

Open Source Frameworks (OSF)

Build Automation with Maven

Open Source Frameworks (OSF)
Master of Science in Engineering (MSE)
Olivier Liechti
olivier.liechti@heig-vd.ch

heig-vd

Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

Agenda

- Maven
- Core concepts
- Lifecycles, phases, plugins and goals (mojos)
- Project relationships
- Maven and Java EE

What is the difference between
a **library** and a **framework**?



What is the difference between
a **ant** and a **maven**?

Maven, a Yiddish word meaning **accumulator of knowledge**, was originally started as an attempt to **simplify the build processes** in the Jakarta Turbine project. There were several projects each with their own Ant build files that were all slightly different and JARs were checked into CVS. We wanted **a standard way to build the projects**, a clear **definition of what the project consisted of**, an easy way to **publish project information** and a way to **share JARs** across several projects.



References

[http://www.sonatype.com/products/maven/
documentation/book-defguide](http://www.sonatype.com/products/maven/documentation/book-defguide)



Getting started...

- Install maven 2
 - <http://maven.apache.org/download.html>
- Create and build a project
- Look at these directories and files
 - ~/.m2/repository
 - ~/.m2/settings.xml
 - \$INSTALLATION/conf/settings.xml

```
mvn archetype:generate \  
  -DarchetypeGroupId=org.apache.maven.archetypes \  
  -DarchetypeArtifactId=maven-archetype-quickstart \  
  -DgroupId=ch.heigvd.osf.cool \  
  -DartifactId=coolProject  
cd coolProject  
mvn install
```

