

# Apache Maven

---

---

This work is licensed under the  
Creative Commons Attribution 2.0 Belgium License.

To view a copy of this license,  
visit <http://creativecommons.org/licenses/by/2.0/be/>  
or send a letter to  
Creative Commons,  
171 Second Street, Suite 300,  
San Francisco, California, 94105,  
USA.



# Planning

---

- Installing Maven and your first build
  - Apache Maven core concepts
  - Building a Java project
  - Working with plugins
  - Properties and resource filtering
  - Building enterprise projects
  - Introduction to plugin development
-

# Apache Maven

---

Installing Maven and doing your first build

---

# Install Apache Maven

---

- Download Apache Maven from <http://maven.apache.org/download.html>
  - Extract the zipfile to your local hard disk, e.g. to the c:\tools\ directory
  - Change environment variables
    - set M2\_HOME to e.g. c:\tools\apache-maven-2.0.9
    - set JAVA\_HOME to JDK installation directory
    - change PATH variable to %M2\_HOME%/bin;%PATH%
  - Test your installation
    - running mvn --version should output version info
-

# Your first Maven build

---

- Create a project directory  
e.g. c:\projects\maven
- Create an file called pom.xml inside this directory

```
<?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/maven-v4_0_0.xsd">

  <modelVersion>4.0.0</modelVersion>
  <groupId>be.anova.abis</groupId>
  <artifactId>maven-pom</artifactId>
  <packaging>pom</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>My first Maven POM</name>

</project>
```

---

# Your first Maven build

---

- Now, run the mvn install command

...

```
[INFO] [install:install]
```

```
[INFO] Installing /home/gert/Projects/test/pom.xml to
```

```
/home/gert/.m2/repository/be/anova/abis/maven-pom/1.0-SNAPSHOT/maven-pom-1.0-SNAPSHOT.pom
```

```
[INFO] -----
```

```
[INFO] BUILD SUCCESSFUL
```

```
[INFO] -----[INFO] Total time: 56 seconds
```

```
[INFO] Finished at: Mon Nov 10 09:51:08 CET 2008
```

```
[INFO] Final Memory: 13M/91M
```

```
[INFO] -----
```

- Great, your first Maven build succeeded!  
Now, let's take a look at what really happened...
-

# Apache Maven

---

Apache Maven concepts

---

# Apache Maven concepts

---

- What is Maven?
  - Maven Core concepts
    - POM
    - Artifact
    - Repository
    - Plugin
    - Lifecycle
  - Maven versus Ant
  - Looking back at our first build
-

# What is Maven?

---

- Software project management framework/tool
    - builds
    - versioning
    - project reports
    - ...
-

# What is Maven?

---

- Design principles
    - convention over configuration
      - standard directory layout for projects
      - one primary output per project
      - standard naming conventions
    - declarative build process definition
      - POM
      - build life cycle
    - reuse of build logic
    - standardized organization of dependencies
-

# POM

---

- POM = Project Object Model
    - defined in pom.xml
    - contains all the information Maven needs to manage your project's build
      - name, group, description
      - dependencies
      - plugins
      - build profiles
      - ...
    - a POM can inherit from another POM
-

# Artifact

---

- Every artifact has
    - a group id
    - an artifact id
    - a version (1.0-SNAPSHOT, 2.1.0, ...)
    - an artifact type (pom, jar, war, ear, ...)
    - (optionally) an artifact classifier
-

# Repository

---

- Artifacts are stored in a repository
    - standard directory layout  
(`<group id>/<artifact id>/<version>`)  
e.g. `be/anova/abis/maven-pom/1.0-SNAPSHOT`
    - standard naming conventions for artifacts  
(`<artifact-id>-<version>`)  
e.g. `maven-pom-1.0-SNAPSHOT.pom`
-

