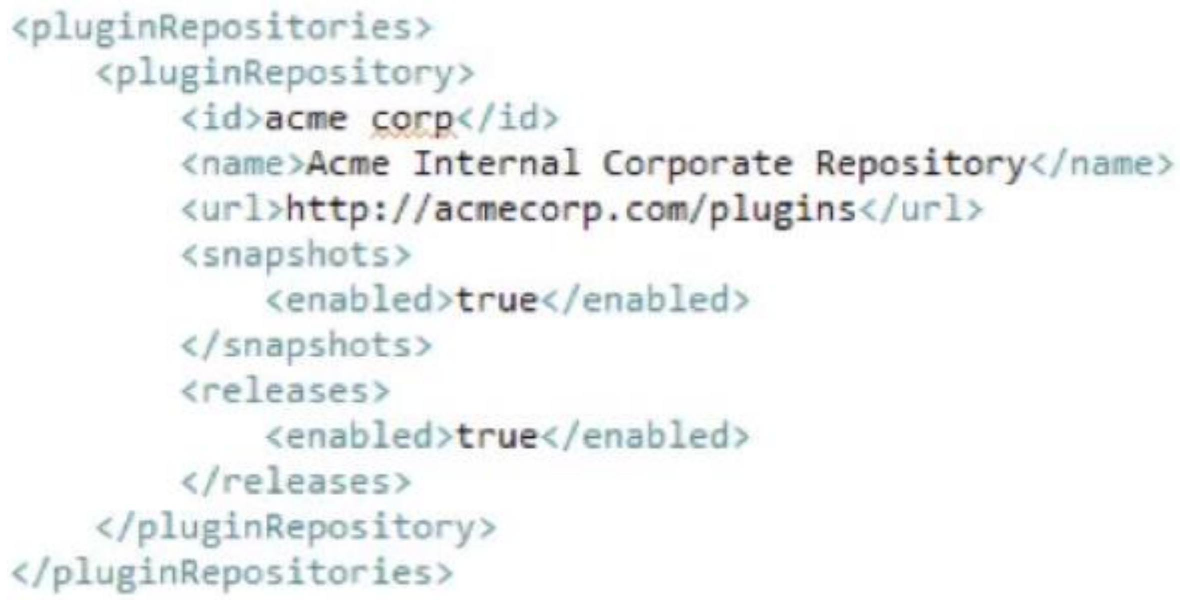
  and it has around 90-95% of everything you would probably want to download. In order to add a different remote repository, this is done the following way:



```
<repositories>
    <repository>
        <id>spring-snapshot</id>
        <name>Spring Maven SNAPSHOT Repository</name>
        <url>http://repo.springsource.org/libs-snapshot</url>
        <snapshots>
            <enabled>true</enabled>
        </snapshots>
        <releases>
            <enabled>false</enabled>
        </releases>
    </repository>
</repositories>
```

Repositories – Remote Repo – For Plugins

- It’s not really different from behavior perspective, how dependency and plugin repositories work, the different is only how you define them in your pom.xml:



```
<pluginRepositories>
    <pluginRepository>
        <id>acme corp</id>
        <name>Acme Internal Corporate Repository</name>
        <url>http://acmecorp.com/plugins</url>
        <snapshots>
            <enabled>true</enabled>
        </snapshots>
        <releases>
            <enabled>true</enabled>
        </releases>
    </pluginRepository>
</pluginRepositories>
```