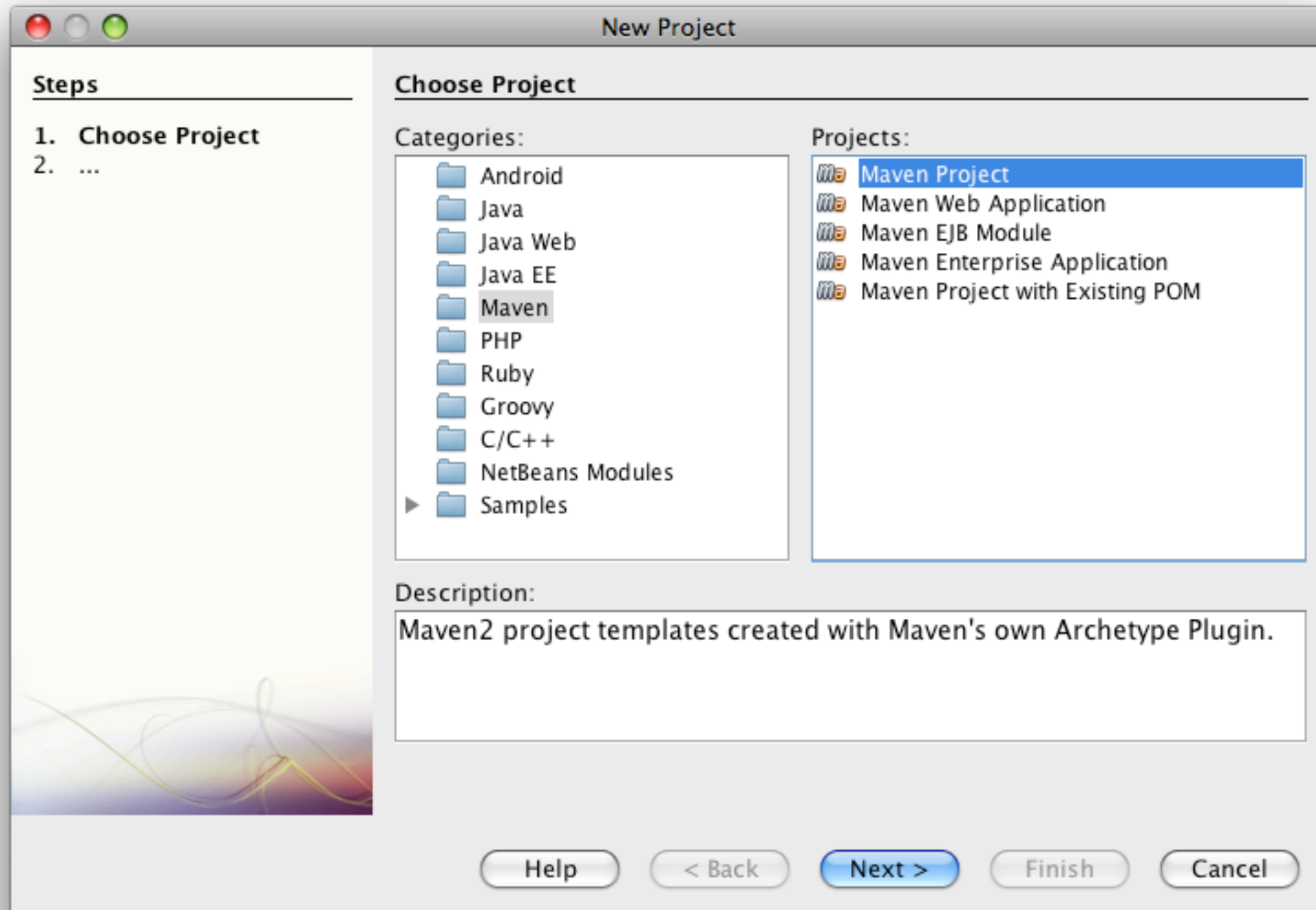

Project Object Model (POM)

- The POM is an XML document that describes the project:
 - What kind of artifacts are we building (jar, war, ear, etc.)?
 - What libraries do we depend on to build the artifact?
 - What are the special actions that need to be done during the build?
 - Where can plug-ins and dependencies be found?
 - Are there relationships with other maven-driven projects?
- In other words, with maven:
 - You declare “properties” about your project.
 - You let maven build the artifact, based on conventions and best practices.

