Lecture 5: Tooling

Olivier Liechti
AMT

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

Agenda

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

15h00- 15h10	10’	**Lecture** Automation tools, continuous integration, continous delivery
15h10 - 15h40	30’	**Lecture** Building Java projects with apache maven
15h40 - 16h10	30’	**Lecture** Building a build pipeline with Jenkins
16h10 - 16h30	20’	**Lecture** Vagrant, Docker & friends... putting it together
16h30 - 18h00	90’	**Lab** Cleaning up your previous work and moving it into a maven + Jenkins environment

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

Automation Tools

Why do we need **automation** tools?

- To **save time** and avoid the need to repeat **boring tasks** again and again. Do you prefer to:
  - spend 2 days of designing and implementing an automation tool (writing scripts, using tools, etc.)
  - spend 2 minutes of repeating manual tasks, every time you do a modification in your project (2 minutes x 3 modifications/per hour x 10 hours/day x 5 days/week x 10 weeks = 60 hours per developer)
- To improve **reliability** and **reproducibility** (because automation reduces the risk of human errors).
- To reduce the “it works on MY machine” syndrome.

From **continuous integration** to continuous delivery

- Agile methodologies have put an emphasis on **continuous integration** since the early days (e.g. it is a core practice in eXtreme Programming).
- The goal is to avoid the **nightmare** of doing an integration only at the end of a project:
  - Think that you are building a complex software in a team. You identify 5 sub-systems and define interfaces between them. One team works on every sub-system, using the interface specification as a contract. If the 5 sub-systems are integrated on the last day, do you think that the software will work flawlessly?
- With continuous integration, the goal is to **integrate at least once a day**, but in practice more than once a day. To do that, having a shared source code repository, automated build tools and automated tests are key.
- With this approach, issues (such as unclear interfaces) can be identified and solved **early** in the project.

From continuous integration to **continuous delivery**

- More recently, continuous integration has evolved towards the notion of continuous delivery.
- The key difference is that:
  - **continuous integration** ends at the creation of a software package (a .jar file, a .war file) that can be handed over to the “ops team”.
  - **continuous delivery** ends at the deployment of a new feature to the end-users. In other words, it also encompasses the release and deployment parts of the process.
- You may have heard that Facebook, Twitter and other companies who run large scale Internet services “**deploy several times a day**”.
- This is only possible because they have built a **highly sophisticated automation infrastructure**, that covers the end to end process.
- If you find this exciting, you should learn about the job of “**Devops**”, which is becoming increasingly important.

The Anatomy of a Continuous Delivery Pipeline

Jenkins cruisecontrol

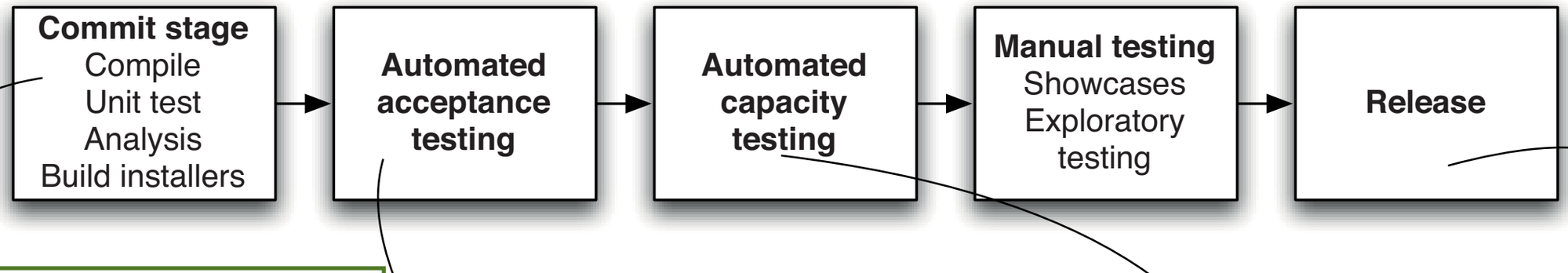


Figure 1.1 *The deployment pipeline*

maven

sonar JUnit

N unit QUnit

SUBVERSION git

Concordion Se

Cucumber

jbehave

bugbuster

Apache JMeter

Chef Puppet

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

But let's start with relatively simple tasks and tools.

heig-vd

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

Apache 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

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

http://www.sonatype.com/resources/books

Getting started...

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

- Install maven 3
  - http://maven.apache.org/download.html
- Create, build and run a project

```
mvn archetype:generate \
   -DarchetypeGroupId=org.apache.maven.archetypes \
   -DarchetypeArtifactId=maven-archetype-quickstart \
   -DgroupId=ch.heigvd.amt.cool \
   -DartifactId=coolProject
cd coolProject
mvn install
java -cp target/coolProject-1.0-SNAPSHOT.jar \
ch.heigvd.amt.cool.App
```

a maven plugin:goal used to generate a new project from a skeleton (an "archtetype")

which archetype do we want to use as a model for our new project?

the name of our new project

now that we have the skeleton, let's ask maven to build the project

we find the result of the build in the **target** directory

Look at these directories and files

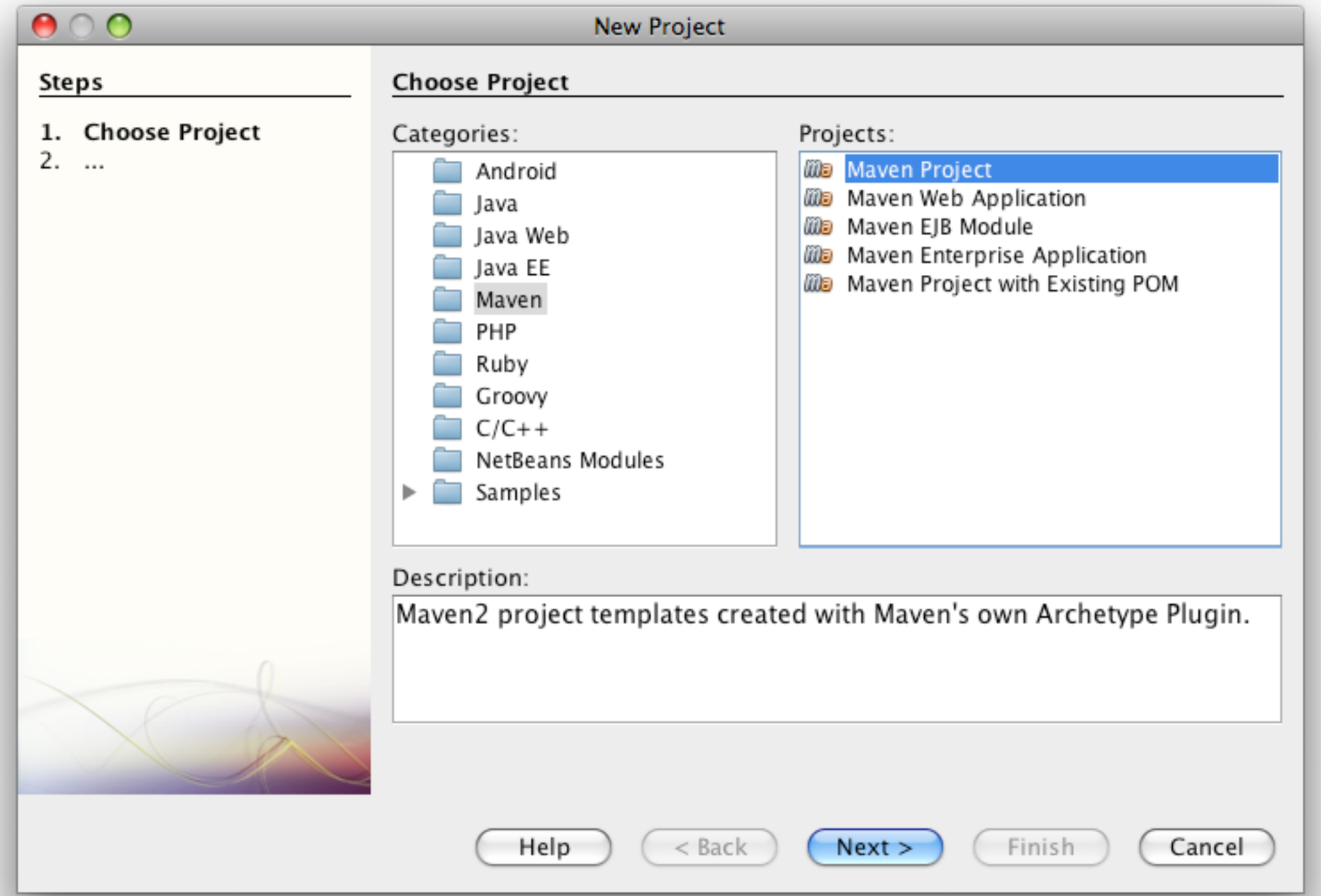
- ~/.m2/repository
- ~/.m2/settings.xml
- $INSTALLATION/conf/settings.xml

this is where maven downloads and caches all lib dependencies for your projects

this is where you setup your own maven preferences (proxies, repositories, passwords, etc.)