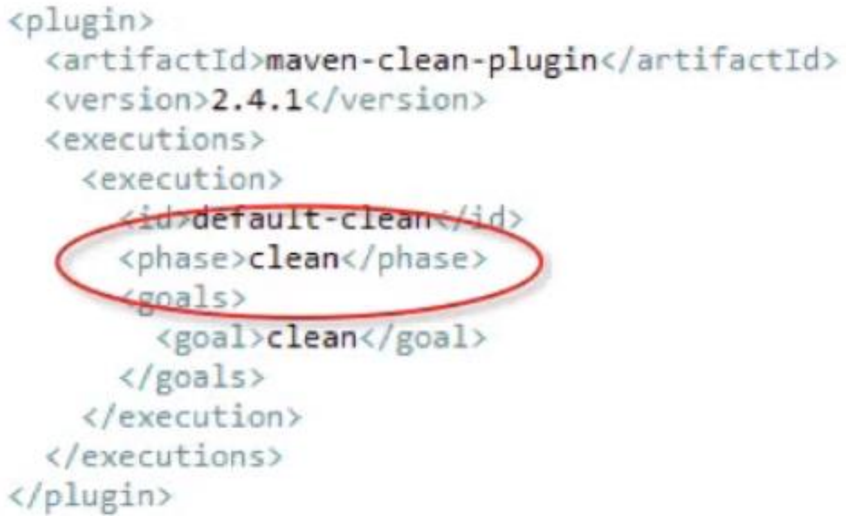
Plugins and Goals

- Goals are really just some configured plugins in our application, knowing that there are some default ones that are coming from the super pom.xml
- Plugins are tied to one of the defined phases, but can usually be overridden to do something more specific for our application
- The compile plugin is defined already for us, but usually this is the most overridden plugin, as it defaults to Java 1.5 no matter what JDK you have installed.
- Source code and javadocs can easily be generated to be in your corporate repository together with your code for use by our colleagues

Plugins and Goals

- The default goals are things like:
  - clean, compile, test, install, package, deploy
- We can configure all those plugins inside our own project pom.xml and choose what exactly they do
- Goals are always tied to some phase and we can change the phase if we want. Example:

If we want a clean to run after we install, we can tie it to the install phase, so we need to change "clean" with "install"



```
<plugin>
  <artifactId>maven-clean-plugin</artifactId>
  <version>2.4.1</version>
  <executions>
    <execution>
      <id>default-clean</id>
      <phase>clean</phase>
      <goals>
        <goal>clean</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

Override default Maven compiler to 1.8 – maven-compiler-plugin

- Maven compiler defaults to 1.5, regardless of what version of JDK is installed on your computer. To override this put this XML code in the pom.xml and update your Maven :

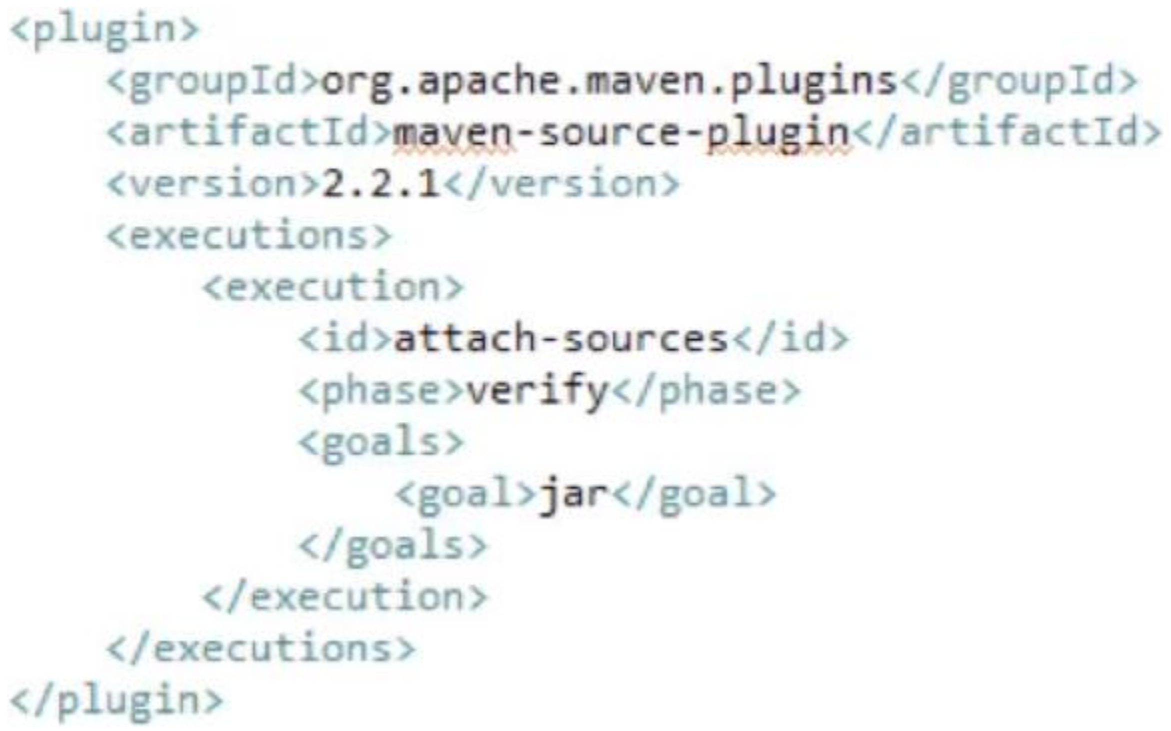
```
<build>
  <plugins>
    <plugin>
       <groupId>org.apache.maven.plugins</groupId>
       <artifactId>maven-compiler-plugin</artifactId>
       <version>3.5.1</version>
       <configuration>
           <source>1.8</source>
           <target>1.8</target>
       </configuration>
    </plugin>
  </plugins>
</build>
```

maven-source-plugin

- maven-source-plugin is used to attach our source code to a jar
- https://maven.apache.org/plugins/maven-source-plugin/
- By default it’s tied to the package phase
  - Frequently overridden to a later phase, if you run package often you don’t want sometimes to loose time to wait for the sources be attached, you normally want it to a later phase like install or deploy, because it’s slowing your build down

maven-source-plugin

- Lets override the maven-source-plugin to be attached to the verify or install phases:
- The id is chosen by yourself

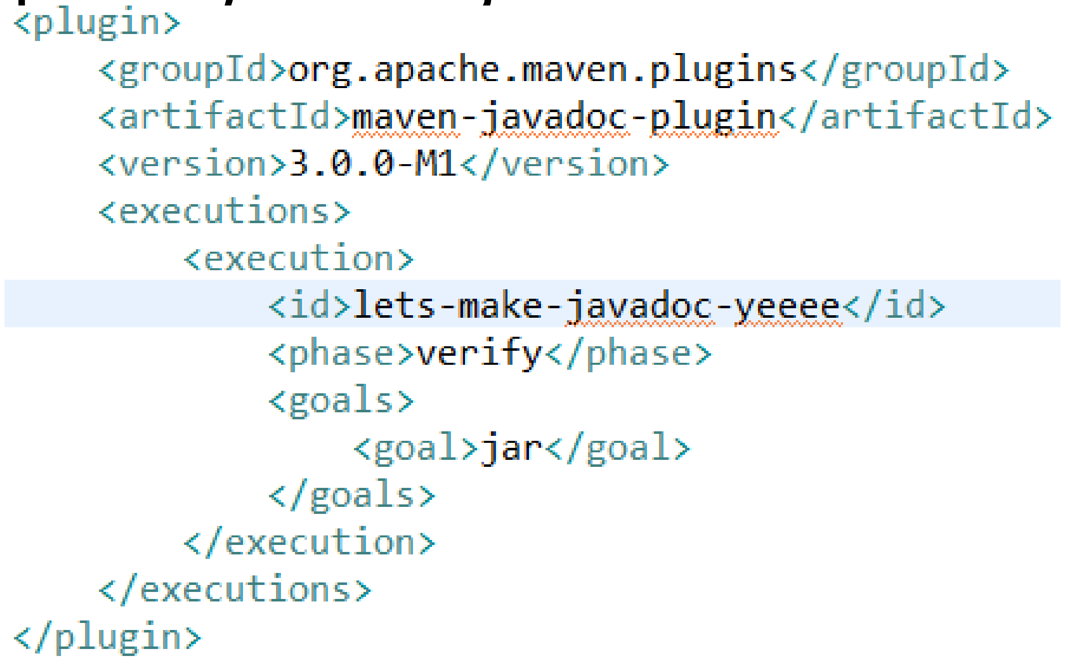


```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-source-plugin</artifactId>
    <version>2.2.1</version>
    <executions>
        <execution>
            <id>attach-sources</id>
            <phase>verify</phase>
            <goals>
                <goal>jar</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