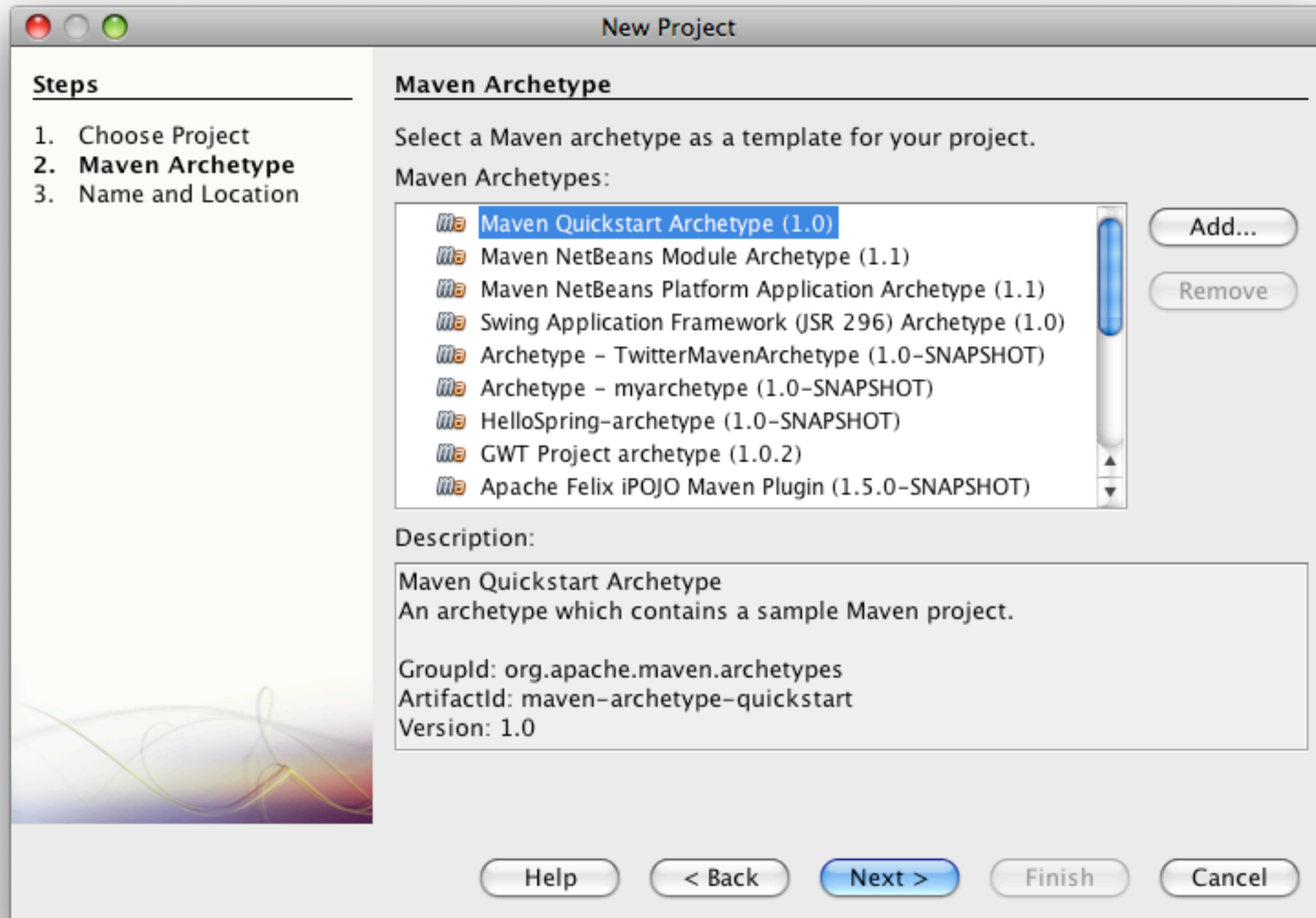
Maven Coordinates

- Every maven project produces one **artifact**, whether it is a jar file, a war file or anything else.
- When a project has a **dependency** on an artifact produced by another project, it needs a way to **reference this artifact**.
- Maven **coordinates** address this need: every project is identified by three values:
 - A group id (used to group related artifacts)
 - An artifact id
 - A version number
- Maven coordinates are the things you will find in every **POM**.
- Maven coordinates are also used to capture **inheritance relationships** between projects (more on this later).