# Repository

---

- Local repository
    - is created automatically in `<user_home>/m2/repository`
    - all dependencies are loaded from this repository first
  - Remote repository
    - is used by Maven to download additional artifacts
    - all downloads are copied to the local repo
    - central repo (<http://repo2.maven.org/maven2/>) is the default, but you can configure additional repos
-

# Plugin

---

- Maven is a plugin execution framework
    - uses the Plexus container project
    - you can build your own plugins and have Maven execute them as part of the build
    - plugins have goals
    - goals are identified by `<plugin id>:<goal id>`  
e.g. `clean:clean`, `compiler:compile`, `install:install`, ...
-

# Plugin

---

- Some common plugins

maven-clean-plugin	clean up after the build
maven-compiler-plugin	compiles Java sources
maven-deploy-plugin	deploy an artifact into a remote repo
maven-install-plugin	install an artifact into the local repo
maven-resources-plugin	copy resources to the output directory
maven-site-plugin	generate a site for the project
maven-surefire-plugin	runs JUnit (or TestNG) tests

- There are a lot plugins out there for generating, building, testing, releasing, packaging, ... projects
-

# Build lifecycle

---

- The build lifecycle consists of phases
    - examples: validate, compile, test, package, integration-test, verify, install, deploy
  - In every build phase, plugin goals are executed
  - Goals are bound to the lifecycle phases
    - based on the packaging
      - example: for pom packaging...
        - package → site:attach-descriptor
        - install → install:install-file
    - by adding a plugin
    - by explicitly configuring it in the POM
-

# Maven versus Ant

---

- Maven versus Ant
    - Ant is excellent for creating build scripts, Maven provide a more holistic solution
    - Maven already has conventions, with Ant you have to build your own conventions
    - Maven is declarative, Ant is procedural
    - Maven comes with its own lifecycle, with Ant you have to build your own
  - You can use Ivy/Antlibs to add some 'missing' features to Ant
-

# Looking back at our first build

---

- Let's now run mvn install again

```
gert@pantoeef:~/Projects/test$ mvn install
[INFO] Scanning for projects...
[INFO] -----
[INFO] Building My first Maven POM
[INFO]   task-segment: [install]
[INFO] -----
[INFO] [site:attach-descriptor]
[INFO] [install:install]
[INFO] Installing /home/gert/Projects/test/pom.xml to ~/.m2/repository/be/anova/abis/maven-
pom/1.0-SNAPSHOT/maven-pom-1.0-SNAPSHOT.pom
[INFO] -----
[INFO] BUILD SUCCESSFUL
[INFO] -----
[INFO] Total time: 2 seconds
[INFO] Finished at: Mon Nov 10 14:24:45 CET 2008
[INFO] Final Memory: 13M/97M
[INFO] -----
```

---

# Looking back at our first build

---

- Do you see...
    - the lifecycle and goals that are determined by the packaging in the pom.xml file?
    - the main artifact being installed in the local repo?
    - the plugins that have been used are already in the local repo and are not downloaded again?
  - Now browse the local repository and look for...
    - your own artifact
    - the artifacts for the plugins that have been used (the group id is org.apache.maven.plugins)
  - Can you also find these at central repo?
-

# Apache Maven

---

Building a Java project

---

# Building a Java project

---

- Creating the project
    - kickstart the project with Maven archetypes
    - standard directory layout
  - Building the project
    - running the build
    - (transitive) dependencies
  - IDE integration
  - Exercise: Building a Java Project
-

# Create a Java Project

---

- There are two ways for creating a Java project
    - option 1: manually
      - create a directory
      - create a pom.xml file and specify packaging 'jar'
      - create the standard directory layout
    - option 2: use a Maven archetype
      - run the mvn archetype:generate command
      - select the maven-quickstart-archetype
      - fill in the group id, artifact id, ... in the wizard
      - the archetype will generate everything from option 1
-

