### Maven:

### Dealing with build failure

If you are running a complex multi-module project build, you can define how the Maven build system should react to errors in one module.

* -fae, --fail-at-end - fails the build after all modules are build; allow all non-impacted builds to continue
* -ff, --fail-fast - Stop at first module build failure
* -fn, --fail-never - NEVER fail the build, regardless of module build result

The -fn and -fae options are useful to verify builds that are running within a continuous integration tool like Jenkins and to see all errors in the build.

Creating a Maven Wrapper

Create a Maven Wrapper for a project with the latest available Maven version.

cd {your-project}

mvn -N io.takari:maven:wrapper

Create a Maven Wrapper for a project with a specified Maven version by using the maven property.

cd {your-project}

mvn -N io.takari:maven:wrapper -Dmaven=3.3.0

executing maven wrapper in unix

./mvnw clean package

Creating a Maven Wrapper

Run maven using following commands

mvn compile

mvn clean package

mvn test

mvn –Dtest=<CLASSNAME> test

mvn package -DSkipTests

### Adding goals to life cycle phases

You can add goals to life cycle phases by configuring more Maven plug-ins and adding them to a life cycle in your pom file. You need to specify which goal should be executed. If the plug-in does not specify the default life cycle it should run, you must also specify the life cycle phase it should run.

<plugin>

<groupId>com.vogella.example</groupId>

<artifactId>vogella-some-maven-plugin</artifactId>

<version>1.0</version>

<executions>

<execution>

<phase>verify</phase>

<goals>

<goal>checklinks</goal>

</goals>

</execution>

</executions>

</plugin>

Using properties

You can specify properties in your build files. You can override them via the command line with the -Dparameter followed directly (without whitespace) by its value. See the next chapter for an example.

Effective POM

With all this POM inheritance it may be hard to know what the total POM file looks like when Maven executes. The total POM file (result of all inheritance) is called the *effective POM*. You can get Maven to show you the effective POM using this command:

Mvn help:effective-pom

This command will make Maven write out the effective POM to the command line prompt.

Maven Settings File

Maven has two settings files. In the settings files you can configure settings for Maven across all Maven POM files. For instance, you can configure:

* Location of local repository
* Active build profile
* Etc.

The settings files are called settings.xml. The two settings files are located at:

* The Maven installation directory: $M2\_HOME/conf/settings.xml
* The user's home directory: ${user.home}/.m2/settings.xml

Both files are optional. If both files are present, the values in the user home settings file overrides the values in the Maven installation settings file.

You can read more about the Maven settings files in the [**Maven Settings Reference**](http://maven.apache.org/settings.html).

## Maven Build Life Cycles, Phases and Goals

When Maven builds a software project it follows a build life cycle. The build life cycle is divided into build phases, and the build phases are divided into build goals. Maven build life cycles, build phases and goals are described in more detail in the [**Maven Introduction to Build Phases**](http://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html), but here I will give you a quick overview.

**Build Life Cycles**  
Maven has 3 built-in build life cycles. These are:

1. default
2. clean
3. site

Each of these build life cycles takes care of a different aspect of building a software project. Thus, each of these build life cycles are executed independently of each other. You can get Maven to execute more than one build life cycle, but they will be executed in sequence, separately from each other, as if you had executed two separate Maven commands.

The default life cycle handles everything related to compiling and packaging your project. The clean life cycle handles everything related to removing temporary files from the output directory, including generated source files, compiled classes, previous JAR files etc. The site life cycle handles everything related to generating documentation for your project. In fact, site can generate a complete website with documentation for your project.

**Build Phases**  
Each build life cycle is divided into a sequence of build phases, and the build phases are again subdivided into goals. Thus, the total build process is a sequence of build life cycle(s), build phases and goals.

You can execute either a whole build life cycle like clean or site, a build phase like install which is part of the default build life cycle, or a build goal like dependency:copy-dependencies. Note: You cannot execute the default life cycle directly. You have to specify a build phase or goal inside thedefault life cycle.

When you execute a build phase, all build phases before that build phase in this standard phase sequence are executed. Thus, executing the install build phase really means executing all build phases before the install phase, and then execute the install phase after that.