Creating a maven project with NetBeans



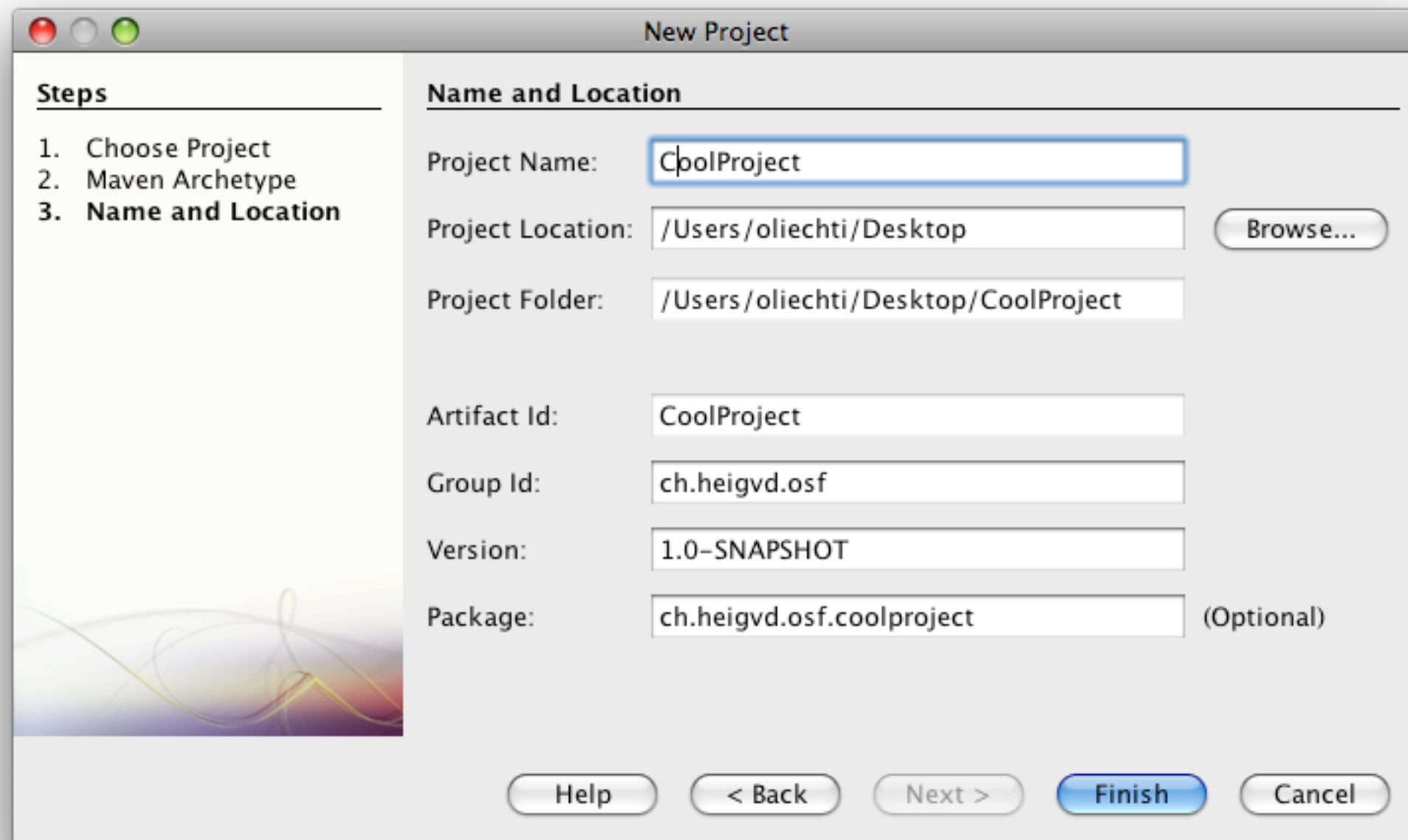
Creating a maven project with NetBeans



Creating a maven project with NetBeans

heig-vd

Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud



The image shows the 'New Project' dialog box in NetBeans. The 'Steps' panel on the left indicates the current step is '3. Name and Location'. The main panel contains several input fields for project configuration. The 'Project Name' field is highlighted with a blue border and contains the text 'CoolProject'. The 'Project Location' field contains '/Users/oliechti/Desktop' and has a 'Browse...' button to its right. The 'Project Folder' field contains '/Users/oliechti/Desktop/CoolProject'. The 'Artifact Id' field contains 'CoolProject'. The 'Group Id' field contains 'ch.heigvd.osf'. The 'Version' field contains '1.0-SNAPSHOT'. The 'Package' field contains 'ch.heigvd.osf.coolproject' and is marked as '(Optional)'. At the bottom, there are five buttons: 'Help', '< Back', 'Next >', 'Finish' (which is highlighted in blue), and 'Cancel'.

Steps

1. Choose Project
2. Maven Archetype
3. **Name and Location**

Name and Location

Project Name:

Project Location:

Project Folder:

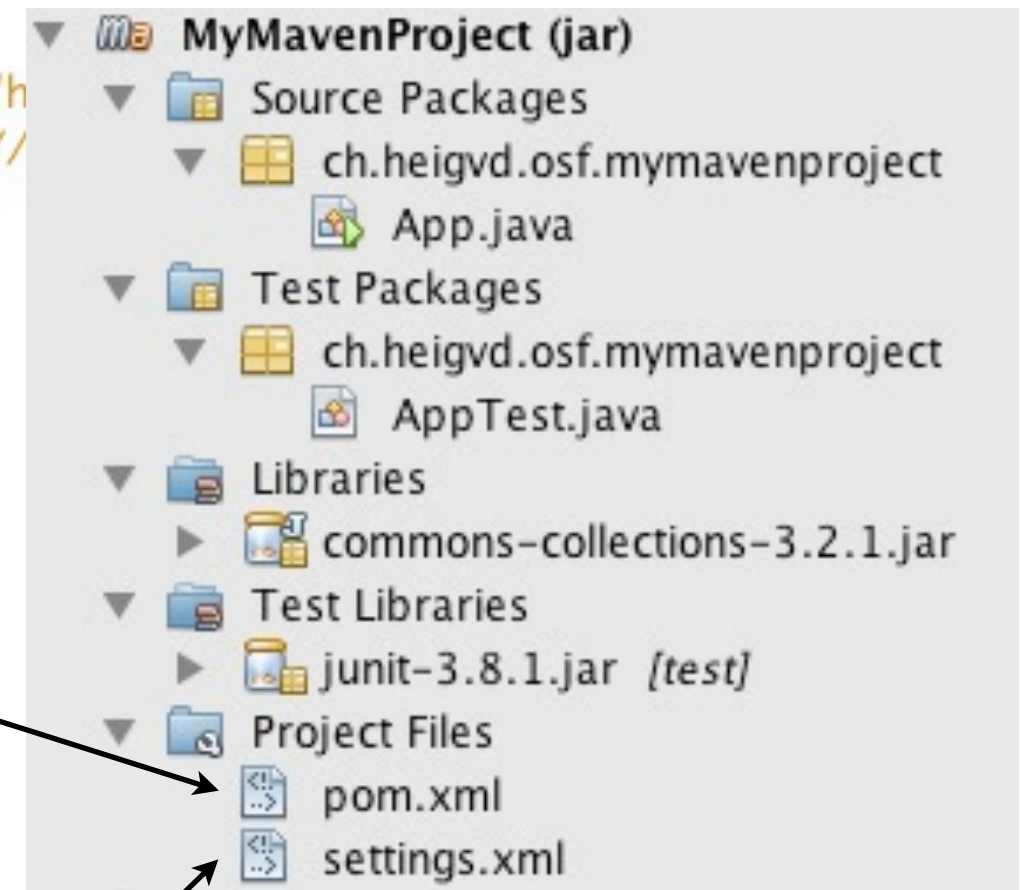
Artifact Id:

Group Id:

Version:

Package: (Optional)


```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://maven.apache.org/POM/4.0.0" 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>ch.heigvd.osf</groupId>
  <artifactId>MyMavenProject</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>MyMavenProject</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>commons-collections</groupId>
      <artifactId>commons-collections</artifactId>
      <version>3.2.1</version>
    </dependency>
  </dependencies>
</project>
```



Shortcut on your maven preferences

Dependencies

- **Scopes**
 - **compile**: you need to compile and it will be packaged with the artifact.
 - **provided**: you need it to compile, but it will not be packaged - it will be provided by the runtime environment.
 - **runtime**: you need it to execute the system, but not to compile it.
 - **test**: you only need it during for running tests.
 - **system**: similar to provided, but path has to be provided.

```
<project>
...
<dependencies>
  <dependency>
    <groupId>org.codehaus.xfire</groupId>
    <artifactId>xfire-java5</artifactId>
    <version>1.2.5</version>
  </dependency>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>3.8.1</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>servlet-api</artifactId>
    <version>2.4</version>
    <scope>provided</scope>
  </dependency>
</dependencies>
...
</project>
```

Local and Remote Repositories

- **Local repository**

- On **every machine**, there is a local repository. It is located in ~/.m2/repository.
- This repository acts as a **cache** (when you fetch libraries during a build, they are kept there for later usage).
- When you build your artifacts, they are “**installed**” here.

- **Remote repositories**

- Remote repositories are used to **share artifacts** between several developers & organizations.
- A remote repository can be **public** (e.g. <http://repo1.maven.org/maven2/>, maven repositories of open source projects, etc.)
- A remote repository can be **private** (i.e. a company manages a repository so that internal developers can share artifacts).

Want to manage your repository?

- <http://archiva.apache.org/>
- <http://nexus.sonatype.org/>



http://www.flickr.com/photos/garryknight/2818736770/sizes/m/#cc_license

“The core of Maven is pretty dumb, it doesn't know how to do much beyond parsing a few XML documents and keeping track of a lifecycle and a few plugins.

Maven has been designed to delegate most responsibility to a set of Maven Plugins which can affect the Maven Lifecycle and offer access to goals.”