# Standard directory layout

---

/	→ project root dir
/src	→ source files
/src/main	→ application source files
/src/main/java	→ Java source files
/src/main/resources	→ other sources (properties, config, ...)
/src/test	→ test source files
/src/test/java	→ Java test source files
/src/test/resources	→ other sources for tests
/target	→ output files (created by build)
/pom.xml	→ POM file

---

# Building a Java project

---

- You can run these commands
    - mvn compile → compiles the sources
    - mvn test → run the unit tests
    - mvn package → build the jar file in the /target folder
    - mvn install → install the jar file in the local repo
  - Running mvn install will do all of those (remember the lifecycle?)
-

# Dependencies

---

- The Java project we generated uses JUnit tests
- In Maven, we use a `<dependency>` to 'use' the additional library
- Maven will download the dependency JAR from a repository and copy it to the local repository

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

---

# Dependencies

---

- A dependency is identified by
    - a group id
    - an artifact id
    - a version
    - an artifact type (defaults to jar)
    - (optionally) a classifier
  - Other options for `<dependency>`
    - `<optional>true</optional>`
    - `<exclusions><exclusion>` to exclude a transitive dependency
-

# Dependencies

---

- Transitive dependencies
  - sometimes, a JAR requires other JARs
  - Maven can lookup these dependencies from the POM
  - example: commons-logging 1.1

```
[INFO] -----  
[INFO] Building maven-java  
[INFO]   task-segment: [dependency:tree]  
[INFO] -----  
[INFO] [dependency:tree]  
[INFO] be.anova.maven:maven-java:jar:1.0-SNAPSHOT  
[INFO] +- commons-logging:commons-logging:jar:1.1:compile  
[INFO] | +- log4j:log4j:jar:1.2.12:compile  
[INFO] | +- logkit:logkit:jar:1.0.1:compile  
[INFO] | +- avalon-framework:avalon-framework:jar:4.1.3:compile  
[INFO] | \- javax.servlet:servlet-api:jar:2.3:compile  
[INFO] +- junit:junit:jar:3.8.1:test
```

---

# Dependencies

---

- Scope
    - determines when/where the dependency is available
      - in the current build
      - as a transitive dependency
    - possible values
      - compile
      - runtime
      - test
      - provided
      - system
      - import
-

# Dependencies

---

- Getting more information about the dependencies
    - use the maven-dependency-plugin
      - mvn dependency:resolve  
shows a list of all artifacts that have been resolved
      - mvn dependency:tree  
shows a tree of (transitive) dependencies
    - use the maven-site-plugin
      - mvn site  
generates the project site with the dependency report
-

# IDE integration

---

- Working with Eclipse & Maven
    - with the maven-eclipse-plugin
      - in Eclipse: add a classpath variable M2\_REPO
      - run the mvn eclipse:eclipse goal
      - import existing projects into the workspace
    - with Eclipse Maven plugin  
(<http://m2eclipse.codehaus.org>)
  - Similar solutions exist for IDEA, Netbeans, ...
-

# Exercise: Building Java projects

---

- A new project
    - Start a new Java project with the Maven archetype
    - Run the Maven build
  - Working with dependencies
    - add Camel 1.5.0  
group org.apache.camel, artifact camel-core
    - check the dependency tree
    - generate the project site with the dependency report
    - use the CamelContext interface inside your class
    - what happens if you change the scope to 'test'?
-

# Apache Maven

---

Working with plugins

---

# Working with plugins

---

- Configuring plugins
  - Adding and configuring a build plugin
    - Configuring plugin execution
  - Adding and configuring a reporting plugin
- Some plugin examples

# Configuring plugins

---

- Adding a build plugin to the POM
  - use the <plugin> element inside <build>
  - specify the plugin's artifact and group id

```
<project ...>
  ...
  <build>
    ...
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
      </plugin>
    </plugins>
  </build>
</project>
```

---

# Configuring plugins

---

- Configure the plugin
  - add a `<configuration>` element with an extra element for every parameter
  - example: configure the compiler plugin for Java 5