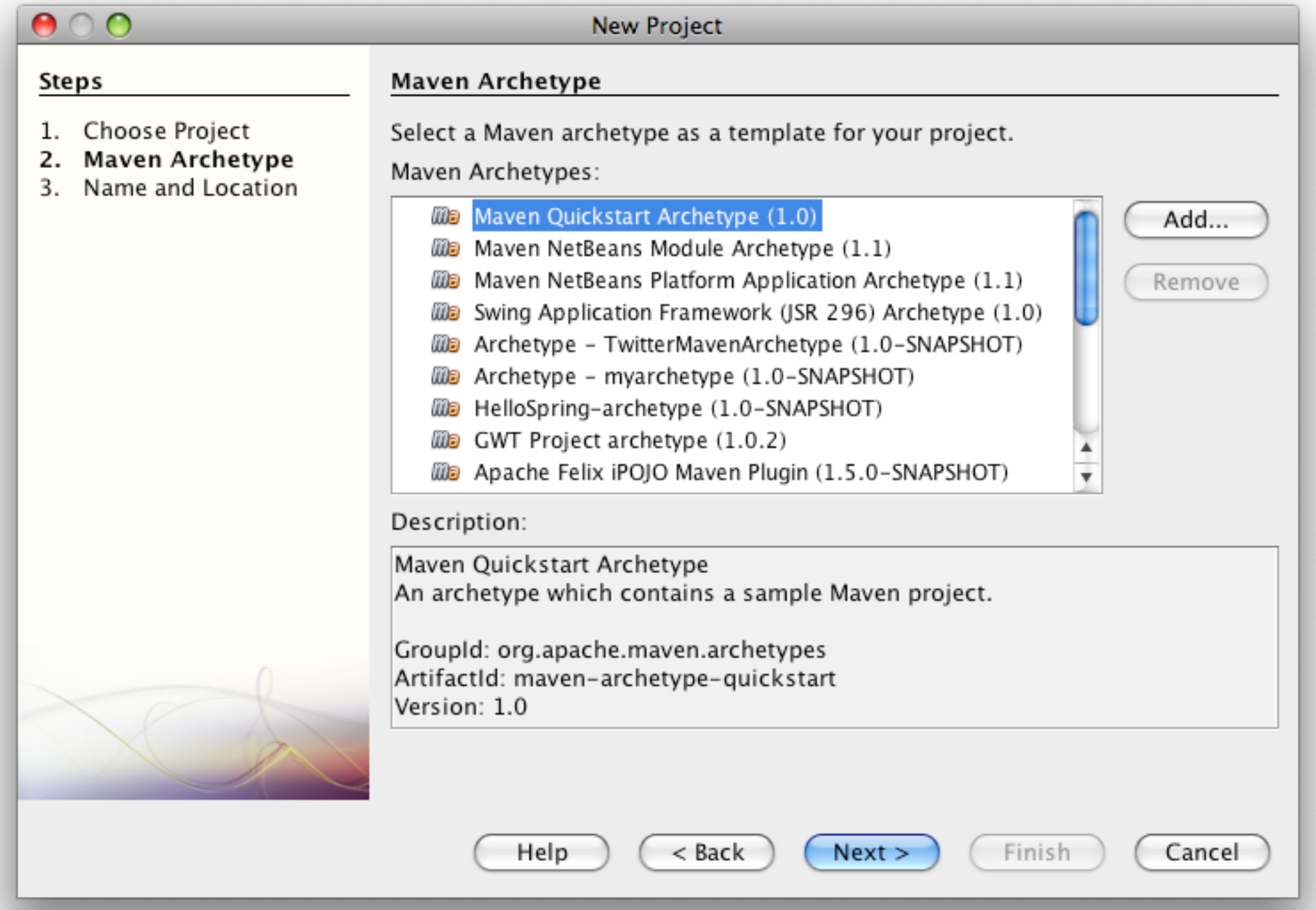
Creating a maven project with NetBeans

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud



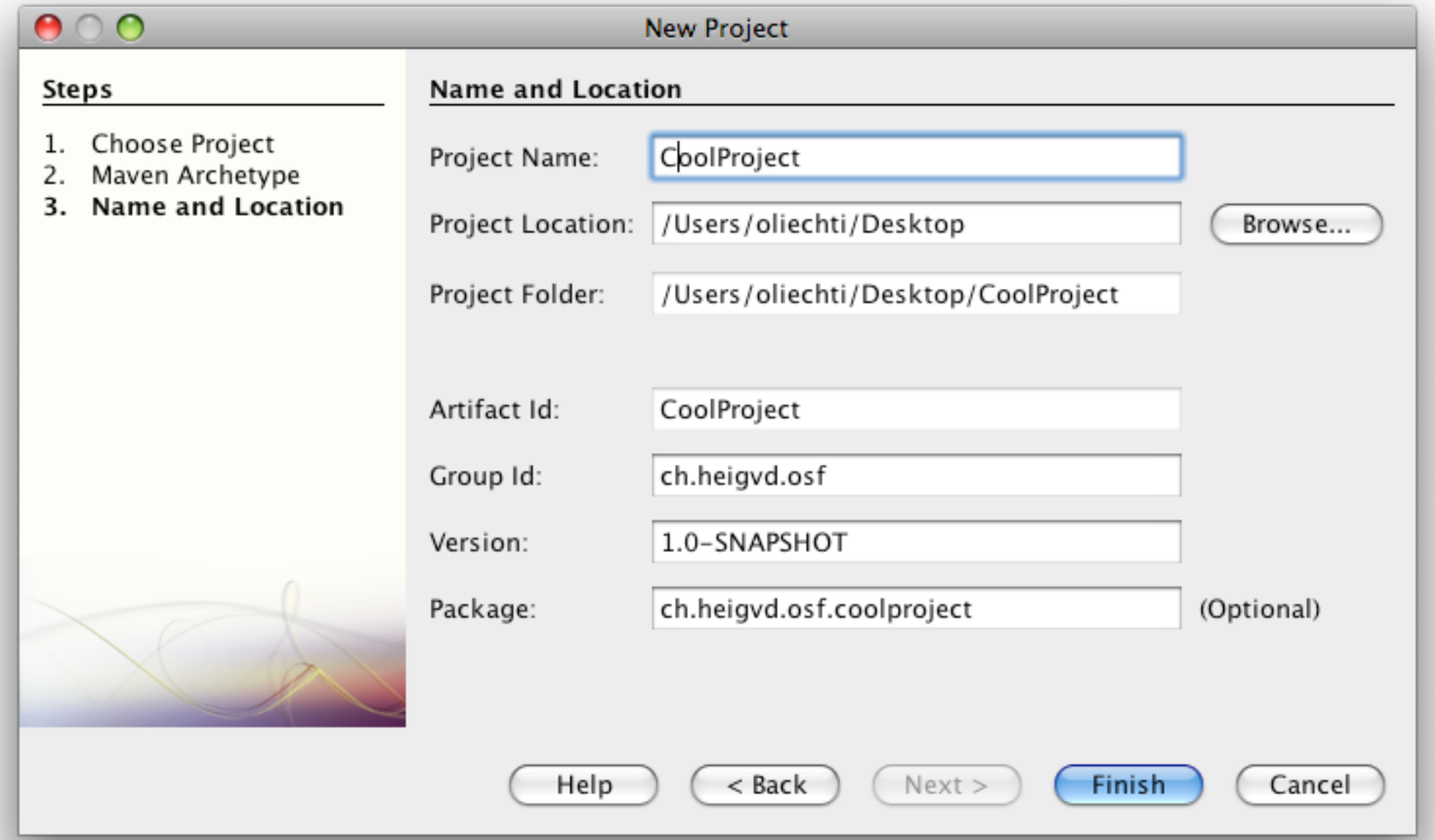
Creating a maven project with NetBeans

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

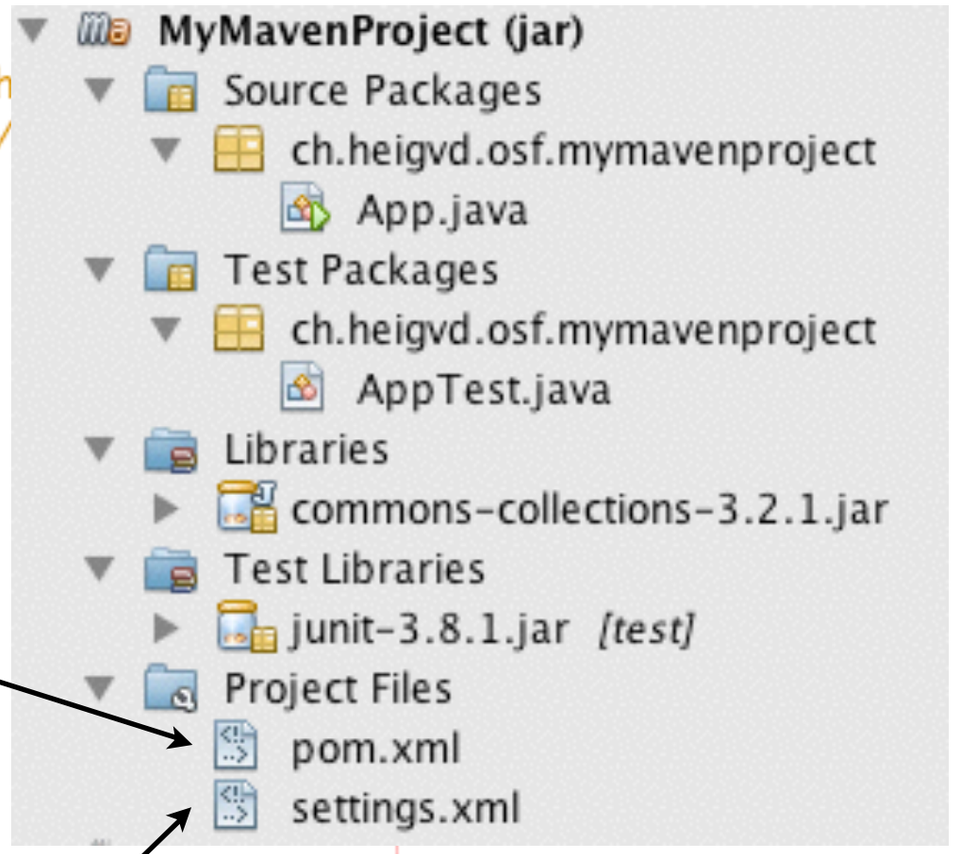


Creating a maven project with NetBeans

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud



```xml
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="h
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://
  <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 (~/.m2/settings.xml)

Project Object Model (POM)

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

- 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.

```
<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>ch.heigvd.amt.cool</groupId>
  <artifactId>coolProject</artifactId>
  <version>1.0-SNAPSHOT</version>

  <packaging>jar</packaging>

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

  <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>
</project>
```

heig-vd
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

The "**coordinates**" for our project. This is how others projects will find us. "SNAPSHOT" means that it's the "bleeding edge" version.

The type of project, i.e. **the type of artifact that the build process will create**. Here, it is a .jar file. Soon, we will generate a .war file or an .ear file. Each project type has its own **build lifecycle**.

What do we need to **build**, **test** and **execute** our software? At the moment, we only need one lib (junit) to test the project. We **declare this dependency with the coordinates of the junit library**.

```xml
<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>ch.heigvd.amt.cool</groupId>
  <artifactId>coolProject</artifactId>
  <version>1.0-SNAPSHOT</version>

  <packaging>jar</packaging>

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

  <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>
</project>
```

heig-vd
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

Maven knows a lot about Java projects and how to build them:

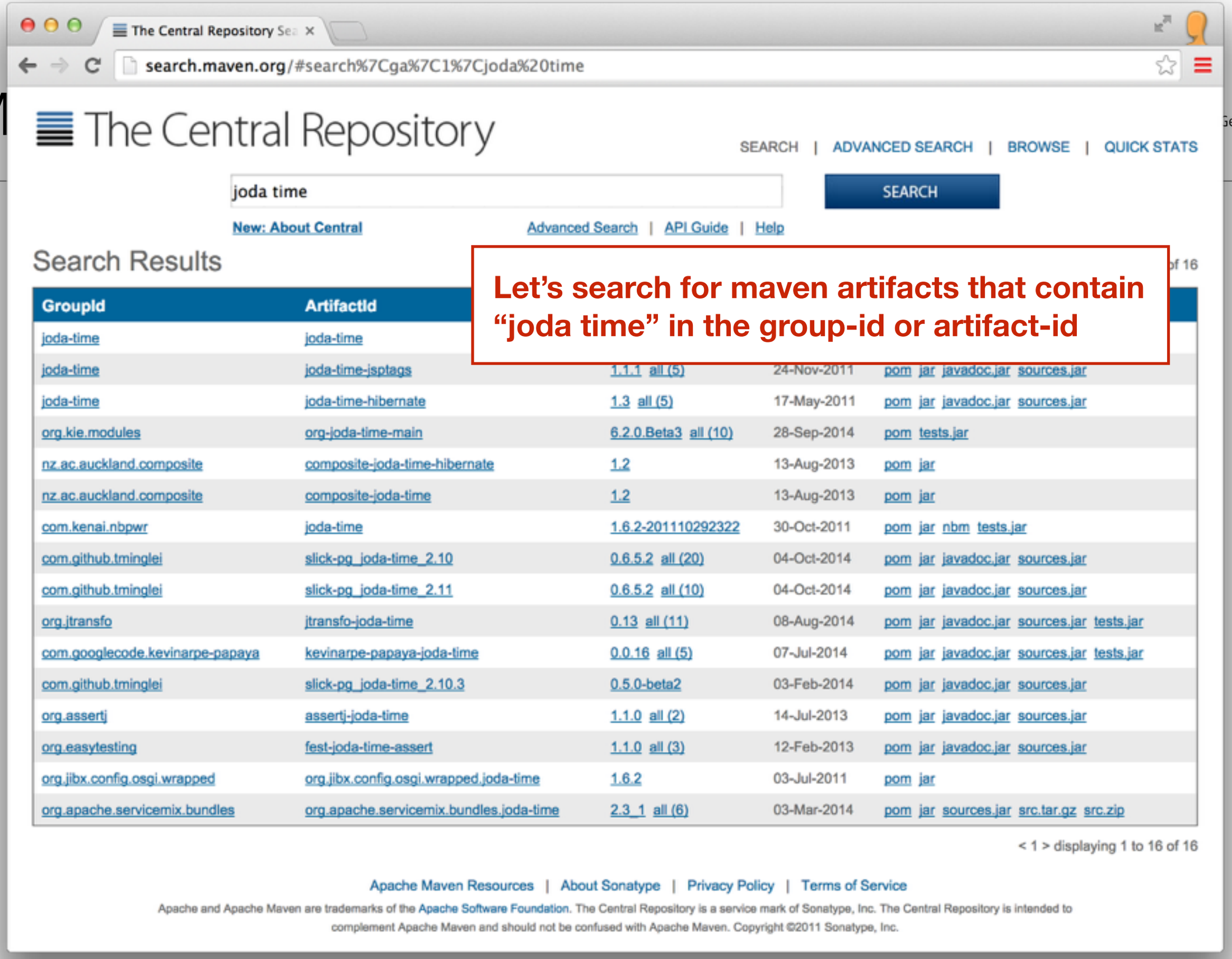
- one **phase** of the build **lifecycle** consists in **compiling** Java source files
- a **convention** is to put these source files in the **./src/main/java** subtree.
- another **phase** of the build lifecycle consists in compiling and running **unit tests**.
- a **convention** is to put the test source files in the **./src/test/java** subtree.
- another phase of the build lifecycle consists in **packaging** the .class files into a **.jar** file.
- a **convention** is to put the outcome of the build process in the **./src/target/** directory.

All of that knowledge is implicit and does not have to be explicitly defined in the POM file!

Maven Coordinates

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

- 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).


The Central Repository Sea
search.maven.org/#search%7Cga%7C1%7Cjoda%20time
The Central Repository
SEARCH | ADVANCED SEARCH | BROWSE | QUICK STATS
joda time
SEARCH
New: About Central
Advanced Search | API Guide | Help
Search Results
Let's search for maven artifacts that contain "joda time" in the group-id or artifact-id
GroupId
ArtifactId
joda-time joda-time
joda-time joda-time-jsptags 1.1.1 all (5) 24-Nov-2011 pom jar javadoc.jar sources.jar
joda-time joda-time-hibernate 1.3 all (5) 17-May-2011 pom jar javadoc.jar sources.jar
org.kie.modules org-joda-time-main 6.2.0.Beta3 all (10) 28-Sep-2014 pom tests.jar
nz.ac.auckland.composite composite-joda-time-hibernate 1.2 13-Aug-2013 pom jar
nz.ac.auckland.composite composite-joda-time 1.2 13-Aug-2013 pom jar
com.kenai.nbpwr joda-time 1.6.2-201110292322 30-Oct-2011 pom jar nbm tests.jar
com.github.tminglei slick-pg_joda-time_2.10 0.6.5.2 all (20) 04-Oct-2014 pom jar javadoc.jar sources.jar
com.github.tminglei slick-pg_joda-time_2.11 0.6.5.2 all (10) 04-Oct-2014 pom jar javadoc.jar sources.jar
org.jtransfo jtransfo-joda-time 0.13 all (11) 08-Aug-2014 pom jar javadoc.jar sources.jar tests.jar
com.googlecode.kevinarpe-papaya kevinarpe-papaya-joda-time 0.0.16 all (5) 07-Jul-2014 pom jar javadoc.jar sources.jar tests.jar
com.github.tminglei slick-pg_joda-time_2.10.3 0.5.0-beta2 03-Feb-2014 pom jar javadoc.jar sources.jar
org.assertj assertj-joda-time 1.1.0 all (2) 14-Jul-2013 pom jar javadoc.jar sources.jar
org.easytesting fest-joda-time-assert 1.1.0 all (3) 12-Feb-2013 pom jar javadoc.jar sources.jar
org.jibx.config.osgi.wrapped org.jibx.config.osgi.wrapped.joda-time 1.6.2 03-Jul-2011 pom jar
org.apache.servicemix.bundles org.apache.servicemix.bundles.joda-time 2.3_1 all (6) 03-Mar-2014 pom jar sources.jar src.tar.gz src.zip
< 1 > displaying 1 to 16 of 16
Apache Maven Resources | About Sonatype | Privacy Policy | Terms of Service
Apache and Apache Maven are trademarks of the Apache Software Foundation. The Central Repository is a service mark of Sonatype, Inc. The Central Repository is intended to complement Apache Maven and should not be confused with Apache Maven. Copyright ©2011 Sonatype, Inc.

M

Gestion

The Central Repository Sea ×

search.maven.org/#search%7Cgav%7C1%7Cg%3A%22joda-time%22%20AND%20a%3A%22joda-time%22

The Central Repository

SEARCH | ADVANCED SEARCH | BROWSE | QUICK STATS

g:"joda-time" AND a:"joda-time"

SEARCH

New: About Central Advanced Search | API Guide | Help

Search Results

< 1 > displaying 1 to 19 of 19

GroupId	ArtifactId	Version	Updated	Download
joda-time	joda-time	2.5	03-Oct-2014	pom jar javadoc.jar sources.jar
joda-time	joda-time	2.4	27-Jul-2014	pom jar javadoc.jar sources.jar
joda-time	joda-time	2.3	16-Aug-2013	pom jar javadoc.jar sources.jar
joda-time	joda-time	2.2	08-Mar-2013	pom jar javadoc.jar sources.jar
joda-time	joda-time	2.1	22-Feb-2012	pom jar javadoc.jar sources.jar
joda-time	joda-time	2.0	01-Aug-2011	pom jar javadoc.jar sources.jar
joda-time	joda-time	1.6.2	14-Sep-2010	pom jar javadoc.jar sources.jar
joda-time	joda-time	1.6.1	08-Aug-2010	pom jar javadoc.jar sources.jar
joda-time	joda-time	1.6		
joda-time	joda-time	1.5.2		
joda-time	joda-time	1.5.1		
joda-time	joda-time	1.5		
joda-time	joda-time	1.4	16-Nov-2006	pom jar
joda-time	joda-time	1.3	26-Aug-2006	pom jar
joda-time	joda-time	1.2.1	13-Feb-2006	pom jar
joda-time	joda-time	1.2	10-Jan-2006	pom jar
joda-time	joda-time	1.1	24-Nov-2005	pom jar
joda-time	joda-time	1.0	24-Nov-2005	pom jar
joda-time	joda-time	0.95	24-Nov-2005	pom jar

For one library, there is one different artifact for every version (the history is kept)

< 1 > displaying 1 to 19 of 19

Dependencies and their scopes

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

- **compile**: you need the library to compile your project. The library will be packaged with your artifact.
- **provided**: you need it to compile your project, but it will not be packaged. Rather, it will be provided by the runtime environment (e.g. Java EE APIs).
- **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

heig-vd
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

- **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?

heig-vd
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

- This is something that you would typically consider when you are developing software in a **company** (you will not need it for your project).
- Imagine that you are working on several customer projects. All of these projects have shared needs, so you are likely to **write reusable libraries** and modules over time.
- What you would like to do, is to be able to **declare a dependency between a customer project and a list of your own reusable libraries**. You would like to use maven for that, we would like to keep the reusable libraries private (distribution via a public maven repo is not an option).
- There are **artifacts repository** software that you can install in your infra:
  - http://www.sonatype.org/nexus/go/
  - http://archiva.apache.org/

Plugins

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

“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.”

http://www.sonatype.com/books/mvnref-book/reference/installation-sect-universal-reuse.html

Plugins

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

- A **plugin** implements a number of actions, called "**goals**" (aka "**mojos**")
  - The `clean` plugin has one goal: `clean:clean`
  - The `jar` plugin has too 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 (**inversion of control!**)