Plugins

- A **plugin** implements a number of actions, called “**goals**” (aka “**mojos**”)
 - The `clean` plugin has one goal: `clean:clean`
 - The `jar` plugin has two goals: `jar:jar` and `jar:test-jar`
- Plugin goals often accept **parameters** (some optional, some required)
- Plugins are contributed by the **community** - you can write your own plugins.
- You can **invoke** a goal directly:

```
mvn clean:clean
```

- You can let maven invoke the right goals at the right time (**hollywood** principle)

```
mvn install
```

Maven Plugins

heig-vd

Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

| Plugin | Packaging types / tools | Tools | Reporting plugins |
|------------------------|-------------------------|-------------------------------|-----------------------------------|
| Core plugins | | | |
| <code>clean</code> | <code>ear</code> | <code>ant</code> | <code>changelog</code> |
| <code>compiler</code> | <code>ejb</code> | <code>antrun</code> | <code>changes</code> |
| <code>deploy</code> | <code>jar</code> | <code>archetype</code> | <code>checkstyle</code> |
| <code>failsafe</code> | <code>rar</code> | <code>assembly</code> | <code>clover</code> |
| <code>install</code> | <code>war</code> | <code>dependency</code> | <code>doap</code> |
| <code>resources</code> | <code>shade</code> | <code>enforcer</code> | <code>docck</code> |
| <code>site</code> | | <code>gpg</code> | <code>javadoc</code> |
| <code>surefire</code> | | <code>help</code> | <code>jxr</code> |
| <code>verifier</code> | | <code>invoker</code> | <code>pmd</code> |
| | | <code>jarsigner</code> | <code>project-info-reports</code> |
| | | <code>one</code> | <code>surefire-report</code> |
| | | <code>patch</code> | |
| | | <code>pdf</code> | |
| | | <code>plugin</code> | |
| | | <code>release</code> | |
| | | <code>reactor</code> | |
| | | <code>remote-resources</code> | |
| | | <code>repository</code> | |
| | | <code>scm</code> | |
| | | <code>source</code> | |
| | | <code>stage</code> | |
| | | <code>toolchains</code> | |

- A **lifecycle** is a sequence of phases (e.g. compile, test, package, etc.)
- A **phase** is where some actions can be attached (e.g. invoke a compiler during the compile phase)
- Maven provides **standard lifecycles**:
 - clean lifecycle
 - default lifecycle
 - site lifecycle
- Lifecycles can be customized depending on the **type of artifact** being built (building .jar does not involve the same steps as building a .war).
- **Convention over configuration**: if you don't specify otherwise, you let maven proceed "as usual" and do not care about the setup of lifecycles and plugins.

Table 4.3. Default Goals for POM Packaging

| Lifecycle Phase | Goal |
|-----------------|------------------------|
| package | site:attach-descriptor |
| install | install:install |
| deploy | deploy:deploy |

Table 4.2. Default Goals for JAR Packaging

| Lifecycle Phase | Goal |
|------------------------|-------------------------|
| process-resources | resources:resources |
| compile | compiler:compile |
| process-test-resources | resources:testResources |
| test-compile | compiler:testCompile |
| test | surefire:test |
| package | jar:jar |
| install | install:install |
| deploy | deploy:deploy |

Table 4.6. Default Goals for WAR Packaging

| Lifecycle Phase | Goal |
|------------------------|-------------------------|
| process-resources | resources:resources |
| compile | compiler:compile |
| process-test-resources | resources:testResources |
| test-compile | compiler:testCompile |
| test | surefire:test |
| package | war:war |
| install | install:install |
| deploy | deploy:deploy |