Run: mvn install

maven-javadoc-plugin

- It simply a plugin to create javadoc archive together with everything else, which can be extremely useful, by default it’s tied to the package phase, and again often overridden



```xml
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-javadoc-plugin</artifactId>
    <version>3.0.0-M1</version>
    <executions>
        <execution>
            <id>lets-make-javadoc-yeeee</id>
            <phase>verify</phase>
            <goals>
                <goal>jar</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

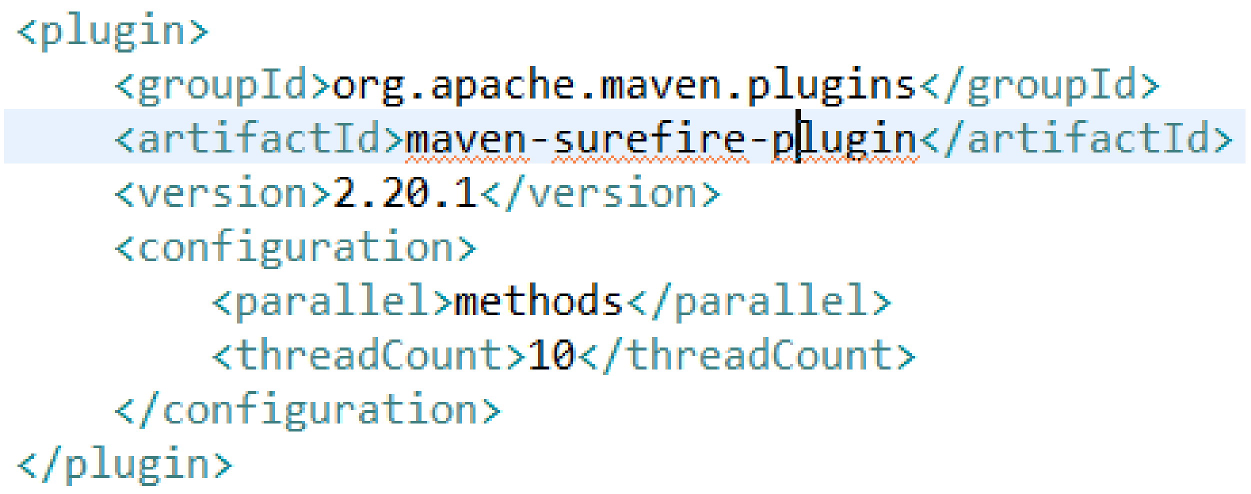
Run: mvn install

maven-surefire-plugin

- This is the plugin that we get normally interested in, when we run tests
- http://maven.apache.org/surefire/maven-surefire-plugin/plugin-info.html
- From JUnit 4.7 onwards you can run your tests in parallel. To do this, you must set the parallel parameter, and may change the threadCount or useUnlimitedThreads attribute

maven-surefire-plugin

- Here is an example of configured maven-surefire-plugin to run in parallel your tests with limited threadCount to 10:

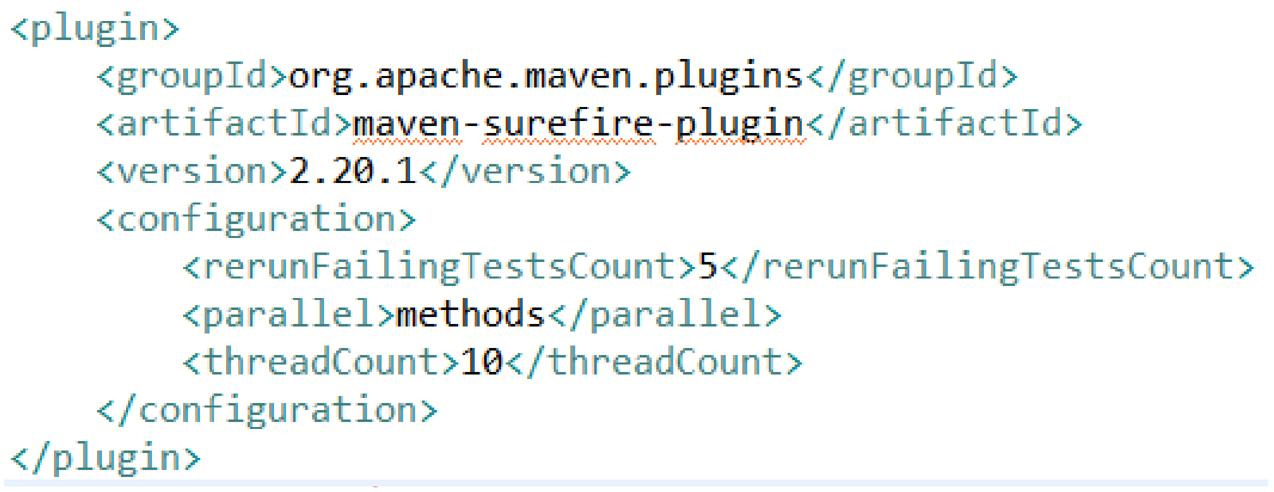