```
mvn install
```

Standard Maven Plugins

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

Plugin
Core plugins
clean
compiler
deploy
failsafe
install
resources
site
surefire
verifier

Packaging types / tools
ear
ejb
jar
rar
war
shade

Tools
ant
antrun
archetype
assembly
dependency
enforcer
gpg
help
invoker
jarsigner
one
patch
pdf
plugin
release
reactor
remote-resources
repository
scm
source
stage
toolchains

Reporting plugins
changelog
changes
checkstyle
clover
doap
docck
javadoc
jxr
pmd
project-info-reports
surefire-report

Lifecycle

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

- 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

http://www.sonatype.com/books/mvnref-book/reference/lifecycle-sect-war.html#

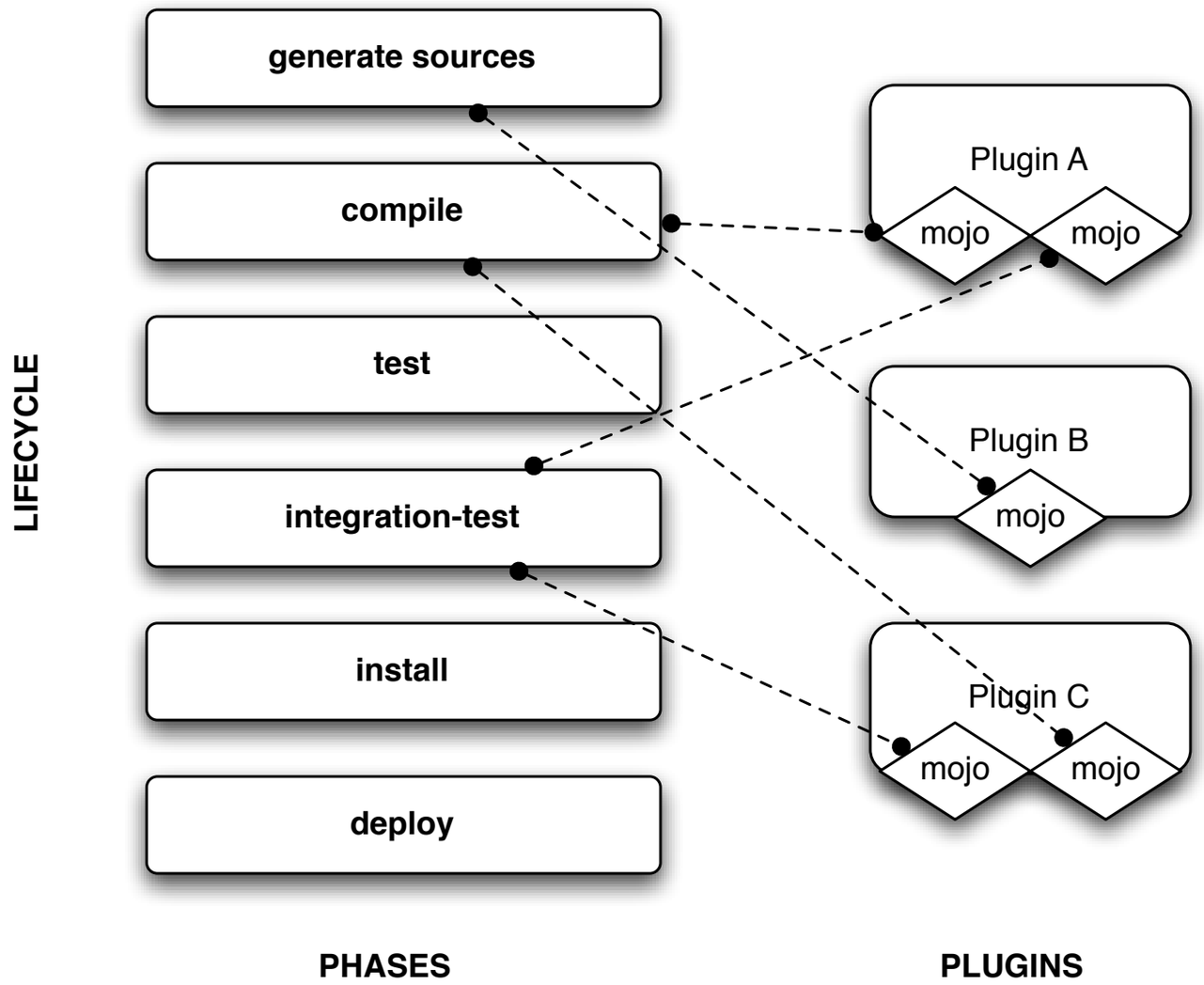
```
mvn install
```

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.



maven deploy has nothing to do with the deployment of a Java EE app!

Profiles

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

- 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

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

```
<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>
```

Resource Filtering

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

- 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

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

```xml
<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

```xml
    <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

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

- 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

Relationships between Projects (1)

heig-vd
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

- *This is something you need to understand if you build an .ear artifact. We will build a .war artifact, so let's not worry too much about this right now.*
- Multi-modules projects
  - A project can be organized in sub-projects.
  - When you built 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)

heig-vd
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

- *This is also an advanced feature. It is included in the slides, but we will most likely not need to dig into it for the project.*
- 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>
```

Relationships between Projects (3)

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

```
<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>
```

heig-vd

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

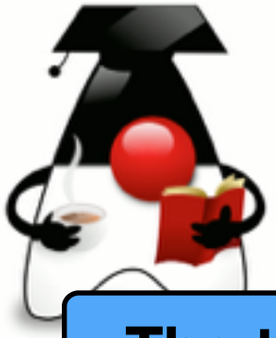
Jenkins

Jenkins

What is Jenkins?

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

- Jenkins is a **continuous integration server**, extensible with plug-ins provided by the community.
- Think of it as the music director ("**chef d'orchestre**"), who controls all the steps of the continuous integration process.



"**Managing the process**". Is it not what maven is supposed to do? Do I have to choose between maven and Jenkins?

The two tools are complementary. Launching a maven build is typically one step of the continuous integration process. But there are other steps that are out of the scope of maven, such as fetching sources from a git repo or deploying to a production server.

Some tasks can be done **either via a maven plugin**, **or via a Jenkins step**. For instance, Glassfish domains can be created both ways. Recommendation: stick with standard maven plugins and handle infrastructure concerns (e.g. Glassfish) with Jenkins.

The Anatomy of a Continuous Delivery Pipeline

Jenkins cruisecontrol

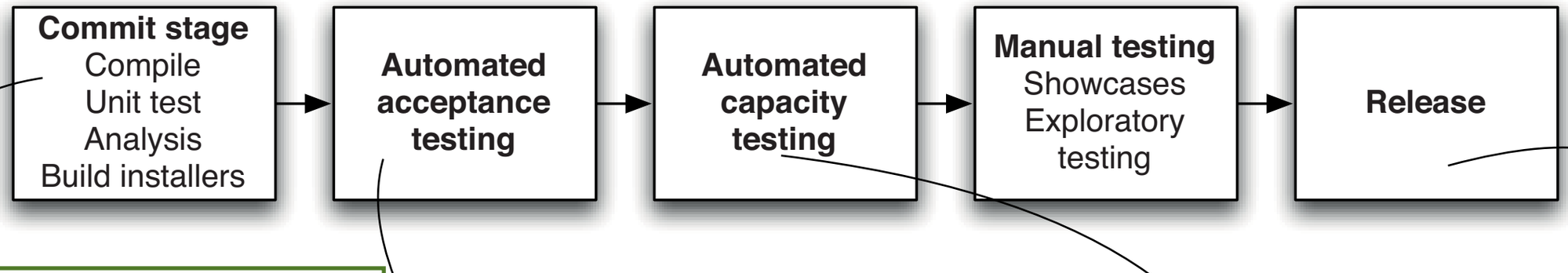


Figure 1.1 *The deployment pipeline*

maven

sonar JUnit

N unit QUnit

SUBVERSION git

Concordion Se

Cucumber

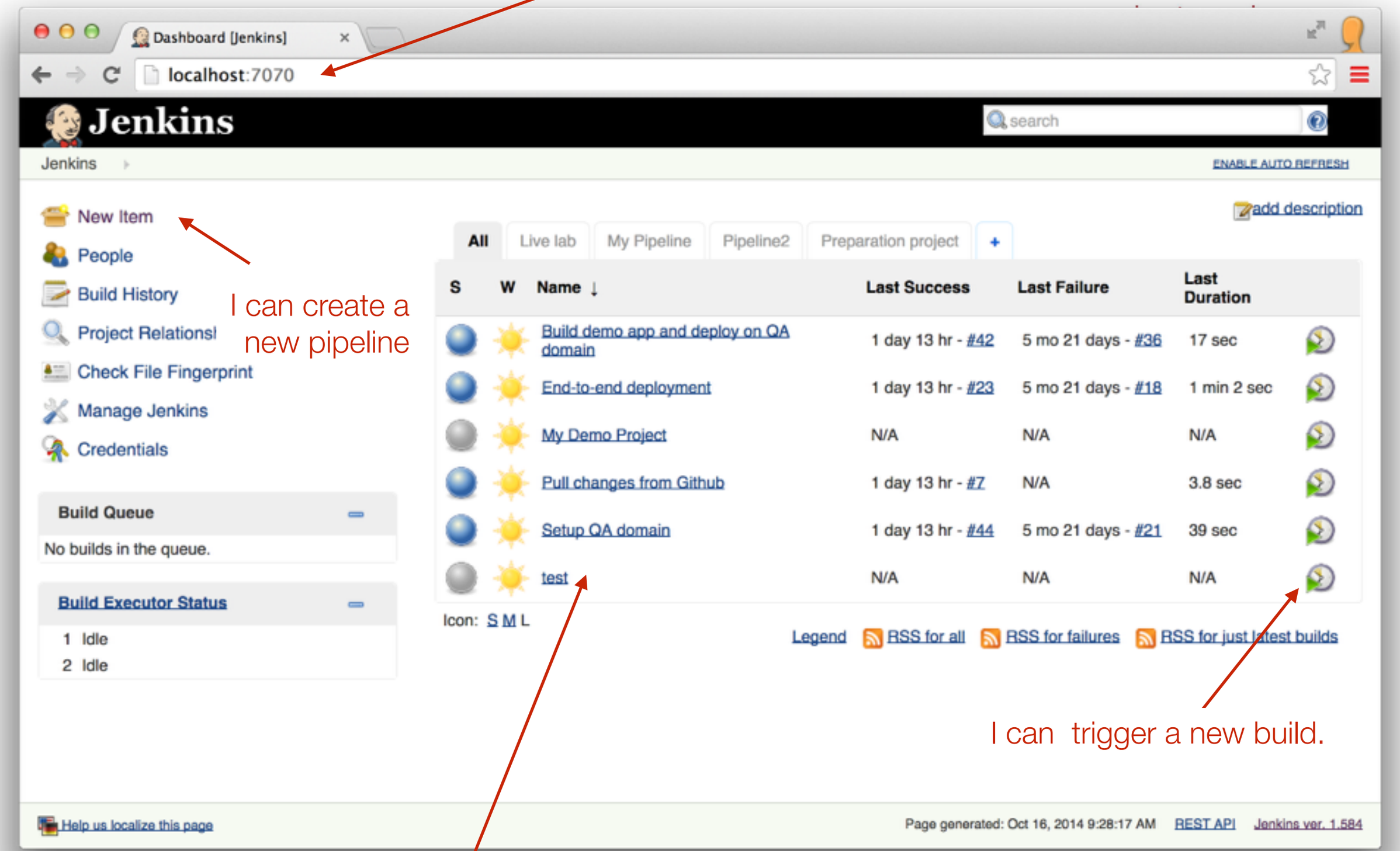
jbehave

bugbuster

Apache JMeter

Chef Puppet

Jenkins is a web app. It can be deployed in an app server OR executed in stand-alone mode.