The default life cycle is of most interest since that is what builds the code. Since you cannot execute the default life cycle directly, you need to execute a build phase or goal from the default life cycle. The default life cycle has an extensive sequence of build phases and goals, ,so I will not describe them all here. The most commonly used build phases are:

|  |  |
| --- | --- |
| Build Phase | Description |
| validate | Validates that the project is correct and all necessary information is available. This also makes sure the dependencies are downloaded. |
| compile | Compiles the source code of the project. |
| test | Runs the tests against the compiled source code using a suitable unit testing framework. These tests should not require the code be packaged or deployed. |
| package | Packs the compiled code in its distributable format, such as a JAR. |
| install | Install the package into the local repository, for use as a dependency in other projects locally. |
| deploy | Copies the final package to the remote repository for sharing with other developers and projects. |

You execute one of these build phases by passing its name to the mvn command. Here is an example:

mvn package

This example executes the package build phase, and thus also all build phases before it in Maven's predefined build phase sequence.

If the standard Maven build phases and goals are not enough to build your project, you can create[**Maven plugins**](http://tutorials.jenkov.com/maven/maven-tutorial.html#maven-plugins) to add the extra build functionality you need.

**Build Goals**  
Build goals are the finest steps in the Maven build process. A goal can be bound to one or more build phases, or to none at all. If a goal is not bound to any build phase, you can only execute it by passing the goals name to the mvn command. If a goal is bound to multiple build phases, that goal will get executed during each of the build phases it is bound to.

## Maven Build Profiles

Maven build profiles enable you to build your project using different configurations. Instead of creating two separate POM files, you can just specify a profile with the different build configuration, and build your project with this build profile when needed.

You can read the full story about build profiles in the Maven POM reference under [**Profiles**](http://maven.apache.org/pom.html#Profiles). Here I will give you a quick overview though.

Maven build profiles are specified inside the POM file, inside the profiles element. Each build profile is nested inside a profile element. Here is an 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/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.jenkov.crawler</groupId>

<artifactId>java-web-crawler</artifactId>

<version>1.0.0</version>

<profiles>

<profile>

<id>test</id>

<activation>...</activation>

<build>...</build>

<modules>...</modules>

<repositories>...</repositories>

<pluginRepositories>...</pluginRepositories>

<dependencies>...</dependencies>

<reporting>...</reporting>

<dependencyManagement>...</dependencyManagement>

<distributionManagement>...</distributionManagement>

</profile>

</profiles>

</project>

A build profile describes what changes should be made to the POM file when executing under that build profile. This could be changing the applications configuration file to use etc. The elements inside the profile element will override the values of the elements with the same name further up in the POM.

Inside the profile element you can see a activation element. This element describes the condition that triggers this build profile to be used. One way to choose what profile is being executed is in the settings.xml file. There you can set the active profile. Another way is to add -P profile-name to the Maven command line. See the profile documentation for more information.

Maven supports the usage of profiles to define different configurations. If you start Maven, you can instruct it to use a certain profile. For this you specify the -P parameter followed directly (without whitespace) by the profile, e.g. -PyourProfile.

<profiles>

<profile>

<id>dev</id>

<activation>

<activeByDefault>true</activeByDefault>

</activation>

<properties>

<db.location>URL\_to\_dev\_system</db.location>

<logo.image>companylogo.png</logo.image>

</properties>

</profile>

<profile>

<id>production</id>

<properties>

<db.location>URL\_to\_prod\_system</db.location>

<logo.image>companylogo2.png</logo.image>

</properties>

</profile>

</profiles>

## Maven and version control systems

Maven generates its output into the target folder of each project. This build output should not get included into your version control system.

Add this directory to your ignore resources. For example, if you use Git as version control system, add the "target/" entry to your .gitignore file in the root of each project.

## Useful properties

| *Useful properties* | |
| --- | --- |
| **Property** | **Description** |
| skipTests | true or false, specifies if tests should be executed or not |
| showWarnings | true or false, defines if the Maven build shows the compiler warnings |

Reference:

<http://www.vogella.com/tutorials/ApacheMaven/article.html>

<http://tutorials.jenkov.com/maven/maven-tutorial.html>