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

---

# Configuring plugins

---

- Adding a reporting plugin to the POM
  - use the `<plugin>` element inside `<reporting>`
  - specify the plugin's artifact and group id

```
<project ...>

  <reporting>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-javadoc-plugin</artifactId>
      </plugin>
    </plugins>
  </reporting>

</project>
```

---

# Configuring plugins

---

- Configure the plugin
  - add a `<configuration>` element with an extra element for every parameter
  - example: configure a document title for the javadoc

```
...  
<plugins>  
  <plugin>  
    <groupId>org.apache.maven.plugins</groupId>  
    <artifactId>maven-javadoc-plugin</artifactId>  
    <configuration>  
      <doctitle>My API's javadoc</doctitle>  
    </configuration>  
  </plugin>  
</plugins>  
...
```

---

# Configuring plugins

---

- Configuring plugin execution
  - many plugins will automatically bind to a phase...
  - ... but you can also configure the goals to be executed in a phase

```
<plugin>
  ...
  <executions>
    <execution>
      <phase>process-resources</phase>
      <configuration>...</configuration>
      <goals>
        <goal>run</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

---

# Some plugin examples

---

- Using the maven-antrun-plugin
  - create an extra directory inside /target

```
<plugin>
  <artifactId>maven-antrun-plugin</artifactId>
  <executions>
    <execution>
      <phase>initialize</phase>
      <configuration>
        <tasks>
          <mkdir dir="${project.build.directory}/extra-dir"/>
        </tasks>
      </configuration>
      <goals>
        <goal>run</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

---

# Some plugin examples

---

- Using the maven-source-plugin
  - generate a sources jar that is published in the repo
  - pop quiz: why do we bind to the verify phase?

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

---

# Some plugin examples

---

- Using Apache Felix' maven-bundle-plugin
  - create an OSGi bundle and configure exports

```
<project ...>
  <packaging>bundle</packaging>
  ...
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.felix</groupId>
        <artifactId>maven-bundle-plugin</artifactId>
        <extensions>>true</extensions>
        <configuration>
          <Export-Package>be.anova.api</Export-Package>
          <Private-Pacakge>be.anova.*</Private-Pacakge>
        </configuration>
      </plugin>
    </plugins>
  </build>
</project>
```

---

# Exercise: configuring plugins

---

- Start a web project
    - use the maven-archetype-webapp
      - what packaging type is used?
      - can you figure out the directory naming conventions for web applications?
  - Add a plugin
    - add the Jetty plugin (group: org.mortbay.jetty, artifact: maven-jetty-plugin)
    - now run `mvn jetty:run` to run the webapp with Jetty
    - can you configure this goal to be run automatically in the verify phase?
-

# Apache Maven

---

Properties and resource filtering

---

# Properties and resource filtering

---

- Properties
  - Defining properties
  - Using properties in the POM
- Resource filtering
  - Configure resource filtering

# Properties

---

- Defining properties
  - built-in properties like pom.version, pom.name, ....
  - every tag inside the <properties> tag become a property as well
  - example: define the version for ActiveMQ and Scala

```
<project ...>  
  
  <properties>  
    <activemq.version>5.1.0</activemq.version>  
    <scala.version>2.7.2</scala.version>  
  </properties>  
  
</project>
```

---

# Properties

---

- Using properties in the POM
  - properties can be used in most places inside the pom
  - the syntax for using a property is  
`${<property name>}`

```
<project ...>
  ...
  <name>Project (built with Scala ${scala.version})</name>
  ...
  <dependencies>
    <dependency>
      <groupId>org.apache.activemq</groupId>
      <artifactId>activemq-core</artifactId>
      <version>${activemq.version}</version>
    </dependency>
  </dependencies>

</project>
```

---

# Resource filtering

---

- Configure resource filtering
  - add a `<resource>` element with `filtering true`
  - every occurrence of the `${<property name>}` syntax will be replaced when copying resources

```
<project ...>

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

</project>
```