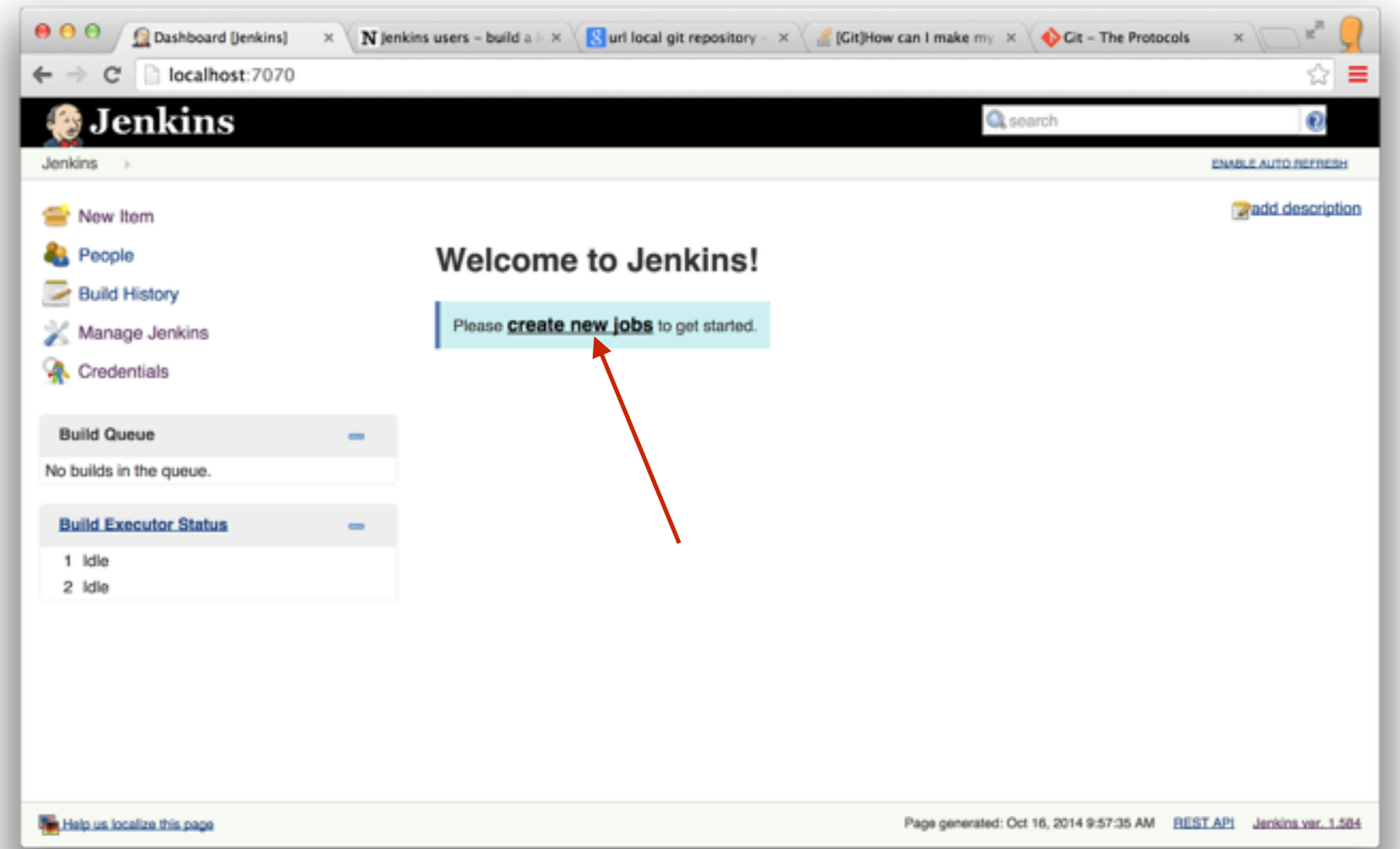


This is list of all my “build pipelines”. What is their current status, was there an issue in the nightly build, etc.

Getting our feet wet...

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

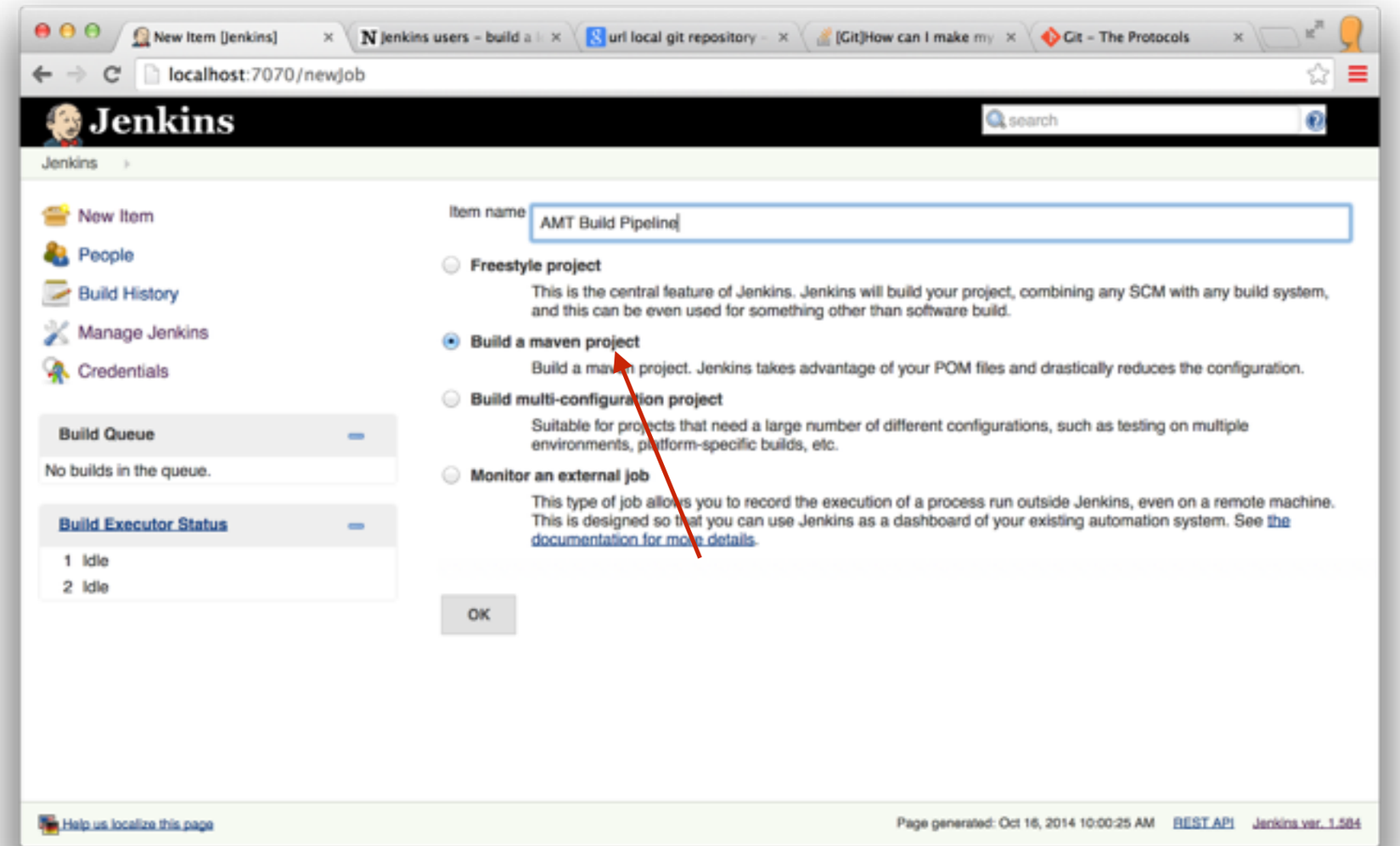
- Start by **downloading the jenkins** war package.
- Run it as a standalone application: `java -jar jenkins.war --httpPort=7070`



Getting our feet wet...

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

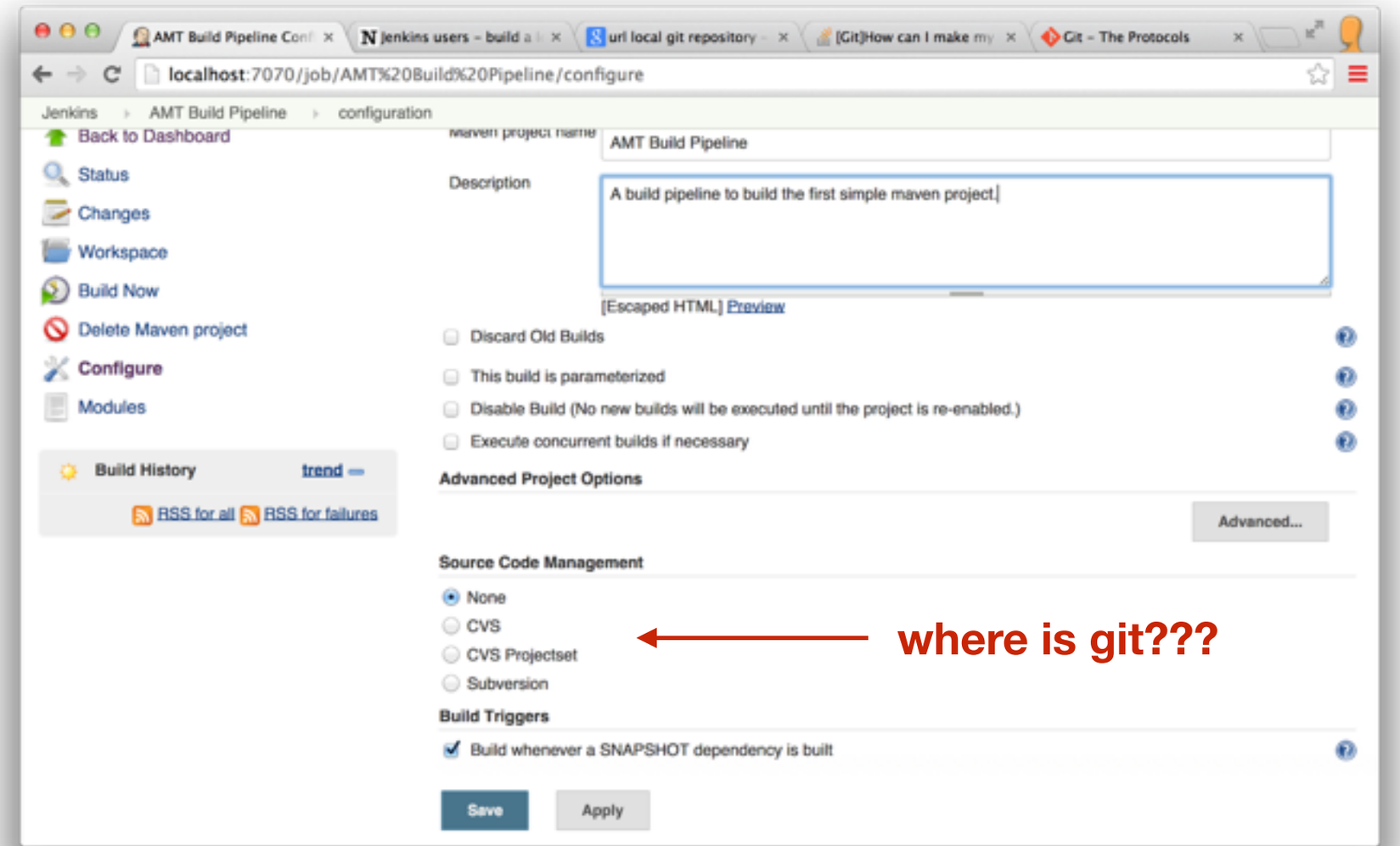
- Create a new job, give it a name.
- We will use the "**Build a maven project**" template.