```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-surefire-plugin</artifactId>
    <version>2.20.1</version>
    <configuration>
        <parallel>methods</parallel>
        <threadCount>10</threadCount>
    </configuration>
</plugin>
```

- Run: mvn test
- Or: mvn surefire:test – if you want it to run as simple task and not execute all other phases

maven-surefire-plugin

- How to make the maven-surefire-plugin to rerun the failed tests, as sometimes they might be “flaky” and we want to make sure they are re-run at least twice for example, to make sure it’s not caused by some network glitch or something like that.



```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-surefire-plugin</artifactId>
    <version>2.20.1</version>
    <configuration>
        <rerunFailingTestsCount>5</rerunFailingTestsCount>
        <parallel>methods</parallel>
        <threadCount>10</threadCount>
    </configuration>
</plugin>
```

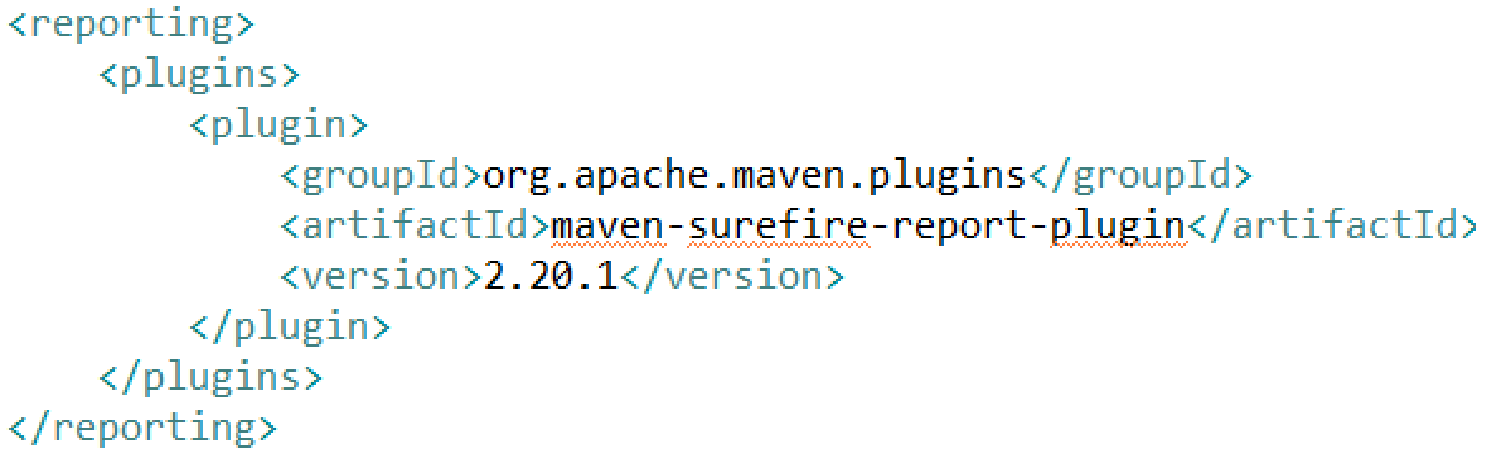
Run: mvn test

make sure you have

at least one failing

maven-surefire-report-plugin

- By default tied to site lifecycle site phase
- This plugin is used to generate report in HTML format that you can open in your browser and check your test results. You have to add it in <reporting> tag and NOT in the <build>:

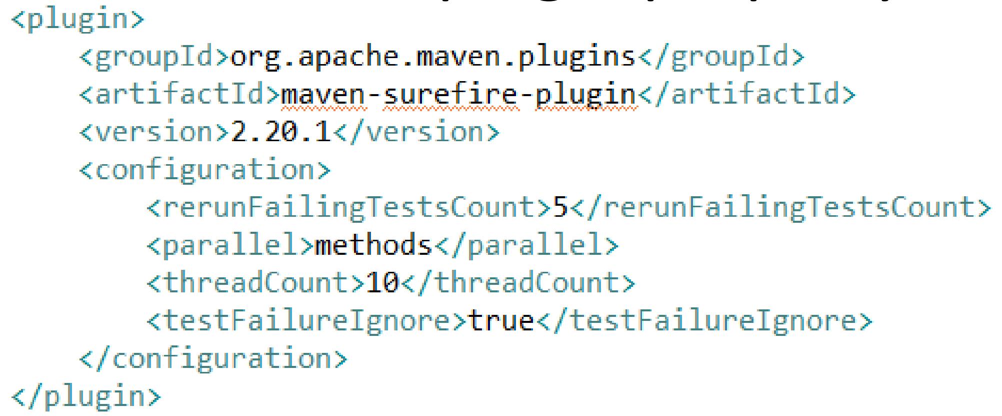