---

# Properties and resource filtering

---

- Improve the simple Java project
  - move the Apache Camel version into a property
  - configure the dependency to use the property
  - add a README.TXT file to the project resources, which contains the Camel version as well as the project version

# Apache Maven

---

Building enterprise projects

---

# Building enterprise projects

---

- Multi-module builds
  - POM inheritance
    - explicit POM inheritance
    - Super POM
  - Improving POM maintainability
    - dependency management
    - plugin management
-

# Multi-module builds

---

- Enterprise projects are often more complex
    - multiple modules for technical reasons
      - a web project that uses a core implementation JAR
      - EAR files can contain one or more WAR, JAR or RAR
      - a JBI SA contains multiple service units
    - multiple modules for project management
      - separate API and implementation JAR files
      - modules developed by separate teams
-

# Multi-module builds

---

- Creating a multi-module build
    - create a project with packaging 'pom'
    - for every module, create
      - a subdirectory with the module Maven project
      - a <module> entry in the root project's POM, referring to the subdirectory
    - modules can have dependencies on each other
      - Maven will determine the build order for the entire set of modules
  - Example: web application
    - 2 modules: web and core module
-

# Multi-module builds

---

- The root project directory contains
  - two directories: core and web
  - the root pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" ...>
  <modelVersion>4.0.0</modelVersion>
  <groupId>be.anova.webapp</groupId>
  <artifactId>parent</artifactId>
  <packaging>pom</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>webapp</name>
  <url>http://maven.apache.org</url>
  <modules>
    <module>web</module>
    <module>core</module>
  </modules>
</project>
```

---

# Multi-module builds

---

- Each subdirectory contains its own pom.xml
  - example: the web pom.xml depends on the core

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>be.anova.webapp</groupId>
  <artifactId>web</artifactId>
  <packaging>war</packaging>
  <name>webapp :: web</name>
  <version>1.0-SNAPSHOT</version>
  <url>http://maven.apache.org</url>
  ...
  <dependencies>
    <dependency>
      <groupId>be.anova.webapp</groupId>
      <artifactId>core</artifactId>
      <version>1.0-SNAPSHOT</version>
    </dependency>
  </dependencies>
</project>
```

---

# Multi-module builds

---

- You can build modules separately or together

```
[INFO] Scanning for projects...
[INFO] Reactor build order:
[INFO]   webapp :: core
[INFO]   webapp :: web
[INFO]   webapp
[INFO] -----
      ...
      (build output for every module + parent project)
      ...
[INFO] -----
[INFO] Reactor Summary:
[INFO] -----
[INFO] webapp :: core ..... SUCCESS [5.594s]
[INFO] webapp :: web ..... SUCCESS [0.506s]
[INFO] webapp ..... SUCCESS [2.121s]
[INFO] -----
[INFO] -----
[INFO] BUILD SUCCESSFUL
[INFO] -----
[INFO] Total time: 8 seconds
[INFO] Finished at: Tue Nov 11 21:36:33 CET 2008
[INFO] Final Memory: 32M/198M
[INFO] -----
```

---

# POM inheritance

---

- A POM can inherit from another POM
    - move common definitions into the parent POM
    - what is inherited?
      - identifiers (you must override group/artifact)
      - dependencies
      - developers and contributors
      - plugin list
      - reports list
      - plugin executions
      - plugin configuration
-

# POM inheritance

---

- POM inheritance
    - configure <parent/> element
      - artifactId
      - groupId
      - version
    - module POM often inherits from root POM
    - other common patterns:
      - prototype parent POM
      - company-wide parent POM
-

# POM inheritance

---

- Example: the core POM from our webapp

```
<project>

  <parent>
    <groupId>be.anova.webapp</groupId>
    <artifactId>parent</artifactId>
    <version>1.0-SNAPSHOT</version>
  </parent>

  <modelVersion>4.0.0</modelVersion>
  <groupId>be.anova.webapp</groupId>
  <artifactId>core</artifactId>
  <name>webapp :: core</name>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>
  <url>http://maven.apache.org</url>
  ...

</project>
```