Configure the maven project template

heig-vd
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

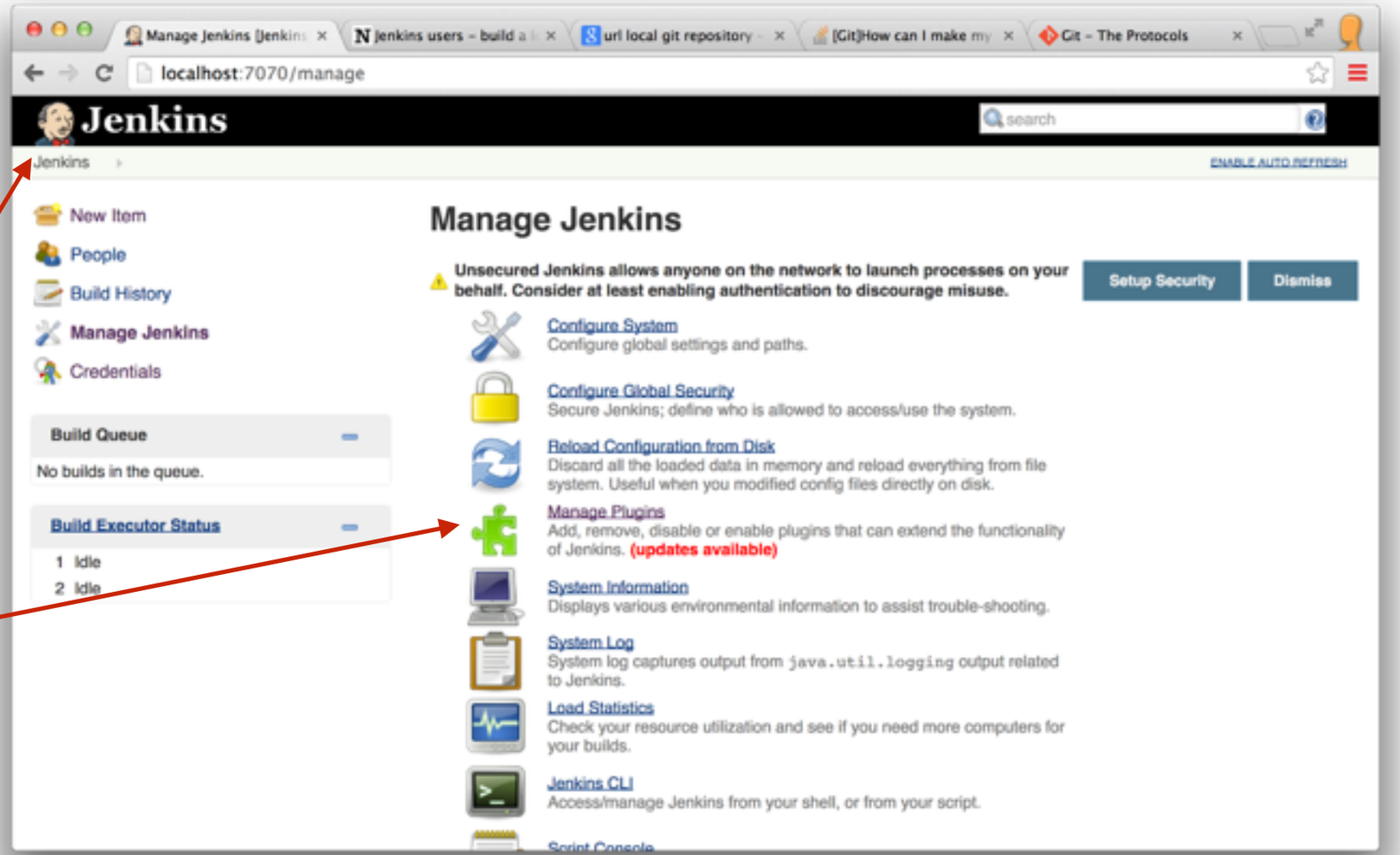
- Typically, the first step in the build process is to retrieve source code from a repository (git, subversion, etc.).
- Out-of-the-box, we don't see any option to use a git repository...



Adding support for Git

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

- The power of Jenkins comes from the multitude of plugins developed by the community.

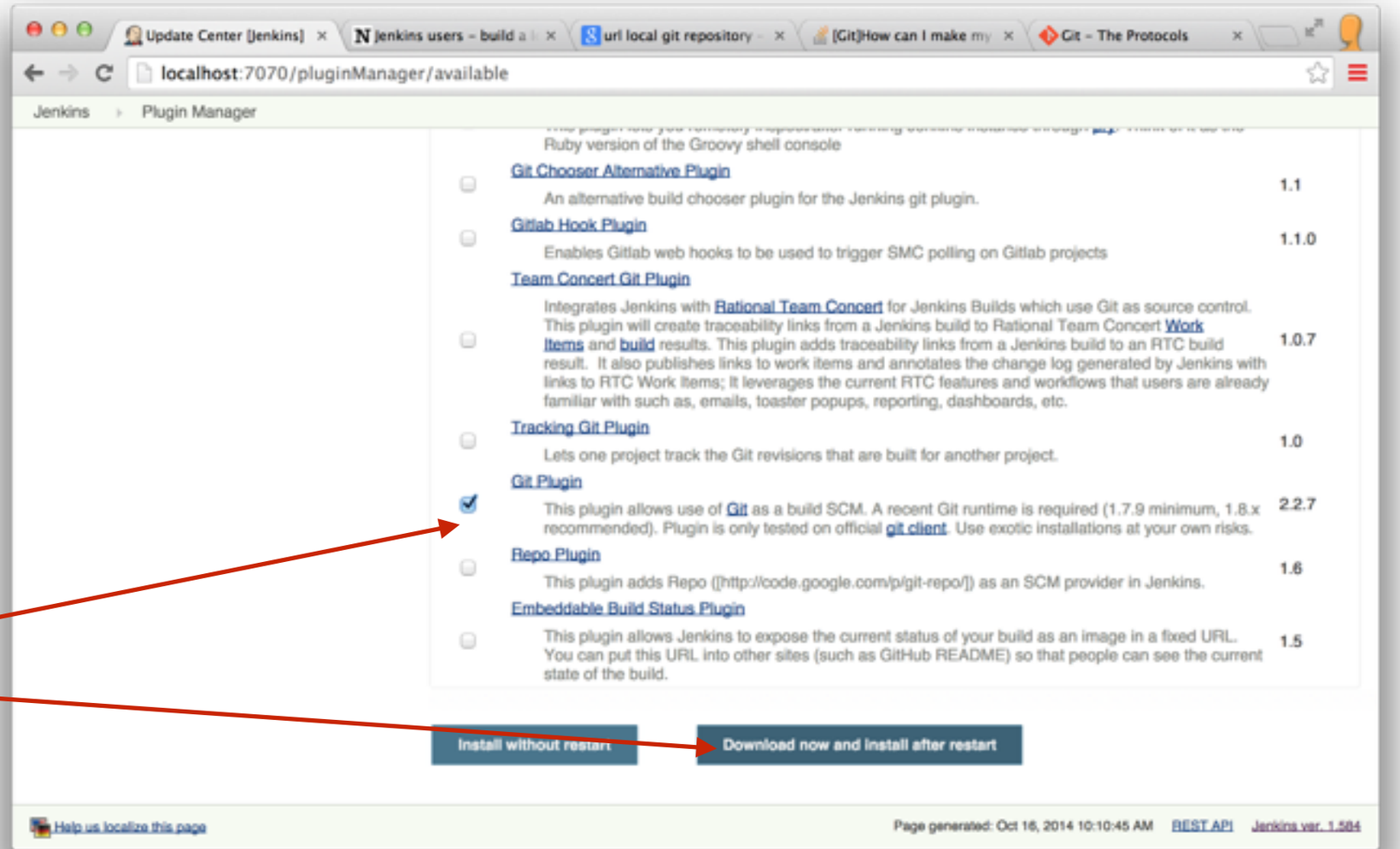


from the Jenkins home, we have an option to add plugins

Adding support for Git

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

- Doing a search on “git” gives us a list of plugins.
- Let’s install the Git Plugin. We might consider GitHub plugins later...

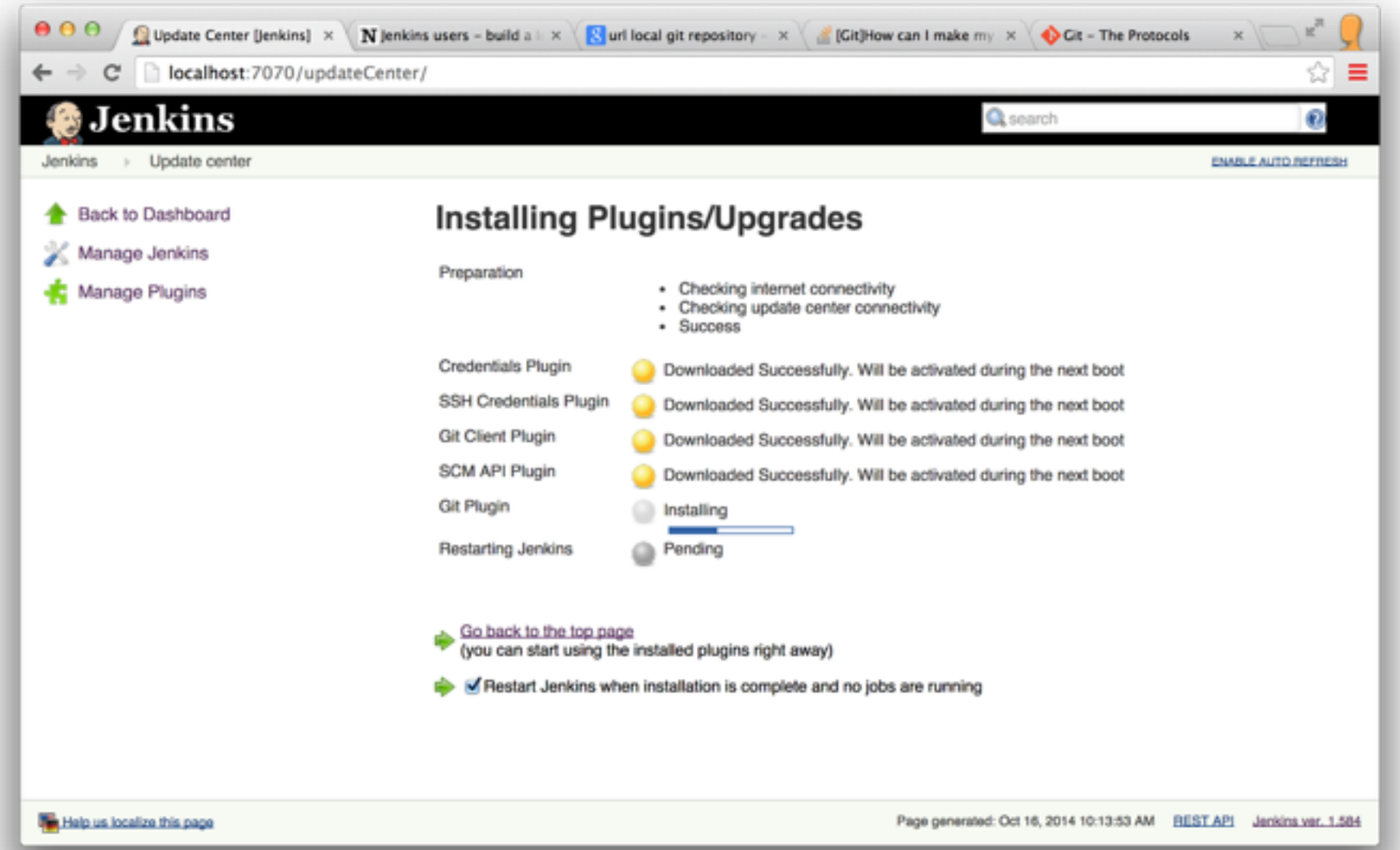


There is a Git Plugin (and many other related ones...)

Adding support for Git

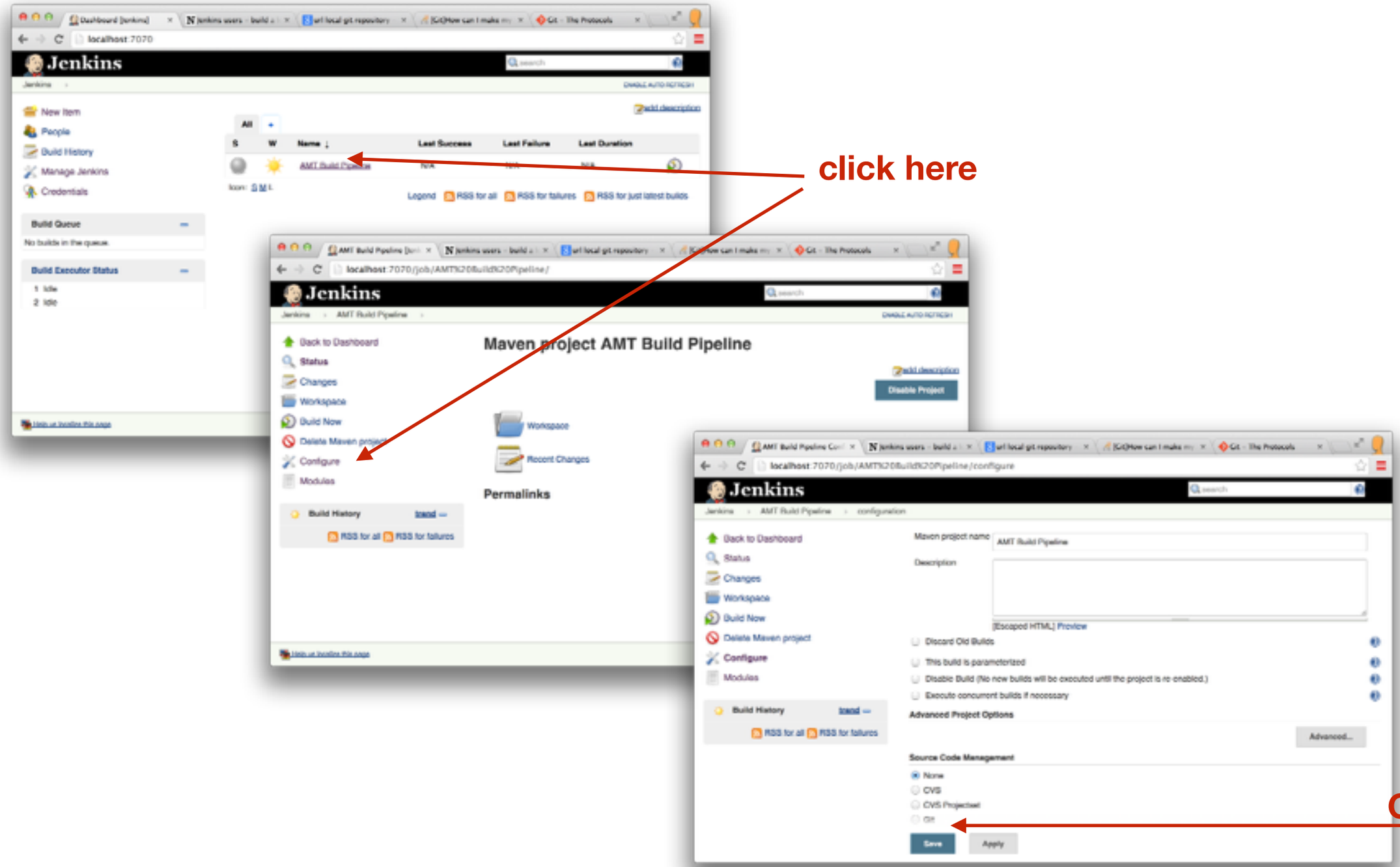
heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

- Doing a search on “git” gives us a list of plugins.
- Let’s install the Git Plugin. We might consider GitHub plugins later...