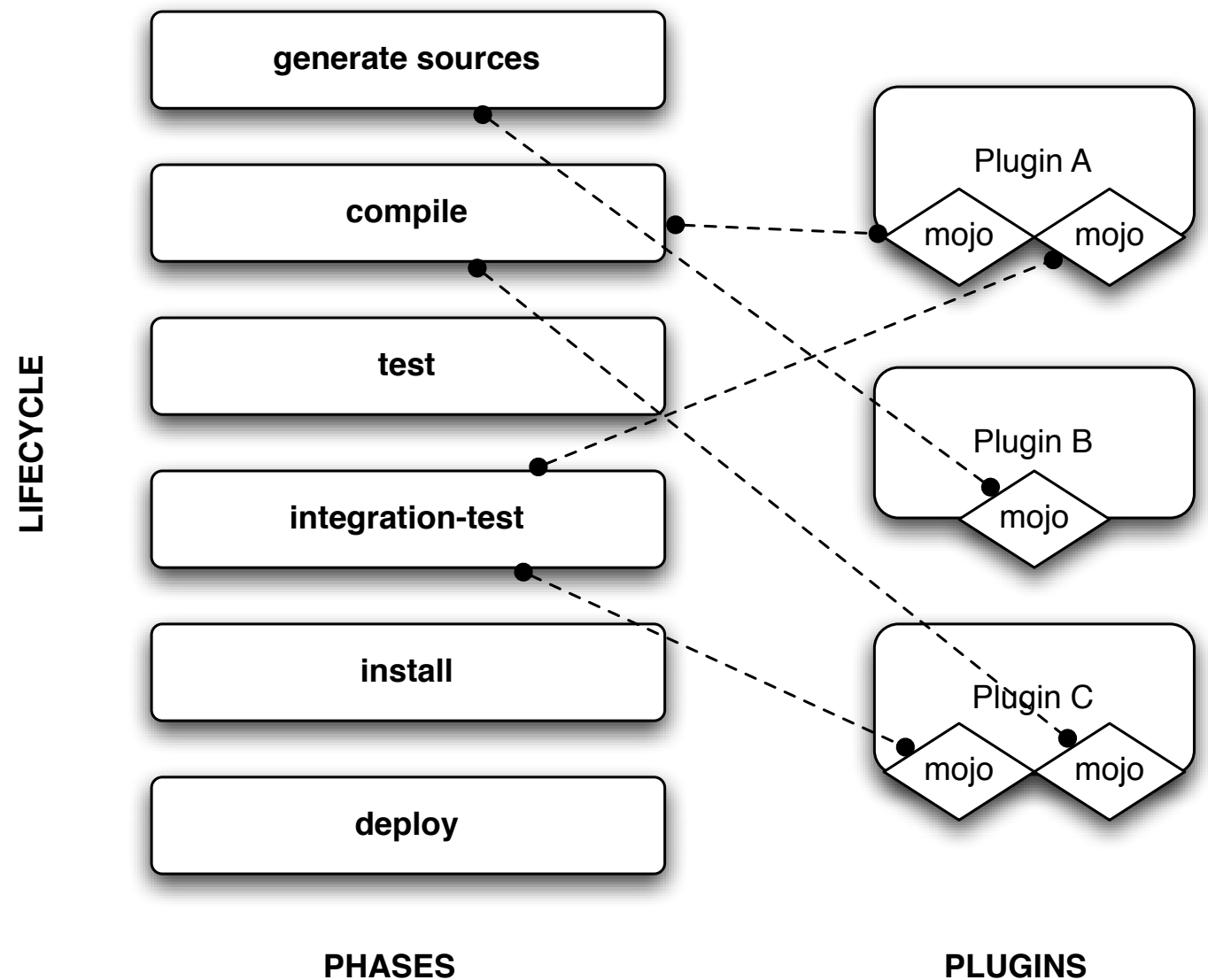
```
mvn install
```

This runs the lifecycle until the install phase; this results in the installation of the artifact in the local maven repository.

All mojos attached to the phases are executed.

```
mvn deploy
```

This runs the lifecycle until the deploy phase; this results in the deployment of the artifact in a remote maven repository.