---

# POM inheritance

---

- The Super POM
    - every POM without a `<parent/>` implicitly inherits from the Super POM
    - is in the `maven-uber.jar`
    - it defines
      - the central repo (`central` is the repository id), both for dependencies and plugins
      - the standard directory layout
      - default versions of the core plugins (starting from Maven 2.0.9)
-

# Dependency management

---

- Defining managed dependencies
  - define dependency version only once inside <dependencyManagement/>

```
<project>
  ...
  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.4</version>
      </dependency>
    </dependencies>
  </dependencyManagement>
</project>
```

---

# Dependency management

---

- Using managed dependencies
  - omit `<version/>` from `<dependency/>`
  - enforce managed version on transitive dependencies

```
<?xml version="1.0"?>
<project>
  ...
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
    </dependency>
  </dependencies>
</project>
```

---

# Plugin management

---

- Same as dependency management, but for plugins
  - define plugin version and configuration centrally  
example: configure checkstyle plugin

```
...  
<build>  
  <pluginManagement>  
    <plugins>  
      <plugin>  
        <artifactId>maven-checkstyle-plugin</artifactId>  
        <version>2.2</version>  
        <configuration>  
          ...  
        </configuration>  
      </plugin>  
    </plugins>  
  </pluginManagement>  
</build>  
...
```

---

# Building enterprise projects

---

- Create a new, multi-module project
    - containing a web and plain java module
    - the web module depends on the plain java module
    - run the build with and without POM inheritance configured in the modules -- what is the difference?
    - use mvn dependency:tree to see the difference between adding camel-core as
      - a dependency in the parent POM
      - a dependency in a module POM with dependency management in the parent POM
-

# Apache Maven

---

Plugin development

---

# Plugin development

---

- Before you start...
  - Concepts and terminology
  - Developing a plug-in
    - Starting a new project
    - Coding the plug-in
  - Using the plug-in
-

# Before you start...

---

- For a one-off task, consider
    - maven-antrun-plugin to run Ant tasks
    - groovy-maven-plugin to run Groovy script
    - ... or some other lightweight/scripting solution
  - Developing a new Maven plugin
    - requires more knowledge about the underlying mechanisms
    - is probably only worth the effort if you can actually reuse the plugin
-

# Concepts and terminology

---

- Maven is built on top of the Plexus IoC container
    - very similar to Spring
    - Maven uses field and setting dependency injection
  - A plugin
    - is a JAR file with a META-INF/maven/plugin.xml file
    - contains Plexus components
  - A Mojo is
    - a component in the Plexus container
    - the Java type for a Maven plugin goal
-

# Developing a plugin

---

- Start a new plugin project
    - manually
      - create a new project with packaging 'maven-plugin'
      - add the maven-plugin-api as a dependency
      - code a java class that extends AbstractMojo and contains Javadoc annotations for generating the plugin.xml
    - using the archetype
      - use mvn archetype:generate and select the maven-archetype-mojo archetype to generate the new plugin project
-

# Developing the plugin

---

- Coding the plugin
    - plugin implementation starts in the `execute()` method
    - javadoc annotations
      - `@goal` to name the goal that matches the Mojo
      - `@phase` to specify default phase of execution
      - `@parameter` can be used to annotate fields to become goal parameters (`@required` to make it a required field, you can specify a default value)
    - IoC: `@component` to inject a Maven component
      - `ArtifactFactory`, `MavenProject`, `ArtifactRepository`, ...
-