Back to our pipeline configuration

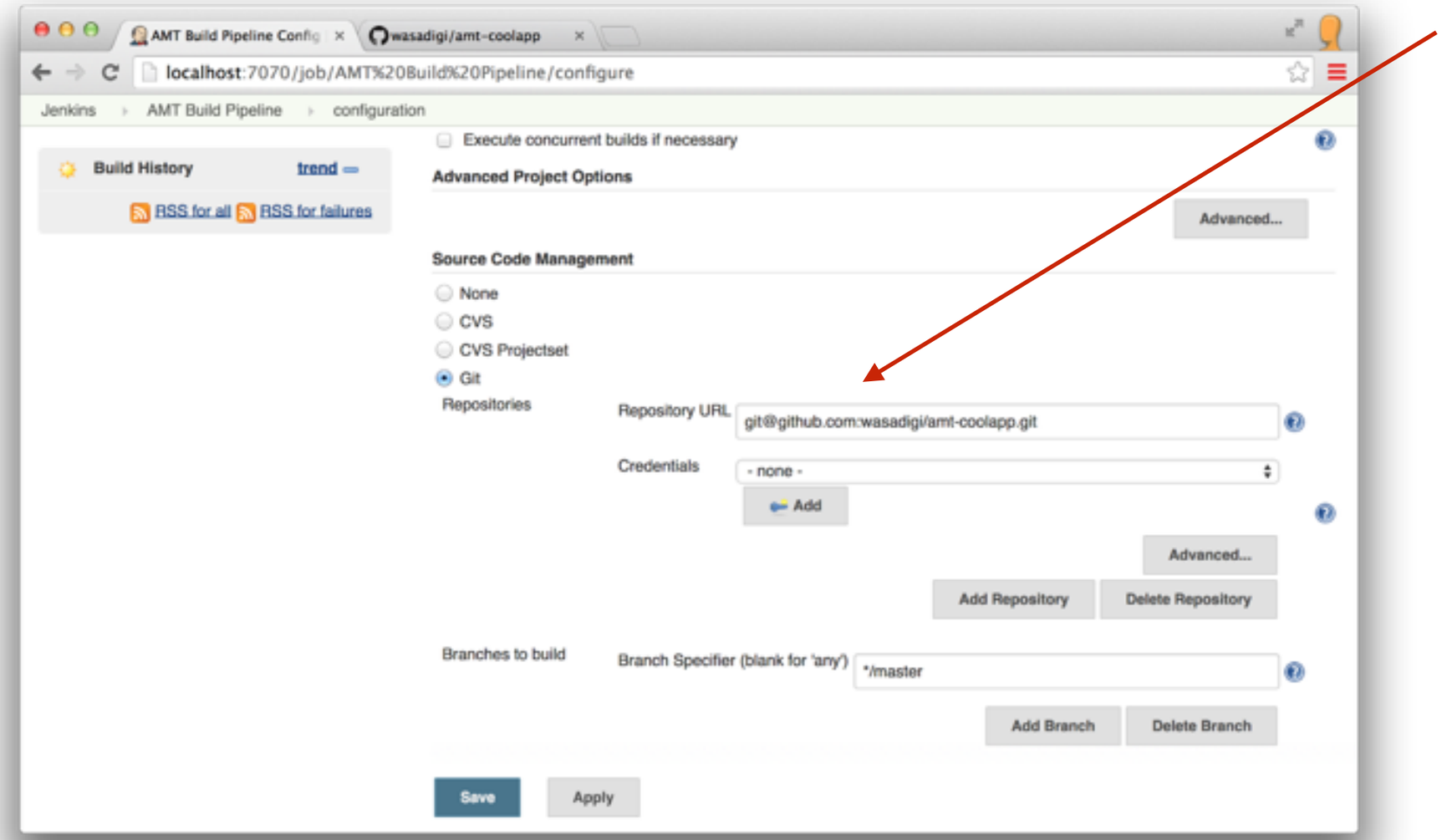
heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud



We need to provide the URL of a repo

heig-vd
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

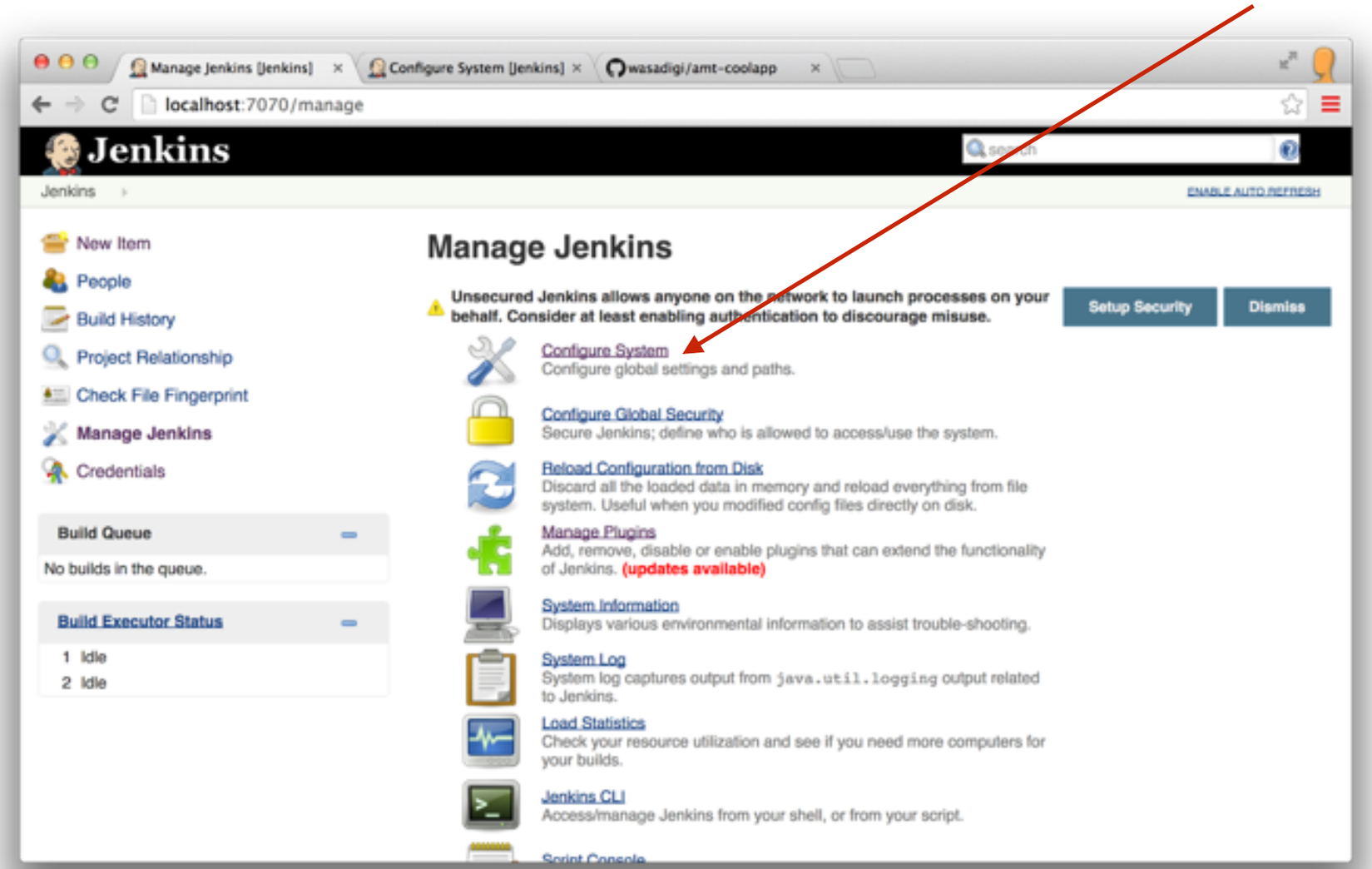
- In general, we use a remote URL (on GitHub, BitBucket, a company git server)
- It is also possible to use a local repo (`file:///mydir/myrepo`), for instance after doing a `git init` in the local directory.



Oops... we need to configure maven!

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

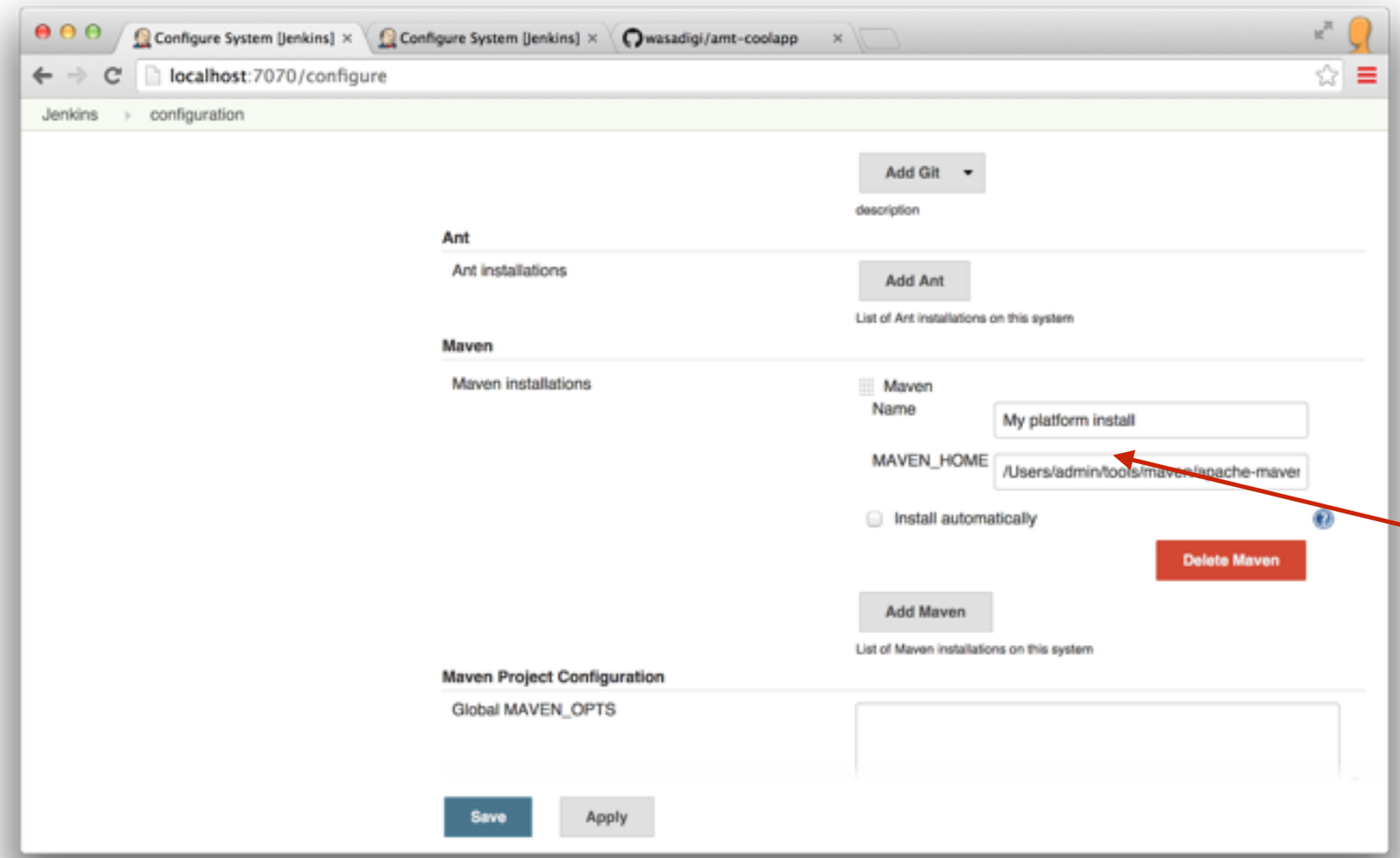
- Jenkins needs to know where maven is installed.
- We can either use our “global” installation, or let Jenkins use a local installation. Let’s use what we already have setup.



Oops... we need to configure maven!

heig-vd
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

- Jenkins needs to know where maven is installed.
- We can either use our “global” installation, or let Jenkins use a local installation. Let’s use what we already have setup.