Relationships between Projects (1)

- Multi-modules projects
 - A project can be organized in sub-projects.
 - When you build the project artifact, you also want to build the artifacts of the sub-projects.
 - If there are dependencies between some of the sub-projects, you need to build the artifacts in the right order.
 - Maven takes care of that, through the “Reactor”.

```
<modules>
  <module>moduleA</module>
  <module>moduleC</module>
  <module>moduleB</module>
</modules>
```


Relationships between Projects (2)

- Inheritance relationships between project
 - A maven project can extend another maven project.
 - This is a way to inherit various properties, such as the versions of the libraries used all sub-projects (“everybody should use log4j version 1.3.2”).
 - The relationship is captured at the beginning of the POM, using maven coordinates.
- Lost in your inheritance tree?
 - Run this goal: `mvn help:effective-pom`

```
<parent>
  <groupId>your.group.id</groupId>
  <artifactId>toto</artifactId>
  <version>1.0-SNAPSHOT</version>
</parent>
```

Inheritance & dependency management

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.sonatype.mavenbook</groupId>
  <artifactId>a-parent</artifactId>
  <version>1.0.0</version>
  ...
  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>5.1.2</version>
      </dependency>
      ...
    </dependencies>
  </dependencyManagement>
```

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.sonatype.mavenbook</groupId>
    <artifactId>a-parent</artifactId>
    <version>1.0.0</version>
  </parent>
  <artifactId>project-a</artifactId>
  ...
  <dependencies>
    <dependency>
      <groupId>mysql</groupId>
      <artifactId>mysql-connector-java</artifactId>
    </dependency>
  </dependencies>
</project>
```



- For the same project, you often want to be able to do **different types of builds**:
 - A developer builds the artifact on his local machine during development.
 - Continuous integration triggers daily builds on a QA server.
 - You want to build the artifact before putting into production.
- For every **different type of build**, you may want to:
 - Use different “components” in the environment (different glassfish domains, different DBs, etc.)
 - Skip some of the phases in the build process.
 - Attach some special plugins to some of the phases
- Profiles allow you to do just that. Within a **profile definition**, you declare the expected behavior. You then **activate** one or more profiles (either via a command line flag, or indirectly through maven properties).

Profiles

```
<project>
  <profiles>
    <profile>
      <build>
        <defaultGoal>...</defaultGoal>
        <finalName>...</finalName>
        <resources>...</resources>
        <testResources>...</testResources>
        <plugins>...</plugins>
      </build>
      <reporting>...</reporting>
      <modules>...</modules>
      <dependencies>...</dependencies>
      <dependencyManagement>...</dependencyManagement>
      <distributionManagement>...</distributionManagement>
      <repositories>...</repositories>
      <pluginRepositories>...</pluginRepositories>
      <properties>...</properties>
    </profile>
  </profiles>
</project>
```

- Very often, you want some parts of your resource files, possibly of your source code, to depend on the execution environment:
 - If I deploy on the QA server, I want to talk to this DB server; if I deploy on the production server, I want to talk to this other DB server.
 - If I deploy on the test server, I want to invoke this web service endpoint; if I deploy on the production server, I want to use this other endpoint.
- Resource filtering is a mechanism, where you can ask maven to **expand properties** in your code base during the build process.

Resource Filtering

```
<profiles>
  <profile>
    <id>production</id>
    <properties>
      <jdbc.driverClassName>oracle.jdbc.driver.OracleDriver</jdbc.driverClassName>
      <jdbc.url>jdbc:oracle:thin:@proddb01:1521:PROD</jdbc.url>
      <jdbc.username>prod_user</jdbc.username>
      <jdbc.password>s00p3rs3cr3t</jdbc.password>
    </properties>
  </profile>
</profiles>
```

POM.xml - you give values to properties

```
<bean id="dataSource" destroy-method="close"
      class="org.apache.commons.dbcp.BasicDataSource">
  <property name="driverClassName" value="${jdbc.driverClassName}"/>
  <property name="url" value="${jdbc.url}"/>
  <property name="username" value="${jdbc.username}"/>
  <property name="password" value="${jdbc.password}"/>
</bean>
</beans>
```

Source - you reference maven properties with \${xxx}

Archetypes

- A maven archetype is a **skeleton** for a certain type of project.
- You have already used archetypes when creating your first maven project.
- Archetype definitions can be **shared** via repositories.
- **Creating you own archetype is a very efficient way to create a SDK.**
- **AppFuse** is an open source project that uses archetypes as a way to pre-integrate several open source frameworks (with a layer on top of them).
- The archetype facility is available through the **archetype plugin**:

<http://maven.apache.org/archetype/maven-archetype-plugin/>



<http://appfuse.org>