# Developing the plugin

---

- Coding the plugin

```
/**
 * @goal run
 * @phase process-sources
 */
public class RunningMojo extends AbstractMojo {

    /**
     * @parameter expression="${project.build.directory}"
     * @required
     */
    private File target;

    /**
     * @component
     */
    protected ArtifactResolver resolver;

    public void execute() throws MojoExecutionException {
        getLog().info("Running the goal now...");
        getLog().info(" --> outputting to " + target.getAbsolutePath());
        getLog().info("...done!");
    }
}
```

---

# Developing the plugin

---

- Building the project with the maven-plugin-plugin
    - automatically for 'maven-plugin' packaging
    - it will generate the plugin.xml file
      - based on the information in the annotations
      - for the plugin prefix, uses naming convention xyz-maven-plugin or maven-xyz-plugin
    - it will generate the components.xml file (used for @component injection by Plexus)
      - based on the information in the annotations
-

# Using the plugin

---

- Same as for any other plugin
  - define it in the <build/> section

```
<plugins>
  <plugin>
    <groupId>be.anova.maven</groupId>
    <artifactId>anova-maven-plugin</artifactId>
    <version>1.0-SNAPSHOT</version>
    <executions>
      <execution>
        <goals>
          <goal>run</goal>
        </goals>
      </execution>
    </executions>
  </plugin>
</plugins>
```

---

# Plugin development

---

- Create a new plugin
    - start the plugin project with the archetype
    - modify the plugin to calculate how much you have learned (a random number between 0 and 10 will do)
    - build your plugin
  - Use your own plugin
    - configure your new plugin in the multi-module project we created
    - be surprised how much (or how little) you learned from creating each module of the build process
-

# Apache Maven

---

Tips & tricks

---

# Tips & tricks

---

- Troubleshooting Maven builds
  - Create a company repository
  - Your own parent POM
  - Build builds as you build software
-

# Tips & tricks

---

- Troubleshooting Maven builds
    - use mvn dependency:tree
      - (optionally) add -Dverbose=true for more detail
    - use mvn -X for debug logging
      - warning: this will give you a LOT of output
    - run module builds separately
      - POM inheritance can lead to dependency inheritance problems
      - mvn help:effective-pom shows the merged POM
-

# Tips & tricks

---

- Create a company repository
    - mirror for central and/or other public repo
    - keep copy of own artifact versions
    - use enterprise repository software
      - Apache Archiva
      - JFrog Artifactory
      - Sonatype Nexus
    - this ensures a reproducible build
      - the version you use will still be available next time
-

# Tips & tricks

---

- Develop your own company parent POM
    - promoting your own standards
    - enforcing some common settings (reporting, checkstyle, ...)
    - as a possibility for adding company-wide configurations later on
  - Be careful not to add to much to this POM
-

# Tips & tricks

---

- Build your builds as you build your software
    - don't use everything you learned today in every POM
    - only add complexity when necessary
      - POM inheritance can really add to the complexity
      - dependency and plugin management only make sense in a large build
      - reporting is only useful if the reports are being used
-

# Apache Maven

---

Wrap up

---

# POM reference

---

- Basic settings
    - <groupId>
    - <artifactId>
    - <version>
    - <packaging>
    - <dependencies>
    - <parent>
    - <dependencyManagement>
    - <modules>
    - <properties>
-

# POM reference

---

- Build settings
    - <build>
      - <sourceDirectory>, <testSourceDirectory>, ..
      - <resources> and <testResources>
      - <plugins>
      - <pluginManagement>
      - <extension>
    - <reporting>
      - <outputDirectory>
      - <plugins>
-

# POM reference

---

- More settings
  - <name>
  - <description>
  - <url>
  - <inceptionYear>
  - <licenses>
  - <organization>
  - <developers>
  - <contributors>

# POM reference

---

- Environment Settings
    - <issueManagement>
    - <ciManagement>
    - <mailingLists>
    - <scm>
    - <prerequisites>
    - <repositories>
    - <pluginRepositories>
    - <distributionManagement>
    - <profiles>
-

# Questions?

---

---

The end

---

---