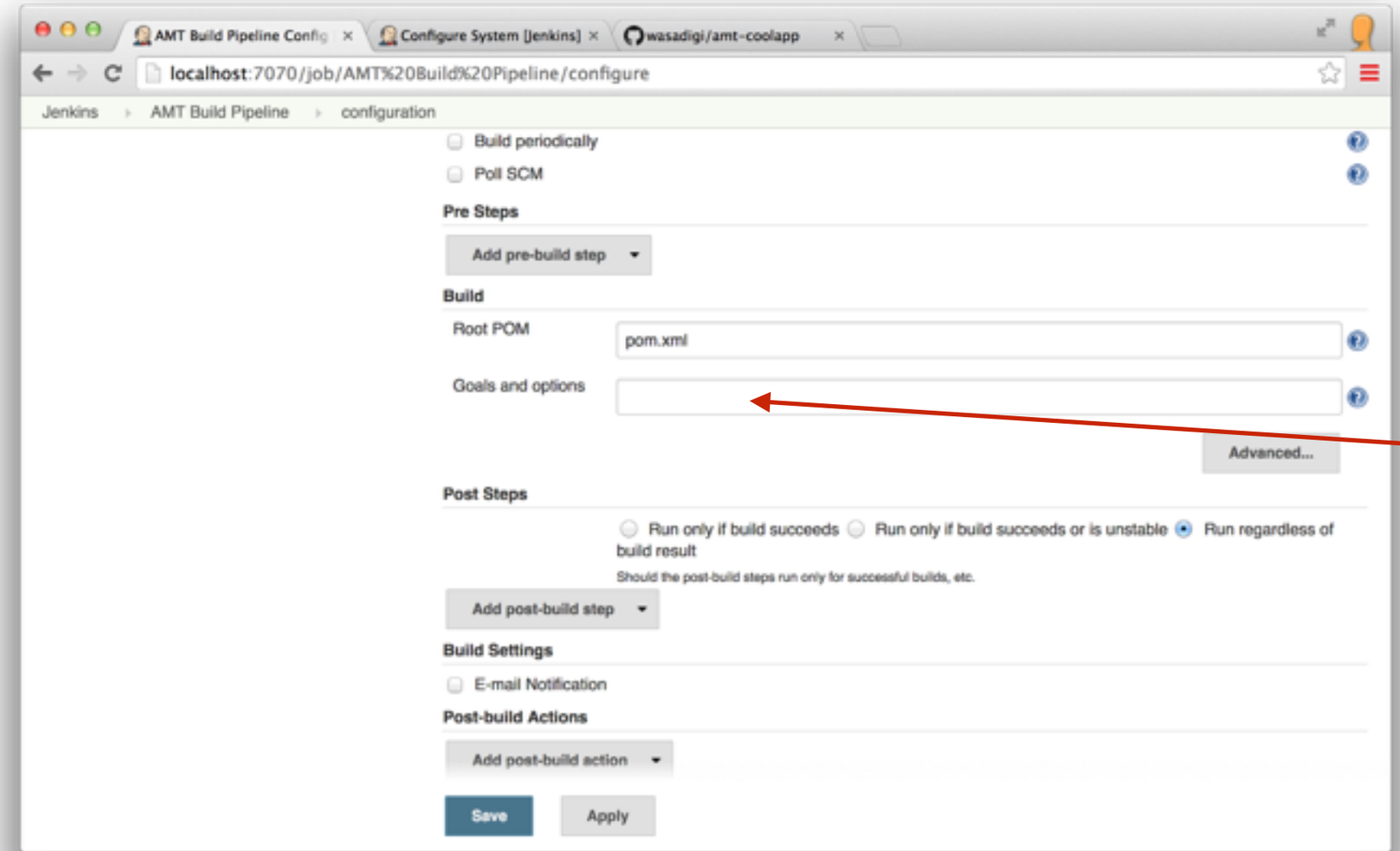


Indicate where you have installed maven on your machine in this field

Back to our pipeline configuration

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

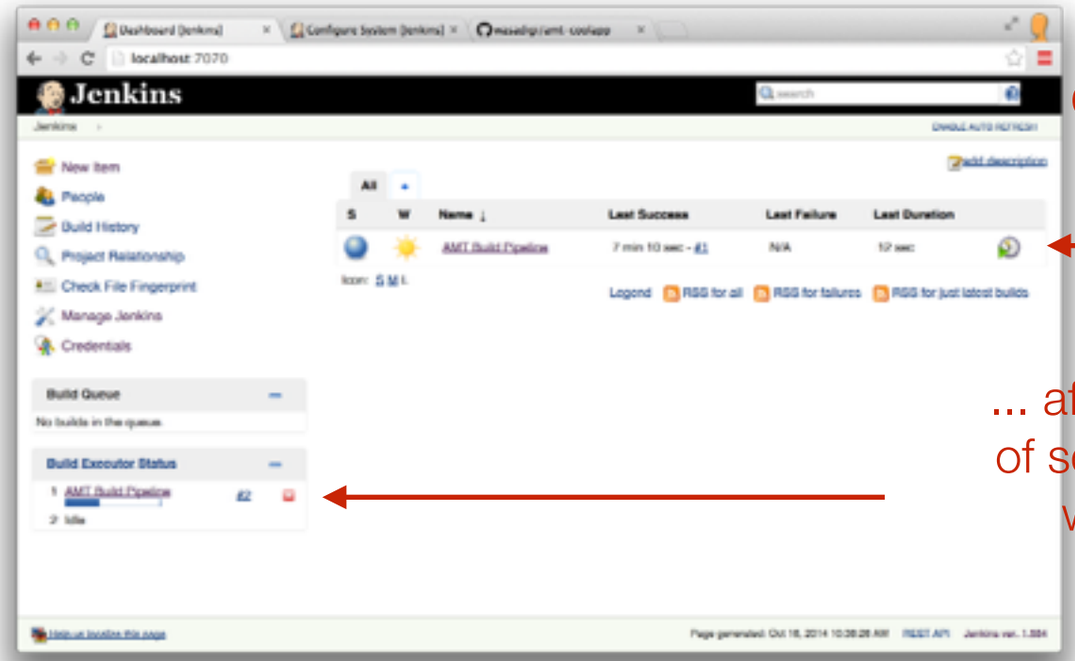
- We are now almost ready to go...



This is empty, so jenkins will simply do a 'mvn install'

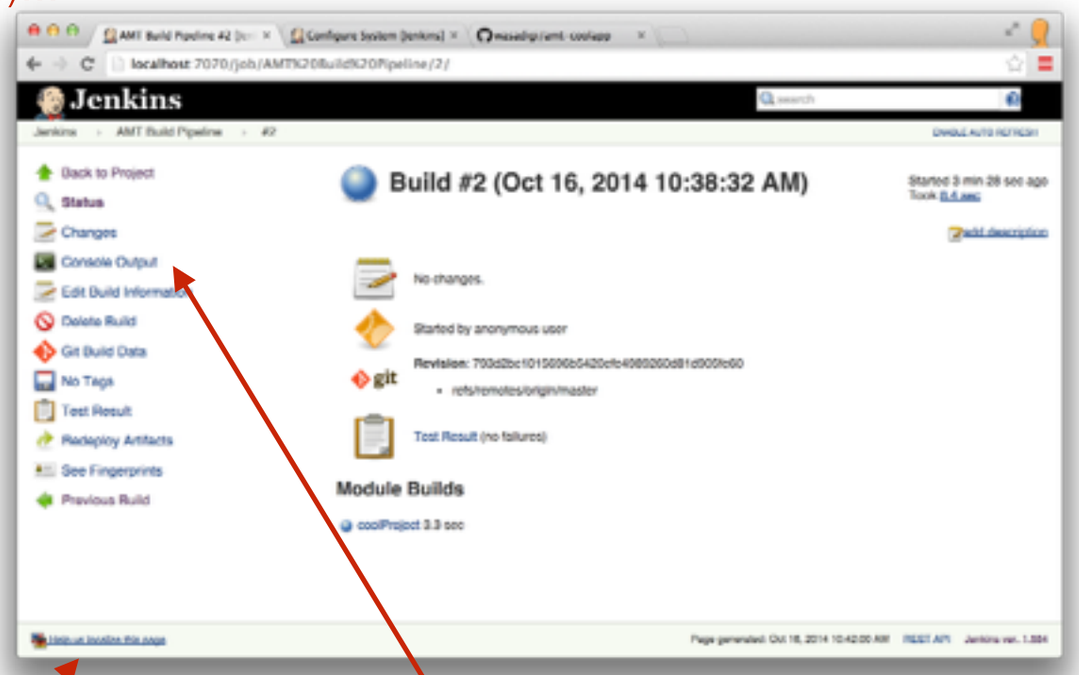
Doing a build and checking the outcome

heig-vd
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

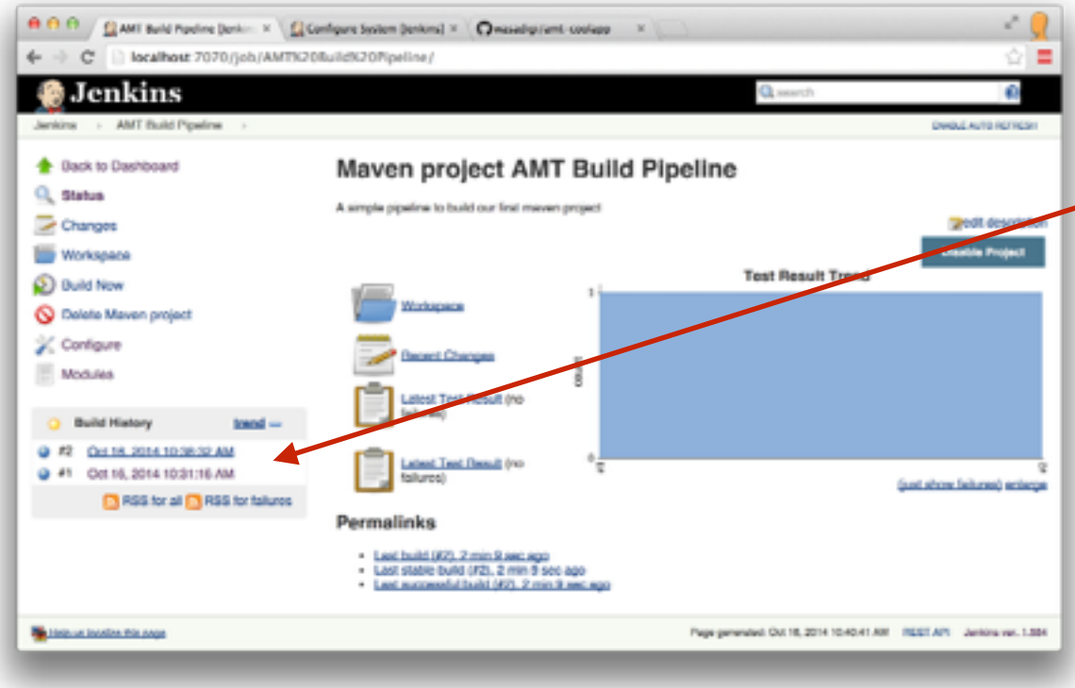


Clicking here will trigger the build (async)...

... after a couple of seconds, you will see a job here.



You see the history of previous builds for this pipeline...



... and you can get the details of one build

Click on Console Output to see what has been executed

Console Output

```
Started by user anonymous
Building in workspace /Users/admin/.jenkins/jobs/AMT Build Pipeline/workspace
 > git rev-parse --is-inside-work-tree # timeout=10
Fetching changes from the remote Git repository
 > git config remote.origin.url git@github.com:wasadigi/amt-coolapp.git # timeout=10
Fetching upstream changes from git@github.com:wasadigi/amt-coolapp.git
 > git --version # timeout=10
 > git fetch --tags --progress git@github.com:wasadigi/amt-coolapp.git +refs/heads/*:refs/remotes/origin/*
 > git rev-parse refs/remotes/origin/master^{commit} # timeout=10
 > git rev-parse refs/remotes/origin/origin/master^{commit} # timeout=10
Checking out Revision 793d2bc1015696b5420efe4989260d81d905fe60 (refs/remotes/origin/master)
 > git config core.sparsecheckout # timeout=10
 > git checkout -f 793d2bc1015696b5420efe4989260d81d905fe60
 > git rev-list 793d2bc1015696b5420efe4989260d81d905fe60 # timeout=10
Parsing POMs
[workspace] $ java -cp /Users/admin/.jenkins/plugins/maven-plugin/WEB-INF/lib/maven31-agent-1.5.jar:/Users/admin/tools/maven/apache-maven-3.1.0/boot/plexus-
classworlds-2.4.2.jar:/Users/admin/tools/maven/apache-maven-3.1.0/conf/logging jenkins.maven3.agent.Maven31Main /Users/admin/tools/maven/apache-
/Users/admin/.jenkins/war/WEB-INF/lib/remoting-2.46.jar /Users/admin/.jenkins/plugins/maven-plugin/WEB-INF/lib/maven31-interceptor-1.5.jar
/Users/admin/.jenkins/plugins/maven-plugin/WEB-INF/lib/maven3-interceptor-commons-1.5.jar 52457
<===[JENKINS REMOTING CAPACITY]===>channel started
Executing Maven:  -B -f /Users/admin/.jenkins/jobs/AMT Build Pipeline/workspace/pom.xml install
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building coolProject 1.0-SNAPSHOT
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] --- maven-resources-plugin:2.6:resources (default-resources) @ coolProject ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] skip non existing resourceDirectory /Users/admin/.jenkins/jobs/AMT Build Pipeline/workspace/src/main/resources
[INFO]
[INFO] --- maven-compiler-plugin:2.5.1:compile (default-compile) @ coolProject ---
[INFO] Nothing to compile - all classes are up to date
[INFO]
[INFO] --- maven-resources-plugin:2.6:testResources (default-testResources) @ coolProject ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] skip non existing resourceDirectory /Users/admin/.jenkins/jobs/AMT Build Pipeline/workspace/src/test/resources
[INFO]
[INFO] --- maven-compiler-plugin:2.5.1:testCompile (default-testCompile) @ coolProject ---
[INFO] Nothing to compile - all classes are up to date
[INFO]
[INFO] --- maven-surefire-plugin:2.12.4:test (default-test) @ coolProject ---
[INFO] Surefire report directory: /Users/admin/.jenkins/jobs/AMT Build Pipeline/workspace/target/surefire-reports

-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running ch.heigvd.amt.cool.AppTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.006 sec

Results :

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

[JENKINS] Recording test results
log4j:WARN No appenders could be found for logger (org.apache.commons.beanutils.converters.BooleanConverter).
log4j:WARN Please initialize the log4j system properly.
[INFO]
[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ coolProject ---
[INFO]
[INFO] --- maven-install-plugin:2.4:install (default-install) @ coolProject ---
[INFO] Installing /Users/admin/.jenkins/jobs/AMT Build Pipeline/workspace/target/coolProject-1.0-SNAPSHOT.jar to
/Users/admin/.m2/repository/ch/heigvd/amt/cool/coolProject/1.0-SNAPSHOT/coolProject-1.0-SNAPSHOT.jar
[INFO] Installing /Users/admin/.jenkins/jobs/AMT Build Pipeline/workspace/pom.xml to /Users/admin/.m2/repository/ch/heigvd/amt/cool/coolProject/1.0-
SNAPSHOT/coolProject-1.0-SNAPSHOT.pom
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 3.647s
[INFO] Finished at: Thu Oct 16 10:38:40 CEST 2014
[INFO] Final Memory: 12M/69M
```