```
<reporting>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-surefire-report-plugin</artifactId>
            <version>2.20.1</version>
        </plugin>
    </plugins>
</reporting>
```

- Run: mvn clean test site

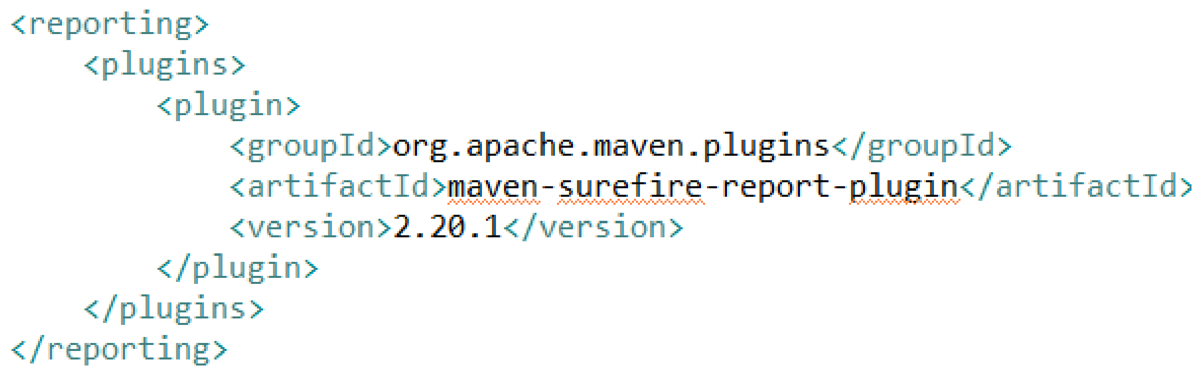
Generating HTML report no matter the failing build due to tests failures

- Set the maven-surefire-plugin property testFailureIgnore as follows:



```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-surefire-plugin</artifactId>
    <version>2.20.1</version>
    <configuration>
        <rerunFailingTestsCount>5</rerunFailingTestsCount>
        <parallel>methods</parallel>
        <threadCount>10</threadCount>
        <testFailureIgnore>true</testFailureIgnore>
    </configuration>
</plugin>
```

- Also the reporting tag with maven-surefire-report-plugin:



```
<reporting>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-surefire-report-plugin</artifactId>
            <version>2.20.1</version>
        </plugin>
    </plugins>
</reporting>
```

Run:

mvn clean test site

and open the target directory of your project, in my case

D:\myfirstworkspace\automation\target\site\index.html

Maven execution example

- mvn package -Dmaven.test.skip=true - running package phase by skipping the test phase. As you can see the properties that you can set in the pom.xml file can also be passed as parameters in the command line during the execution of a command using "-D"(WITHOUT space after it). All properties for each plugin can be seen under their goals, the example above is for maven-surefire-plugin:

http://maven.apache.org/surefire/maven-surefire-plugin/test-mojo.html

Maven execution example

- If you do not want to generate a site, but only a HTML report you can directly call the goal:
  mvn surefire-report:report
- Invokes the execution of the lifecycle phase test prior to executing itself.
- The generated report will appear under your project site directory. In my case:

  D:\myfirstworkspace\automation\target\site\surefire-report.html

There is a known bug that it does not generate the CSS files and it's ugly(https://issues.apache.org/jira/browse/SUREFIRE-616)

Questions