Jenkins asks the Git plugin to fetch sources from the repo

Jenkins does a mvn install

All of this is the output of the mvn command

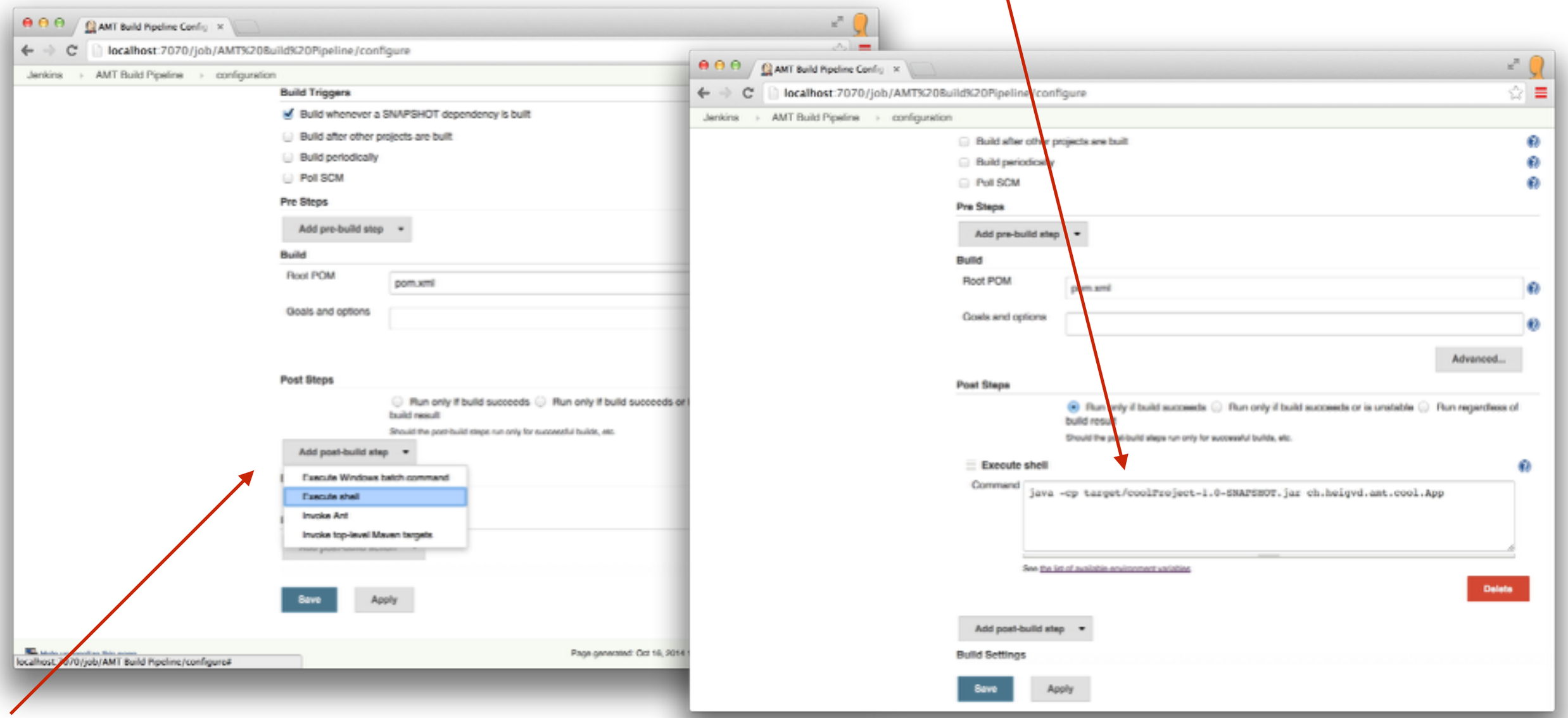
Our .jar file is installed in ~/.m2

Last thing: let's add a "post-build" step

heig-vd
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

- To demonstrate the feature, we will simply execute our Java app after building it (and expect to see the "Hello World!" message printed on the console output).

```
java -cp target/coolProject-1.0-SNAPSHOT.jar ch.heigvd.amt.cool.App
```



heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

Vagrant, Docker & Friends

Remember the RES lab from last year?

heig-vd
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

- You used **Vagrant** to setup and control a (VirtualBox) virtual machine.
  - There was a **Vagrantfile** that was describe the setup of the VM.
  - Doing a "**vagrant up**" was launching the VM
  - Doing a "vagrant ssh" allowed us to connect to the VM.
- You did not have to write the Vagrantfile from scratch, but you had to look at it to understand how the **port mapping** was configured.

Remember the RES lab from last year?

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

- You used **Docker** to run several lightweight containers in the VM.
- Within the VM, controlled via Vagrant, several "**micro VMs**" were launched. Each had its own operating system, its IP address, its file system, etc.
- Each Docker container was **running a single service:** there was a container running the "nginx reverse proxy" service, several containers running the "apache web server" service.
- **Docker containers** are running instances of **Docker images**. There were commands to list, create, delete images and containers.

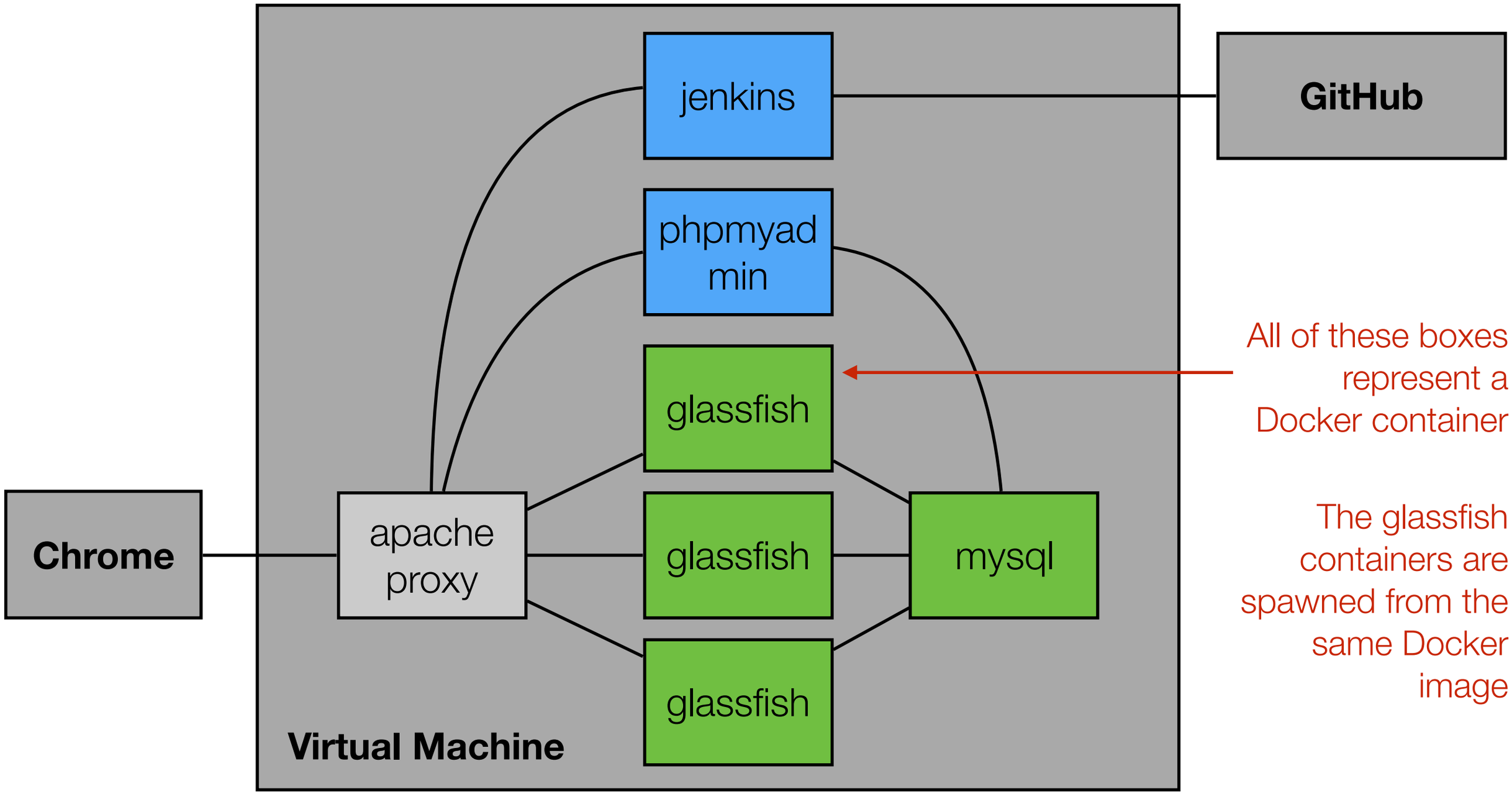
Can we use that for our Java EE work?

heig-vd
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

- We have already encountered **platform-specific issues**. For instance, we have written scripts that use asadmin. If we have two developers in a group who are using Mac OS, resp. Windows, they will have troubles.
- When we deploy an application in production, we will want to have a DMZ and a reverse proxy. Can we test that in our development environment as early as possible?
- Same thing about scalability: can we validate a setup, where we have several Glassfish servers behind a load balancer? Is there a way for a developer to test that on his machine?

The Big Picture

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud



All of these boxes represent a Docker container

The glassfish containers are spawned from the same Docker image

How should we build this?

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

- **Building this environment is going to take time.**
- You will not get an “end-to-end” recipe. You have to get your hands dirty, look for information, try things out by yourself. **We are here to guide you and answer your questions, but not to show you the end to end procedure.**
- It is also going to be an **iterative process**, which means that the environment will gradually become more and more sophisticated.
- Your environment will be **evaluated** at the end of the semester, as part of the second project.
- However, you should **start building it as soon as possible**, and you should **regularly invest time** into this activity because:
  - Having the environment will allow you to save precious time in the two projects!
  - If you wait too long, you will not be able to catch up and to do a proper job.

What is easy, what is hard?

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

- **Vagrant is very easy** to understand and to setup. After **1-2 hours**, you should be able to write your Vagrant file and to really understand what is going on behind the scenes.

- **Docker is way more complex.** Once you have understood core principles, such as the fact that containers are short-lived and meant to run a single service, you should be able to create your first images and run your first containers. **One full-day** of documentation reading and experimentation should give you a good basis.

  - http://docs.docker.com/introduction/understanding-docker/

- One of the things you will need to understand is how to **link docker** containers. Reading the doc will not take a long, but you will need to do a lot of experimentation. I would count at least **2 full days** for that.

  - http://docs.docker.com/userguide/dockerlinks/

  - http://docs.docker.com/articles/networking/

What is easy, what is hard?

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

- **Docker is way more complex.** Once you have understood core principles, such as the fact that containers are short-lived and meant to run a single service, you should be able to create your first images and run your first containers. **One full-day** of documentation reading and experimentation should give you a good basis.
    - http://docs.docker.com/introduction/understanding-docker/
- One of the things you will need to understand is how to **link docker** containers. Reading the doc will not take a long, but you will need to do a lot of experimentation. I would count at least **2 full days** for that.
    - http://docs.docker.com/userguide/dockerlinks/
    - http://docs.docker.com/articles/networking/
- Of course, we are not the first to do something like that. **Do some research, you will find descriptions of environments combining Docker, jenkins and Java EE**. Understand how they work, adapt and extend them based on your needs!

What is easy, what is hard?

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

- Independently from Vagrant and Docker, **Jenkins is pretty easy to understand and to use**:
    - You have seen the basics in the previous slides and this will address your basic needs.
    - Of course, the collection of plug-ins offers **endless** possibilities. So there is always the possibility to add features to your automation pipeline and to iteratively improve it over time.
    - There are a few gotchas and we will share some of them over the next couple of weeks. Be proactive when you encounter issues and be specific when you describe them.
    - One thing that you should try to do ASAP is to integrate the “DB and glassfish domain setup” tasks in a Jenkins pipeline.
- I don’t think that you will need more than a couple of hours to be at ease with the basics.

What is easy, what is hard?

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

- To do simple things with **maven**, you don't need to fully grasp the inner workings of the system (e.g. associations of plugins with phases of lifecycles, project inheritance).
- So at the minimum, what you should really understand is:
  - How to **invoke maven** (by invoking a phase or a plugin goal).
  - How to deal with **dependencies** and be clear on the difference between the **compile** and **provided** scopes.
  - Be clear on what "**install**" and "**deploy**" mean for maven.
  - Be clear on where you will find the output of the build process.
  - How to use **resource filtering** and **profiles** to make your build process customizable and shareable across team members and environments.
- The last part will require some experimentation time. I would say that after 1 full day of study and tests, you have a good grasp of the basics.

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

Next steps...

Lab: getting ready for the project

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

- During the first weeks of the semester, you have gradually built a simple end-to-end Java EE application.
- You have also created a script to automate the setup of your environment (database, glassfish domain, etc.).
- **What I have seen in most of the GitHub repositories:**
  - Partially implemented features, with “hacks” and “TODOs” (e.g. HTML generated in servlets...).
  - Almost no code documentation.
  - IDE-generated header files.
  - French comments, even some french identifiers.
  - Messy repositories, with a lot of “dead code” and legacy files.
  - Poor commit messages.
- **All of that is fine since you were learning and experimenting, but now is the time to clean up (and change habits)!**

You job until next lecture (after vacactions)

heig-vd
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

- Create a **maven project** that will generate a .war artifact.
- **Cleanup** your existing code and move it to the new project structure. Make sure that you can build the .war file using “mvn install” and that the result can be deployed in Glassfish.
- Create a **jenkins pipeline** and make sure that you can build and deploy your project via jenkins.
- Do the **DB and domain setup** in the jenkins pipeline.
- Create a new Github repo and make sure that it is clean. When we get to it, it should be easy for us to know how to build, deploy and run your code (what are the requirements, etc.)
- **This will be the starting point for the first project!** You can work in teams of 2. Please send us an email with the names of the team members and the URL of the GitHub repo.